INTRO TO PROGRAMMING WITH PROGRAMMING FUNDAMENTALS AND PYTHON FOR EVERYONE



April Browne
Butte College

LibreTexts™

Intro to Programming with Programming Fundamentals and Python for Everyone

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden borne from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and were supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

The LibreTexts name, logo, and book covers are trademarked by LibreTexts, Inc. (a 501(c)(3) not-for-profit organization) and protected against unauthorized use.

This text was compiled on 10/14/2026

TABLE OF CONTENTS

Licensing

1: Welcome to Programming

- 1.1: What is a program?
- 1.2: Integrated Development Environment
- 1.3: Terminology - Interpreter and Compiler
- 1.4: Why should you learn to write programs?
- 1.5: Creativity and Motivation
- 1.6: Understanding Programming
- 1.7: The Learning Journey

2: Commands

- 2.1: Hello World
- 2.2: Words and Sentences
- 2.3: Conversing with Python
- 2.4: What could possibly go wrong?
- 2.5: Comments
- 2.6: Comments

3: Principles of Structured Programming

- 3.1: The Building Blocks of Programs
- 3.2: Structured Programming

4: Functions and Events

- 4.1: Modular Programming
- 4.2: Why functions?
- 4.3: Function Calls
- 4.4: Definitions and Uses
- 4.5: Flow of Execution
- 4.6: Parameters and Arguments
- 4.7: Parameters and Arguments

5: Algorithms and Design

- 5.1: Thinking Algorithmically
- 5.2: Program Design
- 5.3: Program Quality

6: Math and Variables

- 6.1: Arithmetic Operators
- 6.2: Order of Operations
- 6.3: Constants and Variables
- 6.4: Identifier Names

6.5: Variable names and Keywords
6.6: Choosing Mnemonic Variable Names
6.7: Statements
6.8: Assignment
6.9: Variables
6.10: Updating Variables
6.11: Python Examples
6.12: Arrays and Lists
6.13: Index Notation
6.14: A list is a sequence
6.15: Lists are mutable
6.16: Debugging

7: Problem Solving and Design

7.1: Pseudocode
7.2: Flowcharts
7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms

8: Decisions

8.1: Relational Operators
8.2: Boolean Expressions
8.3: Conditional Execution
8.4: If Then Else
8.5: Alternative Execution
8.6: Assignment vs Equality
8.7: Code Blocks
8.8: Chained Conditionals

9: Helper Functions

9.1: Adding New Functions
9.2: Definitions and Uses
9.3: Why functions?
9.4: Flow of Execution
9.5: Function Examples
9.6: Hierarchy or Structure Chart

10: Nested Decision

10.1: Logical Operators
10.2: Logical Operators
10.3: Nested If Then Else
10.4: Condition Examples
10.5: Python Examples
10.6: Nested Conditionals

10.7: Catching exceptions Using Try and Except

11: Software Development Life Cycle

11.1: Systems Development Life Cycle

11.2: Software Testing

11.3: Version Control

11.4: Programming Style

12: Variable Scope

12.1: Scope

13: For Loops

13.1: Iteration Control Structures

13.2: For Loop

13.3: Definite loops using for

13.4: Counting and Summing Loops

14: More Math and Random Numbers

14.1: Integer Division and Modulus

14.2: Modulus Operator

14.3: Standard Libraries

14.4: Random Numbers

14.5: Math Functions

15: Nested For Loops

15.1: Nested For Loops

16: Data Types and User Input

16.1: Data Types

16.2: Integer Data Type

16.3: Floating-Point Data Type

16.4: String Data Type

16.5: Boolean Data Type

16.6: Values and Types

16.7: Data Type Conversions

16.8: Input and Output

16.9: Asking the user for input

17: Strings

17.1: Strings

17.2: A string is a sequence

17.3: String Functions

17.4: String Operations

17.5: Getting the length of a string using len

17.6: Traversal through a string with a loop

17.7: String Slices
17.8: Strings are immutable
17.9: Looping and Counting
17.10: The in operator
17.11: String Comparison
17.12: String Methods
17.13: Parsing strings
17.14: String Formatting
17.15: Format operator

18: While Loops

18.1: While Loop
18.2: The while Statement
18.3: Flag Concept
18.4: Infinite Loops
18.5: Loop patterns
18.6: Maximum and Minimum Loops
18.7: Loop Examples
18.8: Python Examples

19: Lists

19.1: Arrays and Lists
19.2: Index Notation
19.3: Displaying Array Members
19.4: Traversing a List
19.5: List operations
19.6: List Slices
19.7: List Methods
19.8: Deleting Elements
19.9: Math Statistics with Arrays
19.10: Searching Arrays
19.11: Sorting Arrays
19.12: Parallel Arrays
19.13: Multidimensional Arrays

20: Function Return

20.1: Return Statement
20.2: Fruitful functions and void functions
20.3: Python Examples

21: File Input/Output

21.1: File Input and Output
21.2: Opening Files

21.3: Text files and Lines
21.4: Reading Files
21.5: Writing Files
21.6: Using try, except, and open
21.7: Loading an Array from a Text File
21.8: Letting the user choose the file name
21.9: Searching through a File
21.10: Python Examples

22: Searching and Sorting Algorithms

22.1: Activity 1 - Searching Algorithm
22.2: Activity 2 - Sorting Algorithm

23: Object Oriented Programming

23.1: Managing Larger Programs
23.2: Getting Started
23.3: Objects and Classes
23.4: Using Objects
23.5: Our First Python Object
23.6: Python Examples

24: Networked Programs

24.1: HyperText Transfer Protocol - HTTP
24.2: The World's Simplest Web Browser
24.3: Retrieving an image over HTTP
24.4: Retrieving web pages with urllib
24.5: Parsing HTML and scraping the web
24.6: Parsing HTML using regular expressions
24.7: Parsing HTML using BeautifulSoup
24.8: Reading binary files using urllib
24.E: Networked Programs (Exercises)
24.G: Networked Programs (Glossary)

25: Python and Web Services

25.1: eXtensible Markup Language - XML
25.2: Looping through Nodes
25.3: JavaScript Object Notation - JSON
25.4: Parsing JSON
25.5: Application Programming Interfaces
25.6: Google geocoding web service
25.7: Security and API usage
25.E: Python and Web Services (Exercises)
25.G: Python and Web Services (Glossary)

26: Algorithms

26.1: Activity 1 - Overview of Algorithm Design and Analysis

26.2: Activity 1 - Introduction to Algorithms and Problem Solving

26.3: Activity 2 - The characteristics of an algorithm

27: Additional Fundamentals of Programming

27.1: Introduction

27.1.1: Computer Hardware Architecture

27.1.2: Writing a Program

27.1.E: Introduction (Exercises)

27.1.G: Introduction (Glossary)

27.2: Introduction to Programming

27.2.1: C++ Examples

27.2.2: C# Examples

27.2.3: Java Examples

27.2.4: JavaScript Examples

27.2.5: Swift Examples

27.2.6: Practice- Introduction to Programming

27.3: Data and Operators

27.3.1: JavaScript Examples

27.3.2: Swift Examples

27.3.3: Practice- Data and Operators

27.3.4: Nothing Data Type

27.3.5: Lvalue and Rvalue

27.3.6: Input-Process-Output Model

27.3.7: C++ Examples

27.3.8: C# Examples

27.3.9: Java Examples

27.4: Functions

27.4.1: Call by Value vs. Call by Reference

27.4.2: Void Data Type

27.4.3: C++ Examples

27.4.4: C# Examples

27.4.5: Java Examples

27.4.6: JavaScript Examples

27.4.7: Swift Examples

27.4.8: Practice- Functions

27.5: Conditions

27.5.1: Case Control Structure

27.5.2: C++ Examples

27.5.3: C# Examples
27.5.4: Java Examples
27.5.5: JavaScript Examples
27.5.6: Swift Examples
27.5.7: Practice- Conditions
27.5.8: Selection Control Structures

27.6: Loops
27.6.1: Do While Loop
27.6.2: Branching Statements
27.6.3: Integer Overflow
27.6.4: C++ Examples
27.6.5: C# Examples
27.6.6: Java Examples
27.6.7: JavaScript Examples
27.6.8: Swift Examples
27.6.9: Practice- Loops

27.7: Arrays and Lists
27.7.1: Arrays and Functions
27.7.2: Dynamic Arrays
27.7.3: C++ Examples
27.7.4: C# Examples
27.7.5: Java Examples
27.7.6: JavaScript Examples
27.7.7: Python Examples
27.7.8: Swift Examples
27.7.9: Practice- Arrays and Lists

27.8: Strings and Files
27.8.1: C++ Examples
27.8.2: C# Examples
27.8.3: Java Examples
27.8.4: JavaScript Examples
27.8.5: Swift Examples
27.8.6: Practice- Strings and Files

27.9: Object-Oriented Programming
27.9.1: Encapsulation
27.9.2: Inheritance and Polymorphism
27.9.3: C++ Examples
27.9.4: C# Examples
27.9.5: Java Examples

27.9.6: JavaScript Examples
27.9.7: Swift Examples
27.9.8: Practice

28: Additional Python 4 Everyone

28.1: A trip to Jupyter

28.1.1: Jupyter Notebooks
28.1.2: Code and Output

28.2: Variables, Expressions, and Statements

28.2.1: Variable names and Keywords
28.2.2: Operators and Operands
28.2.3: Expressions
28.2.4: Order of Operations
28.2.5: Choosing Mnemonic Variable Names
28.2.6: Debugging
28.2.E: Variables, Expressions, and Statements (Exercises)
28.2.G: Glossary

28.3: Conditional Execution

28.3.1: Short-Circuit Evaluation of Logical Expressions
28.3.2: Debugging
28.3.E: Conditional Execution (Exercises)
28.3.G: Conditional Execution (Glossary)

28.4: Functions

28.4.1: Function Calls
28.4.2: Built-in Functions
28.4.3: Type Conversion Functions
28.4.4: Parameters and Arguments
28.4.5: Debugging
28.4.E: Functions (Exercises)
28.4.G: Functions (Glossary)

28.5: Iterations

28.5.1: "Infinite loops" and break
28.5.2: Finishing iterations with continue
28.5.3: Debugging
28.5.E: Iterations (Exercises)
28.5.G: Iterations (Glossary)

28.6: Strings

28.6.1: Debugging
28.6.E: Strings (Exercises)
28.6.G: Strings (Glossary)

28.7: Files
28.7.1: Persistence
28.7.2: Debugging
28.7.E: Files (Exercises)
28.7.G: Files (Glossary)

28.8: Lists
28.8.1: Lists and Functions
28.8.2: Lists and Strings
28.8.3: Parsing lines
28.8.4: Objects and Values
28.8.5: Aliasing
28.8.6: List arguments
28.8.7: Debugging
28.8.E: Lists (Exercises)
28.8.G: Lists (Glossary)

28.9: Object-Oriented Programming
28.9.1: Starting with Programs
28.9.2: Subdividing a Problem - Encapsulation
28.9.3: Classes as Types
28.9.4: Many Instances
28.9.5: Object Lifecycle
28.9.6: Inheritance
28.9.G: Object-Oriented Programming (Glossary)
28.9.S: Object-Oriented Programming (Summary)

Index
Index
Glossary
Detailed Licensing
Detailed Licensing
Detailed Licensing

This page titled Intro to Programming with Programming Fundamentals and Python for Everyone is shared under a license and was authored, remixed, and/or curated by LibreTexts.

Licensing

A detailed breakdown of this resource's licensing can be found in ***Back Matter/Detailed Licensing***.

This page titled Licensing is shared under a license and was authored, remixed, and/or curated by LibreTexts.

1: WELCOME TO PROGRAMMING

1.1: What is a program?

1.2: Integrated Development Environment

1.3: Terminology - Interpreter and Compiler

1.4: Why should you learn to write programs?

1.5: Creativity and Motivation

1.6: Understanding Programming

1.7: The Learning Journey

This page titled 1: Welcome to Programming is shared under a license and was authored, remixed, and/or curated by LibreTexts.

1.1: What is a program?

The definition of a *program* at its most basic is a sequence of Python statements that have been crafted to do something. Even our simple *hello.py* script is a program. It is a one-line program and is not particularly useful, but in the strictest definition, it is a Python program.

It might be easiest to understand what a program is by thinking about a problem that a program might be built to solve, and then looking at a program that would solve that problem.

Lets say you are doing Social Computing research on Facebook posts and you are interested in the most frequently used word in a series of posts. You could print out the stream of Facebook posts and pore over the text looking for the most common word, but that would take a long time and be very mistake prone. You would be smart to write a Python program to handle the task quickly and accurately so you can spend the weekend doing something fun.

For example, look at the following text about a clown and a car. Look at the text and figure out the most common word and how many times it occurs.

```
    the clown ran after the car and the car ran into the tent
    and the tent fell down on the clown and the car
```

Then imagine that you are doing this task looking at millions of lines of text. Frankly it would be quicker for you to learn Python and write a Python program to count the words than it would be to manually scan the words.

The even better news is that I already came up with a simple program to find the most common word in a text file. I wrote it, tested it, and now I am giving it to you to use so you can save some time.

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)

# Code: http://www.py4e.com/code3/words.py
```

You don't even need to know Python to use this program. You will need to get through Chapter 10 of this book to fully understand the awesome Python techniques that were used to make the program. You are the end user, you simply use the program and marvel at its cleverness and how it saved you so much manual effort. You simply type the code into a file called *words.py* and run it or you download the source code from http://www.py4e.com/code3/ and run it.

This is a good example of how Python and the Python language are acting as an intermediary between you (the end user) and me (the programmer). Python is a way for us to exchange useful instruction sequences (i.e., programs) in a common language that can be used by anyone who installs Python on their computer. So neither of us are talking *to Python*, instead we are communicating

with each other *through* Python.

This page titled 1.1: What is a program? is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.9: What is a program? by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

1.2: Integrated Development Environment

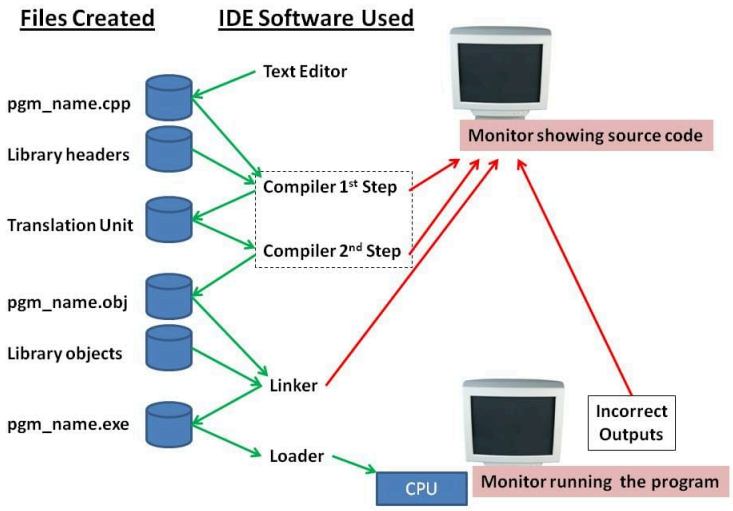
Overview

An **integrated development environment (IDE)** is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools, and a debugger. Most modern IDEs have intelligent code completion. Some IDEs contain a compiler, interpreter, or both. The boundary between an integrated development environment and other parts of the broader software development environment is not well-defined. Sometimes a version control system, or various tools to simplify the construction of a graphical user interface (GUI), are integrated. Many modern IDEs also have a class browser, an object browser, and a class hierarchy diagram, for use in object-oriented software development.[1]

Discussion

High-level language programs are usually written (coded) as ASCII text into a source code file. A unique file extension (Examples: .asm .c .cpp .java .js .py) is used to identify it as a source code file. As you might guess for our examples – Assembly, “C”, “C++”, Java, JavaScript, and Python, however, they are just ASCII text files (other text files usually use the extension of .txt). The source code produced by the programmer must be converted to an executable machine code file specifically for the computer’s CPU (usually an Intel or Intel-compatible CPU within today’s world of computers). There are several steps in getting a program from its source code stage to running the program on your computer. Historically, we had to use several software programs (a text editor, a compiler, a linker, and operating system commands) to make the conversion and run our program. However, today all those software programs with their associated tasks have been **integrated** into one program. However, this one program is really many software items that create an **environment** used by programmers to **develop** software. Thus the name: Integrated Development Environment or IDE.

Programs written in a high-level language are either directly executed by some kind of interpreter or converted into machine code by a compiler (and assembler and linker) for the CPU to execute. JavaScript, Perl, Python, and Ruby are examples of interpreted programming languages. C, C++, C#, Java, and Swift are examples of compiled programming languages.[2] The following figure shows the progression of activity in an IDE as a programmer enters the source code and then directs the IDE to compile and run the program.



Integrated Development Environment or IDE

Upon starting the IDE software the programmer usually indicates the file he or she wants to open for editing as source code. As

they make changes they might either do a “save as” or “save”. When they have finished entering the source code, they usually direct the IDE to “compile & run” the program. The IDE does the following steps:

1. If there are any unsaved changes to the source code file it has the **test editor** save the changes.
2. The **compiler** opens the source code file and does its **first step** which is executing the **pre-processor** compiler directives and other steps needed to get the file ready for the second step. The #include will insert header files into the code at this point. If it encounters an error, it stops the process and returns the user to the source code file within the text editor with an error message. If no problems encountered it saves the source code to a temporary file called a translation unit.
3. The **compiler** opens the translation unit file and does its **second step** which is **converting** the programming language code to machine instructions for the CPU, a data area, and a list of items to be resolved by the linker. Any problems encountered (usually a syntax or violation of the programming language rules) stops the process and returns the user to the source code file within the text editor with an error message. If no problems encountered it saves the machine instructions, data area, and linker resolution list as an object file.
4. The **linker** opens the program object file and links it with the library object files as needed. Unless all linker items are resolved, the process stops and returns the user to the source code file within the text editor with an error message. If no problems encountered it saves the linked objects as an executable file.
5. The IDE directs the operating system’s program called the **loader** to load the executable file into the computer’s memory and have the Central Processing Unit (CPU) start processing the instructions. As the user interacts with the program, entering test data, he or she might discover that the outputs are not correct. These types of errors are called logic errors and would require the user to return to the source code to change the algorithm.

Resolving Errors

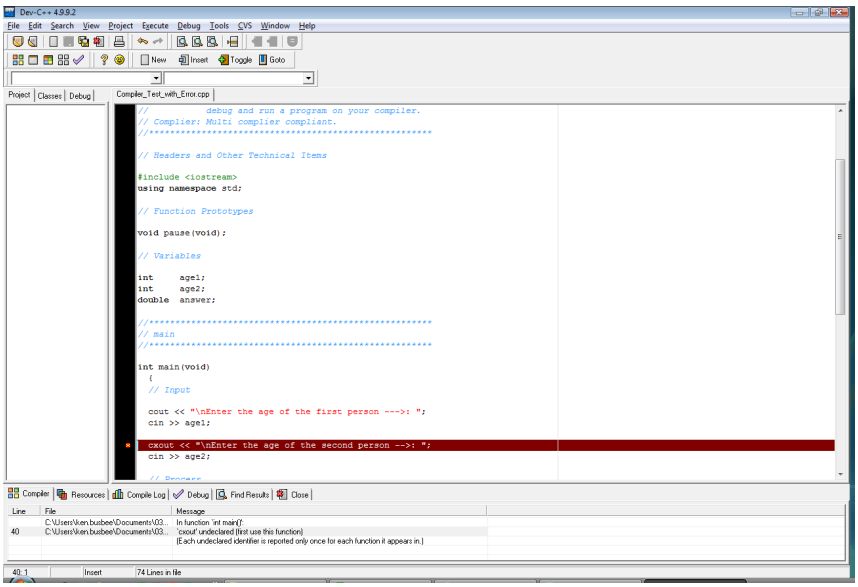
Despite our best efforts at becoming perfect programmers, we will create errors. Solving these errors is known as **debugging** your program. The three types of errors in the order that they occur are:

1. Compiler
2. Linker
3. Logic

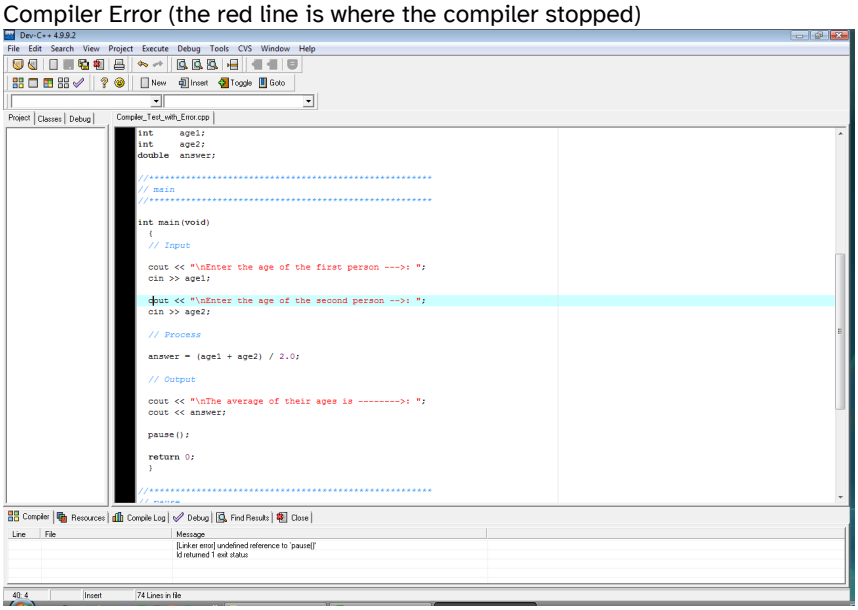
There are two types of compiler errors; pre-processor (1st step) and conversion (2nd step). A review of Figure 1 above shows the four arrows returning to the source code so that the programmer can correct the mistake.

During the conversion (2nd step) the compiler might give a **warning** message which in some cases may not be a problem to worry about. For example: Data type demotion may be exactly what you want your program to do, but most compilers give a warning message. Warnings don’t stop the compiling process but as their name implies, they should be reviewed.

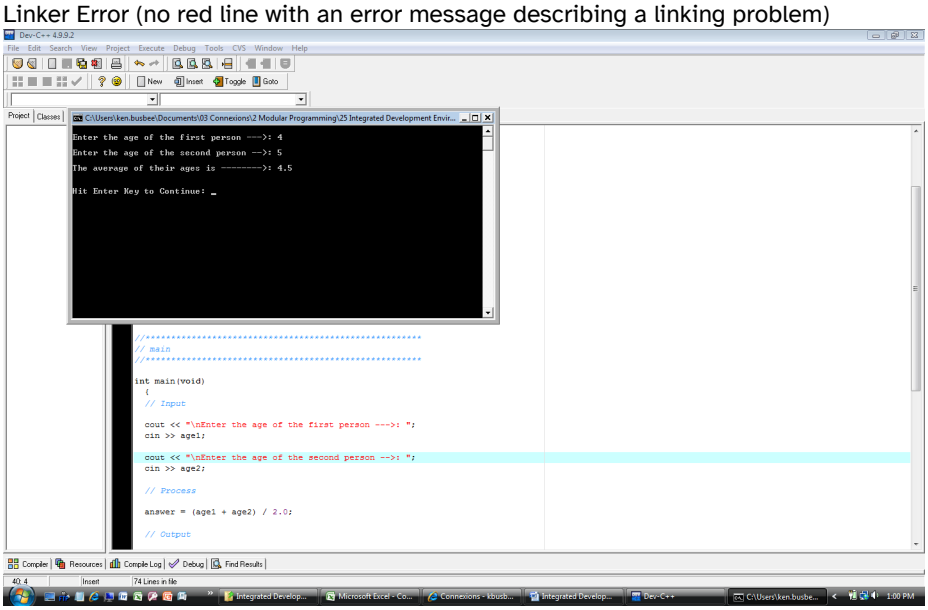
The next three figures show IDE monitor interaction for the **Bloodshed Dev-C++ 5 compiler/IDE**.



Compiler Error (the red line is where the compiler stopped)



Linker Error (no red line with an error message describing a linking problem)



Logic Error (from the output within the “Black Box” area)

Key Terms

compiler
: Converts source code to object code.

debugging
: The process of removing errors from a program. 1) compiler 2) linker 3) logic

linker
: Connects or links object files into an executable file.

loader
: Part of the operating system that loads executable files into memory and directs the CPU to start running the program.

pre-processor
: The first step the compiler does in converting source code to object code.

text editor
: A software program for creating and editing ASCII text files.

warning
: A compiler alert that there might be a problem.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Integrated development environment ↵
2. Wikipedia: Interpreter (computing) ↵

This page titled 1.2: Integrated Development Environment is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

1.3: Terminology - Interpreter and Compiler

Python is a *high-level* language intended to be relatively straightforward for humans to read and write and for computers to read and process. Other high-level languages include Java, C++, PHP, Ruby, Basic, Perl, JavaScript, and many more. The actual hardware inside the Central Processing Unit (CPU) does not understand any of these high-level languages.

The CPU understands a language we call *machine language*. Machine language is very simple and frankly very tiresome to write because it is represented all in zeros and ones:

```
001010001110100100101010000001111
111001100000111010100101011101101
...
```

Machine language seems quite simple on the surface, given that there are only zeros and ones, but its syntax is even more complex and far more intricate than Python. So very few programmers ever write machine language. Instead we build various translators to allow programmers to write in high-level languages like Python or JavaScript and these translators convert the programs to machine language for actual execution by the CPU.

Since machine language is tied to the computer hardware, machine language is not *portable* across different types of hardware. Programs written in high-level languages can be moved between different computers by using a different interpreter on the new machine or recompiling the code to create a machine language version of the program for the new machine.

These programming language translators fall into two general categories: (1) interpreters and (2) compilers.

An *interpreter* reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly. Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python.

Some of the lines of Python tell Python that you want it to remember some value for later. We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later. We use the term *variable* to refer to the labels we use to refer to this stored data.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

In this example, we ask Python to remember the value six and use the label *x* so we can retrieve the value later. We verify that Python has actually remembered the value using *print*. Then we ask Python to retrieve *x* and multiply it by seven and put the newly computed value in *y*. Then we ask Python to print out the value currently in *y*.

Even though we are typing these commands into Python one line at a time, Python is treating them as an ordered sequence of statements with later statements able to retrieve data created in earlier statements. We are writing our first simple paragraph with four sentences in a logical and meaningful order.

It is the nature of an *interpreter* to be able to have an interactive conversation as shown above. A *compiler* needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.

If you have a Windows system, often these executable machine language programs have a suffix of ".exe" or ".dll" which stand for "executable" and "dynamic link library" respectively. In Linux and Macintosh, there is no suffix that uniquely marks a file as executable.

If you were to open an executable file in a text editor, it would look completely crazy and be unreadable:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

It is not easy to read or write machine language, so it is nice that we have *interpreters* and *compilers* that allow us to write in high-level languages like Python or C.

Now at this point in our discussion of compilers and interpreters, you should be wondering a bit about the Python interpreter itself. What language is it written in? Is it written in a compiled language? When we type "python", what exactly is happening?

The Python interpreter is written in a high-level language called "C". You can look at the actual source code for the Python interpreter by going to www.python.org and working your way to their source code. So Python is a program itself and it is compiled into machine code. When you installed Python on your computer (or the vendor installed it), you copied a machine-code copy of the translated Python program onto your system. In Windows, the executable machine code for Python itself is likely in a file with a name like:

```
C:\Python35\python.exe
```

That is more than you really need to know to be a Python programmer, but sometimes it pays to answer those little nagging questions right at the beginning.

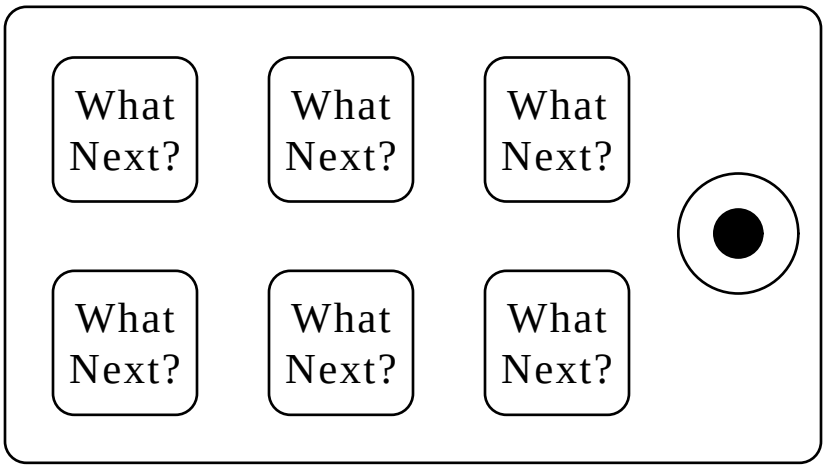
This page titled 1.3: Terminology - Interpreter and Compiler is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.7: Terminology - Interpreter and Compiler by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

1.4: Why should you learn to write programs?

Writing programs (or programming) is a very creative and rewarding activity. You can write programs for many reasons, ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem. This book assumes that *everyone* needs to know how to program, and that once you know how to program you will figure out what you want to do with your newfound skills.

We are surrounded in our daily lives with computers ranging from laptops to cell phones. We can think of these computers as our "personal assistants" who can take care of many things on our behalf. The hardware in our current-day computers is essentially built to continuously ask us the question, "What would you like me to do next?"



Personal Digital Assistant

Programmers add an operating system and a set of applications to the hardware and we end up with a Personal Digital Assistant that is quite helpful and capable of helping us do many different things.

Our computers are fast and have vast amounts of memory and could be very helpful to us if we only knew the language to speak to explain to the computer what we would like it to "do next". If we knew this language, we could tell the computer to do tasks on our behalf that were repetitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing.

For example, look at the first three paragraphs of this chapter and tell me the most commonly used word and how many times the word is used. While you were able to read and understand the words in a few seconds, counting them is almost painful because it is not the kind of problem that human minds are designed to solve. For a computer the opposite is true, reading and understanding text from a piece of paper is hard for a computer to do but counting the words and telling you how many times the most used word was used is very easy for the computer:

```
01    python words.py
02    Enter file:words.txt
03    to 16
```

Our "personal information analysis assistant" quickly told us that the word "to" was used sixteen times in the first three paragraphs of this chapter.

This very fact that computers are good at things that humans are not is why you need to become skilled at talking "computer language". Once you learn this new language, you can delegate mundane tasks to your partner (the computer), leaving more time for you to do the things that you are uniquely suited for. You bring creativity, intuition, and inventiveness to this partnership.

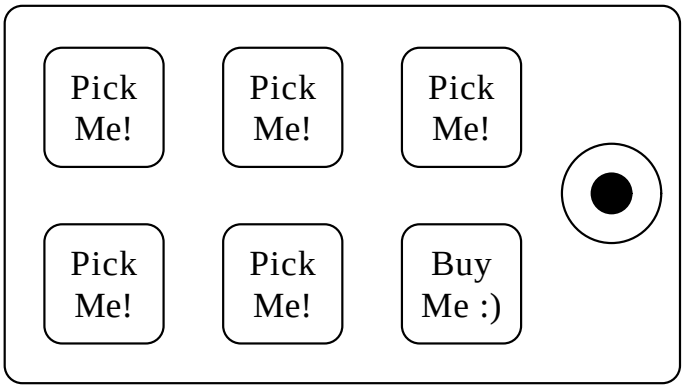
This page titled 1.4: Why should you learn to write programs? is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.1: Why should you learn to write programs? by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

1.5: Creativity and Motivation

While this book is not intended for professional programmers, professional programming can be a very rewarding job both financially and personally. Building useful, elegant, and clever programs for others to use is a very creative activity. Your computer or Personal Digital Assistant (PDA) usually contains many different programs from many different groups of programmers, each competing for your attention and interest. They try their best to meet your needs and give you a great user experience in the process. In some situations, when you choose a piece of software, the programmers are directly compensated because of your choice.

If we think of programs as the creative output of groups of programmers, perhaps the following figure is a more sensible version of our PDA:



Programmers Talking to You

For now, our primary motivation is not to make money or please end users, but instead for us to be more productive in handling the data and information that we will encounter in our lives. When you first start, you will be both the programmer and the end user of your programs. As you gain skill as a programmer and programming feels more creative to you, your thoughts may turn toward developing programs for others.

This page titled 1.5: Creativity and Motivation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.2: Creativity and Motivation by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

1.6: Understanding Programming

In the rest of this book, we will try to turn you into a person who is skilled in the art of programming. In the end you will be a *programmer* - perhaps not a professional programmer, but at least you will have the skills to look at a data/information analysis problem and develop a program to solve the problem.

In a sense, you need two skills to be a programmer:

- First, you need to know the programming language (Python) - you need to know the vocabulary and the grammar. You need to be able to spell the words in this new language properly and know how to construct well-formed "sentences" in this new language.
- Second, you need to "tell a story". In writing a story, you combine words and sentences to convey an idea to the reader. There is a skill and art in constructing the story, and skill in story writing is improved by doing some writing and getting some feedback. In programming, our program is the "story" and the problem you are trying to solve is the "idea".

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as JavaScript or C++. The new programming language has very different vocabulary and grammar but the problem-solving skills will be the same across all programming languages.

You will learn the "vocabulary" and "sentences" of Python pretty quickly. It will take longer for you to be able to write a coherent program to solve a brand-new problem. We teach programming much like we teach writing. We start reading and explaining programs, then we write simple programs, and then we write increasingly complex programs over time. At some point you "get your muse" and see the patterns on your own and can see more naturally how to take a problem and write a program that solves that problem. And once you get to that point, programming becomes a very pleasant and creative process.

We start with the vocabulary and structure of Python programs. Be patient as the simple examples remind you of when you started reading for the first time.

This page titled 1.6: Understanding Programming is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.4: Understanding Programming by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

1.7: The Learning Journey

As you progress through the rest of the book, don't be afraid if the concepts don't seem to fit together well the first time. When you were learning to speak, it was not a problem for your first few years that you just made cute gurgling noises. And it was OK if it took six months for you to move from simple vocabulary to simple sentences and took 5-6 more years to move from sentences to paragraphs, and a few more years to be able to write an interesting complete short story on your own.

We want you to learn Python much more rapidly, so we teach it all at the same time over the next few chapters. But it is like learning a new language that takes time to absorb and understand before it feels natural. That leads to some confusion as we visit and revisit topics to try to get you to see the big picture while we are defining the tiny fragments that make up that big picture. While the book is written linearly, and if you are taking a course it will progress in a linear fashion, don't hesitate to be very nonlinear in how you approach the material. Look forwards and backwards and read with a light touch. By skimming more advanced material without fully understanding the details, you can get a better understanding of the "why?" of programming. By reviewing previous material and even redoing earlier exercises, you will realize that you actually learned a lot of material even if the material you are currently staring at seems a bit impenetrable.

Usually when you are learning your first programming language, there are a few wonderful "Ah Hah!" moments where you can look up from pounding away at some rock with a hammer and chisel and step away and see that you are indeed building a beautiful sculpture.

If something seems particularly hard, there is usually no value in staying up all night and staring at it. Take a break, take a nap, have a snack, explain what you are having a problem with to someone (or perhaps your dog), and then come back to it with fresh eyes. I assure you that once you learn the programming concepts in the book you will look back and see that it was all really easy and elegant and it simply took you a bit of time to absorb it.

This page titled 1.7: The Learning Journey is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.12: The Learning Journey by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

2: COMMANDS

2.1: Hello World
2.2: Words and Sentences
2.3: Conversing with Python
2.4: What could possibly go wrong?
2.5: Comments
2.6: Comments

This page titled 2: Commands is shared under a license and was authored, remixed, and/or curated by LibreTexts.

2.1: Hello World

`Hello world!` Overview

A “**Hello, world**!” program is a computer program that outputs or displays “Hello, world!” to a user. Being a very simple program in most programming languages, it is often used to illustrate the basic syntax of a programming language for a working program, and as such is often the very first program people write.[1]

Discussion

A “Hello, world!” program is traditionally used to introduce novice programmers to a programming language. “Hello, world!” is also traditionally used in a sanity test to make sure that a computer language is correctly installed, and that the operator understands how to use it.[2]

The tradition of using the phrase “Hello, world!” as a test message was influenced by an example program in the seminal book *The C Programming Language*. The example program from that book prints “hello, world” (without capital letters or exclamation mark), and was inherited from a 1974 Bell Laboratories internal memorandum by Brian Kernighan.[3]

In addition to displaying “Hello, world!”, a “Hello, world!” program might include comments. A **comment** is a programmer-readable explanation or annotation in the source code of a computer program. They are added with the purpose of making the source code easier for humans to understand, and are generally ignored by compilers and interpreters. The syntax of comments in various programming languages varies considerably.[4]

Pseudocode

```
01 Function Main
02     ... This program displays "Hello world!"
03     Output "Hello world!"
04 End
```

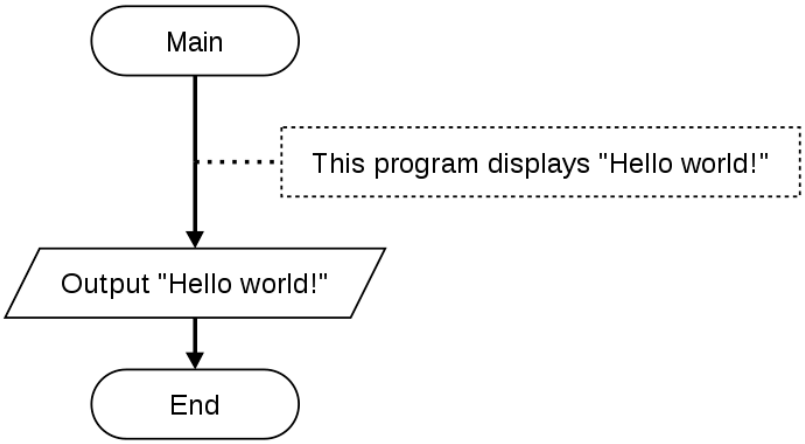
Output

```
01 Hello world!
```

Each code element represents:[5]

- `Function Main` begins the main function
- `...` begins a comment
- `Output` indicates the following value(s) will be displayed or printed
- `"Hello world!"` is the literal string to be displayed
- `End` ends a block of code

Flowchart



Examples

The following pages provide examples of “Hello, world!” programs in different programming languages. Each page includes an explanation of the code elements that comprise the program and links to IDEs you may use to test the program.

Key Terms

comment
: A programmer-readable explanation or annotation in the source code of a computer program.

References

- Wikiversity: Computer Programming
- Flowgorithm – Flowchart Programming Language

---

1. Wikipedia: "Hello, World!" program ↵
2. Wikipedia: "Hello, World!" program ↵
3. Wikipedia: "Hello, World!" program ↵
4. Wikipedia: Comment (computer programming) ↵
5. Wikibooks: Programming Fundamentals/Hello World ↵

This page titled 2.1: Hello World is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

2.2: Words and Sentences

Unlike human languages, the Python vocabulary is actually pretty small. We call this "vocabulary" the "reserved words". These are words that have very special meaning to Python. When Python sees these words in a Python program, they have one and only one meaning to Python. Later as you write programs you will make up your own words that have meaning to you called *variables*. You will have great latitude in choosing your names for your variables, but you cannot use any of Python's reserved words as a name for a variable.

When we train a dog, we use special words like "sit", "stay", and "fetch". When you talk to a dog and don't use any of the reserved words, they just look at you with a quizzical look on their face until you say a reserved word. For example, if you say, "I wish more people would walk to improve their overall health", what most dogs likely hear is, "blah blah blah *walk* blah blah blah blah." That is because "walk" is a reserved word in dog language. Many might suggest that the language between humans and cats has no reserved words[1].

The reserved words in the language where humans talk to Python include the following:

```
and       del       global      not       with
as        elif      if          or        yield
assert    else      import      pass
break     except    in          raise
class     finally   is          return
continue  for       lambda      try
def       from      nonlocal    while
```

That is it, and unlike a dog, Python is already completely trained. When you say "try", Python will try every time you say it without fail.

We will learn these reserved words and how they are used in good time, but for now we will focus on the Python equivalent of "speak" (in human-to-dog language). The nice thing about telling Python to speak is that we can even tell it what to say by giving it a message in quotes:

Code 1.5.1 (Python)

```
print('Hello world!')
```

```
Hello world!
```

And we have even written our first syntactically correct Python sentence. Our sentence starts with the function *print* followed by a string of text of our choosing enclosed in single quotes.

This page titled 2.2: Words and Sentences is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.5: Words and Sentences by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

2.3: Conversing with Python

Now that we have a word and a simple sentence that we know in Python, we need to know how to start a conversation with Python to test our new language skills.

Before you can converse with Python, you must first install the Python software on your computer and learn how to start Python on your computer. That is too much detail for this chapter so I suggest that you consult www.py4e.com where I have detailed instructions and screencasts of setting up and starting Python on Macintosh and Windows systems. At some point, you will be in a terminal or command window and you will type *python* and the Python interpreter will start executing in interactive mode and appear somewhat as follows:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> prompt is the Python interpreter's way of asking you, "What do you want me to do next?" Python is ready to have a conversation with you. All you have to know is how to speak the Python language.

Let's say for example that you did not know even the simplest Python language words or sentences. You might want to use the standard line that astronauts use when they land on a faraway planet and try to speak with the inhabitants of the planet:

```
>>> I come in peace, please take me to your leader
  File "<stdin>", line 1
    I come in peace, please take me to your leader
         ^
SyntaxError: invalid syntax
>>>
```

This is not going so well. Unless you think of something quickly, the inhabitants of the planet are likely to stab you with their spears, put you on a spit, roast you over a fire, and eat you for dinner.

Luckily you brought a copy of this book on your travels, and you thumb to this very page and try again:

```
>>> print('Hello world!')
Hello world!
```

This is looking much better, so you try to communicate some more:

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
  File "<stdin>", line 1
    print 'We will have a feast tonight unless you say
                                                      ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

Code 1.6.1 (Python)

```
print 'We will have a feast tonight unless you say

```

The conversation was going so well for a while and then you made the tiniest mistake using the Python language and Python brought the spears back out.

At this point, you should also realize that while Python is amazingly complex and powerful and very picky about the syntax you use to communicate with it, Python is *not* intelligent. You are really just having a conversation with yourself, but using proper syntax.

In a sense, when you use a program written by someone else the conversation is between you and those other programmers with Python acting as an intermediary. Python is a way for the creators of programs to express how the conversation is supposed to proceed. And in just a few more chapters, you will be one of those programmers using Python to talk to the users of your program.

Before we leave our first conversation with the Python interpreter, you should probably know the proper way to say "good-bye" when interacting with the inhabitants of Planet Python:

```
    >>> good-bye
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    NameError: name 'good' is not defined
    >>> if you don't mind, I need to leave
      File "<stdin>", line 1
        if you don't mind, I need to leave
                   ^
    SyntaxError: invalid syntax
    >>> quit()
```

You will notice that the error is different for the first two incorrect attempts. The second error is different because *if* is a reserved word and Python saw the reserved word and thought we were trying to say something but got the syntax of the sentence wrong.

The proper way to say "good-bye" to Python is to enter *quit()* at the interactive chevron >>> prompt. It would have probably taken you quite a while to guess that one, so having a book handy probably will turn out to be helpful.

This page titled 2.3: Conversing with Python is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.6: Conversing with Python by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

2.4: What could possibly go wrong?

As we saw in our earliest conversations with Python, we must communicate very precisely when we write Python code. The smallest deviation or mistake will cause Python to give up looking at your program.

Beginning programmers often take the fact that Python leaves no room for errors as evidence that Python is mean, hateful, and cruel. While Python seems to like everyone else, Python knows them personally and holds a grudge against them. Because of this grudge, Python takes our perfectly written programs and rejects them as "unfit" just to torment us.

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt ('Hello world')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined

>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
               ^
SyntaxError: invalid syntax
>>>
```

There is little to be gained by arguing with Python. It is just a tool. It has no emotions and it is happy and ready to serve you whenever you need it. Its error messages sound harsh, but they are just Python's call for help. It has looked at what you typed, and it simply cannot understand what you have entered.

Python is much more like a dog, loving you unconditionally, having a few key words that it understands, looking you with a sweet look on its face (>>>), and waiting for you to say something it understands. When Python says "SyntaxError: invalid syntax", it is simply wagging its tail and saying, "You seemed to say something but I just don't understand what you meant, but please keep talking to me (>>>)."

As your programs become increasingly sophisticated, you will encounter three general types of errors:

Syntax errors
: These are the first errors you will make and the easiest to fix. A syntax error means that you have violated the "grammar" rules of Python. Python does its best to point right at the line and character where it noticed it was confused. The only tricky bit of syntax errors is that sometimes the mistake that needs fixing is actually earlier in the program than where Python *noticed* it was confused. So the line and character that Python indicates in a syntax error may just be a starting point for your investigation.

Logic errors
: A logic error is when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another. A good example of a logic error might be, "take a drink from your water bottle, put it in your backpack, walk to the library, and then put the top back on the bottle."

Semantic errors

A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program. The program is perfectly correct but it does not do what you *intended* for it to do. A simple example would be if you were giving a person directions to a restaurant and said, "...when you reach the intersection with the gas station, turn left and go one mile and the restaurant is a red building on your left." Your friend is very late and calls you to tell you that they are on a farm and walking around behind a barn, with no sign of a restaurant. Then you say "did you turn left or right at the gas station?" and they say, "I followed your directions perfectly, I have them written down, it says turn left and go one mile at the gas station." Then you say, "I am very sorry, because while my instructions were syntactically correct, they sadly contained a small but undetected semantic error.".

Again in all three types of errors, Python is merely trying its hardest to do exactly what you have asked.

This page titled 2.4: What could possibly go wrong? is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.11: What could possibly go wrong? by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

2.5: Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called *comments*, and in Python they start with the # symbol:

```
01 # compute the percentage of the hour that has elapsed
02 percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
01 percentage = (minute * 100) / 60     # percentage of an hour
```

Everything from the \# to the end of the line is ignored; it has no effect on the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

This comment is redundant with the code and useless:

```
01 v = 5     # assign 5 to v
```

This comment contains useful information that is not in the code:

```
01 v = 5     # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

This page titled 2.5: Comments is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.11: Comments by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

2.6: Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called *comments*, and in Python they start with the # symbol:

```
01 # compute the percentage of the hour that has elapsed
02 percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
01 percentage = (minute * 100) / 60     # percentage of an hour
```

Everything from the \# to the end of the line is ignored; it has no effect on the program.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.

This comment is redundant with the code and useless:

```
01 v = 5     # assign 5 to v
```

This comment contains useful information that is not in the code:

```
01 v = 5     # velocity in meters/second.
```

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

This page titled 2.6: Comments is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.11: Comments by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

3: PRINCIPLES OF STRUCTURED PROGRAMMING

3.1: The Building Blocks of Programs
3.2: Structured Programming

This page titled 3: Principles of Structured Programming is shared under a license and was authored, remixed, and/or curated by LibreTexts.

3.1: The Building Blocks of Programs

In the next few chapters, we will learn more about the vocabulary, sentence structure, paragraph structure, and story structure of Python. We will learn about the powerful capabilities of Python and how to compose those capabilities together to create useful programs.

There are some low-level conceptual patterns that we use to construct programs. These constructs are not just for Python programs, they are part of every programming language from machine language up to the high-level languages.

input
: Get data from the "outside world". This might be reading data from a file, or even some kind of sensor like a microphone or GPS. In our initial programs, our input will come from the user typing data on the keyboard.

output
: Display the results of the program on a screen or store them in a file or perhaps write them to a device like a speaker to play music or speak text.

sequential execution
: Perform statements one after another in the order they are encountered in the script.

conditional execution
: Check for certain conditions and then execute or skip a sequence of statements.

repeated execution
: Perform some set of statements repeatedly, usually with some variation.

reuse
: Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program.

It sounds almost too simple to be true, and of course it is never so simple. It is like saying that walking is simply "putting one foot in front of the other". The "art" of writing a program is composing and weaving these basic elements together many times over to produce something that is useful to its users.

The word counting program above directly uses all of these patterns except for one.

This page titled 3.1: The Building Blocks of Programs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.10: The Building Blocks of Programs by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

3.2: Structured Programming

Overview

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines in contrast to using simple tests and jumps such as the go to statement, which can lead to "**spaghetti code**" that is potentially difficult to follow and maintain.[1]

Discussion

One of the most important concepts of programming is the ability to control a program so that different lines of code are executed or that some lines of code are executed many times. The mechanisms that allow us to control the flow of execution are called **control structures**. Flowcharting is a method of documenting (charting) the flow (or paths) that a program would execute. There are three main categories of control structures:

- **Sequence** – Very boring. Simply do one instruction then the next and the next. Just do them in a given sequence or in the order listed. Most lines of code are this.
- **Selection** – This is where you select or choose between two or more flows. The choice is decided by asking some sort of question. The answer determines the path (or which lines of code) will be executed.
- **Iteration** – Also known as repetition, it allows some code (one to many lines) to be executed (or repeated) several times. The code might not be executed at all (repeat it zero times), executed a fixed number of times or executed indefinitely until some condition has been met. Also known as looping because the flowcharting shows the flow looping back to repeat the task.

A fourth category describes unstructured code.

- **Branching** – An uncontrolled structure that allows the flow of execution to jump to a different part of the program. This category is rarely used in modular structured programming.

All high-level programming languages have control structures. All languages have the first three categories of control structures (sequence, selection, and iteration). Most have the if then else structure (which belongs to the selection category) and the while structure (which belongs to the iteration category). After these two basic structures, there are usually language variations.

The concept of **structured programming** started in the late 1960's with an article by Edsger Dijkstra. He proposed a "go to less" method of planning programming logic that eliminated the need for the branching category of control structures. The topic was debated for about 20 years. But ultimately – "By the end of the 20th century nearly all computer scientists were convinced that it is useful to learn and apply the concepts of structured programming."[2]

Key Terms

branching
: An uncontrolled structure that allows the flow of execution to jump to a different part of the program.

control structures
: Mechanisms that allow us to control the flow of execution within a program.

iteration
: A control structure that allows some lines of code to be executed many times.

selection
: A control structure where the program chooses between two or more options.

sequence
: A control structure where the program executes the items in the order listed.

spaghetti code
: A pejorative phrase for unstructured and difficult to maintain source code.[3]

structured programming
: A method of planning programs that avoids the branching category of control structures.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Structured programming ↵
2. Wikipedia: Structured programming ↵
3. Wikipedia: Spaghetti code ↵

This page titled 3.2: Structured Programming is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

4: FUNCTIONS AND EVENTS

4.1: Modular Programming

4.2: Why functions?

4.3: Function Calls

4.4: Definitions and Uses

4.5: Flow of Execution

4.6: Parameters and Arguments

4.7: Parameters and Arguments

This page titled 4: Functions and Events is shared under a license and was authored, remixed, and/or curated by LibreTexts.

4.1: Modular Programming

Overview

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.[1]

Concept of Modularization

One of the most important concepts of programming is the ability to group some lines of code into a unit that can be included in our program. The original wording for this was a sub-program. Other names include: macro, sub-routine, procedure, module and function. We are going to use the term **function** for that is what they are called in most of the predominant programming languages of today. Functions are important because they allow us to take large complicated programs and to divide them into smaller manageable pieces. Because the function is a smaller piece of the overall program, we can concentrate on what we want it to do and test it to make sure it works properly. Generally, functions fall into two categories:

1. **Program Control** – Functions used to simply sub-divide and control the program. These functions are unique to the program being written. Other programs may use similar functions, maybe even functions with the same name, but the content of the functions are almost always very different.
2. **Specific Task** – Functions designed to be used with several programs. These functions perform a specific task and thus are usable in many different programs because the other programs also need to do the specific task. Specific task functions are sometimes referred to as building blocks. Because they are already coded and tested, we can use them with confidence to more efficiently write a large program.

The main program must establish the existence of functions used in that program. Depending on the programming language, there is a formal way to:

1. define a function (its **definition** or the code it will execute)
2. **call** a function
3. declare a function (a **prototype** is a declaration to a compiler)

Note: Defining and calling functions are common activities across programming languages. Declaring functions with prototypes is specific to certain programming languages, including C and C++.

Program Control functions normally do not communicate information to each other but use a common area for variable storage. Specific Task functions are constructed so that data can be communicated between the calling program piece (which is usually another function) and the function being called. This ability to communicate data is what allows us to build a specific task function that may be used in many programs. The rules for how the data is communicated in and out of a function vary greatly by programming language, but the concept is the same. The data items passed (or communicated) are called parameters. Thus the wording: **parameter passing**. The four data communication options include:

1. no communication in with no communication out
2. no communication in with some communication out
3. some communication in with some communication out
4. some communication in with no communication out

Program Control Function

The main program piece in many programming languages is a special function with the **identifier name** of main. The special or uniqueness of main as a function is that this is where the program starts executing code and this is where it usually stops executing code. It is often the first function defined in a program and appears after the area used for includes, other technical items, declaration of prototypes, the listing of global constants and variables and any other items generally needed by the program. The code to define the function main is provided; however, it is not prototyped or usually called like other functions

within a program.

Specific Task Function

We often have the need to perform a specific task that might be used in many programs.

General layout of a function in a statically-typed language such as C++, C#, and Java:

```
<return value data type> function identifier name(<data type> <identifier name for input value>) {
    //lines of code;
    return <value>;
}
```

General layout of a function in a dynamically typed language such as JavaScript and Python:

```
function identifier name(<identifier name for input value>) {
    //lines of code;
    return <value>;
}

def function identifier name(<identifier name for input value>):
    //lines of code
    return <value>
```

In some programming languages, functions have a set of **braces** {} used for identifying a group or block of statements or lines of code. Other languages use indenting or some type of begin and end statements to identify a code block. There are normally several lines of code within a function.

Programming languages will either have specific task functions defined before or after the main function, depending on coding conventions for the given language.

When you call a function you use its identifier name and a set of parentheses. You place any data items you are passing inside the parentheses. After our program is compiled and running, the lines of code in the main function are executed, and when it gets to the calling of a specific task function, the control of the program moves to the function and starts executing the lines of code in the function. When it's done with the lines of code, it will return to the place in the program that called it (in our example the function main) and continue with the code in that function.

Program Layout

Most programs have several items before the functions, including:

1. Documentation – Most programs have a comment area at the start of the program with a variety of comments pertinent to the program.
2. Include or import statements used to access standard library functions.
3. Language-specific code such as namespace references or function prototypes.
4. Global or module-level constants and variables, when required.

Key Terms

braces
: Used to identify a block of code in languages such as C++, C#, Java, and JavaScript.

function
: What modules are called in many predominant programming languages of today.

function call
A function's using or invoking of another function.

function definition
The code that defines what a function does.

function prototype
A function's communications declaration to a compiler.

identifier name
The name given by the programmer to identify a function or other program items such as variables.

modularization
The ability to group some lines of code into a unit that can be included in our program.

parameter passing
How the data is communicated in to and out of a function.

program control
Functions used to simply subdivide and control the program.

specific task
Functions designed to be used with several programs.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Modular programming ↩

This page titled 4.1: Modular Programming is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

4.2: Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as "find the smallest value in a list of values". Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named `min` which takes a list of values as its argument and returns the smallest value in the list.

This page titled 4.2: Why functions? is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.11: Why functions? by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

4.3: Function Calls

In the context of programming, a *function* is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a *function call*:

```
>>> type(32)
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the *argument* of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the `type` function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the *return value*.

This page titled 4.3: Function Calls is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.1: Function Calls by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

4.4: Definitions and Uses

Pulling together the code fragments from the previous section, the whole program looks like this:

Code 4.7.1 (Python)

```
%%python3
def print_lyrics():
    print(\"I'm a lumberjack, and I'm okay.\")
    print('I sleep all night and I work all day.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: http://www.py4e.com/code3/lyrics.py

```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Exercise 2: Edit the code block and move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Exercise 3: Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

This page titled 4.4: Definitions and Uses is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.7: Definitions and Uses by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

4.5: Flow of Execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the *flow of execution*.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function *definitions* do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

This page titled 4.5: Flow of Execution is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.8: Flow of Execution by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

4.6: Parameters and Arguments

Overview

A **parameter** is a special kind of variable used in a function to refer to one of the pieces of data provided as input to the function. These pieces of data are the values of the **arguments** with which the function is going to be called/invoked. An ordered list of parameters is usually included in the definition of a function, so that, each time the function is called, its arguments for that call are evaluated, and the resulting values can be assigned to the corresponding parameters.[1]

Discussion

Recall that the modular programming approach separates the functionality of a program into *independent* modules. To separate the functionality of one function from another, each function is given its own unique input variables, called parameters. The parameter values, called arguments, are passed to the function when the function is called. Consider the following function pseudocode:

```
01 Function CalculateCelsius (Real fahrenheit)
02     Declare Real celsius
03
04     Assign celsius = (fahrenheit - 32) * 5 / 9
05 Return Real celsius
```

If the CalculateCelsius function is called passing in the value 100, as in `CalculateCelsius(100)`, the parameter is `fahrenheit` and the argument is `100`. The terms parameter and argument are often used interchangeably. However, parameter refers to the variable identifier (fahrenheit) while argument refers to the variable value (100).

Functions may have no parameters or multiple parameters. Consider the following function pseudocode:

```
01 Function DisplayResult (Real fahrenheit, Real celsius)
02     Output fahrenheit & "° Fahrenheit is " & celsius & "° Celsius"
03 End
```

If the DisplayResult function is called passing in the values 98.6 and 37.0, as in `DisplayResults(98.6, 37.0)`, the argument or value for the fahrenheit parameter is 98.6 and the argument or value for the celsius parameter is 37.0. Note that the arguments are passed positionally. Calling `DisplayResults(37.0, 98.6)`would result in incorrect output, as the value of fahrenheit would be 37.0 and the value of celsius would be 98.6.

Some programming languages, such as Python, support named parameters. When calling functions using named parameters, parameter names and values are used, and positions are ignored. When names are not used, arguments are identified by position. For example, any of the following function calls would be valid:

```
01 CalculateCelsius(98.6, 37.0)
02 CalculateCelsius(fahrenheit=98.6, celsius=37.0)
03 CalculateCelsius(celsius=37.0, fahrenheit=98.6)
```

Key Terms

argument
: A value provided as input to a function.

parameter
A variable identifier provided as input to a function.

References

- Wikiversity: Computer Programming

---

1. Wikipedia: Parameter (computer programming) ↵

This page titled 4.6: Parameters and Arguments is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

4.7: Parameters and Arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called *parameters*. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `Spam  '*4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

This page titled 4.7: Parameters and Arguments is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.9: Parameters and Arguments by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

5: ALGORITHMS AND DESIGN

5.1: Thinking Algorithmically
5.2: Program Design
5.3: Program Quality

This page titled 5: Algorithms and Design is shared under a license and was authored, remixed, and/or curated by LibreTexts.

5.1: Thinking Algorithmically



Figure 12.1: Think algorithmic thoughts.

Professional engineers are well known for their ability to solve complex problems. Throughout the rest of this book, we will continue to learn how to use MATLAB as a tool to solve problems. However, before we jump back into MATLAB syntax and usage, it is important to remember the end goal: **to learn how to solve problems algorithmically.**

In the spirit of learning how to think algorithmically, in this chapter you will be presented with material that will help you start to *think* like a computer. The key to it all is to remember that computers are really, really, really, dumb.

Learning Goals

In this chapter, we will take a break from MATLAB to learn how to think algorithmically. That means we will explore the following:

- How to approach solutions to problems
- What an algorithm is
- Where algorithms arise in everyday life and in engineering contexts
- How to “think algorithmically”
- How to write pseudocode
- How to create flowcharts

The 8 AM Attendance Problem

Getting to a class that is held early in the morning is a problem that some college freshmen and professors can relate to. Although, we professors typically complain about it much less (*and some of us, myself included, even enjoy early classes!*). Based on the attendance of classes I have taught at 8 AM, it **really is a problem that doesn’t have a definite solution yet!** There are a ton of things that you need to do, to solve the problem of attending class at 8 AM. First, you have to start by waking up at a specific time that is often earlier than the time you are accustomed to waking up at. Presumably, your solution to this problem will look

similar to my solution. Certain details may be different but you get the idea.



Figure 12.2: PSA - Science says you shouldn't use your phone as an alarm clock!

Professor's Solution - Speech

I wake up using a cheap digital alarm clock I have on my bedside table set for 6 AM. It cost me about $4 and as a bonus, I avoid the temptation to check my smartphone before I go to bed. I found that I sleep better that way. I am not really a "snooze" guy, so I just pop up out of bed as soon as the alarm goes off. I take a shower, appreciating the clean, reliable, hot water, being delivered to my wall. If only my ancestors could imagine such luxury, such magnificence! Next, I check the weather on the internet, then put on some weather-appropriate clothing for bike riding. I hop on my bike and ride towards Colorado State University. I am fortunate that my commuting route doubles as a scenic route. It travels through an area of Fort Collins known for its beautiful turn-of-the-century houses and great, big, old, happy, trees. As I pedal through Old Town, I remind myself how lucky I am to live in a city with drivers that are educated on bicycle safety and are respectful to riders. I arrive at CSU, lock my bike, and get to class at 7:50 am.

What is your solution?

Discussion 12.1: Your 8 AM Algorithm

This interactive element is not available in this format. Visit the link or scan the QR code to access it online.

View this Interactive Content online

Congratulations you just wrote out an algorithm! You just authored a list of step-by-step instructions to solve a problem. That is what an algorithm is!

An algorithm is a set of instructions for solving a problem, step-by-step.

Millions of algorithms are present in our everyday life, it is just not common for us to acknowledge them as such. When we bake, we follow recipes that are just baking algorithms. That one is obvious but there are many more examples! When you are buying groceries you automatically gravitate towards the checkout line that *appears* as if it will take the least amount of time. Think about how complicated *that* algorithm is. You would likely prefer to be behind two people with three items each than one person with

50 items. Most of the algorithms that humans follow are just outside of our conscious thought so we don't realize that they are there even though our human algorithms govern almost every decision we make and every action we take.

How to write algorithms people (or computers) can actually follow

Part of learning how to be a professional engineer is learning how to do things efficiently, in a way that is repeatable by others. It is important that if you solve a unique problem, that others don't need to solve it all over again.

Now, if I wanted to describe to someone my solution to the 8 AM attendance problem, I could write it out as I did above (see Professor's Solution). However, there are some language flourishes that are unnecessary, take up space, and *do not contribute to your understanding of how I arrive at the solution*. If we strip those away, it then becomes clear that there are a lot of details that are missing. For example, if you wanted to copy my solution exactly, you may wonder what types of clothes I wear and when I wear them. Also, you may wonder if I ride my bike to work in all weather conditions. Do I ever drive? Etc.

For your algorithms to be usable by other people, **the algorithm needs to be as specific as possible**. You need to write algorithms down in such a way that a computer (sometimes, but less often, a human) can follow the steps to complete the solution. It is important to write your algorithms so that *nothing is implied*! To make sure I am writing my algorithms appropriately, I pretend that I am writing them for a really, really, dumb person. Even though we say things with computers are "smart" (i.e. smartphones), in reality, computers are shockingly dumb. They are just really *fast* which we can confuse as smart. For now, when crafting algorithms, remember to be as specific as you possibly can.

It is important to write your algorithms so that nothing is implied!

Let's break down the Professor's Solution to the problem as an ordered list of instructions that should be followed step-by-step:

Professor's solution - Algorithm

Below, is the Professor's Solution to the 8 AM attendance problem.

Instructions to ensure success when following the algorithm that solves the 8 AM attendance problem:

- Steps should be read from the top to bottom (line numbers are displayed in yellow to the left)
- Follow the instructions on each line to their logical conclusion
- Either *continue one line down* OR *follow the instruction on line designating skip location*

```
1  Wake up from alarm at 6am
2  Take a shower
3  Check the weather to get the forecast
4  If forecasted temperature >= 70 continue, else go to line 7
5      Wear t-shirt and shorts
6      Skip to line 17
7  If forecast >= 50 continue, else go to line 10
8      Wear longsleeve shirt and jeans
9      Skip to line 17
10 If forecast >= 30 coninue, else go to line 13
11     Wear jacket and jeans
12     Skip to line 17
13 Else
14     Wear sweatervest and slacks
15     Leave house at 7:30am, drive car to CSU
16     Skip to line 19
17 Leave house at 7:20am to ride bike to CSU
18 Lock bike safely
19 Arrive at class by 7:50am
```

Figure 12.3: Professor's solution to the 8 AM problem.

Let's take a look at line 4: `If forecast >= 70 continue, else go to line 7`

Translated from Algorithm-ese into English, that is equivalent to saying:

If the temperature is **forecast** to be **greater than or equal to** 70 degrees F, **continue** to line 8 which tells you to wear a t-shirt and shorts. Or **else** you should consider other clothing options.

Let's say the forecast indicates it will be 61 degrees today. Therefore when I evaluate my algorithm, the statement `If forecast >= 70 continue,` evaluates to false. It isn't true, so I shouldn't follow the instruction. This is where the algorithm following rule **Follow the instructions on each line to their logical conclusion** comes into play. Since it is not true that it is 61 degrees, I should follow the else go to line 7 bit. So I skip to line 7 and am on my merry way.

Question 12.1: Follow the Logic

This interactive element is not available in this format. Visit the link or scan the QR code to access it online.

View this Interactive Content online

Bugs

Notice how the steps are written in such a way, **that you can not mess them up if you follow the instructions**. If you start from the top and follow the instructions, you can't both ride the bike AND drive the car at the same time. You will not be instructed to wear both a t-shirt and a long sleeve, etc. You may think the way of writing `go to line 7` is unnecessary because you are smart and can understand that some things are implied. However, it is crucial to ensure that someone following the algorithm does not wear multiple items of clothing if they are dumb.

> *A bug in your algorithm is an erroneous instruction(s) included in your steps that causes your algorithm to behave in a way you didn't anticipate.*



Figure 12.4: Always check for bugs in your code.

Remember, it might be useful to you to think about writing algorithms for a very, very, very dumb person. For example, consider the following Professor's Solution - Algorithm with the following, small, yet critical change (figure 12.5 below). Now, read it and

pretend that you are exceptionally dumb and just follow the instructions blindly. Furthermore, let's say that the forecast is 61 degrees F. Before continuing reading the text, look at the picture below and see if you can find the bug.

```
1  Wake up from alarm at 6am
2  Take a shower
3  Check the weather to get the forecast
4  If forecasted temperature >= 70
5         Wear t-shirt and shorts
6  If forecast >= 50
7         Wear longsleeve shirt and jeans
8  If forecase >= 30
9         Wear jacket and jeans
10 Else
11        Wear sweatervest and slacks
12        Leave house at 7:30am, drive car to CSU
13 Leave house at 7:20am to ride bike to CSU
14 Lock bike safely
15 Arrive at class by 7:50am
```

Figure 12.5: Can you find the bug?

The dumb person MIGHT start at the top of the list, notice that the forecast is NOT greater than or equal to 70, deduce that the tabs imply that you should only follow the instructions if the preceding statement is true, then skip to line 6 (doubtful, but let's say that this particular dumb person is having a good day). When the dumb person gets to line 6, they notice that hey, `If forecast >= 50` is true so they continue and put on a long sleeve shirt and jeans. When they get to line 8, they notice that `If forecast >= 30` is ALSO true! So they put on a jacket and jeans over the long sleeve shirt and jeans they already have on. *That is the bug!* We didn't intend for someone to remember that an algorithm is a step-by-step process that should be written with nothing implied. Wear two pairs of jeans or two tops!

Question 12.2: Find the bug

This interactive element is not available in this format. Visit the link or scan the QR code to access it online.

View this Interactive Content online

Decisions and Loops

Almost always, an algorithm needs to make a decision. We saw that in the 8 am attendance problem solution. The algorithm helped choose clothing based on the variable forecast. As mentioned earlier, computers are fast but really dumb. We can't just ask a computer what we should wear. We need to be specific in the steps it needs to follow to determine what we wear. Whenever we need to have a computer make a **decision**, we use an `if` statement. (e.g. `if forecast >= 60`)

Another thing that computers are really good at is doing boring, repetitive tasks over and over. For example, let's think about a computer program that assigns people to a military draft. Let's say that we need to draft 20,000 men into the armed services. We can write a computer program that randomly assigns all eligible men a draft number, then randomly chooses 20,000 numbers. If a human had to do this, it would take a very long time, but a computer can get this done very quickly! Whenever we need a computer to do a task over and over again (aka: **loop**) we use a `for` statement.

Decisions - `if` Statements

`if` statements are actually very simple and elegant ways to get computers to make decisions. They work by evaluating some conditional statements. If the statement evaluates to true, you do what follows. Optionally, you can include an **else** to specify what to do if the if statement evaluates to false.

Generically, if statements look like this:

```
If condition then do

sequence 1

Else do

sequence 2
```

For example, consider line 4 in the pseudocode in figure 12.5:

```
If forecast >= 70, continue, else go to line 7.
```

The decision is made based on the conditional statement. If the forecast temperature is greater than or equal to 70 degrees you continue, else you go to line 7. That is the decision! Hopefully, you know and understand how this works, and if you do, you are already thinking algorithmically!

Consider that a student designs an algorithm that informs them how much they should be studying each school day for an exam. The algorithm makes a decision about how much the student should study based on how far away the exam is. The student "runs" the algorithm once a day in the morning. Consider that the exam is 9 days away.

```
 1 Check exam date
 2 If exam is >= 3 weeks away, continue, else skip to line 5
 3        Spend 15 minutes studying
 4        Finish!
 5 If exam is >= 2 weeks away, continue, else skip to line 8
 6        Spend 30 min studying
 7        Finish!
 8 If exam is >= 1 week away, continue, else skip to line 11
 9        Spend 1 hour studying
10        Finish!
11 If exam is >= 2 days away
12        Spend 2 hours studying
13        Finish!
```

Question 12.3: Exam studying algorithim

This interactive element is not available in this format. Visit the link or scan the QR code to access it online.

View this Interactive Content online

Loops - for Statements

for statements allow us to write down repetitive actions algorithmically. Consider the generic for statement to get an idea of how it works:

```
for number of iterations

sequence repeats a number of iterations
```

The idea of a for statement is that you already have an idea of how many iterations need to be performed, then you algorithmically define the sequence that is to be performed beneath it. Let's return to the military drafting example to illustrate this. Consider the algorithm below that assigns all eligible men a draft number, then randomly chooses 20,000 numbers.

```
1 Assign all eligble men a draft number
2 number_eligible = total number of men eligible
3 For draft = 1 to 20,000
4         choose a random number between 1 and number_eligble
5         draft the person with the chosen number
```

Figure 12.6: for statement example algorithm

Looking at figure 12.6 above, hopefully, you can see why it is called a **loop**. Starting at the top, we first assign all eligible men a draft number. Let's say there are 150,000 eligible men, so the next line tells us to set the variable number_eligible to 150,000. On line three we have our for statement and *enter the loop*. The statement is telling us to start drafting at 1 and continue until 20,000. On line 4 we choose a random number, then on line 5, we draft the person with that number. Now the loop comes into play. Because we are in a for loop, we *automatically* increment draft by one and continue repeating lines 4 and 5 19,999 more times.

I understand that for loops can be tricky. Take some time to think about this section and if you are still confused supplement your reading with the Wikipedia page on for loops. Don't skip your brain workouts!

Nested Statements

The last thing we should mention about algorithms is that you can *nest* statements within each other. For example, look at the following algorithm below in figure 12.7.

```
1 If country is at war, continue, else skip to line 8
2         Assign all eligble men a draft number
3         number_eligible = total number of men eligible
4         For draft = 1 to 20,000
5                 choose a random number between 1 and number_eligble
6                 draft the person with the chosen number
7                 finish
8 Well there is no war going on so relax!
```

Figure 12.7: Example of nesting statements

Notice how we *nested* a for statement *within* an if statement. Nesting is a way to allow your algorithms to only loop if certain conditions are met (for within an if), or to make certain decisions a number of times (if within a for).

Again, this can be a confusing concept but once you start to use it in your programming it will make more sense. We will see the

concept of nesting pop up in later chapters. For now, just know that it is an option.

Good Algorithm Checklist

All good algorithms have the following characteristics:

- Remember that an algorithm is a step-by-step process that should be written with *nothing* implied.
- Consider the variables that will be used in the algorithm. For example, in the 8 am the solution, the variable was forecast. Thinking about this ahead of time will make writing computer algorithms in MATLAB easier.
- Check for bugs. To do this it is best to pretend you are *really* dumb (this can actually be difficult to do). Make sure that the algorithm performs exactly the way that you want it to.
- If there are decisions or loops necessary, make sure that they are written out in such a way that they are easy to read and understand. Only use them if necessary.

Communicating Algorithms

There are probably other ways to communicate algorithms, but for this class, we will only consider two: pseudocode and flowcharts. I highly recommend getting used to both ways of communicating algorithms as they are often most useful when used in conjunction. I have worked at several engineering companies and I can safely affirm that engineers often use both pseudocode and flowcharts in conjunction.

The last thing to note is that there are no *rule*s when it comes to pseudocode or flowcharting there are just guidelines. I once had a student that used to draw elaborate pictures for her “flowcharts”. Elephants meant something I can’t remember, but it was crazy. Her technique worked for her, but it was impossible for anyone else to follow. The guidelines that I will present below are standardized and should be understandable by your professors, tutors, or TAs.

Pseudocode

Lucky for us we have already seen examples of pseudocode. **Pseudocode (seen in figures 12.3, 12.5, 12.6, 12.7) is just an implementation of an algorithm in plain English**. Writing pseudocode can be helpful because you *think* about how your algorithm will work before you start to worry about programming it. The only real rule for pseudocode is that it needs to describe the logic of the algorithm so that when you go to translate the algorithm into a computer programming language, it is simply a task of changing English to MATLAB.

Flowcharting

Another common way to write down algorithms is using a flowchart. Before we talk about some general rules for flowcharting, look at the flowchart for the 8 am problem shown below in figure X. I think the best way to learn about how to flowchart, is to look at a flowchart and discover its logic on your own.

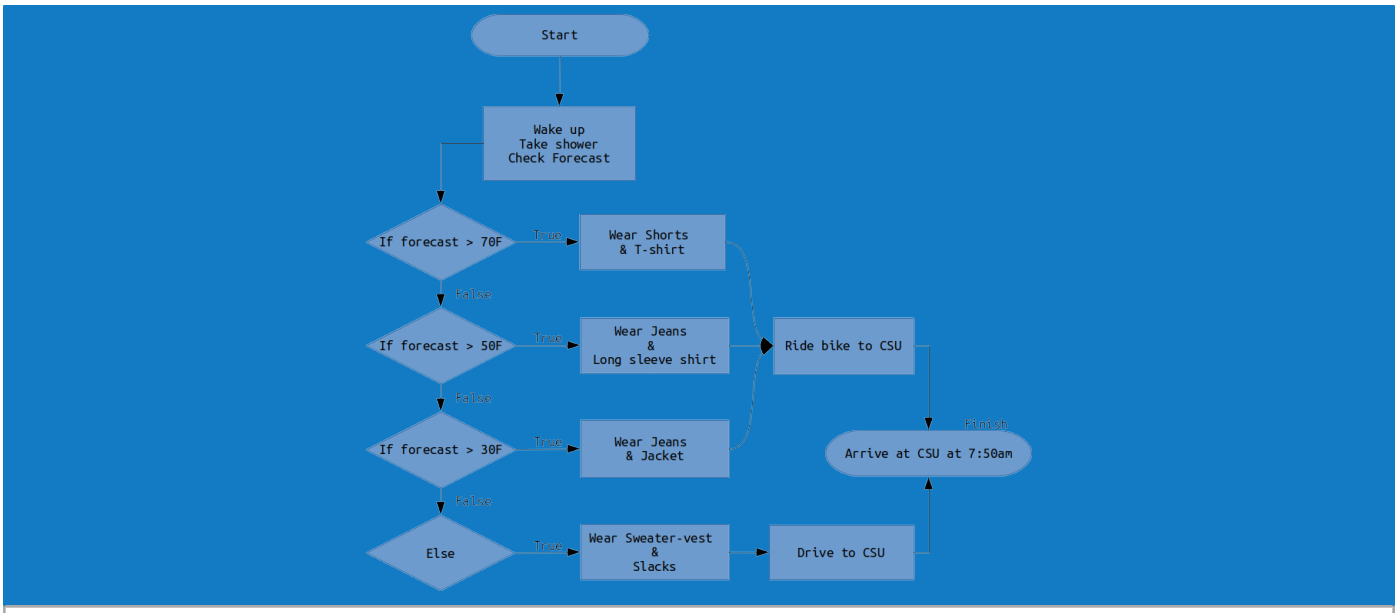


Figure 12.8: Flowchart showing solution to the 8 AM Attendance Problem

As you can see, flowcharts are nice because they are pictures with arrows which makes it easy to follow along. Just like your pseudocode, in order for flowcharts to be effective, they need to fully describe the logic of the algorithm. Again, that makes copying the algorithm into MATLAB (or whatever programming language) much easier.

There are some generally accepted standards for flowcharting presented below in figure 12.9.

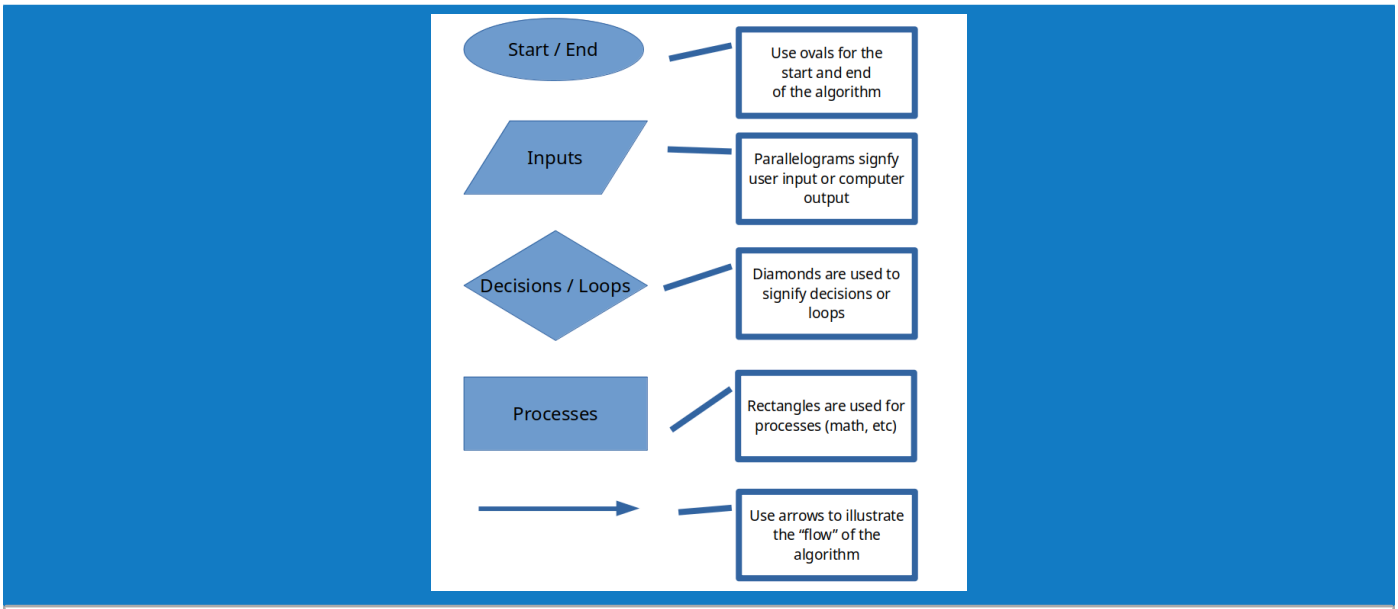


Figure 12.9: Flowchart standards

The only real rule with flowcharts is that they are useful to you. Honestly, I prefer flowcharting to pseudocode but in my professional life, I use both to help organize my thoughts and force me to think algorithmically. Whichever you decide to use is a matter of personal taste, but all good engineers *will use at least one of these techniques*.

Think about an algorithm that only you would know (or very few of your classmates), and explain it "algorithmically" as either a flowchart OR pseudo-code. The only requirement is that it should have some decisions to make and should contain no bugs. My suggestion is to avoid recipes, they are kind of boring and aren't really unique TO YOU. Perhaps you are a gifted gardener and know exactly how to prune apple trees? Are you into repairing cars and know the best procedure to follow to diagnose and

replace an alternator? I am willing to bet that there is something that you know how to do that a lot of your classmates would not understand (it doesn't matter how nerdy or weird it is).

Discussion 12.2: Algorithms in life

This interactive element is not available in this format. Visit the link or scan the QR code to access it online.

View this Interactive Content online

This page titled 5.1: Thinking Algorithmically is shared under a CC BY-NC 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- Chapter 15: Thinking Algorithmically is licensed CC BY-NC 4.0.

5.2: Program Design

Overview

Program design consists of the steps a programmer should do before they start coding the program in a specific language. These steps when properly documented will make the completed program easier for other programmers to maintain in the future. There are three broad areas of activity:

- Understanding the Program
- Using Design Tools to Create a Model
- Develop Test Data

Understanding the Program

If you are working on a project as one of many programmers, the system analyst may have created a variety of documentation items that will help you understand what the program is to do. These could include screen layouts, narrative descriptions, documentation showing the processing steps, etc. If you are not on a project and you are creating a simple program you might be given only a simple description of the purpose of the program. Understanding the purpose of a program usually involves understanding its:

- Inputs
- Processing
- Outputs

This **IPO** approach works very well for beginning programmers. Sometimes, it might help to visualize the program running on the computer. You can imagine what the monitor will look like, what the user must enter on the keyboard and what processing or manipulations will be done.

Using Design Tools to Create a Model

At first, you will not need a hierarchy chart because your first programs will not be complex. But as they grow and become more complex, you will divide your program into several modules (or functions).

The first modeling tool you will usually learn is **pseudocode**. You will document the logic or algorithm of each function in your program. At first, you will have only one function, and thus your pseudocode will follow closely the IPO approach above.

There are several methods or tools for planning the logic of a program. They include: flowcharting, hierarchy or structure charts, pseudocode, HIPO, Nassi-Schneiderman charts, Warnier-Orr diagrams, etc. Programmers are expected to be able to understand and do flowcharting and pseudocode. These methods of developing the model of a program are usually taught in most computer courses. Several standards exist for flowcharting and pseudocode and most are very similar to each other. However, most companies have their own documentation standards and styles. Programmers are expected to be able to quickly adapt to any flowcharting or pseudocode standards for the company at which they work. The other methods that are less universal require some training which is generally provided by the employer that chooses to use them.

Later in your programming career, you will learn about using application software that helps create an information system and/or programs. This type of software is called Computer-Aided Software Engineering (CASE).

Understanding the logic and planning the algorithm on paper before you start to code is a very important concept. Many students develop poor habits and skipping this step is one of them.

Develop Test Data

Test data consists of the programmer providing some input values and predicting the outputs. This can be quite easy for a simple program and the test data can be used to check the model to see if it produces the correct results.

Key Terms

IPO
: Inputs – Processing – Outputs

pseudocode
: English-like statements used to convey the steps of an algorithm or function.

test data
: Providing input values and predicting the outputs.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 5.2: Program Design is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

5.3: Program Quality

Overview

Program quality describes fundamental properties of the program's source code and executable code, including reliability, robustness, usability, portability, maintainability, efficiency, and readability.

Discussion

Whatever the approach to development may be, the final program must satisfy some fundamental properties. The following properties are among the most important:

- **Reliability**: how often the results of a program are correct. This depends on the conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero or off-by-one errors).
- **Robustness**: how well a program anticipates problems due to errors (not bugs). This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, user error, and unexpected power outages.
- **Usability**: the ergonomics of a program: the ease with which a person can use the program for its intended purpose or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness, and completeness of a program's user interface.
- **Portability**: the range of computer hardware and operating system platforms on which the source code of a program can be compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behavior of the hardware and operating system, and availability of platform specific compilers (and sometimes libraries) for the language of the source code.
- **Maintainability**: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a program over the long term.
- **Efficiency**/performance: the measure of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes careful management of resources, for example cleaning up temporary files and eliminating memory leaks.
- **Readability**: the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability. Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code. Unreadable code often leads to bugs, inefficiencies, and duplicated code.

Key Terms

efficiency
: The measure of system resources a program consumes.

maintainability
: The ease with which a program can be modified by its present or future developers.

portability
: The range of computer hardware and operating system platforms on which the source code of a program can be compiled/interpreted and run.

readability
: The ease with which a human reader can comprehend the purpose, control flow, and operation of source code.

reliability
: How often the results of a program are correct.

robustness
: How well a program anticipates problems due to errors.

usability
: The ease with which a person can use the program.

References

- https://en.Wikipedia.org/wiki/Computer_programming

This page titled 5.3: Program Quality is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6: MATH AND VARIABLES

6.1: Arithmetic Operators
6.2: Order of Operations
6.3: Constants and Variables
6.4: Identifier Names
6.5: Variable names and Keywords
6.6: Choosing Mnemonic Variable Names
6.7: Statements
6.8: Assignment
6.9: Variables
6.10: Updating Variables
6.11: Python Examples
6.12: Arrays and Lists
6.13: Index Notation
6.14: A list is a sequence
6.15: Lists are mutable
6.16: Debugging

This page titled 6: Math and Variables is shared under a license and was authored, remixed, and/or curated by LibreTexts.

6.1: Arithmetic Operators

Overview

The basic **arithmetic operations** are addition, subtraction, multiplication, and division. Arithmetic is performed according to an order of operations.[1]

Discussion

An operator performs an action on one or more operands. The common arithmetic operators are:

Action	Common Symbol
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Modulus (associated with integers)	`%`

These arithmetic operators are binary that is they have two operands. The operands may be either constants or variables.

```
age + 1
```

This expression consists of one operator (addition) which has two operands. The first is represented by a variable named age and the second is a literal constant. If age had a value of 14 then the expression would evaluate (or be equal to) 15.

These operators work as you have learned them throughout your life with the exception of division and modulus. We normally think of division as resulting in an answer that might have a fractional part (a floating-point data type). However, division, when both operands are of the integer data type, may act differently. Please refer to the next section on “Integer Division and Modulus”.

Arithmetic Assignment Operators

Many programming languages support a combination of the assignment (`=`) and arithmetic operators (`+`, `-`, `*`, `/`, `%`). Various textbooks call them “compound assignment operators” or “combined assignment operators”. Their usage can be explained in terms of the assignment operator and the arithmetic operators. In the table, we will use the variable age and you can assume that it is of integer data type.

Arithmetic assignment examples:	Equivalent code:
`age += 14;`	`age = age + 14;`
`age -= 14;`	`age = age - 14;`
`age *= 14;`	`age = age * 14;`
`age /= 14;`	`age = age / 14;`
`age %= 14;`	`age = age % 14;`

Pseudocode

```
01 Function Main
```

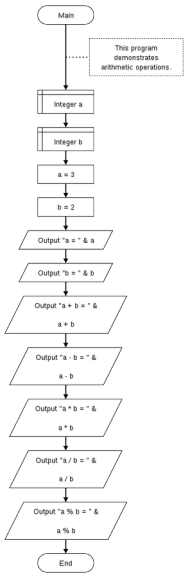
```
    ... This program demonstrates arithmetic operations.
    Declare Integer a
    Declare Integer b

    Assign a = 3
    Assign b = 2
    Output "a = " & a
    Output "b = " & b
    Output "a + b = " & a + b
    Output "a - b = " & a - b
    Output "a * b = " & a * b
    Output "a / b = " & a / b
    Output "a % b = " & a % b
End
```

Output

```
a = 3
b = 2
a + b = 5
a - b = 1
a * b = 6
a / b = 1.5
a % b = 1
```

Flowchart



References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Flowgorithm – Flowchart Programming Language

---

1. Wikipedia: Arithmetic operators ↵

This page titled 6.1: Arithmetic Operators is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6.2: Order of Operations

Overview

The order of operations (or operator precedence) is a collection of rules that reflect conventions about which procedures to perform first in order to evaluate a given mathematical expression.[1]

Discussion

Single values by themselves are important; however, we need a method of manipulating values (processing data). Scientists wanted an accurate machine for manipulating values. They wanted a machine to process numbers or calculate answers (that is, compute the answer). Prior to 1950, dictionaries listed the definition of computers as " humans that do computations". Thus, all of the terminology for describing data manipulation is math oriented. Additionally, the two fundamental data type families (the integer family and floating-point family) consist entirely of number values.

An Expression Example with Evaluation

Let's look at an example: 2 + 3 * 4 + 5 is our expression but what does it equal?

1. the symbols of + meaning addition and * meaning multiplication are our operators
2. the values 2, 3, 4 and 5 are our operands
3. precedence says that multiplication is higher than addition
4. thus, we evaluate the 3 * 4 to get 12
5. now we have: 2 + 12 + 5
6. the associativity rules say that addition goes left to right, thus we evaluate the 2 +12 to get 14
7. now we have: 14 + 5
8. finally, we evaluate the 14 + 5 to get 19; which is the value of the expression

Parentheses would change the outcome. (2 + 3) * (4 + 5) evaluates to 45.

Parentheses would change the outcome. (2 + 3) * 4 + 5 evaluates to 25.

Operator Precedence Chart

Each computer language has some rules that define precedence and associativity. They often follow rules we may have already learned. Multiplication and division come before addition and subtraction is a rule we learned in grade school. This rule still works.

Order of Operations[2]

- Parentheses
- Exponents
- Multiplication / Division
- Addition / Subtraction

A common mnemonic to remember this rule is *PEMDAS*, or *Please Excuse My Dear Aunt Sally*. Precedence rules may vary from one programming language to another. You should refer to the reference sheet that summarizes the rules for the language that you are using. It is often called an Operator Precedence, Precedence of Operators, or Order of Operations chart. You should review this chart as needed when evaluating expressions.

A valid expression consists of operand(s) and operator(s) that are put together properly. Why the (s)? Some operators are:

1. Unary – only have one operand
2. Binary – have two operands, one on each side of the operator

3. Trinary – have two operator symbols that separate three operands

Most operators are binary, that is they require two operands. Some precedence charts indicate of which operators are unary and trinary and thus all others are binary.

Key Terms

associativity
: Determines the order in which the operators of the same precedence are allowed to manipulate the operands.

evaluation
: The process of applying the operators to the operands and resulting in a single value.

expression
: A valid sequence of operand(s) and operator(s) that reduces (or evaluates) to a single value.

operand
: A value that receives the operator's action.

operator
: A language-specific syntactical token (usually a symbol) that causes an action to be taken on one or more operands.

parentheses
: Change the order of evaluation in an expression. You do what's in the parentheses first.

precedence
: Determines the order in which the operators are allowed to manipulate the operands.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Order of operations ↵
2. Wikipedia: Order of operations ↵

This page titled 6.2: Order of Operations is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6.3: Constants and Variables

Overview

A **constant** is a value that cannot be altered by the program during normal execution, i.e., the value is constant. When associated with an identifier, a constant is said to be "named," although the terms "constant" and "named constant" are often used interchangeably. This is contrasted with a **variable**, which is an identifier with a value that can be changed during normal execution, i.e., the value is variable.[1]

Discussion

Understanding Constants

A **constant** is a data item whose value cannot change during the program's execution. Thus, as its name implies – the value is constant.

A **variable** is a data item whose value can change during the program's execution. Thus, as its name implies – the value can vary.

Constants are used in two ways. They are:

1. literal constant
2. defined constant

A literal constant is a **value** you type into your program wherever it is needed. Examples include the constants used for initializing a variable and constants used in lines of code:

```
01 21
02 12.34
03 'A'
04 "Hello world!"
05 false
06 null
```

In addition to literal constants, most textbooks refer to symbolic constants or named constants as a constant represented by a name. Many programming languages use ALL CAPS to define named constants.

Language	Example

C++	`#define PI 3.14159`
	or
	`const double PI = 3.14159;`

C#	`const double PI = 3.14159;`
Java	`const double PI = 3.14159;`
JavaScript	`const PI = 3.14159;`
Python	`PI = 3.14159`
Swift	`let pi = 3.14159`

Technically, Python does not support named constants, meaning that it is possible (but never good practice) to change the value of a constant later. There are workarounds for creating constants in Python, but they are beyond the scope of a first-semester textbook.

Defining Constants and Variables

Named constants must be assigned a value when they are defined. Variables do not have to be assigned initial values. Variables once defined may be assigned a value within the instructions of the program.

Language	Example
C++	`double value = 3;`
C#	`double value = 3;`
Java	`double value = 3;`
JavaScript	`var value = 3;` `let value = 3;`
Python	`value = 3`
Swift	`var value:Int = 3`

Key Terms

constant
: A data item whose value cannot change during the program's execution.

variable
: A data item whose value can change during the program's execution.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Constant (computer programming) ↵

This page titled 6.3: Constants and Variables is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6.4: Identifier Names

Overview

Within programming a variety of items are given descriptive names to make the code more meaningful to us as humans. These names are called “Identifier Names”. Constants, variables, type definitions, functions, etc. when declared or defined are identified by a name. These names follow a set of rules that are imposed by:

1. the language’s technical limitations
2. good programming practices
3. common industry standards for the language

Discussion

Technical to Language

- Use only allowable characters (in many languages the first character must be alphabetic or underscore, can continue with alphanumeric or underscore)
- Can’t use reserved words
- Length limit

These attributes vary from one programming language to another. The allowable characters and reserved words will be different. The length limit refers to how many characters are allowed in an identifier name and often is compiler dependent and may vary from compiler to compiler for the same language. However, all programming languages have some form of the technical rules listed here.

Good Programming Techniques

- Meaningful
- Be case consistent

Meaningful identifier names make your code easier for another to understand. After all what does “p” mean? Is it pi, price, pennies, etc. Thus do not use cryptic (look it up in the dictionary) identifier names.

Some programming languages treat upper and lower case letters used in identifier names as the same. Thus: pig and Pig are treated as the same identifier name. Unknown to you the programmer, the compiler usually forces all identifier names to upper case. Thus: pig and Pig both get changed to PIG. However, not all programming languages act this way. Some will treat upper and lower case letters as being different things. Thus: pig and Pig are two different identifier names. If you declare it as pig and then reference it in your code later as Pig – you get a different variable or perhaps a compiler error. To avoid the problem altogether, we teach students to **be case consistent**. Use an identifier name only one way and spell it (upper and lower case) the same way every time within your program.

Industry Rules

Almost all programming languages and most coding shops have a standard code formatting style guide programmers are expected to follow. Among these are three common identifier casing standards:

- camelCase – each word is capitalized except the first word, with no intervening spaces
- PascalCase – each word is capitalized including the first word, with no intervening spaces
- snake_case – each word is lowercase with an underscores separating words

C++, Java, and JavaScript typically use camelCase, with PascalCase reserved for libraries and classes. C# uses primarily PascalCase with camelCase parameters. Python uses snake_case for most identifiers. In addition, the following rules apply:

- Do not start with an underscore (used for technical programming)
- CONSTANTS IN ALL UPPER CASE (often UPPER_SNAKE_CASE).

These rules are decided by the industry (those who are using the programming language).

Key Terms

camel case
: The practice of writing compound words or phrases such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation.

Pascal case
: The practice of writing compound words or phrases such that each word or abbreviation in the phrase begins with a capital letter, including the first letter, with no intervening spaces or punctuation.

reserved word
: Words that cannot be used by the programmer as identifier names because they already have a specific meaning within the programming language.

snake case
: The practice of writing compound words or phrases in which the elements are separated with one underscore character (_) and no spaces, with each element's initial letter usually lowercased within the compound and the first letter either upper or lower case.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 6.4: Identifier Names is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6.5: Variable names and Keywords

Programmers generally choose names for their variables that are meaningful and document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

The underscore character (`_`) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

It turns out that `class` is one of Python's *keywords*. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 33 keywords:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

This page titled 6.5: Variable names and Keywords is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.3: Variable names and Keywords by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

6.6: Choosing Mnemonic Variable Names

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print(c)

```

```
hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)

```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)

```

The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently. Humans will most quickly understand the *intent* of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.

We call these wisely chosen variable names "mnemonic variable names". The word *mnemonic*[2] means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer's ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are only 33 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names.

Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

```
for word in words:
    print(word)
```

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code *must* be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

```
for slice in pizza:
```

```
    print(slice)
```

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. It is pretty clear that Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices.

But if our program is truly about reading data and looking for words in the data, `pizza` and `slice` are very un-mnemonic variable names. Choosing them as variable names distracts from the meaning of the program.

After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you:

```
for word in words:
    print word
```

The parts of the code that are defined by Python (`for`, `in`, `print`, and `:`) are in bold and the programmer-chosen variables (`word` and `words`) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

This page titled 6.6: Choosing Mnemonic Variable Names is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.12: Choosing Mnemonic Variable Names by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

6.7: Statements

A *statement* is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: expression statement with "print" and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For the script below, notice that the assignment statement (in this case x = 2) produces no output.

Code 2.4.1 (Python)

```
01 print(1)
02 x = 2
03 print(x)
04
```

This page titled 6.7: Statements is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.4: Statements by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

6.8: Assignment

Overview

An **assignment** statement sets and/or re-sets the value stored in the storage location(s) denoted by a variable name; in other words, it copies a value into the variable.[1]

Discussion

The assignment operator allows us to change the value of a modifiable data object (for beginning programmers this typically means a variable). It is associated with the concept of moving a value into the storage location (again usually a variable). Within most programming languages the symbol used for assignment is the equal symbol. But bite your tongue, when you see the = symbol you need to start thinking: assignment. The assignment operator has two operands. The one to the left of the operator is usually an identifier name for a variable. The one to the right of the operator is a value.

Simple Assignment

```
01 age = 21
```

The value 21 is moved to the memory location for the variable named: age. Another way to say it: age is assigned the value 21.

Assignment with an Expression

```
01 total_cousins = 4 + 3 + 5 + 2
```

The item to the right of the assignment operator is an expression. The expression will be evaluated and the answer is 14. The value 14 would be assigned to the variable named: total_cousins.

Assignment with Identifier Names in the Expression

```
01 students_period_1 = 25
02 students_period_2 = 19
03 total_students = students_period_1 + students_period_2;
```

The expression to the right of the assignment operator contains some identifier names. The program would fetch the values stored in those variables; add them together and get a value of 44; then assign the 44 to the total_students variable.

Key Terms

assignment
: An operator that changes the value of a modifiable data object.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Assignment (computer science) ↩

This page titled 6.8: Assignment is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6.9: Variables

One of the most powerful features of a programming language is the ability to manipulate *variables*. A variable is a name that refers to a value.

An *assignment statement* creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

This example makes three assignments. The first assigns a string to a new variable named `message`; the second assigns the integer `17` to `n`; the third assigns the (approximate) value of π to `pi`.

To display the value of a variable, you can use a print statement:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

This page titled 6.9: Variables is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.2: Variables by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

6.10: Updating Variables

A common pattern in assignment statements is an assignment statement that updates a variable, where the new value of the variable depends on the old.

```
01 x = x + 1
```

This means "get the current value of x, add 1, and then update x with the new value."

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
01 >>> x = x + 1
02 NameError: name 'x' is not defined
```

Before you can update a variable, you have to *initialize* it, usually with a simple assignment:

```
01 >>> x = 0
02 >>> x = x + 1
```

Updating a variable by adding 1 is called an *increment*; subtracting 1 is called a *decrement*.

This page titled 6.10: Updating Variables is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.1: Updating Variables by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

6.11: Python Examples

Overview

The following examples demonstrate data types, arithmetic operations, and input in Python.

Data Types

```
# This program demonstrates variables, literal constants, and data types.

i = 1234567890
f = 1.23456789012345
s = "string"
b = True

print("Integer i =", i)
print("Float f =", f)
print("String s =", s)
print("Boolean b =", b)
```

Output

```
Integer i = 1234567890
Float f = 1.23456789012345
String s = string
Boolean b = true
```

Discussion

Each code element represents:

- `#` begins a comment
- `i = , d = , s =, b =` assign literal values to the corresponding variables
- `print()` calls the print function

Arithmetic

```
# This program demonstrates arithmetic operations.

a = 3
b = 2

print("a =", a)
print("b =", b)
print("a + b =", (a + b))
print("a - b =", (a - b))
print("a * b =", a * b)
print("a / b =", a / b)
print("a % b =", (a % b))
```

Output

```
a = 3
b = 2
a + b = 5
a - b = 1
a * b = 6
a / b = 1.5
a % b = 1
```

Discussion

Each new code element represents:

- `+, -, *, /, and %` represent addition, subtraction, multiplication, division, and modulus, respectively.

Temperature

```
# This program converts an input Fahrenheit temperature to Celsius.

print("Enter Fahrenheit temperature:")
fahrenheit = float(input())

celsius = (fahrenheit - 32) * 5 / 9

print(str(fahrenheit) + "° Fahrenheit is " + str(celsius) + "° Celsius")
```

Output

```
Enter Fahrenheit temperature:
 100
100.0° Fahrenheit is 37.77777777777778° Celsius
```

Discussion

Each new code element represents:

- `input()` reads the next line from standard input
- `float()` converts the input to a floating-point value

References

- Wikiversity: Computer Programming

This page titled 6.11: Python Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6.12: Arrays and Lists

Overview

An array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.[1]

Depending on the language, array types may overlap (or be identified with) other data types that describe aggregates of values, such as lists and strings. Array types are often implemented by array data structures, but sometimes by other means, such as hash tables, linked lists, or search trees.[2] In Python, the built-in array data structure is a list.

Discussion

An array is a sequenced collection of elements of the same data type with a single identifier name. Python lists are similar to arrays in other languages but are not restricted to a single data type. The term 'array' as used in this chapter will generally also apply to Python lists unless otherwise noted.

Arrays can have multiple axes (more than one axis). Each axis is a **dimension**. Thus a single-dimension array is also known as a **list**. A two-dimension array is commonly known as a **table** (a spreadsheet like Excel is a two dimension array). In real life, there are occasions to have data organized into multiple-dimension arrays. Consider a theater ticket with section, row, and seat (three dimensions). Most single-dimension arrays are visualized vertically.

Most programmers are familiar with a special type of array called a string. Strings are basically a single dimension array of characters. Unlike other single dimension arrays, we usually envision a string as a horizontal stream of characters and not vertically as a list.

We refer to the individual values as members (or elements) of the array. Programming languages implement the details of arrays differently. Because there is only one identifier name assigned to the array, we have operators that allow us to reference or access the individual members of an array. The operator commonly associated with referencing array members is the **index** operator. It is important to learn how to define an array and initialize its members.

Defining an Array

Language	Example
C++	`int ages[] = {49, 48, 26, 19, 16};`
C#	`int[] ages = {49, 48, 26, 19, 16};`
Java	`int[] ages = {49, 48, 26, 19, 16};`
JavaScript	`var ages = [49, 48, 26, 19, 16];`
Python	`ages = [49, 48, 26, 19, 16]`
Swift	`var ages:[Int] = [49, 48, 26, 19, 16]`

This is the **defining of storage space**. The square brackets `[]` are used here to create the array with five integer members and the identifier name of ages. The assignment with braces (that is a block) establishes the initial values assigned to the members of the array. Note the use of the sequence or comma operator. We could have also done something similar to:

Language	Example	Initial Values
C++	`int ages[5];`	undefined
C#	`int[] ages = new int[5];`	0

Java	`int[] ages = new int[5];`	0
JavaScript	`var ages = Array(5);`	undefined
Python	`ages = [None] * 5`	None

This would have declared the storage space of five integers with the identifier name of ages but their initial values would have been unknown values or initialized as indicated, depending on the programming language. We could assign values later in our program by doing the following (leaving off the semicolons in Python):

```
01 ages[0] = 49;
02 ages[1] = 48;
03 ages[2] = 26;
04 ages[3] = 19;
05 ages[4] = 16;
```

Note: The members of the array go from 0 to 4; **NOT** 1 to 5. This is explained in more detail on the next page.

Key Terms

dimension
: An axis of an array.

list
: A single dimension array.

table
: A two-dimension array.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Array data structure ↩
2. Wikipedia: Array data type ↩

This page titled 6.12: Arrays and Lists is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

Overview

Index notation is used to specify the elements of an array.[1] Most current programming languages use square brackets `[]` as the array index operator. Older programming languages, such as FORTRAN, COBOL, and BASIC, often use parentheses `()` as the array index operator.

Discussion

Example:

Language	Example
C++	`int ages[] = {49, 48, 26, 19, 16};` `int myAge = ages[2];`
C#	`int[] ages = {49, 48, 26, 19, 16};` `int myAge = ages[2];`
Java	`int[] ages = {49, 48, 26, 19, 16};` `int myAge = ages[2];`
JavaScript	`var ages = [49, 48, 26, 19, 16];` `int myAge = ages[2];`
Python	`ages = [49, 48, 26, 19, 16]` `my_age = ages[2]`
Swift	`var ages:[Int] = [49, 48, 26, 19, 16]` `var my_age = ages[2]`

As an operator, square brackets either provide the value held by the member of the array (Rvalue) or change the value of member (Lvalue). In the above example, the member that is two offsets from the front of the array (the value 26) is assigned to the variable named myAge. The dereference operator of [2] means to go the 2nd **offset** from the front of the ages array and get the value stored there. In this case, the value would be 26. In most current programming languages, the array members (or elements) are referenced starting at **zero**. The more common way for people to reference a list is by starting with position **one**. Consider:

Position	Index	Miss America	Other Contests
zero offsets from the front	`ages[0]`	Winner	1st Place
one offset from the front	`ages[1]`	1st Runner Up	2nd Place
two offsets from the front	`ages[2]`	2nd Runner Up	3rd Place
three offsets from the front	`ages[3]`	3rd Runner Up	4th Place
four offsets from the front	`ages[4]`	4th Runner Up	5th Place

Saying that my cousin is the 2nd Runner-Up in the Miss America contest sounds so much better than saying that she was in 3rd Place. We would be talking about the same position in the array of the five finalists.

```
ages[3] = 20;
```

This is an example of changing an array's value by assigning 20 to the 4th member of the array and replacing the value 19 with 20. This is an Lvalue context because the array is on the left side of the assignment operator.

Key Terms

array member
: An element or value in an array.

index
: An operator that allows us to reference a member of an array.

offset
: The method of referencing array members by starting at zero.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Index notation ↩

This page titled 6.13: Index Notation is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

6.14: A list is a sequence

Like a string, a *list* is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called *element*s or sometimes *item*s.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[` and `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is *nested*.

A list that contains no elements is called an empty list; you can create one with empty brackets, `[]`.

As you might expect, you can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

This page titled 6.14: A list is a sequence is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.1: A list is a sequence by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

6.15: Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> print(cheeses[0])
Cheddar
```

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

The one-eth element of `numbers`, which used to be 123, is now 5.

You can think of a list as a relationship between indices and elements. This relationship is called a *mapping*; each index "maps to" one of the elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an `IndexError`.

- If an index has a negative value, it counts backward from the end of the list.

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

This page titled 6.15: Lists are mutable is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.2: Lists are mutable by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

6.16: Debugging

At this point, the syntax error you are most likely to make is an illegal variable name, like `class` and `yield`, which are keywords, or `odd~job` and `US$`, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

```
>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so `LaTeX` is not the same as `latex`.

At this point, the most likely cause of a semantic error is the order of operations. For example, to evaluate $1/2\pi$, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get $\pi/2$, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

This page titled 6.16: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.13: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

7: PROBLEM SOLVING AND DESIGN

7.1: Pseudocode

7.2: Flowcharts

7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms

This page titled 7: Problem Solving and Design is shared under a license and was authored, remixed, and/or curated by LibreTexts.

7.1: Pseudocode

Overview

Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm.[1]

Discussion

Pseudocode is one method of designing or planning a program. **Pseudo** means false, thus pseudocode means false code. A better translation would be the word fake or imitation. Pseudocode is fake (not the real thing). It looks like (imitates) real code but it is NOT real code. It uses English statements to describe what a program is to accomplish. It is fake because no compiler exists that will translate the pseudocode to any machine language. Pseudocode is used for documenting the program or module design (also known as the algorithm).

The following outline of a simple program illustrates pseudocode. We want to be able to enter the ages of two people and have the computer calculate their average age and display the answer.

Outline using Pseudocode

```
Input
    display a message asking the user to enter the first age
    get the first age from the keyboard
    display a message asking the user to enter the second age
    get the second age from the keyboard

Processing
    calculate the answer by adding the two ages together and dividing by two

Output
    display the answer on the screen
    pause so the user can see the answer
```

After developing the program design, we use the pseudocode to write code in a language (like C++, Java, Python, etc.) where you must follow the rules of the language (syntax) in order to code the logic or algorithm presented in the pseudocode. Pseudocode usually does not include other items produced during programming design such as identifier lists for variables or test data.

There are other methods for planning and documenting the logic for a program. One method is HIPO. It stands for Hierarchy plus Input Process Output and was developed by IBM in the 1960s. It involved using a hierarchy (or structure) chart to show the relationship of the sub-routines (or functions) in a program. Each sub-routine had an IPO piece. Since the above problem/task was simple, we did not need to use multiple sub-routines, thus we did not produce a hierarchy chart. We did incorporate the IPO part of the concept for the pseudocode outline.

Key Terms

pseudo
: Means false and includes the concepts of fake or imitation.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Pseudocode ↩

This page titled 7.1: Pseudocode is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

7.2: Flowcharts

Overview

A **flowchart** is a type of diagram that represents an algorithm, workflow or process. The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows. This diagrammatic representation illustrates a solution model to a given problem. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.[1]

Discussion

Common flowcharting symbols and examples follow. When first reading this section, focus on the simple symbols and examples. Return to this section in later chapters to review the advanced symbols and examples.

Simple Flowcharting Symbols

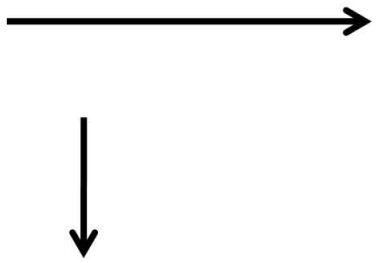
Terminal

The rounded rectangles, or terminal points, indicate the flowchart's starting and ending points.



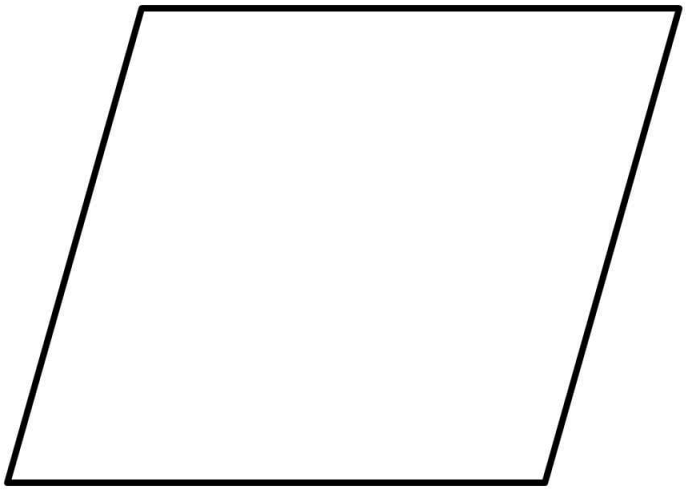
Flow Lines

Note: The default flow is left to right and top to bottom (the same way you read English). To save time arrowheads are often only drawn when the flow lines go contrary the normal.



Input/Output

The parallelograms designate input or output operations.



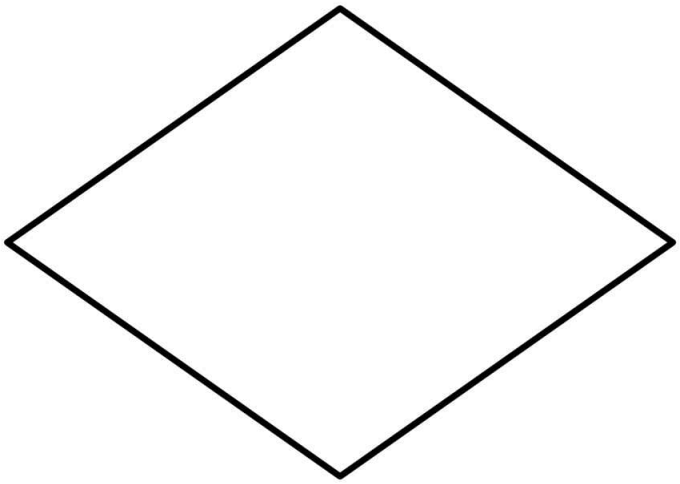
Process

The rectangle depicts a process such as a mathematical computation, or a variable assignment.



Decision

The diamond is used to represent the true/false statement being tested in a decision symbol.

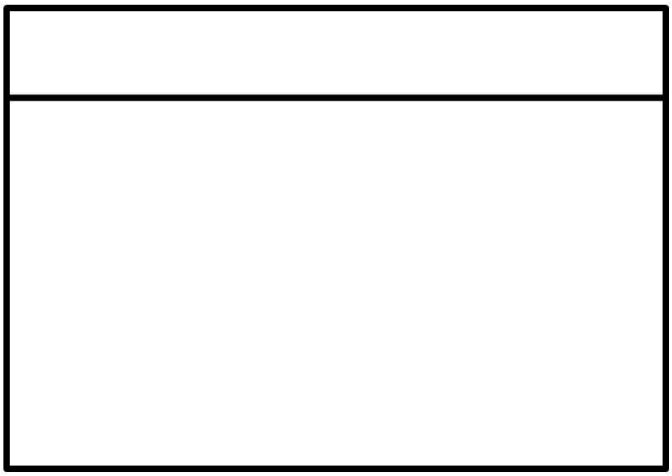


Advanced Flowcharting Symbols

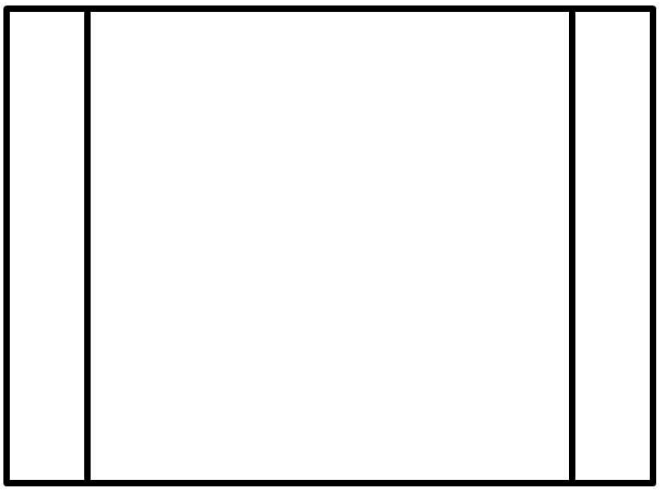
Module Call

A program module is represented in a flowchart by rectangle with some lines to distinguish it from process symbol. Often programmers will make a distinction between program control and specific task modules as shown below.

Local module: usually a program control function.



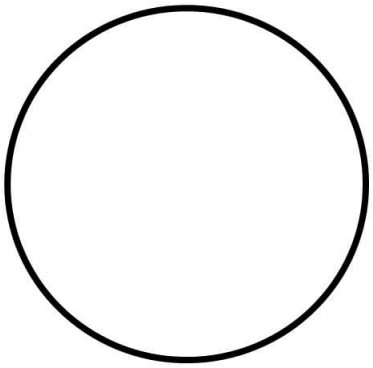
Library module: usually a specific task function.



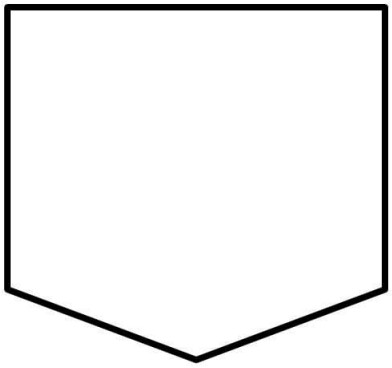
Connectors

Sometimes a flowchart is broken into two or more smaller flowcharts. This is usually done when a flowchart does not fit on a single page, or must be divided into sections. A connector symbol, which is a small circle with a letter or number inside it, allows you to connect two flowcharts on the same page. A connector symbol that looks like a pocket on a shirt, allows you to connect to a flowchart on a different page.

On-Page Connector



Off-Page Connector



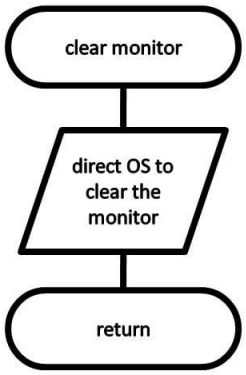
Simple Examples

We will demonstrate various flowcharting items by showing the flowchart for some pseudocode.

Functions

pseudocode: Function with no parameter passing

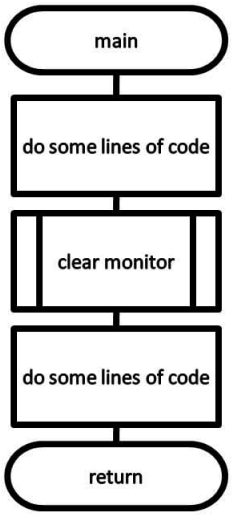
```
01 Function clear monitor
02     Pass In: nothing
03     Direct the operating system to clear the monitor
04     Pass Out: nothing
05 End function
```



Function clear monitor

pseudocode: Function main calling the clear monitor function

```
Function main
    Pass In: nothing
    Doing some lines of code
    Call: clear monitor
    Doing some lines of code
    Pass Out: value zero to the operating system
End function
```



Function main

Sequence Control Structures

The next item is pseudocode for a simple temperature conversion program. This demonstrates the use of both the on-page and off-page connectors. It also illustrates the sequence control structure where nothing unusual happens. Just do one instruction after another in the sequence listed.

pseudocode: Sequence control structure

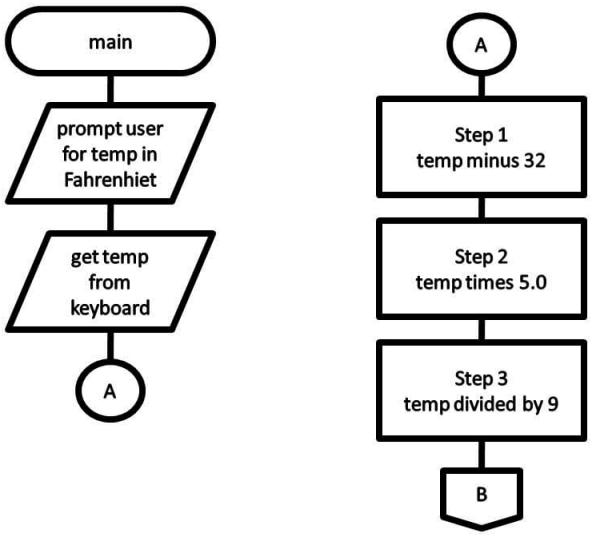
```
Filename: Solution_Lab_04_Pseudocode.txt
Purpose:  Convert Temperature from Fahrenheit to Celsius
Author:   Ken Busbee; © 2008 Kenneth Leroy Busbee
Date:     Dec 24, 2008

Pseudocode = IPO Outline

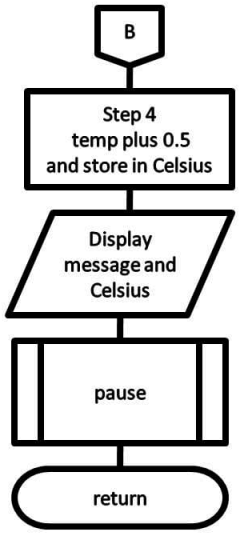
input
    display a message asking user for the temperature in Fahrenheit
    get the temperature from the keyboard

processing
    calculate the Celsius by subtracting 32 from the Fahrenheit
    temperature then multiply the result by 5 then
    divide the result by 9. Round up or down to the whole number.
    HINT: Use 32.0 when subtracting to ensure floating-point accuracy.

output
    display the celsius with an appropriate message
    pause so the user can see the answer
```



Sequence control structure



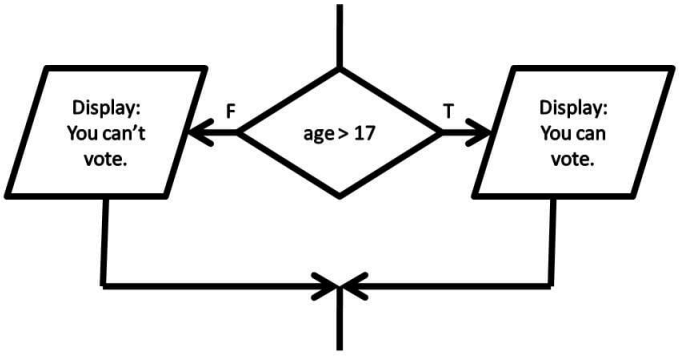
Sequence control structured continued

Advanced Examples

Selection Control Structures

pseudocode: If then Else

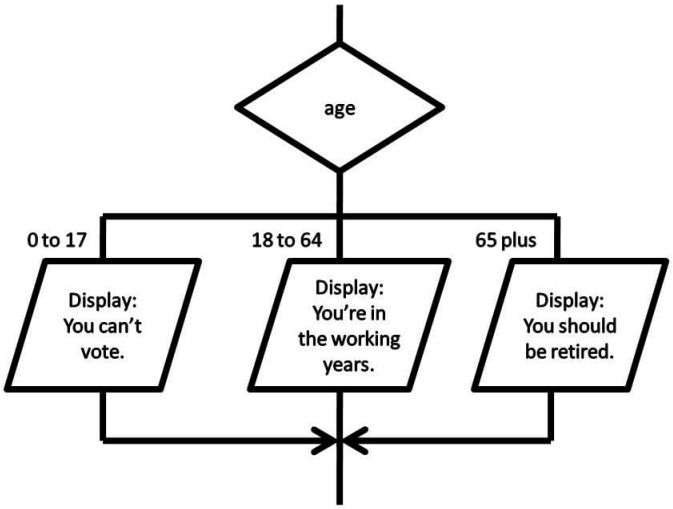
```
If age > 17
    Display a message indicating you can vote.
Else
    Display a message indicating you can't vote.
Endif
```



If then Else control structure

pseudocode: Case

```
Case of age
    0 to 17   Display "You can't vote."
    18 to 64  Display "You are in your working years."
    65 +      Display "You should be retired."
End case
```

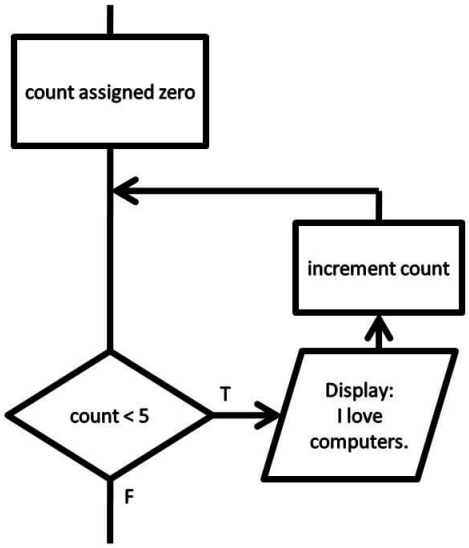


Case control structure

Iteration (Repetition) Control Structures

pseudocode: While

```
count assigned zero
While count < 5
    Display "I love computers!"
    Increment count
End while
```

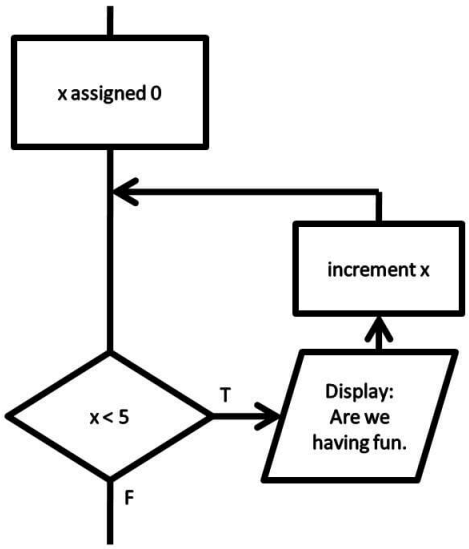


While control structure

pseudocode: For

```
For x starts at 0, x < 5, increment x
    Display "Are we having fun?"
End for
```

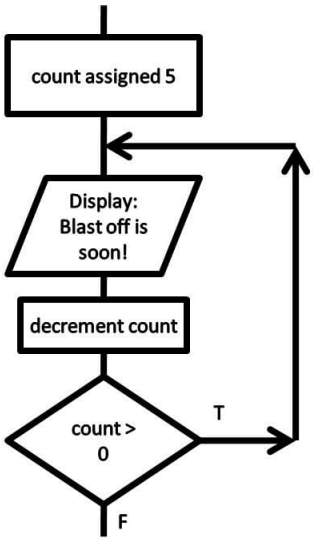
The for loop does not have a standard flowcharting method and you will find it done in different ways. The for loop as a counting loop can be flowcharted similar to the while loop as a counting loop.



For control structure

pseudocode: Do While

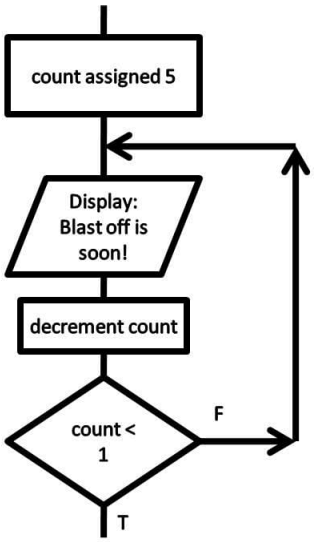
```
count assigned five
Do
    Display "Blast off is soon!"
    Decrement count
While count > zero
```



Do While control structure

pseudocode: Repeat Until

```
count assigned five
Repeat
    Display "Blast off is soon!"
    Decrement count
Until count < one
```



Repeat Until control structure

Key Terms

decision symbol
: A diamond used in flowcharting for asking a question and making a decision.

flow lines
: Lines (sometimes with arrows) that connect the various flowcharting symbols.

flowcharting
: A programming design tool that uses graphical elements to visually depict the flow of logic within a function.

input/output symbol
: A parallelogram used in flowcharting for input/output interactions.

process symbol
: A rectangle used in flowcharting for normal processes such as assignment.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Flowchart ↩

This page titled 7.2: Flowcharts is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms

Introduction

The student will learn how to design an algorithm using either a pseudo code or flowchart. Pseudo code is a mixture of English like statements, some mathematical notations and selected keywords from a programming language. It is one of the tools used to design and develop the solution to a task or problem. Pseudo codes have different ways of representing the same thing and emphasis is on the clarity and not style.

Pseudo Code

Pseudo code is an informal high-level description of the operating principle of a computer program or other algorithm. It is a procedure for solving a problem in terms of the actions to be executed and the order in which those actions are to be executed.

Pseudo code uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. It typically omits details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and some subroutines. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation.

The purpose of using pseudo code is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm. It is commonly used in textbooks and scientific publications that are documenting various algorithms, and also in planning of computer program development, for sketching out the structure of the program before the actual coding takes place.

Example 7.3.1

In the following example, the pseudo code is on program that can add 2 numbers together then display the result.

Solution

```
01 Start program
02 Enter two numbers A, B
03 Add the two numbers together
04 Print sum
05 End program
```

Flow Chart

A flow chart is a type of diagram that represents an algorithm, workflow or process. It shows the steps in the form of boxes of various kinds and their order by connecting them with arrows. The diagrammatic representation illustrates a solution model to a given problem.

Flowcharts can be used in the analysis, design, documenting or managing a process or program in various fields. They are also used in designing and documenting complex processes or programs.

Flowchart building blocks

1. They are represented as circles, ovals or rounded (fillet) rectangles. They contain the word “Start” or “End”, or another phrase signaling the start or end of a process, such as “submit inquiry” or “receive product”.

2. They show “flow of control”. For example an arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to. The line for the arrow can be solid or dashed. The meaning of the arrow with dashed line may differ from one flowchart to another and can be defined in the legend.
3. These are represented as rectangles.
4. Subroutines are represented as rectangles with double-struck vertical edges. They are used to show complex processing steps which may be detailed in a separate flowchart.
5. These are represented as a parallelogram.
6. These are represented as a hexagon. They normally show operations which have no effect other than preparing a value for a subsequent conditional or decision step (see below).
7. These are normally represented as a diamond (rhombus) showing where a decision is necessary. They commonly test a Yes/No question or True/False condition. It has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False.
8. It is generally represented with a black blob, showing where multiple control flows converge in a single exit flow. A junction symbol will have more than one arrow coming into it, but only one going out.
9. They are represented by an identifying label inside a circle. Normally used in complex or multi-sheet diagrams to substitute for arrows. For each label, the “outflow” connector must always be unique, but there may be any number of “inflow” connectors. In this case, a junction in control flow is implied.
10. It is normally represented by a double transverse line with any number of entry and exit arrows.They can be used whenever two or more control flows must operate simultaneously. The exit flows are activated concurrently when all of the entry flows have reached the concurrency symbol. A concurrency symbol with a single entry flow is a fork; one with a single exit flow is a join.

NB

All processes should flow from top to bottom and left to right.

Example 7.3.2

Problem: Calculate and report the grade-point average for a class.

Discussion: The average grade equals the sum of all grades divided by the number of students.

- **Output:** Average grade
- **Input:** Student grades
- **Processing:** Find the sum of the grades; count the number of students; calculate average

PSEUDO CODE

Program: Determine the average grade of a class

```
01 Initialize Counter and Sum to 0
02 Do While there are more data
03     Get the next Grade
04     Add the Grade to the Sum
05     Increment the Counter
06 Loop
07 Computer Average = Sum / Counter
08 Display Average
```

FLOW CHART

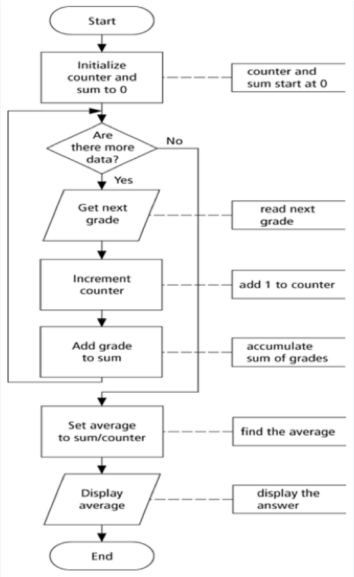


Figure 7.3.1: The Average Grade of a Class Flowchart

Conclusion

This section covered the algorithm development tools, that is, the pseudo codes, flowcharts and how to design and develop them. These tools are not hinged to any programming language but can be implemented in any language of choice. Basically these are the initial steps of designing a program for any given task. Examples for both are given to try and show the learner how they look like. On the flow charts, it has further included the building blocks of a flow chart.

Assessment

1. Give two important differences between the flowchart and the pseudocode.
2. Give two examples of selection control structures. Use flowchart.
3. Draw the different types of symbols used in the flowchart. Explain the role of each.

This page titled 7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 1.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms is licensed CC BY-SA 4.0.

8: DECISIONS

8.1: Relational Operators
8.2: Boolean Expressions
8.3: Conditional Execution
8.4: If Then Else
8.5: Alternative Execution
8.6: Assignment vs Equality
8.7: Code Blocks
8.8: Chained Conditionals

This page titled 8: Decisions is shared under a license and was authored, remixed, and/or curated by LibreTexts.

8.1: Relational Operators

Overview

A **relational operator** is a programming language construct or operator that tests or defines some kind of relation between two entities. These include numerical equality (e.g., 5 = 5) and inequalities (e.g., 4 ≥ 3).[1]

Discussion

The relational operators are often used to create a test expression that controls program flow. This type of expression is also known as a Boolean expression because they create a Boolean answer or value when evaluated. There are six common relational operators that give a Boolean value by comparing (showing the relationship) between two operands. If the operands are of different data types, implicit promotion occurs to convert the operands to the same data type.

Operator symbols and/or names can vary with different programming languages. Most programming languages use relational operators similar to the following:

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equality (equal to)
!= or <>	inequality (not equal to)

Examples:

- 9 < 25
- 9 < 3
- 9 > 14
- 9 <= 17
- 9 >= 25
- 9 == 13
- 9 != 13
- 9 !< 25
- 9 <> 25

Note: Be careful. In math you are familiar with using the symbol = to mean equal and ≠ to mean not equal. In many programming languages the ≠ is not used and the = symbol means assignment.

Key Terms

relational operator
: An operator that gives a Boolean value by evaluating the relationship between two operands.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Relational operator ↩

This page titled 8.1: Relational Operators is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

8.2: Boolean Expressions

A *boolean expression* is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
{}
```

`True` and `False` are special values that belong to the class `bool`; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

The `==` operator is one of the *comparison operators*; the others are:

```
      x != y               # x is not equal to y
      x > y                # x is greater than y
      x < y                # x is less than y
      x >= y               # x is greater than or equal to y
      x <= y               # x is less than or equal to y
      x is y               # x is the same as y
      x is not y           # x is not the same as y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations. A common error is to use a single equal sign (`=`) instead of a double equal sign (`==`). Remember that `=` is an assignment operator and `==` is a comparison operator. There is no such thing as `=<` or `=>`.

This page titled 8.2: Boolean Expressions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

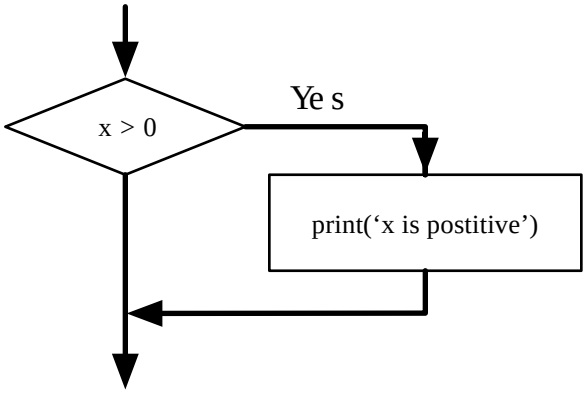
- 3.1: Boolean Expressions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

8.3: Conditional Execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. *Conditional statements* give us this ability. The simplest form is the `if` statement:

```
01 if x > 0 :
02     print('x is positive')
```

The boolean expression after the `if` statement is called the *condition*. We end the `if` statement with a colon character (:) and the line(s) after the if statement are indented.



If Logic

If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

`if` statements have the same structure as function definitions or `for` loops[1]. The statement consists of a header line that ends with the colon character (:) followed by an indented block. Statements like this are called *compound statements* because they stretch across more than one line.

There is no limit on the number of statements that can appear in the body, but there must be at least one. Occasionally, it is useful to have a body with no statements (usually as a place holder for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
01 if x < 0 :
02     pass          # need to handle negative values!
```

If you enter an `if` statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements, as shown below:

```
01 >>> x = 3
02 >>> if x < 10:
03 ...     print('Small')
04 ...
05 Small
06 >>>
```

This page titled 8.3: Conditional Execution is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.3: Conditional Execution by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

8.4: If Then Else

Overview

The **if–then–else** construct, sometimes called if-then, is a two-way selection structure common across many programming languages. Although the syntax varies from language to language, the basic structure looks like:[1]

```
If (boolean condition) Then
    (consequent)
Else
    (alternative)
End If
```

Discussion

We are going to introduce the control structure from the selection category that is available in every high level language. It is called the **if then else** structure. Asking a question that has a true or false answer controls the if then else structure. It looks like this:

```
if the answer to the question is true
    then do this
else because it is false
    do this
```

In most languages, the question (called a test expression) is a Boolean expression. The Boolean data type has two values – true and false. Let's rewrite the structure to consider this:

```
if expression is true
    then do this
else because it is false
    do this
```

Some languages use reserved words of: "if", "then" and "else". Many eliminate the "then". Additionally the "do this" can be tied to true and false. You might see it as:

```
if expression is true
    action true
else
    action false
```

And most languages infer the "is true" you might see it as:

```
if expression
    action true
else
    action false
```

The above four forms of the control structure are saying the same thing. The else word is often not used in our English speaking today. However, consider the following conversation between a mother and her child.

Child asks, “Mommy, may I go out side and play?”

Mother answers, “If your room is clean then you may go outside and play or else you may go sit on a chair for five minutes as punishment for asking me the question when you knew your room was dirty.”

Let’s note that all of the elements are present to determine the action (or flow) that the child will be doing. Because the question (your room is clean) has only two possible answers (true or false) the actions are **mutually exclusive**. Either the child 1) goes outside and plays or 2) sits on a chair for five minutes. One of the actions is executed; never both of the actions.

One Choice – Implied Two-Way Selection

Often the programmer will want to do something only if the expression is true, that is with no false action. The lack of a false action is also referred to as a “null else” and would be written as:

```
01 if expression
02     action true
03 else
04     do nothing
```

Because the “else do nothing” is implied, it is usually written in short form like:

```
01 if expression
02     action true
```

Key Terms

if then else
: A two-way selection control structure.

mutually exclusive
: Items that do not overlap. Example: true or false.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Conditional (computer programming) ↩

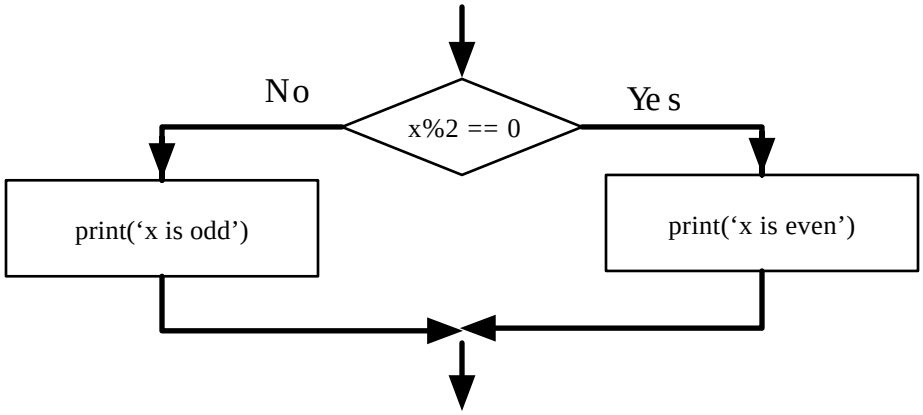
This page titled 8.4: If Then Else is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

8.5: Alternative Execution

A second form of the `if` statement is *alternative execution*, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

If the remainder when `x` is divided by 2 is `0`, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.



If-Then-Else Logic

Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called *branches*, because they are branches in the flow of execution.

This page titled 8.5: Alternative Execution is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.4: Alternative Execution by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

8.6: Assignment vs Equality

Overview

Assignment sets and/or re-sets the value stored in the storage location denoted by a variable name.[1]**Equality** is a relational operator that tests or defines the relationship between two entities.[2]

Discussion

Most control structures use a test expression that executes either selection (as in the: if then else) or iteration (as in the while; do while; or for loops) based on the truthfulness or falseness of the expression. Thus, we often talk about the Boolean expression that is controlling the structure. Within many programming languages, this expression must be a Boolean expression and is governed by a tight set of rules. However, in many programming languages, each data type can be used as a Boolean expression because each data type can be demoted into a Boolean value by using the rule/concept that zero and nothing represent false and all non-zero values represent true.

Within various languages, we have the potential added confusion of the equals symbol = as an operator that does not represent the normal math meaning of equality that we have used for most of our life. The equals symbol typically means: assignment. To get the equality concept of math we often use two equal symbols to represent the relational operator of equality. Let's consider:

```
01 If (pig = 'y')
02     Output "Pigs are good"
03 Else
04     Output "Pigs are bad."
```

The test expression of the control structure will always be true because the expression is an assignment (not the relational operator of ==). It assigns the 'y' to the variable pig, then looks at the value in pig and determines that it is not zero; therefore the expression is true. And it will always be true and the else part will never be executed. This is not what the programmer had intended. The correct syntax for a Boolean expression is:

```
01 If (pig == 'y')
02     Output "Pigs are good"
03 Else
04     Output "Pigs are bad."
```

This example reminds you that you must be careful in creating your test expressions so that they are indeed a question, usually involving the relational operators. Some programming languages will generate a warning or an error when an assignment is used in a Boolean expression, and some do not.

Don't get caught using assignment for equality.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Assignment (computer science) ↩
2. Wikipedia: Relational operator ↩

This page titled 8.6: Assignment vs Equality is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

8.7: Code Blocks

Overview

A **code block**, sometimes referred to as a compound statement, is a lexical structure of source code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called a block-structured programming language. Blocks are fundamental to structured programming, where control structures are formed from blocks.[1]

Discussion

The Need for a Compound Statement

Within many programming languages, there can be only **one statement listed as the action part of a control structure:**

```
01 if (expression)
02     statement
03 else
04     statement
```

Often, we will want to do more than one statement. This problem is overcome by creating a code block or compound statement. For programming languages that use curly braces {} to designate code blocks, a compound if-then-else statement would be similar to:

```
01 if(expression)
02 {
03     statement;
04     statement;
05 }
06 else
07 {
08     statement;
09     statement;
10 }
```

Because programmers often forget that they can have **only one statement listed as the action part of a control structure**; the programming industry encourages the use of indentation (to see the action parts clearly) and the use of compound statements (braces) **always**, even when there is only one statement. Thus:

```
01 if(expression)
02 {
03     statement;
04 }
05 else
06 {
07     statement;
08 }
```

By writing code in this manner, if the programmer modifies the code by adding more statements to either the action true or the action false; they will not introduce either compiler or logic errors. Using indentation and braces should become standard practice

in any language that requires the use of compound statements with control structures.

Indentation and End Block

Other programming languages require explicit designation of code blocks through either indentation or some type of end block statement. For example, Python uses indentation to indicate the statements in a code block:

```
01 if expression:
02     statement
03     statement
04 else:
05     statement
06     statement
```

Lua uses an end block reserved word:

```
01 if expression then
02     statement
03     statement
04 else
05     statement
06     statement
07 end
```

The general if-then-else structure in each of these programming languages is similar, as is the required or expected indentation. The difference is in the syntax used to designate the code blocks.

Key Terms

block
: Another name for a compound statement.

compound statement

A unit of code consisting of zero or more statements.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Block (programming)↵

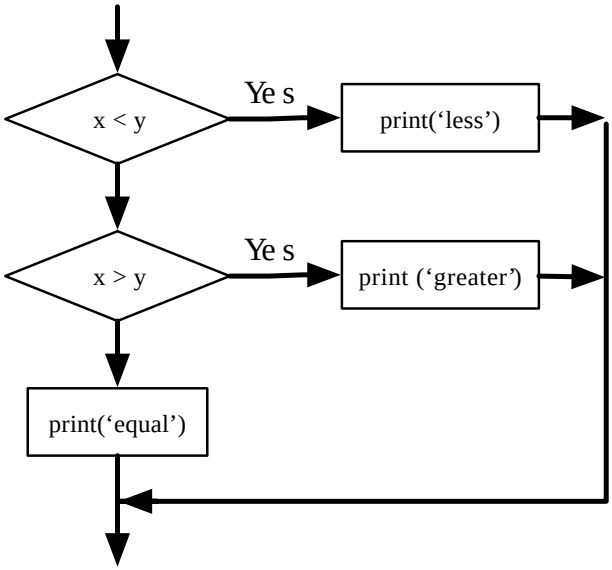
This page titled 8.7: Code Blocks is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

8.8: Chained Conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a *chained conditional*:

```
01 if x < y:
02     print('x is less than y')
03 elif x > y:
04     print('x is greater than y')
05 else:
06     print('x and y are equal')
```

`elif` is an abbreviation of "else if." Again, exactly one branch will be executed.



If-Then-ElseIf Logic

There is no limit on the number of `elif` statements. If there is an `else` clause, it has to be at the end, but there doesn't have to be one.

Code 3.5.1 (Python)

```
01 choice = input()
02 if choice == 'a':
03     print('Bad guess')
04 elif choice == 'b':
05     print('Good guess')
06 elif choice == 'c':
07     print('Close, but not correct')
08
09 print (' Done!')
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

This page titled 8.8: Chained Conditionals is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.5: Chained Conditionals by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

9: HELPER FUNCTIONS

9.1: Adding New Functions
9.2: Definitions and Uses
9.3: Why functions?
9.4: Flow of Execution
9.5: Function Examples
9.6: Hierarchy or Structure Chart

This page titled 9: Helper Functions is shared under a license and was authored, remixed, and/or curated by LibreTexts.

9.1: Adding New Functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A *function definition* specifies the name of a new function and the sequence of statements that execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program.

Here is an example:

```
01 def print_lyrics():
02     print("I'm a lumberjack, and I'm okay.")
03     print('I sleep all night and I work all day.')
```

`def` is a keyword that indicates that this is a function definition. The name of the function is `print_lyrics`. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions that take arguments as their inputs.

The first line of the function definition is called the *header*; the rest is called the *body*. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

If you type a function definition in interactive mode, the interpreter prints ellipses (...) to let you know that the definition isn't complete:

```
01 >>> def print_lyrics():
02 ...     print("I'm a lumberjack, and I'm okay.")
03 ...     print('I sleep all night and I work all day.')
04 ...
```

To end the function, you have to enter an empty line (this is not necessary in a script).

Defining a function creates a variable with the same name.

```
01 >>> print(print_lyrics)
02 <function print_lyrics at 0xb7e99e9c>
03 >>> print(type(print_lyrics))
04 <class 'function'>
```

The value of `print_lyrics` is a *function object*, which has type "function".

The syntax for calling the new function is the same as for built-in functions:

```
01 >>> print_lyrics()
02 I'm a lumberjack, and I'm okay.
03 I sleep all night and I work all day.
```

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that's not really how the song goes.

This page titled 9.1: Adding New Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.6: Adding New Functions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

9.2: Definitions and Uses

Pulling together the code fragments from the previous section, the whole program looks like this:

Code 4.7.1 (Python)

```
%%python3
def print_lyrics():
    print(\"I'm a lumberjack, and I'm okay.\")
    print('I sleep all night and I work all day.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: http://www.py4e.com/code3/lyrics.py

```

This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

Exercise 2: Edit the code block and move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Exercise 3: Move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

This page titled 9.2: Definitions and Uses is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.7: Definitions and Uses by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

9.3: Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as "find the smallest value in a list of values". Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named `min` which takes a list of values as its argument and returns the smallest value in the list.

This page titled 9.3: Why functions? is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.11: Why functions? by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

9.4: Flow of Execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the *flow of execution*.

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.

Function *definitions* do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

This page titled 9.4: Flow of Execution is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.8: Flow of Execution by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

9.5: Function Examples

Overview

The following pseudocode and flowchart examples take the Temperature program from the previous chapter and separate the functionality into independent functions for input, processing, and output, as GetFahrenheit, CalculateCelsius, and DisplayResult, respectively.

Discussion

As independent functions, each function acts as a miniature program, with its own input, processing, and output. As you review the following code, note which functions have parameters (input) and which functions have return values (output). Parameters and return values will be discussed in the next few pages.

Function	Purpose	Parameters (input)	Return Value (output)
Main	main program	none	none
GetFahrenheit	input	none	fahrenheit
CalculateCelsius	processing	fahrenheit	celsius
DisplayResult	output	fahrenheit, celsius	none

Pseudocode

```
Function Main
    ... This program asks the user for a Fahrenheit temperature,
    ... converts the given temperature to Celsius,
    ... and displays the results.

    Declare Real fahrenheit
    Declare Real celsius

    Assign fahrenheit = GetFahrenheit()
    Assign celsius = CalculateCelsius(fahrenheit)
    Call DisplayResult(fahrenheit, celsius)
End

Function GetFahrenheit
    Declare Real fahrenheit

    Output "Enter Fahrenheit temperature:"
    Input fahrenheit
Return Real fahrenheit

Function CalculateCelsius (Real fahrenheit)
    Declare Real celsius

    Assign celsius = (fahrenheit - 32) * 5 / 9
Return Real celsius

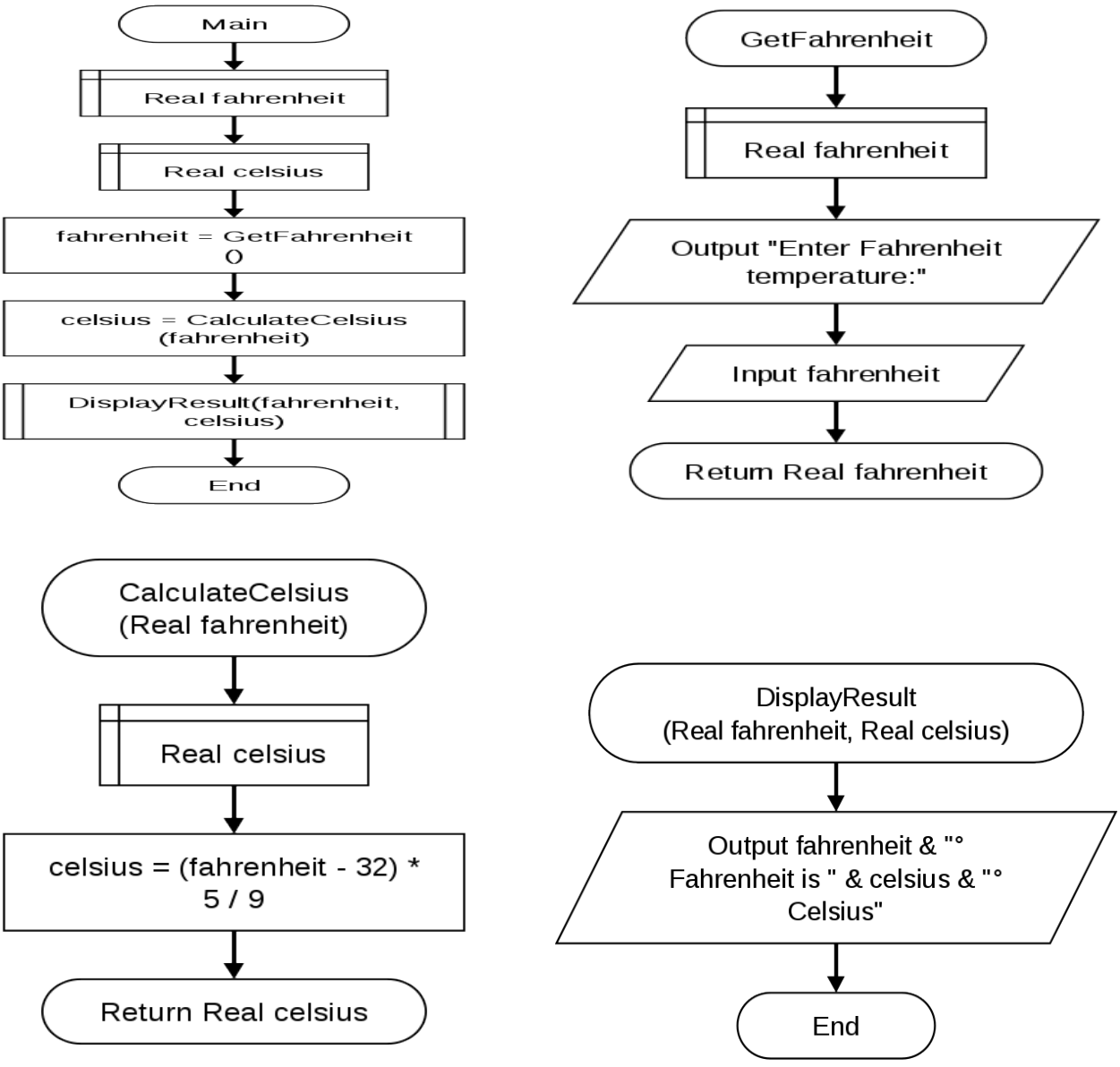
Function DisplayResult (Real fahrenheit, Real celsius)
    Output fahrenheit & "° Fahrenheit is " & celsius & "° Celsius"
```

```
29 End
```

Output

```
01 Enter Fahrenheit temperature:
02  100
03 100° Fahrenheit is 37.7777777777778° Celsius
```

Flowchart



References

- Wikiversity: Computer Programming
- Flowgorithm – Flowchart Programming Language

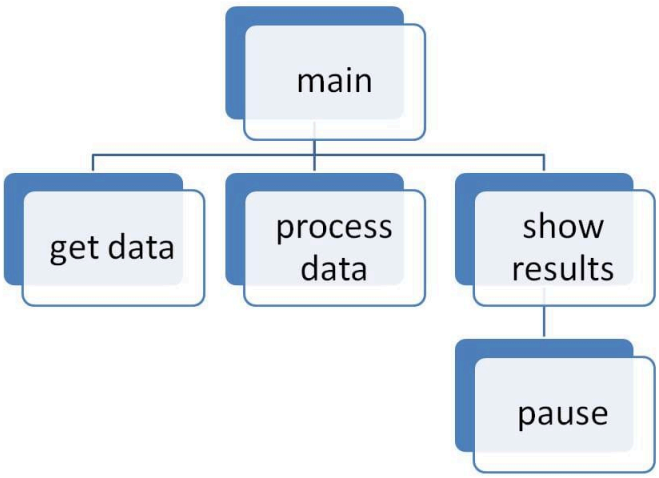
This page titled 9.5: Function Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

9.6: Hierarchy or Structure Chart

Overview

The **hierarchy chart** (also known as a **structure chart**) shows the relationship between various modules. Its name comes from its general use in showing the organization (or structure) of a business. The President at the top, then vice presidents on the next level, etc. Within the context of a computer program, it shows the relationship between modules (or functions). Detail logic of the program is not presented. It does represent the organization of the functions used within the program showing which functions are calling on a subordinate function. Those above are calling those on the next level down.

Hierarchy charts are created by the programmer to help document a program. They convey the big picture of the modules (or functions) used in a program.



Hierarchy or Structure chart for a program that has five functions.

Key Terms

hierarchy chart
: Convey the relationship or big picture of the various functions in a program.

structure chart
: Another name for a hierarchy chart.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 9.6: Hierarchy or Structure Chart is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

10: NESTED DECISION

10.1: Logical Operators
10.2: Logical Operators
10.3: Nested If Then Else
10.4: Condition Examples
10.5: Python Examples
10.6: Nested Conditionals
10.7: Catching exceptions Using Try and Except

This page titled 10: Nested Decision is shared under a license and was authored, remixed, and/or curated by LibreTexts.

10.1: Logical Operators

Overview

A **logical operator** is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator.[1] Common logical operators include AND, OR, and NOT.

Discussion

Within most languages, expressions that yield Boolean data type values are divided into two groups. One group uses the relational operators within their expressions and the other group uses logical operators within their expressions.

The logical operators are often used to help create a test expression that controls program flow. This type of expression is also known as a Boolean expression because they create a Boolean answer or value when evaluated. There are three common logical operators that give a Boolean value by manipulating other Boolean operand(s). Operator symbols and/or names vary with different programming languages:

Language	AND	OR	NOT
C++	`&&`	`||`	`!`
C#	`&&`	`||`	`!`
Java	`&&`	`||`	`!`
JavaScript	`&&`	`||`	`!`
Python	`and`	`or`	`not`
Swift	`&&`	`||`	`!`

The vertical dashes or piping symbol is found on the same key as the backslash \. You use the SHIFT key to get it. It is just above the Enter key on most keyboards. It may be a solid vertical line on some keyboards and show as a solid vertical line on some print fonts.

In most languages there are strict rules for forming proper logical expressions. An example is:

```
6 > 4 && 2 <= 14
6 > 4 and 2 <= 14
```

This expression has two relational operators and one logical operator. Using the precedence of operator rules the two “relational comparison” operators will be done before the “logical and” operator. Thus:

```
true && true
True and True
```

The final evaluation of the expression is: true.

We can say this in English as: It is true that six is greater than four and that two is less than or equal to fourteen.

When forming logical expressions programmers often use parentheses (even when not technically needed) to make the logic of the expression very clear. Consider the above complex Boolean expression rewritten:

```
(6 > 4) && (2 <= 14)
(6 > 4) and (2 <= 14)
```

Most programming languages recognize any non-zero value as true. This makes the following a valid expression:

```
6 > 4 && 8
6 > 4 and 8
```

But remember the order of operations. In English, this is six is greater than four and eight is not zero. Thus,

```
true && true
True and True
```

To compare 6 to both 4 and 8 would instead be written as:

```
6 > 4 && 6 > 8
6 > 4 and 6 > 8
```

This would evaluate to false as:

```
true && false
True and False
```

Truth Tables

A common way to show logical relationships is in truth tables.

Logical and (&&)

x	**y**	**x and y**
false	false	false
false	true	false
true	false	false
true	true	true

Logical or (||)

x	**y**	**x or y**
false	false	false
false	true	true
true	false	true
true	true	true

Logical not (!)

x	**not x**
false	true
true	false

Examples

I call this example of why I hate “and” and love “or”.

Every day as I came home from school on Monday through Thursday; I would ask my mother, “May I go outside and play?” She would answer, “If your room is clean and your homework is done then you may go outside and play.” I learned to hate the word “and”. I could manage to get one of the tasks done and have some time to play before dinner, but both of them... well, I hated

“and”.

On Friday my mother took a more relaxed viewpoint and when asked if I could go outside and play she responded, “If your room is clean or your homework is done then you may go outside and play.” I learned to clean my room quickly on Friday afternoon. Well, needless to say, I loved “or”.

For the next example, just imagine a teenager talking to their mother. During the conversation, mom says, “After all, your Dad is reasonable!” The teenager says, “Reasonable. (short pause) Not.”

Maybe college professors will think that all their students studied for the exam. Ha ha! Not. Well, I hope you get the point.

Examples:

- 25 < 7 || 15 > 36
- 15 > 36 || 3 < 7
- 14 > 7 && 5 <= 5
- 4 > 3 && 17 <= 7
- ! false
- ! (13 != 7)
- 9 != 7 && ! 0
- 5 > 1 && 7

More examples:

- 25 < 7 or 15 > 36
- 15 > 36 or 3 < 7
- 14 > 7 and 5 <= 5
- 4 > 3 and 17 <= 7
- not False
- not (13 != 7)
- 9 != 7 and not 0
- 5 > 1 and 7

Key Terms

logical operator
: An operator used to create complex Boolean expressions.

truth tables
: A common way to show logical relationships.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Logical connective ↩

This page titled 10.1: Logical Operators is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

10.2: Logical Operators

There are three *logical operators*: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example,

```
x > 0 and x < 10
```

is true only if `x` is greater than 0 *and* less than 10.

`n%2 == 0 or n%3 == 0` is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the `not` operator negates a boolean expression, so `not (x > y)` is true if `x > y` is false; that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

```
01 >>> 17 and True
02 True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it until you are sure you know what you are doing.

This page titled 10.2: Logical Operators is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.2: Logical Operators by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

10.3: Nested If Then Else

Overview

Two-way selection structures may be **nested** inside other two-way selection structures, resulting in **multi-way selection**.

Discussion

We are going to first introduce the concept of nested control structures. Nesting is a concept that places one item inside of another. Consider:

```
01 if expression
02     true action
03 else
04     false action
```

This is the basic form of the if then else control structure. Now consider:

```
01 if age is less than 18
02     you can't vote
03     if age is less than 16
04         you can't drive
05     else
06         you can drive
07 else
08     you can vote
09     if age is less than 21
10         you can't drink
11     else
12         you can drink
```

As you can see we simply included as part of the “true action” a statement and another if then else control structure. We did the same (nested another if then else) for the “false action”. In our example, we nested if then else control structures. Nesting could have an if then else within a while loop. Thus, the concept of nesting allows the mixing of the different categories of control structures.

Multiway Selection

One of the drawbacks of two-way selection is that we can only consider two choices. But what do you do if you have more than two choices? Consider the following which has four choices:

```
01 if age equal to 18
02     you can now vote
03 else
04     if age equal to 39
05         you are middle-aged
06     else
07         if age equal to 65
08             you can consider retirement
09         else
```

```
10              your age is unimportant
```

You get an appropriate message depending on the value of age. The last item is referred to as the default. If the age is not equal to 18, 39 or 65 you get the default message. To simplify the code structure, this is most often written as:

```
01 if age equal to 18
02     you can now vote
03 else if age equal to 39
04     you are middle-aged
05 else if age equal to 65
06     you can consider retirement
07 else
08     your age is unimportant
```

Key Terms

multiway selection
: Using control structures to be able to select from more than two choices.

nested control structures Placing one control structure inside of another.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 10.3: Nested If Then Else is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

10.4: Condition Examples

Temperature

Pseudocode

```
Function Main
    Declare String choice

    Assign choice = GetChoice()
    If Choice = "C" Or Choice = "c"
        Call ProcessCelsius()
    False:
        If Choice = "F" Or Choice = "f"
            Call ProcessFahrenheit()
        False:
            Output "You must enter C to convert to Celsius or F to convert to Fahrenheit!"
        End
    End
End

Function CalculateCelsius (Real fahrenheit)
    Declare Real celsius

    Assign celsius = (fahrenheit - 32) * 5 / 9
Return Real celsius

Function CalculateFahrenheit (Real celsius)
    Declare Real fahrenheit

    Assign fahrenheit = celsius * 9 / 5 + 32
Return Real fahrenheit

Function DisplayResult (Real temperature, String fromScale, Real result, String toScale)
    Output temperature & "° " & fromScale & " is " & result & "° " & toScale
End

Function GetChoice
    Declare String choice

    Output "Enter F to convert to Fahrenheit or C to convert to Celsius:"
    Input Choice
Return String choice

Function GetTemperature (String scale)
    Declare Real temperature

    Output "Enter " & scale & " temperature:"
    Input temperature
Return Real temperature

Function ProcessCelsius
    Declare Real temperature
```

```
    Declare Real result

    Assign temperature = GetTemperature("Celsius")
    Assign result = CalculateCelsius(temperature)
    Call DisplayResult(temperature, "Fahrenheit", result, "Celsius")
End

Function ProcessFahrenheit
    Declare Real temperature
    Declare Real result

    Assign temperature = GetTemperature("Fahrenheit")
    Assign result = CalculateFahrenheit(temperature)
    Call DisplayResult(temperature, "Celsius", result, "Fahrenheit")
End
```

Output

```
Enter C to convert to Celsius or F to convert to Fahrenheit:
 c
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7777777777778° Celsius

Enter C to convert to Celsius or F to convert to Fahrenheit:
 f
Enter Celsius temperature:
 100
100° Celsius is 212° Fahrenheit

Enter C to convert to Celsius or F to convert to Fahrenheit:
 x
You must enter C to convert to Celsius or F to convert to Fahrenheit.
```

Flowchart

File:Flowgorithm ProcessCelsius.svg	Conditions	File:Flowgorithm ProcessFahrenheit.svg	Conditions
File:Flowgorithm GetChoice.svg	Conditions	File:Flowgorithm GetTemperature.svg	Conditions
File:Flowgorithm CalculateCelsius.svg	Conditions	File:Flowgorithm CalculateFahrenheit.svg	Conditions

File:Flowgorithm Conditions DisplayResult.svg

References

- Wikiversity: Computer Programming

This page titled 10.4: Condition Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

10.5: Python Examples

Temperature

```
# This program asks the user to select Fahrenheit or Celsius conversion
# and input a given temperature. Then the program converts the given
# temperature and displays the result.
#
# References:
#     https://www.mathsisfun.com/temperature-conversion.html
#     https://en.wikibooks.org/wiki/Python_Programming


def get_choice():
    print("Enter C to convert to Celsius or F to convert to Fahrenheit:")
    choice = input()
    return choice


def process_celsius():
    temperature = get_temperature("Fahrenheit")
    result = calculate_celsius(temperature)
    display_result (temperature, "Fahrenheit", result, "Celsius")


def process_fahrenheit():
    temperature = get_temperature("Celsius")
    result = calculate_fahrenheit(temperature)
    display_result (temperature, "Celsius", result, "Fahrenheit")


def get_temperature(scale):
    print("Enter " + scale + " temperature:")
    temperature = float(input())
    return temperature


def calculate_celsius(fahrenheit):
    celsius = (fahrenheit - 32) * 5 / 9
    return celsius


def calculate_fahrenheit(celsius):
    fahrenheit = celsius * 9 / 5 + 32
    return fahrenheit


def display_result(temperature, fromScale, result, toScale):
    print(str(temperature) + str("° ") + fromScale + " is " + str(result) + "° " + toScale)


def main():
    choice = get_choice()
    if choice == "C" or choice == "c":
```

```
        process_celsius ()
    elif choice == "F" or choice == "f":
        process_fahrenheit ()
    else:
        print("You must enter C to convert to Celsius or F to convert to Fahrenheit.")


main()
```

Output

```
Enter C to convert to Celsius or F to convert to Fahrenheit:
 c
Enter Fahrenheit temperature:
 100
100.0° Fahrenheit is 37.77777777777778° Celsius

Enter C to convert to Celsius or F to convert to Fahrenheit:
 f
Enter Celsius temperature:
 100
100.0° Celsius is 212.0° Fahrenheit

Enter C to convert to Celsius or F to convert to Fahrenheit:
 x
You must enter C to convert to Celsius or F to convert to Fahrenheit.
```

References

- Wikiversity: Computer Programming

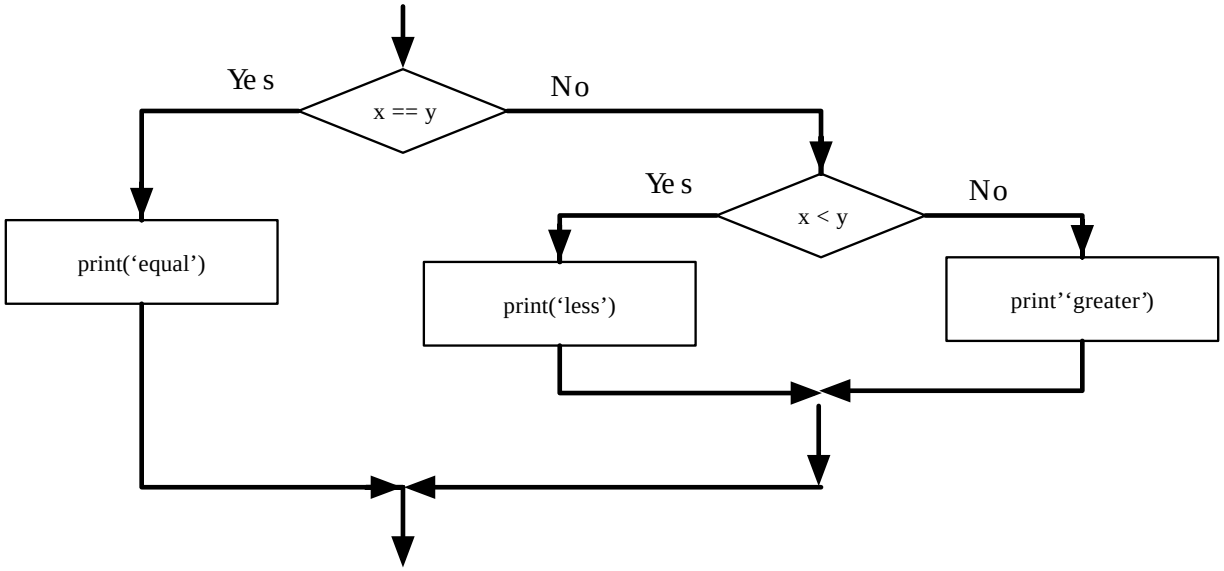
This page titled 10.5: Python Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

10.6: Nested Conditionals

One conditional can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another `if` statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.



Nested If Statements

Although the indentation of the statements makes the structure apparent, *nested conditionals* become difficult to read very quickly. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

The `print` statement is executed only if we make it past both conditionals, so we can get the same effect with the `and` operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

This page titled 10.6: Nested Conditionals is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.6: Nested Conditionals by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

10.7: Catching exceptions Using Try and Except

Earlier we saw a code segment where we used the `input` and `int` functions to read and parse an integer number entered by the user. We also saw how treacherous doing this could be:

```
>>> prompt = "What...is the airspeed velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

When we are executing these statements in the Python interpreter, we get a new prompt from the interpreter, think "oops", and move on to our next statement.

However if you place this code in a Python script and this error occurs, your script immediately stops in its tracks with a traceback. It does not execute the following statement.

Here is a sample program to convert a Fahrenheit temperature to a Celsius temperature:

Code 3.7.1 (Python)

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: http://www.py4e.com/code3/fahren.py

```

If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222
```

```
python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "try / except". The idea of `try` and `except` is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.

You can think of the `try` and `except` feature in Python as an "insurance policy" on a sequence of statements.

We can rewrite our temperature converter as follows:

Code 3.7.2 (Python)

```
inp = input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Please enter a number')

# Code: http://www.py4e.com/code3/fahren2.py

```

Python starts by executing the sequence of statements in the `try` block. If all goes well, it skips the `except` block and proceeds. If an exception occurs in the `try` block, Python jumps out of the `try` block and executes the sequence of statements in the `except` block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222
```

```
python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Handling an exception with a `try` statement is called *catching* an exception. In this example, the `except` clause prints an error message. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

This page titled 10.7: Catching exceptions Using Try and Except is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.7: Catching exceptions Using Try and Except by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

11: SOFTWARE DEVELOPMENT LIFE CYCLE

11.1: Systems Development Life Cycle
11.2: Software Testing
11.3: Version Control
11.4: Programming Style

This page titled 11: Software Development Life Cycle is shared under a license and was authored, remixed, and/or curated by LibreTexts.

11.1: Systems Development Life Cycle

Overview

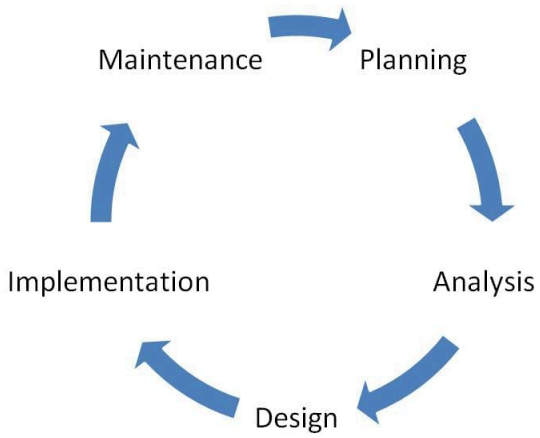
The **Systems Development Life Cycle** describes a process for planning, creating, testing, and deploying an information system.[1]

Discussion

The Systems Development Life Cycle is the big picture of creating an information system that handles a major task (referred to as an application). The **applications** usually consist of many programs. An example would be the Department of Defense supply system, the customer system used at your local bank, the repair parts inventory system used by car dealerships. There are thousands of applications that use an information system created just to help solve a business problem.

Another example of an information system would be the "101 Computer Games" software you might buy at any of several retail stores. This is an entertainment application, that is we are applying the computer to do a task (entertain you). The software actually consists of many different programs (checkers, chess, tic tac toe, etc.) that were most likely written by several different programmers.

Computer professionals that are in charge of creating applications often have the job title of **System Analyst**. The major steps in creating an application include the following and start at **Planning** step.



Systems Development Life Cycle

During the **Design** phase, the System Analyst will document the inputs, processing, and outputs of each program within the application. During the **Implementation** phase, programmers would be assigned to write the specific programs using a programming language decided by the System Analyst. Once the system of programs is tested the new application is installed for people to use. As time goes by, things change and a specific part or program might need repair. During the **Maintenance** phase, it goes through a mini planning, analysis, design, and implementation. The programs that need modification are identified and programmers change or repair those programs. After several years of use, the system usually becomes obsolete. At this point, a major revision of the application is done. Thus the cycle repeats itself.

Key Terms

applications
: An information system or collection of programs that handles a major task.

implementation
: The phase of a Systems Development Life Cycle where the programmers would be assigned to write specific programs.

life cycle
: Systems Development Life Cycle: Planning – Analysis – Design – Implementation – Maintenance

system analyst
: Computer professional in charge of creating applications.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Systems development life cycle ↵

This page titled 11.1: Systems Development Life Cycle is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

11.2: Software Testing

Overview

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:[1]

- meets the requirements that guided its design and development
- responds correctly to all kinds of inputs
- performs its functions within an acceptable time
- is sufficiently usable
- can be installed and run in its intended environments
- achieves the general result its stakeholders desire

Discussion

Test data consists of the user providing some input values and predicting the outputs. This can be quite easy for a simple program and the test data can be used twice.

1. to check the model to see if it produces the correct results (**model checking**)
2. to check the coded program to see if it produces the correct results (**code checking**)

Test data is developed by using the algorithm of the program. This algorithm is usually documented during the program design with either flowcharting or pseudocode. Here is the pseudocode in outline form describing the inputs, processing, and outputs for a program used to calculate gross pay for hourly work.

Pseudocode using an IPO Outline for Calculating Gross Pay

```
Input
    display a message asking user for their hours worked
    get the hours from the keyboard
    display a message asking user for their pay rate
    get the rate from the keyboard

Processing
    calculate the gross pay by:
        multiplying the hours worked by the hourly rate

Output
    display the gross pay on the monitor
    pause so the user can see the answer
```

Creating Test Data and Model Checking

Test data is used to verify that the inputs, processing, and outputs are working correctly. As test data is initially developed it can verify that the documented algorithm (pseudocode in the example we are doing) is correct. It helps us understand and even visualize the inputs, processing, and outputs of the program.

Inputs: I worked 37.5 hours this week and my hourly rate is $15.50 per hour. We should verify that the pseudocode is prompting the user for this data.

Processing: Using my solar powered handheld calculator, I can calculate the gross pay would be: 37.5 * 15.50 or $581.25. We should verify that the pseudocode is performing the correct calculations.

Output: Only the significant information (total gross pay) is displayed for the user to see. We should verify that the appropriate information is being displayed.

Testing the Coded Program – Code Checking

The test data can be developed and used to test the algorithm that is documented (in our case our pseudocode) during the program design phase. Once the program is code with compiler and linker errors resolved, the programmer gets to play user and should test the program using the test data developed. When you run your program, how will you know that it is working properly? Did you properly plan your logic to accomplish your purpose? Even if your plan was correct, did it get converted correctly (coded) into the chosen programming language? The answer (or solution) to all of these questions is our test data.

By developing test data we are predicting what the results should be, thus we can verify that our program is working properly. When we run the program we would enter the input values used in our test data. Hopefully, the program will output the predicted values. If not then our problem could be any of the following:

1. The plan (IPO outline or another item) could be wrong
2. The conversion of the plan to code might be wrong
3. The test data results were calculated wrong

Resolving problems of this nature can be the most difficult problems a programmer encounters. You must review each of the above to determine where the error is lies. Fix the error and re-test your program.

Key Terms

code checking
: Using test data to check the coded program in a specific language (like C++).

model checking
: Using test data to check the design model (usually done in pseudocode).

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Software testing ↵

This page titled 11.2: Software Testing is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

11.3: Version Control

Overview

Version control, also known as revision control or source control, is the management of changes to documents, computer programs, large websites, and other collections of information. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.[1]

Version control systems (VCS) most commonly run as stand-alone applications, but may also be embedded in various types of software, including integrated development environments (IDEs).

Discussion

Version control implements a systematic approach to recording and managing changes in files. At its simplest, version control involves taking 'snapshots' of your file at different stages. This snapshot records information about when the snapshot was made, and also about what changes occurred between different snapshots. This allows you to 'rewind' your file to an older version. From this basic aim of version control, a range of other possibilities is made available.[2]

Version control allows you to:[3]

- Track developments and changes in your files
- Record the changes you made to your file in a way that you will be able to understand later
- Experiment with different versions of a file while maintaining the original version
- 'Merge' two versions of a file and manage conflicts between versions
- Revert changes, moving 'backward' through your history to previous versions of your file

Version control is particularly useful for facilitating collaboration. One of the original motivations behind version control systems was to allow different people to work on large projects together. Using version control to collaborate allows for a greater deal of flexibility and control than many other solutions. As an example, it would be possible for two people to work on a file at the same time and then merge these together. If there were 'conflicts' between the two versions, the version control system would allow you to see these conflicts and make an active decision about how to 'merge' these different versions into a new 'third' document. With this approach you would also retain a 'history' of the previous version should you wish to revert back to one of these later on.[4]

Popular version control systems include:[5]

- Git
- Helix VCS
- Microsoft Team Foundation Server
- Subversion

The remainder of this lesson focuses on using the Git version control system.

Git

Git is a version control system for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for source code management in software development, but it can be used to keep track of changes in any set of files. Git was created by Linus Torvalds in 2005 for development of the Linux kernel and is free and open source software.[6]

Free public git repositories are available from:

- Bitbucket

- GitHub

Cloning an existing repository requires only a URL to the repository and the following git command:

- `git clone <url>`

Once cloned, repositories are synchronized by pushing and pulling changes. If the original source repository has been modified, the following git command is used to pull those changes to the local repository:

- `git pull`

Local changes must be added and committed, and then pushed to the remote repository. *Note the period (dot) at the end of the first command.*

- `git add .`
- `git commit -m "reason for commit"`
- `git push`

If there are conflicts between the local and remote repositories, the changes should be merged and then pushed. If necessary, local changes may be forced upon the remote server using:

- `git push --force`

Key Terms

branch
: A separate working copy of files under version control which may be developed independently from the origin.

clone
: Create a new repository containing the revisions from another repository.

commit
: To write or merge the changes made in the working copy back to the repository.

merge
: An operation in which two sets of changes are applied to a file or set of files.

push
: Copy revisions from the current repository to a remote repository.

pull
: Copy revisions from a remote repository to the current repository.

version control
: The management of changes to documents, computer programs, large websites, and other collections of information.

References

- Programming Historian: An Introduction to Version Control Using GitHub Desktop

---

1. Wikipedia: Version control ↵
2. Programming Historian: An Introduction to Version Control Using GitHub Desktop ↵
3. Programming Historian: An Introduction to Version Control Using GitHub Desktop ↵
4. Programming Historian: An Introduction to Version Control Using GitHub Desktop ↵
5. G2Crowd: Best Version Control Systems ↵
6. Wikipedia: Git ↵

This page titled 11.3: Version Control is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

11.4: Programming Style

Overview

Programming style is a set of rules or guidelines used when writing the source code for a computer program. Following a particular programming style will help programmers read and understand source code conforming to the style, and help to avoid introducing errors.[1]

Discussion

Within the programming industry there is a desire to make software programs easy to maintain. The desire centers on money. Simply put, it costs less money to maintain a well written program. One important aspect of program maintenance is making source code listings clear and as easy to read as possible. To that end we will consider the following:

1. Documentation
2. Vertical Alignment
3. Comments
4. Indentation
5. Meaningful Identifier Names Consistently Typed
6. Appropriate use of Typedef

The above items are not needed in order for the source code to compile. Technically the compiler does not read the source code the way humans read the source code. But that is exactly the point; the desire is to make the source code easier for humans to read. You should not be confused between what is possible (technically will run) and what is okay (acceptable good programming practice that leads to readable code).

For each of these items, check style guides for your selected programming language to determine standards and best practices. The following are general guidelines to consider.

Documentation

Documentation is usually placed at the top of the program using several comment lines. The amount of information would vary based on the requirements or standards of the company who is paying its employees or independent contractors to write the code.

Vertical Alignment

You see this within the documentation area. All of the items are aligned up within the same column. This vertical alignment occurs again when variables are defined. When declaring variables or constants many textbooks put several items on one line; like this:

```
01 float  length, width, height;
```

However common this is in textbooks, it would generally not be acceptable to standards used in most companies. You should declare each item on its own line; like this:

```
01 float  length;
02 float  width;
03 float  height;
```

This method of using one item per line is more readable by humans. It is quicker to find an identifier name because you can read the list vertically faster than searching horizontally. Some programmers list them in alphabetic order.

The lines of code inside functions are also aligned vertically and typically indented two or four spaces from the left. The indentation helps set the block off visually.

Comments

Experts have varying viewpoints on whether, and when, comments are appropriate in source code. Some assert that source code should be written with few comments, on the basis that the source code should be self-explanatory or self-documenting. Others suggest code should be extensively commented, with over 50% of the non-whitespace characters in source code being contained within comments).[2]

In between these views is the assertion that comments are neither beneficial nor harmful by themselves, and what matters is that they are correct and kept in sync with the source code, and omitted if they are superfluous, excessive, difficult to maintain or otherwise unhelpful.[3]

Indentation

For languages that use curly braces, there are two common indentation styles:

```
01 function(parameters) {
02     // code
03 }
```

```
01 function(parameters)
02 {
03     // code
04 }
```

In either case, it is important to maintain vertical alignment between the start of the code block and the closing curly brace.

The number of spaces used for indenting blocks of code is typically two or four spaces. Care should be taken to ensure that the IDE or code editor inserts spaces rather than tab characters for indents.

Meaningful Identifier Names Consistently Typed

As the name implies “identifier names” should clearly identify who (or what) you are talking about. Calling your spouse “Snooky” may be meaningful to only you. Others might need to see her full name (Jane Mary Smith) to appropriately identify who you are talking about. The same concept in programming is true. Variables, constants, functions, and other identifiers should use meaningful names. Additionally, those names should be typed consistently in terms of upper and lower case as they are used in the program. Don’t define a variable as: Pig and then type it later on in your program as: pig.

A good rule of thumb for identifiers in procedural programs (as opposed to object-oriented programs) is to use verb-noun combinations for function identifiers and use noun or adjective-noun combinations for constant and variable identifiers. If a function name requires two verbs or two nouns to fully describe the function, it should probably be split into separate functions.

Key Terms

braces
: Used to identify a block of code in languages such as C++, C#, Java, and JavaScript.

consistent
: A rule that says to type identifier names in upper and lower case consistently throughout your source code.

comments
: Information inserted into a source code file for documentation of the program.

documentation
: A method of preserving information useful to others in understanding an information system or part thereof.

indention
: A method used to make sections of source code more visible.

meaningful
: A rule that says identifier names must be easily understood by another reading the source code.

vertical alignment
: A method of listing items vertically so that they are easier to read quickly.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Programming style ↵
2. Wikipedia: Comment (computer programming) ↵
3. Wikipedia: Comment (computer programming) ↵

This page titled 11.4: Programming Style is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

12: VARIABLE SCOPE

12.1: Scope

This page titled 12: Variable Scope is shared under a license and was authored, remixed, and/or curated by LibreTexts.

12.1: Scope

Overview

The **scope** of an identifier name binding – an association of a name to an entity, such as a variable – is the region of a computer program where the binding is valid: where the name can be used to refer to the entity. Such a region is referred to as a scope block. In other parts of the program, the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound).[1]

Discussion

Scope is the area of the program where an item (be it variable, constant, function, etc.) that has an identifier name is recognized. In our discussion, we will use a variable and the place within a program where the variable is defined determines its scope.

Global scope (and by extension global data storage) occurs when a variable is defined "outside of a function". When compiling the program it creates the storage area for the variable within the program's **data area as part of the object code**. The object code has a machine code piece, a data area, and linker resolution instructions. Because the variable has global scope it is available to all of the functions within your source code. It can even be made available to functions in other object modules that will be linked to your code; however, we will forgo that explanation now. A key wording change should be learned at this point. Although the variable has global scope, technically it is available only from **the point of definition to the end of the program source code**. That is why most variables with global scope are placed near the top of the source code before any functions. This way they are available to all of the functions.

Local scope (and by extension local data storage) occurs when a variable is defined "inside of a function". When compiling, the compiler creates machine instructions that will direct the creation of storage locations on an area known as the **stack which is part of the computer's memory**. These memory locations exist until the function completes its task and returns to its calling function. In assembly language, we talk about items being pushed onto the stack and popped off the stack when the function terminates. Thus, the stack is a reusable area of memory being used by all functions and released as functions terminate. Although the variable has local scope, technically it is available only from **the point of definition to the end of the function**. The parameter passing of data items into a function establishes them as local variables. Additionally, any other variables or constants needed by the function usually occur near the top of the function definition so that they are available during the entire execution of the function's code.

Scope is an important concept for modularization. Program control functions may use global scope for variables and constants placing them near the top of the program before any functions. Specific task functions use only local scope variables by passing data as needed into the function with parameter passing and creating local variables and constants as needed. Any information that needs to be communicated back to the calling function is again done via parameter passing. This **closed communications model** that passes all data into and out of a function creates an important predecessor concept for **encapsulation** which is used in object-oriented programming.

Key Terms

data area
: A part of an object code file used for storage of data.

global scope
: Data storage defined outside of a function.

local scope
: Data storage defined inside of a function.

scope
: The area of a source code file where an identifier name is recognized.

stack
: A part of the computer's memory used for storage of data.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Scope (computer science) ↵

This page titled 12.1: Scope is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

13: FOR LOOPS

13.1: Iteration Control Structures
13.2: For Loop
13.3: Definite loops using for
13.4: Counting and Summing Loops

This page titled 13: For Loops is shared under a license and was authored, remixed, and/or curated by LibreTexts.

13.1: Iteration Control Structures

Overview

In **iteration control structures**, a statement or block is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as `while`, `repeat`, `for`, or `do..until`.[1]

Discussion

The basic attribute of an iteration control structure is to be able to repeat some lines of code. The visual display of iteration creates a circular loop pattern when flowcharted, thus the word “loop” is associated with iteration control structures. Iteration can be accomplished with test before loops, test after loops, and counting loops. A question using Boolean concepts usually controls how often the loop will execute.

Iteration (Repetition) Control Structures

pseudocode: While

```
01 count assigned zero
02 While count < 5
03     Display "I love computers!"
04     Increment count
05 End
```

pseudocode: Do While

```
01 count assigned five
02 Do
03     Display "Blast off is soon!"
04     Decrement count
05 While count > zero
```

pseudocode: Repeat Until

```
01 count assigned five
02 Repeat
03     Display "Blast off is soon!"
04     Decrement count
05 Until count < one
```

pseudocode: For

```
01 For x starts at 0, x < 5, increment x
02     Display "Are we having fun?"
03 End
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Structured programming ↵

This page titled 13.1: Iteration Control Structures is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

13.2: For Loop

Overview

A **for loop** is a control flow statement for specifying iteration, which allows code to be executed repeatedly. A for loop has two parts: a header specifying the iteration, and a body which is executed once per iteration. The header often declares an explicit loop counter or loop variable, which allows the body to know which iteration is being executed. For loops are typically used when the number of iterations is known before entering the loop. For loops can be thought of as shorthands for while loops which increment and test a loop variable.[1]

Discussion

Introduction to Test Before Loops

There are two commonly used test before loops in the iteration (or repetition) category of control structures. They are: while and for. This module covers the: for.

Understanding Iteration in General – for

In many programming languages, the for loop is used exclusively for counting; that is to repeat a loop action as it either counts up or counts down. There is a starting value and a stopping value. The question that controls the loop is a test expression that compares the starting value to the stopping value. This expression is a Boolean expression and is usually using the relational operators of either less than (for counting up) or greater than (for counting down). The term loop comes from the circular looping motion that occurs when using flowcharting. The basic form of the for loop (counting up) is as follows:

```
for
    initialization of the starting value
    starting value is less than the stopping value
    some statements or action
    some statements or action
    some statements or action
    increment the starting value
```

It might be best to understand the for loop by understanding a while loop acting like a counting loop. Let's consider;

```
initialization of the starting value
while the starting value is less than the stopping value
    some statements or action
    some statements or action
    some statements or action
    increment the starting value
```

Within the for control structure, there are four attributes to a properly working loop. They are:

- Initializing the flag – done once
- Test expression
- Action or actions
- Update of the flag

The initialization of the flag is not technically part of the while control structure, but it is usually part of the for control structure. The English phrasing is, "For x is 1; x less than 3; do the following actions; increment x; loop back to the test expression". This is

doing the action on the true. When the test expression is false, you stop the loop and go on with the next item in the program. Notice, because this is a test before loop the action **might not happen**. It is called a test before loop because the test comes before the action. It is also sometimes called a pre-test loop, meaning the test is pre (or Latin for before) the action and update.

An Example

```
01 For counter = 0, counter < 5, counter += 1
02     Output "I love ice cream!"
```

The four attributes of a test before loop (remember the for loop is one example of a test before loop) are present.

- The initialization of the flag to a value of 0.
- The test is the less than relational comparison of the value in the flag variable to the constant value of 5.
- The action part consists of the 1 line of output.
- The update of the flag is done with the increment operator.

Using indentation with the alignment of the loop actions is the normal industry practice.

Infinite Loops

At this point, it is worth mentioning that good programming always provides for a method to ensure that the loop question will eventually be false so that the loop will stop executing and the program continues with the next line of code. However, if this does not happen, then the program is in an infinite loop. Infinite loops are a bad thing. Consider the following code:

```
01 For counter = 0, counter < 5
02     Output "I love ice cream!"
```

The programmer assigned a value to the flag during the initialization step which is correct. However, they forgot to update the flag (the update step is missing). Every time the test expression is asked it will always be true. Thus, an infinite loop because the programmer did not provide a way to exit the loop (he forgot to update the flag).

Key Terms

for
: A test before iteration control structure typically used for counting.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: For loop ↩

This page titled 13.2: For Loop is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

13.3: Definite loops using for

Sometimes we want to loop through a *set* of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a *definite* loop using a `for` statement. We call the `while` statement an *indefinite* loop because it simply loops until some condition becomes `False`, whereas the `for` loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax of a `for` loop is similar to the `while` loop in that there is a `for` statement and a loop body:

Code 5.6.1 (Python)

```
%%python3

friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

In Python terms, the variable `friends` is a list[1] of three strings and the `for` loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Translating this `for` loop to English is not as direct as the `while`, but if you think of friends as a *set*, it goes like this: "Run the statements in the body of the for loop once for each friend *in* the set named friends."

Looking at the `for` loop, *for* and *in* are reserved Python keywords, and `friend` and `friends` are variables.

```
for friend in friends:
    print('Happy New Year:', friend)
```

In particular, `friend` is the *iteration variable* for the for loop. The variable `friend` changes for each iteration of the loop and controls when the `for` loop completes. The *iteration variable* steps successively through the three strings stored in the `friends` variable.

This page titled 13.3: Definite loops using for is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.6: Definite loops using for by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

13.4: Counting and Summing Loops

For example, to count the number of items in a list, we would write the following `for` loop:

Code 5.8.1 (Python)

```
%%python3

count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)

```

We set the variable `count` to zero before the loop starts, then we write a `for` loop to run through the list of numbers. Our *iteration* variable is named `itervar` and while we do not use `itervar` in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add 1 to the current value of `count` for each of the values in the list. While the loop is executing, the value of `count` is the number of values we have seen "so far".

Once the loop completes, the value of `count` is the total number of items. The total number "falls in our lap" at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Another similar loop that computes the total of a set of numbers is as follows:

Code 5.8.1 (Python)

```
%%python3

total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)

```

In this loop we *do* use the *iteration variable*. Instead of simply adding one to the `count` as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable `total`, it contains the "running total of the values so far". So before the loop starts `total` is zero because we have not yet seen any values, during the loop `total` is the running total, and at the end of the loop `total` is the overall total of all the values in the list.

As the loop executes, `total` accumulates the sum of the elements; a variable used this way is sometimes called an *accumulator*.

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions `len()` and `sum()` that compute the number of items in a list and the total of the items in the list respectively.

This page titled 13.4: Counting and Summing Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.8: Counting and Summing Loops by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

14: MORE MATH AND RANDOM NUMBERS

14.1: Integer Division and Modulus
14.2: Modulus Operator
14.3: Standard Libraries
14.4: Random Numbers
14.5: Math Functions

This page titled 14: More Math and Random Numbers is shared under a license and was authored, remixed, and/or curated by LibreTexts.

14.1: Integer Division and Modulus

Overview

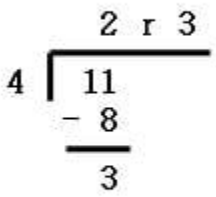
In **integer division** and **modulus**, the dividend is divided by the divisor into an integer quotient and a remainder. The integer quotient operation is referred to as integer division, and the integer remainder operation is the modulus.[1][2]

Discussion

By the time we reach adulthood, we normally think of division as resulting in an answer that might have a fractional part (a floating-point data type). This type of division is known as floating-point division. However, division, when both operands are of the integer data type, may act differently, depending on the programming language, and is called: **integer division**. Consider:

```
11 / 4
```

Because both operands are of the integer data type the evaluation of the expression (or answer) would be 2 with no fractional part (it gets thrown away). Again, this type of division is called **integer division** and it is what you learned in grade school the first time you learned about division.



Integer division as learned in grade school.

In the real world of data manipulation there are some things that are always handled in whole units or numbers (integer data type). **Fractions just don't exist.** To illustrate our example: I have 11 dollar coins to distribute equally to my 4 children. How many do they each get? The answer is 2, with me still having 3 left over (or with 3 still remaining in my hand). The answer is not 2 ¾ each or 2.75 for each child. The dollar coins are not divisible into fractional pieces. Don't try thinking out of the box and pretend you're a pirate. Using an axe and chopping the 3 remaining coins into pieces of eight. Then, giving each child 2 coins and 6 pieces of eight or 2 6/8 or 2 ¾ or 2.75. If you do think this way, I will change my example to cans of tomato soup. I dare you to try and chop up three cans of soup and give each kid ¾ of a can. Better yet, living things like puppy dogs. After you divide them up with an axe, most children will not want the ¾ of a dog.

What is **modulus**? It's the other part of the answer for integer division. It's the remainder. Remember in grade school you would say, "Eleven divided by four is two remainder three." In many programming languages, the symbol for the modulus operator is the percent sign (%).

```
11 % 4
```

Thus, the answer or value of this expression is 3 or the remainder part of integer division.

Many compilers require that you have integer operands on both sides of the modulus operator or you will get a compiler error. In other words, it does not make sense to use the modulus operator with floating-point operands.

Don't let the following items confuse you.

`6 / 24 which is different from 6 % 24`

How many times can you divide 24 into 6? Six divided by 24 is zero. This is different from: What is the remainder of 6 divided by 24? Six, the remainder part is given by modulus.

Evaluate the following division expressions:

1. 14 / 4
2. 5 / 13
3. 7 / 2.0

Evaluate the following modulus expressions:

1. 14 % 4
2. 5 % 13
3. 7 % 2.0

Key Terms

integer division
: Division with no fractional parts.

modulus
: The remainder part of integer division.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Division (mathematics) ↵
2. Wikipedia: Modulo operation ↵

This page titled 14.1: Integer Division and Modulus is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

14.2: Modulus Operator

The *modulus operator* works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

Code 2.8.1 (Python)

```
quotient = 7 // 3
print(quotient)

remainder = 7 % 3
print(remainder)

```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then `x` is divisible by `y`.

You can also extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly, `x % 100` yields the last two digits.

This page titled 14.2: Modulus Operator is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.8: Modulus Operator by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

14.3: Standard Libraries

Overview

Many common or standard functions, whose definitions have already been written, are ready to be used in any program. They are organized into a group of functions (think of them as several books) and are collectively called a **standard library**. There are many functions organized into several libraries For example, within most programming languages many math functions exist and have been coded (and placed into libraries). These functions were written by programmers and tested to ensure that they work properly. In most cases, the functions were reviewed by several people to double and triple check to ensure that they did what was expected. We have the advantage of using these functions with confidence that they will work properly in our programs, thus saving us time and money.

Discussion

The main program must establish the existence of functions used in that program. Depending on the programming language, there is a formal way to:

1. define a function
2. declare a function (a prototype is a declaration to a compiler)
3. call a function

When we create functions in our program, we usually see them in the following order in our source code listing:

1. declare the function (prototype)
2. call the function
3. define the function

When we use functions created by others that have been organized into a library, we include a header file in our program which contains the prototypes for the functions. Just like functions that we create, we see them in the following order in our source code listing:

1. declaring the function (prototype provided in the include file)
2. call the function (with parameter passing of values)
3. define the function (it is either defined in the header file or the linker program provides the actual object code from a Standard Library object area)

In most cases, the user can look at the prototype and understand exactly how the communications (parameter passing) into and out of the function will occur when the function is called. Let's look at the math example of absolute value.

Language	Example
C++	`#include <cmath>` `std::abs(number);`
C#	`Math.Abs(number);`
Java	`Java.lang.Math.abs(number)`
JavaScript	`Math.abs(number);`
Python	`abs(number)`
Swift	`abs(number)`

Not wanting to have a long function name the designers named it: **abs** instead of "absolute". This might seem to violate the identifier naming rule of using meaningful names, however, when identifier names are established for standard libraries they are often shortened to a name that is easily understood by all who would be using them. If I had two integer variables named apple

and banana; and I wanted to store the absolute value of banana into apple; then a line of code to call this function would be:

```
apple = abs(banana);
```

Let's say it in English, pass the function absolute the value stored in variable banana and assign the returning value from the function to the variable apple. Thus, if you know the prototype you can usually properly call the function and use its returning value (if it has one) without ever seeing the definition of the code (i.e. the source code that tells the function how to get the answer; that is written by someone else; and either included in the header file or compiled and placed into an object library; and linked during the linking step of the Integrated Development Environment (IDE).

Key Terms

abs
: A function within a standard library which stands for absolute value.

confidence
: The reliance that Standard Library functions work properly.

standard library
: A set of specific task functions that have been added to the programming language for universal use.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 14.3: Standard Libraries is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

14.4: Random Numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be *deterministic*. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use *algorithms* that generate *pseudorandom* numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

The `random` module provides functions that generate pseudorandom numbers (which I will simply call "random" from here on).

The function `random` returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call `random`, you get the next number in a long series. To see a sample, run this loop:

```
01 import random
02
03 for i in range(10):
04     x = random.random()
05     print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
01 0.11132867921152356
02 0.5950949227890241
03 0.04820265884996877
04 0.841003109276478
05 0.997914947094958
06 0.04842330803368111
07 0.7416295948208405
08 0.510535245390327
09 0.27447040171978143
10 0.028511805472785867
```

Exercise 1: Run the program on your system and see what numbers you get. Run the program more than once and see what numbers you get.

Code 4.4.1 (Python)

```
01 import random
02
03 for i in range(10):
04     x = random.random()
05     print(x)
```

The `random` function is only one of many functions that handle random numbers. The function `randint` takes the parameters `low` and `high`, and returns an integer between `low` and `high` (including both).

```
01 >>> random.randint(5, 10)
02 5
```

```
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

The `random` module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

This page titled 14.4: Random Numbers is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.4: Random Numbers by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

14.5: Math Functions

Python has a `math` module that provides most of the familiar mathematical functions. Before we can use the module, we have to import it:

```
>>> import math
```

This statement creates a *module object* named math. If you print the module object, you get some information about it:

```
>>> print(math)
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called *dot notation*.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called `log` that computes logarithms base `e`.

The second example finds the sine of `radians`. The name of the variable is a hint that `sin` and the other trigonometric functions (`cos`, `tan`, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

The expression `math.pi` gets the variable `pi` from the math module. The value of this variable is an approximation of π, accurate to about 15 digits.

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

This page titled 14.5: Math Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.5: Math Functions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

15: NESTED FOR LOOPS

15.1: Nested For Loops

This page titled 15: Nested For Loops is shared under a license and was authored, remixed, and/or curated by LibreTexts.

15.1: Nested For Loops

Overview

Nested for loops places one for loop inside another for loop. The inner loop is repeated for each iteration of the outer loop.

Discussion

Nested Control Structures

We are going to first introduce the concept of nested control structures. Nesting is a concept that places one item inside of another. Consider:

```
01 if expression
02     true action
03 else
04     false action
```

This is the basic form of the if then else control structure. Now consider:

```
01 if age is less than 18
02     you can't vote
03     if age is less than 16
04         you can't drive
05     else
06         you can drive
07 else
08     you can vote
09     if age is less than 21
10         you can't drink
11     else
12         you can drink
```

As you can see we simply included as part of the “true action” a statement and another if then else control structure. We did the same (nested another if then else) for the “false action”. In our example, we nested if then else control structures. Nesting could have an if then else within a while loop. Thus, the concept of nesting allows the mixing of the different categories of control structures.

Many complex logic problems require using nested control structures. By nesting control structures (or placing one inside another) we can accomplish almost any **complex logic** problem.

An Example – Nested for loops

Here is an example of a 10 by 10 multiplication table:

```
01          1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |
02       ------------------------------------------------------------
03    1 !   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 |
04    2 !   2 |   4 |   6 |   8 |  10 |  12 |  14 |  16 |  18 |  20 |
05    3 !   3 |   6 |   9 |  12 |  15 |  18 |  21 |  24 |  27 |  30 |
```

```
  4 !   4 |   8 |  12 |  16 |  20 |  24 |  28 |  32 |  36 |  40 |
  5 !   5 |  10 |  15 |  20 |  25 |  30 |  35 |  40 |  45 |  50 |
  6 !   6 |  12 |  18 |  24 |  30 |  36 |  42 |  48 |  54 |  60 |
  7 !   7 |  14 |  21 |  28 |  35 |  42 |  49 |  56 |  63 |  70 |
  8 !   8 |  16 |  24 |  32 |  40 |  48 |  56 |  64 |  72 |  80 |
  9 !   9 |  18 |  27 |  36 |  45 |  54 |  63 |  72 |  81 |  90 |
 10 !  10 |  20 |  30 |  40 |  50 |  60 |  70 |  80 |  90 | 100 |
```

We might also see that the answers could be designed as a collection of cells (each cell being exactly six spaces wide). The pseudocode to produce part of the table is:

```
For row = 1, row <= 3, row += 1
    For column = 1, column <= 3, column += 1
        Output row * column
        Output "\t"
    Output "\n"
```

Key Terms

complex logic
: Often solved with nested control structures.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 15.1: Nested For Loops is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

16: DATA TYPES AND USER INPUT

16.1: Data Types
16.2: Integer Data Type
16.3: Floating-Point Data Type
16.4: String Data Type
16.5: Boolean Data Type
16.6: Values and Types
16.7: Data Type Conversions
16.8: Input and Output
16.9: Asking the user for input

This page titled 16: Data Types and User Input is shared under a license and was authored, remixed, and/or curated by LibreTexts.

16.1: Data Types

Overview

A **data type** is a classification of data which tells the compiler or interpreter how the programmer intends to use the data. Most programming languages support various types of data, including integer, real, character or string, and Boolean.[1]

Discussion

Our interactions (inputs and outputs) with a program are treated in many languages as a stream of bytes. These bytes represent data that can be interpreted as representing values that we understand. Additionally, within a program, we process this data in various ways such as adding them up or sorting them. This data comes in different forms. Examples include:

- your name – a string of characters
- your age – usually an integer
- the amount of money in your pocket – usually a value measured in dollars and cents (something with a fractional part)

A major part of understanding how to design and code programs is centered in understanding the types of data that we want to manipulate and how to manipulate that data.

Common data types include:

Data Type	Represents	Examples
integer	whole numbers	`-5`, `0`, `123`
floating point (real)	fractional numbers	`-87.5`, `0.0`, `3.14159`
string	A sequence of characters	`"Hello world!"`
Boolean	logical true or false	`true`, `false`
nothing	no data	`null`

The common data types usually exist in most programming languages and act or behave similarly from language to language. Additional complex and/or composite data types may exist and vary from language to language.

Pseudocode

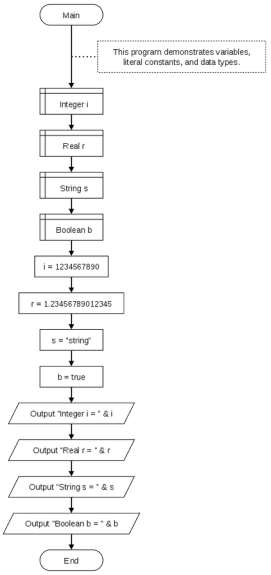
```
01 Function Main
02     ... This program demonstrates variables, literal constants, and data types.
03
04     Declare Integer i
05     Declare Real r
06     Declare String s
07     Declare Boolean b
08
09     Assign i = 1234567890
10     Assign r = 1.23456789012345
11     Assign s = "string"
12     Assign b = true
13
14     Output "Integer i = " & i
15     Output "Real r = " & r
```

```
    Output "String s = " & s
    Output "Boolean b = " & b
End
```

Output

```
Integer i = 1234567890
Real r = 1.23456789012345
String s = string
Boolean b = true
```

Flowchart



Key Terms

Boolean
: A data type representing logical true or false.

data type
: Defines a set of values and a set of operations that can be applied on those values.

floating point
: A data type representing numbers with fractional parts.

integer
: A data type representing whole numbers.

string
: A data type representing a sequence of characters.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Flowgorithm – Flowchart Programming Language

1. Wikipedia: Data type ↵

This page titled 16.1: Data Types is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

16.2: Integer Data Type

Overview

An **integer data type** represents some range of mathematical integers. Integral data types may be of different sizes and may or may not be allowed to contain negative values. Integers are commonly represented in a computer as a group of binary digits (bits). The size of the grouping varies so the set of integer sizes available varies between different types of computers and different programming languages.[1]

Discussion

The integer data type basically represents whole numbers (no fractional parts). The integer values jump from one value to another. There is nothing between 6 and 7. It could be asked why not make all your numbers floating point which allow for fractional parts. The reason is threefold. First, some things in the real world are not fractional. A dog, even with only 3 legs, is still one (1) dog not ¾ of a dog. Second, the integer data type is often used to control program flow by counting, thus the need for a data type that jumps from one value to another. Third, integer processing is significantly faster within the CPU than is floating point processing.

The integer data type has similar attributes and acts or behaves similarly in all programming languages that support it.

Language	Reserved Word	Size	Range
C++	`short`	16 bits / 2 bytes	-32,768 to32,767
C++	`int`	varies	depends on compiler
C++	`long`	32 bits / 4 bytes	-2,147,483,648 to 2, 147,483,647
C++	`long long`	64 bits / 8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
C#	`short`	16 bits / 2 bytes	-32,768 to32,767
C#	`int`	32 bits / 4 bytes	-2,147,483,648 to 2, 147,483,647
C#	`long`	64 bits / 8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Java	`short`	16 bits / 2 bytes	-32,768 to32,767
Java	`int`	32 bits / 4 bytes	-2,147,483,648 to 2, 147,483,647
Java	`long`	64 bits / 8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
JavaScript	N/A		
Python	`int()`		no limit
Swift	`Int`	varies	depends on platform
Swift	`Int32`	32 bits / 4 bytes	-2,147,483,648 to 2, 147,483,647
Swift	`Int64`	64 bits / 8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

For C++ and Swift the size of a default integer varies with the compiler being used and the computer. This effect is known as being **machine dependent**. These variations of the integer data type are an annoyance for a beginning programmer. For a beginning programmer, it is more important to understand the general attributes of the integer data type that apply to most programming languages.

JavaScript does not support an integer data type, but the `Math.round()` function may be used to return the value of a number rounded to the nearest integer.[2]

Python 3 integers are not limited in size, however, `sys.maxsize` may be used to determine the maximum practical size of a list

or string index.[3]

Key Terms

machine dependent
: An attribute of a programming language that changes depending on the computer's CPU.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Integer (computer science) ↵
2. Mozilla: Math.round() ↵
3. Python.org: Integers ↵

This page titled 16.2: Integer Data Type is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

16.3: Floating-Point Data Type

Overview

A **floating-point data type** uses a formulaic representation of real numbers as an approximation so as to support a trade-off between range and precision. For this reason, floating-point computation is often found in systems which include very small and very large real numbers, which require fast processing times. A number is, in general, represented approximately to a fixed number of significant digits and scaled using an exponent in some fixed base.[1]

Discussion

The floating-point data type is a family of data types that act alike and differ only in the size of their domains (the allowable values). The floating-point family of data types represents number values with fractional parts. They are technically stored as two integer values: a **mantissa** and an **exponent**. The floating-point family has the same attributes and acts or behaves similarly in all programming languages. They can always store negative or positive values thus they always are signed; unlike the integer data type that could be unsigned. The **domain** for floating-point data types varies because they could represent very large numbers or very small numbers. Rather than talk about the actual values, we mention the **precision**. The more bytes of storage the larger the mantissa and exponent, thus more precision.

Language	Reserved Word	Size	Precision	Range
C++	`float`	32 bits / 4 bytes	7 decimal digits	±3.40282347E+38
C++	`double`	64 bits / 8 bytes	15 decimal digits	±1.79769313486231570E+308
C#	`float`	32 bits / 4 bytes	7 decimal digits	±3.40282347E+38
C#	`double`	32 bits / 4 bytes	15 decimal digits	±1.79769313486231570E+308
Java	`float`	32 bits / 4 bytes	7 decimal digits	±3.40282347E+38
Java	`double`	32 bits / 4 bytes	15 decimal digits	±1.79769313486231570E+308
JavaScript	`Number`	64 bits / 8 bytes	15 decimal digits	±1.79769313486231570E+308
Python	`float()`	64 bits / 8 bytes	15 decimal digits	±1.79769313486231570E+308
Swift	`Float`	32 bits / 4 bytes	7 decimal digits	±3.40282347E+38
Swift	`Double`	64 bits / 8 bytes	15 decimal digits	±1.79769313486231570E+308

Key Terms

double
: The most often used floating-point family data type used.

mantissa exponent
: The two integer parts of a floating-point value.

precision
: The effect on the domain of floating-point values given a larger or smaller storage area in bytes.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Floating-point arithmetic ↵

This page titled 16.3: Floating-Point Data Type is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

16.4: String Data Type

Overview

A **string data type** is traditionally a sequence of characters, either as a literal constant or as some kind of variable. The latter may allow its elements to be mutated and the length changed, or it may be fixed (after creation). A string is generally considered a data type and is often implemented as an array data structure of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding.[1]

Discussion

Depending on programming language and precise data type used, a variable declared to be a string may either cause storage in memory to be statically allocated for a predetermined maximum length or employ dynamic allocation to allow it to hold a variable number of elements. When a string appears literally in source code, it is known as a string literal or an anonymous string.[2]

The **character** data type represents individual or single characters. Characters comprise a variety of symbols such as the alphabet (both upper and lower case) the numeral digits (0 to 9), punctuation, etc. All computers store character data in a one-byte field as an integer value. Because a byte consists of 8 bits, this one-byte field has 2^8 or 256 possibilities using the positive values of 0 to 255.

C++, C#, and Java differentiate between single characters and strings using single quotes and double quotes, respectively. JavaScript, Python, and Swift do not differentiate between characters and strings and use either single quotes or double quotes to define string literals.

Language	Reserved Word	Example
C++	`char`	`'A'`
C++	`string`	`"Hello world!"`
C#	`char`	`'A'`
C#	`String`	`"Hello world!"`
Java	`char`	`'A'`
Java	`String`	`"Hello world!"`
JavaScript	`String`	`'Hello world!'`, `"Hello world!"`
Python	`str()`	`'Hello world!'`, `"Hello world!"`
Swift	`Character`	`"A"`
Swift	`String`	`"Hello world!"`

Most computing devices use the **ASCII** (stands for American Standard Code for Information Interchange and is pronounced “ask-key”) Character Set which has established values for 0 to 127. For the values of 128 to 255 they usually use the Extended ASCII Character Set. When we hit the capital A on the keyboard, the keyboard sends a byte with the bit pattern equal to an integer 65. When the byte is sent from the memory to the monitor, the monitor converts the integer value of 65 to into the symbol of the capital A to display on the monitor.

For now, we will address only the use of strings and characters as constants. Most modern compilers that are part of an Integrated Development Environment (IDE) will color the source code to help the programmer see different features more readily. Beginning programmers will use string constants to send messages to standard output.

Key Terms

ASCII
: American Standard Code for Information Interchange

character
: A data type representing single text characters like the alphabet, numeral digits, punctuation, etc.

double quote marks
: Used to create string type data within most programming languages.

single quote marks
: Used to create character type data within languages that differentiate between string and character data types.

string
: A series or array of characters as a single piece of data.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: String (computer science) ↵
2. Wikipedia: String (computer science) ↵

This page titled 16.4: String Data Type is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

16.5: Boolean Data Type

Overview

A **Boolean data type** has one of two possible values (usually denoted true and false), intended to represent the two truth values of logic and Boolean algebra. It is named after George Boole, who first defined an algebraic system of logic in the mid 19th century. The Boolean data type is primarily associated with conditional statements, which allow different actions by changing control flow depending on whether a programmer-specified Boolean condition evaluates to true or false.[1]

Discussion

The Boolean data type is also known as the logical data type and represents the concepts of true and false. The name “Boolean” comes from the mathematician George Boole; who in 1854 published: An Investigation of the Laws of Thought. Boolean algebra is the area of mathematics that deals with the logical representation of true and false using the numbers 0 and 1. The importance of the Boolean data type within programming is that it is used to control programming structures (if then else, while loops, etc.) that allow us to implement “choice” into our algorithms.

The Boolean data type has the same attributes and acts or behaves similarly in all programming languages. However, while all languages recognize false as 0, some languages define true as -1 rather than 1. This is the result of storing the Boolean values as an integer and using a one’s complement representation that negates all bits rather than only the rightmost bit. To simplify processing, most programming languages recognize any non-zero value as being true.

Language	Reserved Word	True	False
C++	`bool`	`true`	`false`
C#	`bool` or `Boolean`	`true`	`false`
Java	`bool`	`true`	`false`
JavaScript	`Boolean()`	`true`	`false`
Python	`bool()`	`True`	`False`
Swift	`Bool`	`true`	`false`

Key Terms

Boolean
: A data type representing the concepts of true or false.

one’s complement
: The value obtained by inverting all the bits in the binary representation of a number (swapping 0s for 1s and vice versa).

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Boolean data type ↩

This page titled 16.5: Boolean Data Type is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

16.6: Values and Types

A *value* is one of the basic things a program works with, like a letter or a number. The values we have seen so far are `1`, `2`, and "Hello, World!"

These values belong to different *types*: `2` is an integer, and "Hello, World!" is a *string*, so called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The `print` statement also works for integers. We use the `python` command to start the interpreter.

```
python
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Not surprisingly, strings belong to the type `str` and integers belong to the type `int`. Less obviously, numbers with a decimal point belong to a type called `float`, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>
```

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

They're strings.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in `1,000,000`. This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

Well, that's not what we expected at all! Python interprets `1,000,000` as a comma-separated sequence of integers, which it prints with spaces between.

This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

This page titled 16.6: Values and Types is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.1: Values and Types by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

16.7: Data Type Conversions

Overview

Changing a data type of a value is referred to as “type conversion”. There are two ways to do this:

1. **Implicit** – the change is implied
2. **Explicit** – the change is explicitly done with an operator or function

The value being changed may be:

1. **Promotion** – going from a smaller domain to a larger domain
2. **Demotion** – going from a larger domain to a smaller domain

Discussion

Implicit Type Conversion

Automatic conversion of a value from one data type to another by a programming language, without the programmer specifically doing so, is called implicit type conversion. It happens whenever a binary operator has two operands of different data types. Depending on the operator, one of the operands is going to be converted to the data type of the other. It could be promoted or demoted depending on the operator.

Implicit Promotion

```
01 55 + 1.75
```

In this example, the integer value 55 is converted to a floating-point value (most likely double) of 55.0. It was promoted.

Implicit Demotion

In programming languages that have explicit integer data types (C++, C#, Java), care must be taken to avoid implicit demotion. For example:

```
int money;
money = 23.16;
```

In this example, the variable money is an integer. We are trying to move a floating-point value 23.16 into an integer storage location. This is demotion and the floating-point value usually gets truncated to 23.

Promotion

Promotion is never a problem because the lower data type (smaller range of allowable values) is a subset of the higher data type (larger range of allowable values). Promotion often occurs with three of the standard data types: character, integer, and floating-point. The allowable values (or domains) progress from one type to another. That is, the character data type values are a subset of integer values and integer values are a subset of floating-point values; and within the floating-point values, float values are a subset of double. Even though character data represent the alphabetic letters, numeral digits (0 to 9) and other symbols (a period, $, comma, etc.) their bit pattern also represent integer values from 0 to 255. This progression allows us to promote them up the chain from character to integer to float to double.

Demotion

Demotion represents a potential problem with truncation or unpredictable results often occurring. How do you fit an integer value of 456 into a character value? How do you fit the floating-point value of 45656.453 into an integer value? Most compilers give a warning if it detects demotion happening. A compiler warning does not stop the compilation process. It does warn the programmer to check to see if the demotion is reasonable.

If I calculate the number of cans of soup to buy based on the number of people I am serving (say 8) and the servings per can (say 2.3), I would need 18.4 cans. I might want to demote the 18.4 into an integer. It would truncate the 18.4 into 18 and because the value 18 is within the domain of an integer data type, it should demote with the **truncation** side effect.

If I tried demoting a double that contained the number of stars in the Milky Way galaxy into an integer, I might have a get an **unpredictable result** (assuming the number of stars is larger than allowable values within the integer domain).

Explicit Type Conversion

Most languages have a method for the programmer to change or cast a value from one data type to another; called **explicit type conversion**. Some languages support a cast operator. The cast operator is a unary operator; it only has one operand and the operand is to the right of the operator. The operator is a set of parentheses surrounding the new data type. Other languages have functions that perform explicit type conversion. In each of the following examples, the expression value would be 3.

Language	Floating-Point to Integer Type Conversion Example
C++	`(int) 3.14`
C#	`Convert.ToInt32(3.14)`
Java	`Math.floor(3.14)`
JavaScript	`Math.floor(3.14)`
Python	`int(3.14)`
Swift	`Int(3.14)`

In each of the following examples, the expression value would be 3.14.

Language	String to Floating-Point Type Conversion Example
C++	`#include <string.h>` `std::stod("3.14")`
C#	`Convert.ToDouble("3.14")`
Java	`Double.parseDouble("3.14")`
JavaScript	`parseFloat("3.14")`
Python	`float("3.14")`
Swift	`Double("3.14")`

Key Terms

demotion
: Going from a larger domain to a smaller domain.

explicit
: Changing a value's data type with the cast operator.

implicit
: A value that has its data type changed automatically.

promotion
: Going from a smaller domain to a larger domain.

truncation
: The fractional part of a floating-point data type that is dropped when converted to an integer.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 16.7: Data Type Conversions is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

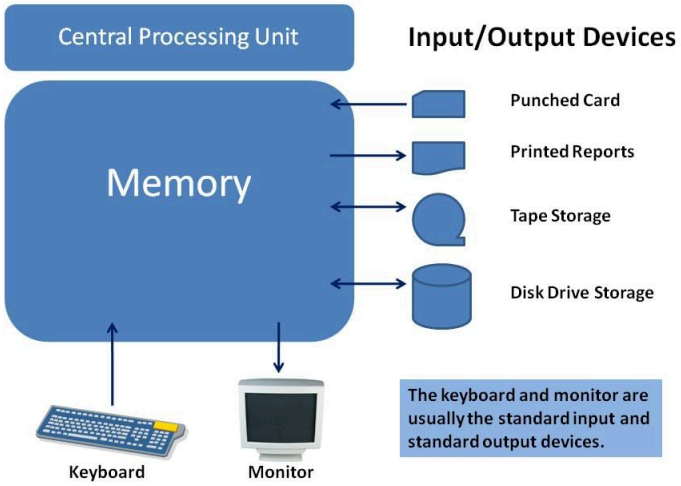
16.8: Input and Output

Overview

Input and output, or I/O is the communication between an information processing system, such as a computer, and the outside world, possibly a human or another information processing system. Inputs are the signals or data received by the system and outputs are the signals or data sent from it.[1]

Discussion

Every task we have the computer do happens inside the central processing unit (CPU) and the associated memory. Once our program is loaded into memory and the operating system directs the CPU to start executing our programming statements the computer looks like this:



CPU – Memory – Input/Output Devices

Our program now loaded into memory has basically two areas:

- Machine instructions – our instructions for what we want done
- Data storage – our variables that we using in our program

Often our program contains instructions to interact with the input/output devices. We need to move data into (read) and/or out of (write) the memory data area. A **device** is a piece of equipment that is electronically connected to the memory so that data can be transferred between the memory and the device. Historically this was done with punched cards and printouts. Tape drives were used for electronic storage. With time we migrated to using disk drives for storage with keyboards and monitors (with monitor output called soft copy) replacing punch cards and printouts (called hard copy).

Most computer operating systems and by extension programming languages have identified the keyboard as the **standard input device** and the monitor as the **standard output device**. Often the keyboard and monitor are treated as the default device when no other specific device is indicated.

Key Terms

device

A piece of equipment that is electronically connected to the memory so that data can be transferred between the

memory and the device.

escape code
A code directing an output device to do something.

extraction
Aka reading or getting data from an input device.

insertion
Aka writing or sending data to an output device.

standard input
The keyboard.

standard output
The monitor.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Input/output ↵

This page titled 16.8: Input and Output is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

16.9: Asking the user for input

Sometimes we would like to take the value for a variable from the user via their keyboard. Python provides a built-in function called `input` that gets input from the keyboard[1]. When this function is called, the program stops and waits for the user to type something. When the user presses `Return` or `Enter`, the program resumes and `input` returns what the user typed as a string.

```
>>> inp = input()
Some silly stuff
>>> print(inp)
Some silly stuff
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to input. You can pass a string to `input` to be displayed to the user before pausing for input:

Code 2.10.1 (Python)

```
name = input('What is your name?"..string.escape("\n").."')
print(name)

```

The sequence `\n` at the end of the prompt represents a *newline*, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to `int` using the `int()` function:

Code 2.10.2 (Python)

```
prompt = 'What...is the airspeed velocity of an unladen
swallow?"..string.escape("\n").."'
speed = input(prompt)

int(speed)

int(speed) + 5

```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

We will see how to handle this kind of error later.

This page titled 16.9: Asking the user for input is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.10: Asking the user for input by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17: STRINGS

17.1: Strings
17.2: A string is a sequence
17.3: String Functions
17.4: String Operations
17.5: Getting the length of a string using len
17.6: Traversal through a string with a loop
17.7: String Slices
17.8: Strings are immutable
17.9: Looping and Counting
17.10: The in operator
17.11: String Comparison
17.12: String Methods
17.13: Parsing strings
17.14: String Formatting
17.15: Format operator

This page titled 17: Strings is shared under a license and was authored, remixed, and/or curated by LibreTexts.

17.1: Strings

Overview

A **string** is traditionally a sequence of characters, either as a literal constant or as some kind of variable. The latter may allow its elements to be mutated and the length changed, or it may be fixed (after creation). A string is generally considered a data type and is often implemented as an array data structure of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding.[1]

Discussion

Recall from String Data Type earlier in the book that, depending on programming language and precise data type used, a variable declared to be a string may either cause storage in memory to be statically allocated for a predetermined maximum length or employ dynamic allocation to allow it to hold a variable number of elements. When a string appears literally in source code, it is known as a string literal or an anonymous string.[2]

Most data is more complex than just one character, integer, etc. Programming languages develop other methods to represent and store data that are more complex. A complex data type of array is the first most students encounter. An array is a sequenced collection of elements of the same data type with a single identifier name. This definition perfectly describes our string data type concept. The simplest array is called a one-dimensional array; also know as a list because we usually list the members or elements vertically. However, strings are viewed as a one-dimensional array that visualize as listed horizontally. Strings are an array of character data.

In the "C" programming language all strings were handled as an array of characters that end in an ASCII null character (the value `0` or the first character in the ASCII character code set). This approach required programmers to manually process string length and manage string storage. Buffer overflows were common. A **buffer overflow**, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.[3]

Most current programming languages implement strings as a data type or class where strings are stored as a length controlled array. String length and storage are handled by the compiler or interpreter, reducing program errors.

Language	Reserved Word
C++	`string`
C#	`String`
Java	`String`
JavaScript	`String`
Python	`str()`
Swift	`String`

Key Terms

array
: A sequenced collection of elements of the same data type with a single identifier name.

buffer overflow
: An anomaly where a program overruns a memory storage location and overwrites adjacent memory locations.

concatenation
: Combining two strings into one string.

string class
: A complex data item that uses object oriented programming.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: String (computer science) ↵
2. Wikipedia: String (computer science) ↵
3. Wikipedia: Buffer overflow ↵

This page titled 17.1: Strings is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

17.2: A string is a sequence

A string is a *sequence* of characters. You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement extracts the character at index position 1 from the `fruit` variable and assigns it to the `letter` variable.

The expression in brackets is called an *index*. The index indicates which character in the sequence you want (hence the name).

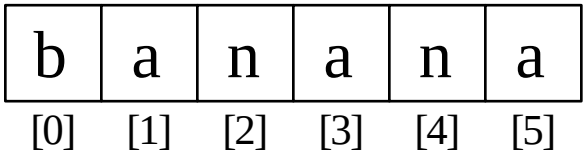
But you might not get what you expect:

```
>>> print(letter)
a
```

For most people, the first letter of "banana" is b, not a. But in Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

So b is the 0th letter ("zero-eth") of "banana", a is the 1th letter ("one-eth"), and n is the 2th ("two-eth") letter.



String Indexes

You can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

This page titled 17.2: A string is a sequence is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.1: A string is a sequence by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.3: String Functions

Overview

String functions are used in computer programming languages to manipulate a string or query information about a string.[1]

Discussion

Most current programming languages include built-in or library functions to process strings. Common examples include case conversion, comparison, concatenation, find, join, length, reverse, split, substring, and trim.

Function	C++	C#	Java
case	`tolower()`, `toupper()`, etc.	`ToLower()`, `ToUpper()`, etc.	`toLowerCase()`, `toUpperCase()`, etc.
comparison	`<`, `>`, `==`, etc.	`<`, `>`, `==`, etc.	`<`, `>`, `==`, etc.
concatenation	`+`, `+=`	`+`, `+=`	`+`, `+=`
find	`find()`	`IndexOf()`	`indexOf()`
join	N/A	`Join()`	`join()`
length	`length()`	`Length`	`length()`
replace	`replace()`	`Replace()`	`replace()`
reverse	`reverse()`	`Reverse()`	N/A
split	`strtok()`	`Split()`	`split()`
substring	`substr()`	`Substring()`	`substring()`
trim	N/A	`Trim()`	`trim()`

Function	JavaScript	Python	Swift
case	`toLowerCase()`, `toUpperCase()`, etc.	`lower()`, `upper()`, etc.	`lowercased()`, `uppercased()`
comparison	`<`, `>`, `==`, etc.	`<`, `>`, `==`, etc.	`<`, `>`, `==`, etc.
concatenation	`+`, `+=`	`+`, `+=`	`+`, `+=`
find	`indexOf()`	`find()`	`firstIndex()`
join	`join()`	`join()`	`joined()`
length	`length`	`len()`	`count`
replace	`replace()`	`replace()`	`replacingOccurrences()`
reverse	N/A	`string[::-1]`	`reversed()`
split	`split()`	`split()`	`split()`
substring	`substring()`	`string[start:end]`	`string[start...end]`
trim	`trim()`	`strip()`	`trimmingCharacters()`

Key Terms

concatenate

Join character strings end-to-end.[2]

trim

Remove leading and trailing spaces from a string.[3]

References

1. Wikipedia: Comparison of programming languages (string functions) ↵
2. Wikipedia: Concatenation ↵
3. Wikipedia: Trimming (computer programming) ↵

This page titled 17.3: String Functions is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

17.4: String Operations

The + operator works with strings, but it is not addition in the mathematical sense. Instead it performs *concatenation*, which means joining the strings by linking them end to end. For example:

Code 2.9.1 (Python)

```
first = 10
second = 15
print(first+second)

first = '100'
second = '150'
print(first + second)


```

The output of this program is `100150`.

This page titled 17.4: String Operations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.9: String Operations by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.5: Getting the length of a string using len

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

This page titled 17.5: Getting the length of a string using len is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.2: Getting the length of a string using len by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.6: Traversal through a string with a loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a *traversal*. One way to write a traversal is with a `while` loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index \< len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

Exercise 1: Write a `while` loop that starts at the last character in the string and works its way backwards to the first character in the string, printing each letter on a separate line, except backwards.

Another way to write a traversal is with a `for` loop:

```
for char in fruit:
    print(char)
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

This page titled 17.6: Traversal through a string with a loop is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.3: Traversal through a string with a loop by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.7: String Slices

A segment of a string is called a *slice*. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

The operator returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an *empty string*, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Exercise 2: Given that `fruit` is a string, what does `fruit[:]` mean?

This page titled 17.7: String Slices is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.4: String Slices by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.8: Strings are immutable

It is tempting to use the operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an *object* is the same thing as a value, but we will refine that definition later. An *item* is one of the values in a sequence.

The reason for the error is that strings are *immutable*, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

This page titled 17.8: Strings are immutable is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.5: Strings are immutable by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.9: Looping and Counting

The following program counts the number of times the letter a appears in a string:

Code 6.6.1 (Python)

```
%%python3

word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)

```

This program demonstrates another pattern of computation called a *counter*. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result: the total number of a's.

Exercise 3:

Encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments.

This page titled 17.9: Looping and Counting is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.6: Looping and Counting by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.10: The in operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

This page titled 17.10: The in operator is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.7: The in operator by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.11: String Comparison

The comparison operators work on strings. To see if two strings are equal:

```
01 if word == 'banana':
02     print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

```
01 if word < 'banana':
02     print('Your word,' + word + ', comes before banana.')
03 elif word > 'banana':
04     print('Your word,' + word + ', comes after banana.')
05 else:
06     print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters, so:

```
01 Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

This page titled 17.11: String Comparison is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.8: String Comparison by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.12: String Methods

Strings are an example of Python *objects*. An object contains both data (the actual string itself) and *methods*, which are effectively functions that are built into the object and are available to any *instance* of the object.

Python has a function called `dir` which lists the methods available for an object. The `type` function shows the type of an object and the `dir` function shows the available methods.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
>>>
```

While the `dir` function lists the methods, and you can use `help` to get some simple documentation on a method, a better source of documentation for string methods would be https://docs.python.org/3.5/library/...string-methods.

Calling a *method* is similar to calling a function (it takes arguments and returns a value) but the syntax is different. We call a method by appending the method name to the variable name using the period as a delimiter.

For example, the method `upper` takes a string and returns a new string with all uppercase letters:

Instead of the function syntax `upper(word)`, it uses the method syntax `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

This form of dot notation specifies the name of the method, `upper`, and the name of the string to apply the method to, `word`. The empty parentheses indicate that this method takes no argument.

A method call is called an *invocation*; in this case, we would say that we are invoking `upper` on the `word`.

For example, there is a string method named `find` that searches for the position of one string within another:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

The `find` method can find substrings as well as characters:

```
>>> word.find('na')
2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

One common task is to remove white space (spaces, tabs, or newlines) from the beginning and end of a string using the `strip` method:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Some methods such as *startswith* return boolean values.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

You will note that `startswith` requires case to match, so sometimes we take a line and map it all to lowercase before we do any checking using the `lower` method.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
>>> line.lower().startswith('h')
True
```

In the last example, the method `lower` is called and then we use `startswith` to see if the resulting lowercase string starts with the letter "h". As long as we are careful with the order, we can make multiple method calls in a single expression.

Exercise 4:

There is a string method called `count` that is similar to the function in the previous exercise. Read the documentation of this method at https://docs.python.org/3.5/library/...string-methods and write an invocation that counts the number of times the letter a occurs in "banana".

This page titled 17.12: String Methods is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.9: String Methods by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.13: Parsing strings

Often, we want to look into a string and find a substring. For example if we were presented a series of lines formatted as follows:

`From stephen.marquard@ uct.ac.za Sat Jan 5 09:14:16 2008`

and we wanted to pull out only the second half of the address (i.e., `uct.ac.za`) from each line, we can do this by using the `find` method and string slicing.

First, we will find the position of the at-sign in the string. Then we will find the position of the first space *after* the at-sign. And then we will use string slicing to extract the portion of the string which we are looking for.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

We use a version of the `find` method which allows us to specify a position in the string where we want `find` to start looking. When we slice, we extract the characters from "one beyond the at-sign through up to *but not including* the space character".

The documentation for the `find` method is available at

https://docs.python.org/3.5/library/...string-methods.

This page titled 17.13: Parsing strings is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.10: Parsing strings by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

17.14: String Formatting

Overview

String formatting uses a process of string interpolation (variable substitution) to evaluate a string literal containing one or more placeholders, yielding a result in which the placeholders are replaced with their corresponding values.[1]

Discussion

Most current programming languages provide one or more string formatting functions that use a template string with placeholders and optional alignment, width, and precision indicators to generate formatted output.

Language	Function	Examples
C++	`snprintf()`	`snprintf(str, sizeof(str), "Hello %s!", name);` `snprintf(str, sizeof(str), "$%.2f", value);`
C#	`Format()`	`String.Format("Hello {0}!", name);` `String.Format("{0:$0.00}", value);`
Java	`format()`	`String.format("Hello %s!", name);` `String.format("$%.2f", value);`
JavaScript	template literal	`` `Hello ${name}`; `` `` `$${value.toFixed(2)}`; ``
Python	`format()`	`"Hello {}!".format(name)` `"${:.2f}".format(value)`
Swift	interpolation `String()`	`"Hello \(name)!"` `String(format:"%.2f", value)`

String interpolation, like string concatenation, may lead to security problems. If user input data is improperly escaped or filtered, the system may be exposed to code injection.[2]

Key Terms

code injection
: The exploitation of a computer bug that is caused by processing invalid data.[3]

formatting
: Modifying the way the output is displayed.

string interpolation
: Evaluating a string literal containing one or more placeholders, yielding a result in which the placeholders are replaced with their corresponding values.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: String interpolation ↵
2. Wikipedia: String interpolation ↵
3. Wikipedia: Code injection ↵

This page titled 17.14: String Formatting is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

17.15: Format operator

The *format operator*, % allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the *format string*, which contains one or more *format sequences* that specify how the second operand is formatted. The result is a string.

For example, the format sequence "%d" means that the second operand should be formatted as an integer (d stands for "decimal"):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string "42", which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple[1]. Each format sequence is matched with an element of the tuple, in order.

The following example uses "%d" to format an integer, "%g" to format a floating-point number (don't ask why), and "%s" to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple must match the number of format sequences in the string. The types of the elements also must match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

The format operator is powerful, but it can be difficult to use. You can read more about it at

https://docs.python.org/3.5/library/...ing-formatting.

This page titled 17.15: Format operator is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.11: Format operator by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

18: WHILE LOOPS

18.1: While Loop
18.2: The while Statement
18.3: Flag Concept
18.4: Infinite Loops
18.5: Loop patterns
18.6: Maximum and Minimum Loops
18.7: Loop Examples
18.8: Python Examples

This page titled 18: While Loops is shared under a license and was authored, remixed, and/or curated by LibreTexts.

18.1: While Loop

Overview

A **while loop** is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.[1]

Discussion

Introduction to Test Before Loops

There are two commonly used test before loops in the iteration (or repetition) category of control structures. They are: while and for. This module covers the: while.

Understanding Iteration in General – while

The concept of iteration is connected to possibly wanting to repeat an action. Like all control structures we ask a question to control the execution of the loop. The term loop comes from the circular looping motion that occurs when using flowcharting. The basic form of the while loop is as follows:

```
01 initialization of the flag
02 while the answer to the question is true then do
03     some statements or action
04     some statements or action
05     some statements or action
06     update the flag
```

In most programming languages the question (called a test expression) is a Boolean expression. The Boolean data type has two values – true and false. Let's rewrite the structure to consider this:

```
01 initialization of the flag
02 while the expression is true then do
03     some statements or action
04     some statements or action
05     some statements or action
06     update the flag
```

Within the while control structure there are four attributes to a properly working loop. They are:

- Initializing the flag
- Test expression
- Action or actions
- Update of the flag

The initialization of the flag is not technically part of the control structure, but a necessary item to occur before the loop is started. The English phrasing is, "While the expression is true, do the following actions". This is looping on the true. When the test expression is false, you stop the loop and go on with the next item in the program. Notice, because this is a test before loop the action **might not happen**. It is called a test before loop because the test comes before the action. It is also sometimes called a pre-test loop, meaning the test is pre (or Latin for before) the action and update.

Human Example of the while Loop

Consider the following one-way conversation from a mother to her child.

Child: The child says nothing, but mother knows the child had Cheerios for breakfast and history tells us that the child most likely spilled some Cheerios on the floor.

Mother says: “While it is true that you see (As long as you can see) a Cheerio on the floor, pick it up and put it in the garbage.”

Note: All of the elements are present to determine the action (or flow) that the child will be doing (in this case repeating). Because the question (can you see a Cheerios) has only two possible answers (true or false) the action will continue while there are Cheerios on the floor. Either the child 1) never picks up a Cheerio because they never spilled any or 2) picks up a Cheerio and keeps picking up Cheerios one at a time while he can see a Cheerio on the floor (that is until they are all picked up).

Infinite Loops

At this point, it is worth mentioning that good programming always provides for a method to ensure that the loop question will eventually be false so that the loop will stop executing and the program continues with the next line of code. However, if this does not happen, then the program is in an infinite loop. Infinite loops are a bad thing. Consider the following code:

Pseudocode infinite loop

```
01 loop_response = 'y'
02 While loop_response == 'y'
03     Output "What is your age? "
04     Input user_age
05     Output "What is your friend's age? "
06     Input friend_age
07     Output "Together your ages add up to: "
08     Output user_age + friend_age
```

The programmer assigned a value to the flag before the loop which is correct. However, they forgot to update the flag. Every time the test expression is asked it will always be true. Thus, an infinite loop because the programmer did not provide a way to exit the loop (he forgot to update the flag). Consider the following code:

```
01 loop_response = 'y';
02 While loop_response = 'y'
03     Output "What is your age? "
04     Input user_age
05     Output "What is your friend's age? "
06     Input friend_age
07     Output "Together your ages add up to: "
08     Output user_age + friend_age
09     Output "Do you want to try again? y or n "
10     Input loop_response
```

No matter what the user replies during the flag update, the test expression does not do a relational comparison but does an assignment. It assigns ‘y’ to the variable and asks if ‘y’ is true? Since all non-zero values are treated as representing true, the answer to the test expression is true. Viola, you have an infinite loop.

Counting Loops

The examples above are for an event controlled loop. The flag updating is an event where someone decides if they want the loop to execute again. Often the initialization sets the flag so that the loop will execute at least once.

Another common usage of the while loop is as a counting loop. Consider:

```
01 counter = 0
02 While counter < 5
03     Output "I love ice cream!"
04     counter += 1
```

The variable counter is said to be controlling the loop. It is set to zero (called initialization) before entering the while loop structure and as long as it is less than 5 (five); the loop action will be executed. But part of the loop action uses the increment operator to increase counter's value by one. After executing the loop five times (once for counter's values of: 0, 1, 2, 3 and 4) the expression will be false and the next line of code in the program will execute. A counting loop is designed to execute the action (which could be more than one statement) a set of given number of times. In our example, the message is displayed five times on the monitor. It is accomplished by making sure all four attributes of the while control structure are present and working properly. The attributes are:

- Initializing the flag
- Test expression
- Action or actions
- Update of the flag

Missing an attribute might cause an infinite loop or give undesired results (does not work properly).

Infinite Loops

Consider:

```
01 counter = 0;
02 while counter < 5
03     Output "I love ice cream!"
```

Missing the flag update usually causes an infinite loop.

Variations on Counting

In the following example, the integer variable age is said to be controlling the loop (that is the flag). We can assume that age has a value provided earlier in the program. Because the while structure is a test before loop; it is possible that the person's age is 0 (zero) and the first time we test the expression it will be false and the action part of the loop would never be executed.

```
01 While 0 < age
02     Output "I love candy!"
03     age -= 1
```

Consider the following variation assuming that age and counter are both integer data type and that age has a value:

```
01 counter = 0;
02 While counter < age
03     Output "I love corn chips!"
04     counter += 1
```

This loop is a counting loop similar to our first counting loop example. The only difference is instead of using a literal constant (in other words 5) in our expression, we used the variable age (and thus the value stored in age) to determine how many times to execute the loop. However, unlike our first counting loop example which will always execute exactly 5 times; it is possible that the person's age is 0 (zero) and the first time we test the expression it will be false and the action part of the loop would never be executed.

Key Terms

counting controlled
: Using a variable to count up or down to control a loop.

event controlled
: Using user input to control a loop.

infinite loop
: A sequence of instructions which loops endlessly, either due to the loop having no terminating condition, having one that can never be met, or one that causes the loop to start over.[2]

initialize item
: An attribute of iteration control structures.

loop attributes
: Items associated with iteration or looping control structures.

might not happen
: Indicating that test before loops might not execute the action.

while
: A test before iteration control structure.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: While loop ↵
2. Wikipedia: Infinite loop ↵

This page titled 18.1: While Loop is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

18.2: The while Statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the `while` statement. Here is a simple program that counts down from five and then says "Blastoff!".

Code 5.2.1 (Python)

```
%%python3

n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')

```

You can almost read the `while` statement as if it were English. It means, "While `n` is greater than `0`, display the value of `n` and then reduce the value of `n` by 1. When you get to `0`, exit the `while` statement and display the word `Blastoff!`"

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `True` or `False`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step
    1.

This type of flow is called a *loop* because the third step loops back around to the top. We call each time we execute the body of the loop an *iteration*. For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. We call the variable that changes each time the loop executes and controls when the loop finishes the *iteration variable*. If there is no iteration variable, the loop will repeat forever, resulting in an *infinite loop*.

This page titled 18.2: The while Statement is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.2: The while Statement by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

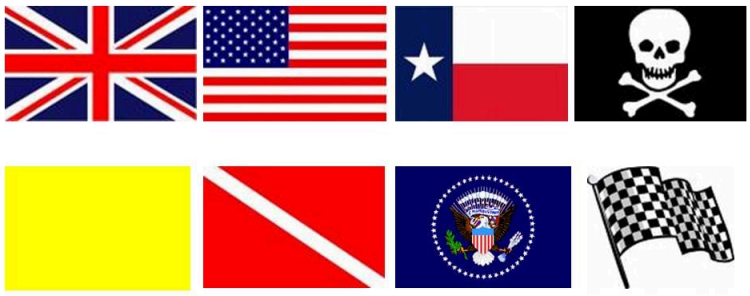
18.3: Flag Concept

Overview

Flags are commonly used to control or to indicate the intermediate state or outcome of particular operations.[1]

Discussion

For centuries flags have been used as a signal to let others know something about the group or individual that is displaying, flying or waving the flag. There are country flags and state flags. Ships at sea flew the flag of their country. Pirates flew the skull and crossbones. A yellow flag was used for quarantine, usually the plague. Even pirates stayed away. Today, some people might recognize the flag used by scuba divers. The Presidents of most countries have a flag. At a race car event, they use the checkered flag to indicate the race is over.



Various Flags

Computer programming uses the concept of a flag in the same way that physical flags are used. A flag is anything that signals some information to the person looking at it.

Computer Implementation

Any variable or constant that holds data can be used as a flag. You can think of the storage location as a flagpole. The value stored within the variable conveys some meaning and you can think of it as being the flag. An example might be a variable named: gender which is of the character data type. The two values commonly stored in the variable are: 'F' and 'M', meaning female and male. Then, somewhere within a program we might look at the variable to make a decision:

flag controlling an if then control structure

```
01 if gender equals 'F'
02     display "Are you pregnant?"
03     get answer from user store in pregnant variable
```

Looking at the flag implies comparing the value in the variable to another value (a constant or the value in another variable) using a relational operator (in our above example: equality).

Control structures are “controlled” by using a **test expression** which is usually a **Boolean expression**. Thus, the flag concept of “looking” at the value in the variable and comparing it to another value is fundamental to understanding how all control structures work.

Two Flags with the Same Meaning

Sometimes we will use an iteration control structure of do while to allow us to decide if we want to do the loop action again. A variable might be named “loop_response” with the user prompted for their answer of ‘y’ for yes or ‘n’ for no. Once the answer is retrieved from the keyboard and stored in our flag variable of “loop_response” the test expression to control the loop might be:

simple flag comparison

```
01 loop_response equals 'y'
```

This is fine but what if the user accidentally has on the caps lock. Then the response of ‘Y’ would not have the control structure loop and perform the action again. The solution lies in looking at the flag twice. Consider:

complex flag comparison

```
01 loop_response equals 'y' or loop_response equals 'Y'
```

We look to see if the flag is either a lower case y or an upper case Y by using a more complex Boolean expression with both relational and logical operators.

Multiple Flags in One Byte

Within assembly language programming and in many technical programs that control special devices; the use of a single byte to represent several flags is common. This is accomplished by having each one of the 8 bits that make up the byte represent a flag. Each bit has a value of either 1 or 0 and can represent true and false, on or off, yes or no, etc.

Key Terms

flag
: A variable used to store information that will normally be used to control the program.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Bit field ↵

This page titled 18.3: Flag Concept is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

18.4: Infinite Loops

An endless source of amusement for programmers is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop because there is no *iteration variable* telling you how many times to execute the loop.

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to `0`. Other times a loop is obviously infinite because it has no iteration variable at all.

This page titled 18.4: Infinite Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.3: Infinite Loops by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

18.5: Loop patterns

Often we use a `for` or `while` loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

This page titled 18.5: Loop patterns is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.7: Loop patterns by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

18.6: Maximum and Minimum Loops

To find the largest value in a list or sequence, we construct the following loop:

Code 5.9.1 (Python)

```
%%python3

largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)

```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable `largest` is best thought of as the "largest value we have seen so far". Before the loop, we set `largest` to the constant `None`. `None` is a special constant value which we can store in a variable to mark the variable as "empty".

Before the loop starts, the largest value we have seen so far is `None` since we have not yet seen any values. While the loop is executing, if `largest` is `None` then we take the first value we see as the largest so far. You can see in the first iteration when the value of `itervar` is 3, since `largest` is `None`, we immediately set `largest` to be 3.

After the first iteration, `largest` is no longer `None`, so the second part of the compound logical expression that checks `itervar > largest` triggers only when we see a value that is larger than the "largest so far". When we see a new "even larger" value we take that new value for `largest`. You can see in the program output that `largest` progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable `largest` now does contain the largest value in the list.

To compute the smallest number, the code is very similar with one small change:

Code 5.9.1 (Python)

```
%%python3

smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
```

```
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)

```

Again, `smallest` is the "smallest so far" before, during, and after the loop executes. When the loop has completed, `smallest` contains the minimum value in the list.

Again as in counting and summing, the built-in functions `max()` and `min()` make writing these exact loops unnecessary.

The following is a simple version of the Python built-in `min()` function:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

In the function version of the smallest code, we removed all of the `print` statements so as to be equivalent to the `min` function which is already built in to Python.

This page titled 18.6: Maximum and Minimum Loops is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.9: Maximum and Minimum Loops by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

18.7: Loop Examples

Temperature

Pseudocode

```
Function Main
    Call WhileLoop
    Call DoLoop
    Call ForLoop
End

Function WhileLoop
    Declare Real fahrenheit
    Declare Real celsius

    Assign fahrenheit = 0
    Call DisplayHeading()
    While fahrenheit <= 100
        Assign celsius = CalculateCelsius(fahrenheit)
        Call DisplayResult(fahrenheit, celsius)
        Assign fahrenheit = fahrenheit + 10
    End
End

Function DoLoop
    Declare Real fahrenheit
    Declare Real celsius

    Call DisplayHeading()
    Assign fahrenheit = 0
    Loop
        Assign celsius = CalculateCelsius(fahrenheit)
        Call DisplayResult(fahrenheit, celsius)
        Assign fahrenheit = fahrenheit + 10
    Do fahrenheit <= 100
End

Function ForLoop
    Declare Real fahrenheit
    Declare Real celsius

    Call DisplayHeading()
    For fahrenheit = 0 to 100 step 10
        Assign celsius = CalculateCelsius(fahrenheit)
        Call DisplayResult(fahrenheit, celsius)
    End
End

Function DisplayHeading
    Output "F°    C°"
End

```

```
Function CalculateCelsius (Real fahrenheit)
    Declare Real celsius

    Assign celsius = (fahrenheit - 32) * 5 / 9
Return Real celsius

Function DisplayResult (Real fahrenheit, Real celsius)
    Output fahrenheit & " = " & celsius
End
```

Output

```
F°      C°
0       -17.777777777778
10      -12.222222222222
20      -6.66666666666667
30      -1.11111111111111
40      4.44444444444444
50      10
60      15.5555555555556
70      21.1111111111111
80      26.6666666666667
90      32.2222222222222
100     37.7777777777778

F°      C°
0       -17.777777777778
10      -12.222222222222
20      -6.66666666666667
30      -1.11111111111111
40      4.44444444444444
50      10
60      15.5555555555556
70      21.1111111111111
80      26.6666666666667
90      32.2222222222222
100     37.7777777777778

F°      C°
0       -17.777777777778
10      -12.222222222222
20      -6.66666666666667
30      -1.11111111111111
40      4.44444444444444
50      10
60      15.5555555555556
70      21.1111111111111
80      26.6666666666667
90      32.2222222222222
100     37.7777777777778
```

Flowchart

File:Flowgorithm Loops Main

File:Flowgorithm While Loop.svg

File:Flowgorithm Do Loop.svg

File:Flowgorithm For Loop.svg

File:Flowgorithm Loops DisplayHeading.svg

File:Flowgorithm Conditions CalculateCelsius.svg

File:Flowgorithm Loops DisplayResult.svg

References

- Wikiversity: Computer Programming

This page titled 18.7: Loop Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

18.8: Python Examples

Temperature

```
# This program displays a temperature conversion table showing Fahrenheit
# temperatures from 0 to 100, in increments of 10, and the corresponding
# Celsius temperatures using While, Do While, and For loops.
#
# References:
#     https://www.mathsisfun.com/temperature-conversion.html
#     https://en.wikibooks.org/wiki/Python_Programming


def while_loop():
    display_heading()
    fahrenheit = 0
    while fahrenheit <= 100:
        celsius = calculate_celsius(fahrenheit)
        display_result(fahrenheit, celsius)
        fahrenheit += 10
    print()


def do_loop():
    display_heading()
    fahrenheit = 0
    while True:
        celsius = calculate_celsius(fahrenheit)
        display_result(fahrenheit, celsius)
        fahrenheit += 10
        if not(fahrenheit <= 100):
            break
    print()


def for_loop():
    display_heading()
    for fahrenheit in range(0, 101, 10):
        celsius = calculate_celsius(fahrenheit)
        display_result(fahrenheit, celsius)
    print()


def display_heading():
    print("F°\tC°")


def calculate_celsius(fahrenheit):
    celsius = (fahrenheit - 32) * 5 / 9
    return celsius


def display_result(fahrenheit, celsius):
    print(str(fahrenheit) + "\t" + str(celsius))
```

```
51
52
53 def main():
54     while_loop()
55     do_loop()
56     for_loop()
57
58
59 main()
```

Output

```
01 F°       C°
02 0        -17.77777777777778
03 10       -12.222222222222221
04 20       -6.666666666666667
05 30       -1.1111111111111112
06 40       4.444444444444445
07 50       10.0
08 60       15.555555555555555
09 70       21.11111111111111
10 80       26.666666666666668
11 90       32.22222222222222
12 100      37.77777777777778
13
14 F°       C°
15 0        -17.77777777777778
16 10       -12.222222222222221
17 20       -6.666666666666667
18 30       -1.1111111111111112
19 40       4.444444444444445
20 50       10.0
21 60       15.555555555555555
22 70       21.11111111111111
23 80       26.666666666666668
24 90       32.22222222222222
25 100      37.77777777777778
26
27 F°       C°
28 0        -17.77777777777778
29 10       -12.222222222222221
30 20       -6.666666666666667
31 30       -1.1111111111111112
32 40       4.444444444444445
33 50       10.0
34 60       15.555555555555555
35 70       21.11111111111111
36 80       26.666666666666668
37 90       32.22222222222222
38 100      37.77777777777778
```

References

- Wikiversity: Computer Programming

This page titled 18.8: Python Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

19: LISTS

19.1: Arrays and Lists
19.2: Index Notation
19.3: Displaying Array Members
19.4: Traversing a List
19.5: List operations
19.6: List Slices
19.7: List Methods
19.8: Deleting Elements
19.9: Math Statistics with Arrays
19.10: Searching Arrays
19.11: Sorting Arrays
19.12: Parallel Arrays
19.13: Multidimensional Arrays

This page titled 19: Lists is shared under a license and was authored, remixed, and/or curated by LibreTexts.

19.1: Arrays and Lists

Overview

An array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.[1]

Depending on the language, array types may overlap (or be identified with) other data types that describe aggregates of values, such as lists and strings. Array types are often implemented by array data structures, but sometimes by other means, such as hash tables, linked lists, or search trees.[2] In Python, the built-in array data structure is a list.

Discussion

An array is a sequenced collection of elements of the same data type with a single identifier name. Python lists are similar to arrays in other languages but are not restricted to a single data type. The term 'array' as used in this chapter will generally also apply to Python lists unless otherwise noted.

Arrays can have multiple axes (more than one axis). Each axis is a **dimension**. Thus a single-dimension array is also known as a **list**. A two-dimension array is commonly known as a **table** (a spreadsheet like Excel is a two dimension array). In real life, there are occasions to have data organized into multiple-dimension arrays. Consider a theater ticket with section, row, and seat (three dimensions). Most single-dimension arrays are visualized vertically.

Most programmers are familiar with a special type of array called a string. Strings are basically a single dimension array of characters. Unlike other single dimension arrays, we usually envision a string as a horizontal stream of characters and not vertically as a list.

We refer to the individual values as members (or elements) of the array. Programming languages implement the details of arrays differently. Because there is only one identifier name assigned to the array, we have operators that allow us to reference or access the individual members of an array. The operator commonly associated with referencing array members is the **index** operator. It is important to learn how to define an array and initialize its members.

Defining an Array

Language	Example
C++	`int ages[] = {49, 48, 26, 19, 16};`
C#	`int[] ages = {49, 48, 26, 19, 16};`
Java	`int[] ages = {49, 48, 26, 19, 16};`
JavaScript	`var ages = [49, 48, 26, 19, 16];`
Python	`ages = [49, 48, 26, 19, 16]`
Swift	`var ages:[Int] = [49, 48, 26, 19, 16]`

This is the **defining of storage space**. The square brackets `[]` are used here to create the array with five integer members and the identifier name of ages. The assignment with braces (that is a block) establishes the initial values assigned to the members of the array. Note the use of the sequence or comma operator. We could have also done something similar to:

Language	Example	Initial Values
C++	`int ages[5];`	undefined
C#	`int[] ages = new int[5];`	0

Java	`int[] ages = new int[5];`	0
JavaScript	`var ages = Array(5);`	undefined
Python	`ages = [None] * 5`	None

This would have declared the storage space of five integers with the identifier name of ages but their initial values would have been unknown values or initialized as indicated, depending on the programming language. We could assign values later in our program by doing the following (leaving off the semicolons in Python):

```
01 ages[0] = 49;
02 ages[1] = 48;
03 ages[2] = 26;
04 ages[3] = 19;
05 ages[4] = 16;
```

Note: The members of the array go from 0 to 4; **NOT** 1 to 5. This is explained in more detail on the next page.

Key Terms

dimension
: An axis of an array.

list
: A single dimension array.

table
: A two-dimension array.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Array data structure ↩
2. Wikipedia: Array data type ↩

This page titled 19.1: Arrays and Lists is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

Overview

Index notation is used to specify the elements of an array.[1] Most current programming languages use square brackets `[]` as the array index operator. Older programming languages, such as FORTRAN, COBOL, and BASIC, often use parentheses `( )` as the array index operator.

Discussion

Example:

Language	Example
C++	`int ages[] = {49, 48, 26, 19, 16};` `int myAge = ages[2];`
C#	`int[] ages = {49, 48, 26, 19, 16};` `int myAge = ages[2];`
Java	`int[] ages = {49, 48, 26, 19, 16};` `int myAge = ages[2];`
JavaScript	`var ages = [49, 48, 26, 19, 16];` `int myAge = ages[2];`
Python	`ages = [49, 48, 26, 19, 16]` `my_age = ages[2]`
Swift	`var ages:[Int] = [49, 48, 26, 19, 16]` `var my_age = ages[2]`

As an operator, square brackets either provide the value held by the member of the array (Rvalue) or change the value of member (Lvalue). In the above example, the member that is two offsets from the front of the array (the value 26) is assigned to the variable named myAge. The dereference operator of [2] means to go the 2nd **offset** from the front of the ages array and get the value stored there. In this case, the value would be 26. In most current programming languages, the array members (or elements) are referenced starting at **zero**. The more common way for people to reference a list is by starting with position **one**. Consider:

Position	Index	Miss America	Other Contests
zero offsets from the front	`ages[0]`	Winner	1st Place
one offset from the front	`ages[1]`	1st Runner Up	2nd Place
two offsets from the front	`ages[2]`	2nd Runner Up	3rd Place
three offsets from the front	`ages[3]`	3rd Runner Up	4th Place
four offsets from the front	`ages[4]`	4th Runner Up	5th Place

Saying that my cousin is the 2nd Runner-Up in the Miss America contest sounds so much better than saying that she was in 3rd Place. We would be talking about the same position in the array of the five finalists.

```
ages[3] = 20;
```

This is an example of changing an array's value by assigning 20 to the 4th member of the array and replacing the value 19 with 20. This is an Lvalue context because the array is on the left side of the assignment operator.

Key Terms

array member
: An element or value in an array.

index
: An operator that allows us to reference a member of an array.

offset
: The method of referencing array members by starting at zero.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Index notation ↩

This page titled 19.2: Index Notation is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

19.3: Displaying Array Members

Overview

To **display** all **array members**, visit each element using a `for` loop and output the element using index notation and the loop control variable.

Discussion

Accessing Array Members

Assume an integer array named ages with five values of 49, 48, 26, 19, and 16, respectively. In pseudocode, this might be written as:

```
Declare Integer Array ages[5]
Assign ages = [49, 48, 26, 19, 16]
```

To display all elements of the array in order, we might write:

```
Output ages[0]
Output ages[1]
Output ages[2]
Output ages[3]
Output ages[4]
```

While this works for short arrays, it is not very efficient, and quickly becomes overwhelming for longer arrays. One of the principles of software development is **don't repeat yourself (DRY)**. Violations of the DRY principle are typically referred to as WET solutions, which is commonly taken to stand for either "write everything twice", "we enjoy typing" or "waste everyone's time".[1]

Rather than repeating ourselves, we can use a for loop to visit each element of the array and use the loop control variable as the array index. Consider the following pseudocode:

```
01 Declare Integer Array ages[5]
02 Declare Integer index
03
04 Assign ages = [49, 48, 26, 19, 16]
05
06 For index = 0 to 4
07     Output ages[index]
08 End
```

This approach is much more efficient from a programming perspective, and also results in a smaller program. But there is still one more opportunity for improvement. Most programming languages support a built-in method for determining the size of an array. To reduce potential errors and required maintenance, the loop control should be based on the size of the array rather than a hard-coded value. Consider:

```
01 Declare Integer Array ages[5]
02 Declare Integer index
03
04 Assign ages = [49, 48, 26, 19, 16]
05
```

```
For index = 0 to Size(ages) - 1
    Output ages[index]
End
```

This method allows for **flexible coding**. By writing the for loop in this fashion, we can change the declaration of the array by adding or subtracting members and we don't need to change our for loop code.

Key Terms

don't repeat yourself
: A principle of software development aimed at reducing repetition of software patterns, replacing it with abstractions, or repetition of the same data, using data normalization to avoid redundancy.[2]

flexible coding Using the size of an array to determine the number of loop iterations required.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

---

1. Wikipedia: Don't repeat yourself↵
2. Wikipedia: Don't repeat yourself↵

This page titled 19.3: Displaying Array Members is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

19.4: Traversing a List

The most common way to traverse the elements of a list is with a `for` loop. The syntax is the same as for strings:

Code 8.3.1 (Python)

```
%%python3

cheeses = ['feta','gorgonzola','brie','mozzarella']

for cheese in cheeses:
    print(cheese)

```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

Code 8.3.1 (Python)

```
%%python3
numbers = [96, 23, 40, 76, 7]

for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
    print(numbers[i])
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to $n - 1$, where n is the length of the list. Each time through the loop, `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A `for` loop over an empty list never executes the body:

```
for x in empty:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

This page titled 19.4: Traversing a List is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.3: Traversing a List by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

19.5: List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats four times. The second example repeats the list three times.

This page titled 19.5: List operations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.4: List operations by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

19.6: List Slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

This page titled 19.6: List Slices is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.5: List Slices by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

19.7: List Methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

This example leaves `t2` unmodified.

`sort` arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

This page titled 19.7: List Methods is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.6: List Methods by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

19.8: Deleting Elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

`pop` modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the `del` operator:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

The return value from `remove` is `None`.

To remove more than one element, you can use `del` with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

As usual, the slice selects all the elements up to, but not including, the second index.

This page titled 19.8: Deleting Elements is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.7: Deleting Elements by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

19.9: Math Statistics with Arrays

Overview

Statistics is a branch of mathematics dealing with the collection, organization, analysis, interpretation, and presentation of data. Common statistical methods include mean (or average) and standard deviation.[1]

Discussion

Arrays are an important complex data type used in almost all programming. We continue to concentrate on simple one dimension arrays also called a list. Most programmers develop a series of user-defined specific task functions that can be used with an array for normal processing. These functions are usually passed the array along with the number of elements within the array. Some functions also pass another piece of data needed for that particular functions task.

This module covers the totaling of the members of an integer array member. The Latin name for totaling is summa, sometimes shortened to the word **sum**. In the example below, the sum function totals the array passed to it. Other mathematical functions often associated with statistics such as: average, count, minimum, maximum, standard deviation, etc. are often developed for processing arrays.

Pseudocode

```
01 Function Main
02     Declare Integer Array ages[5]
03     Declare Integer total
04
05     Assign ages = [49, 48, 26, 19, 16]
06
07     Assign total = sum(ages)
08
09     Output "Total age is: " & total
10 End
11
12 Function sum (Integer Array array)
13     Declare Integer total
14     Declare Integer index
15
16     Assign total = 0
17     For index = 0 to Size(array) - 1
18         Assign total = total + array[index]
19     End
20 Return Integer total
```

Output

```
01 Total age is: 158
```

Key Terms

sum

Latin for summa or a total.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Statistics ↵

This page titled 19.9: Math Statistics with Arrays is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

19.10: Searching Arrays

Overview

Linear search or sequential search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.[1]

Discussion

Finding a specific member of an array means searching the array until the member is found. It's possible that the member does not exist and the programmer must handle that possibility within the logic of his or her algorithm.

"The linear search is a very simple algorithm. Sometimes called a sequential search, it uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm will search to the end of the array."[2]

Two specific linear searches can be made for the maximum (largest) value in the array or the minimum (smallest) value in the array. Maximum and minimum are also known as max and min. Note that the following max and min functions assume an array size >= 1.

Pseudocode

```
Function Main
    Declare Integer Array ages[5]
    Declare Integer maximum
    Declare Integer minimum

    Assign ages = [49, 48, 26, 19, 16]

    Assign maximum = max(ages)
    Assign minimum = min(ages)

    Output "Maximum age is: " & maximum
    Output "Minimum age is: " & minimum
End

Function max (Integer Array array)
    Declare Integer maximum
    Declare Integer index

    Assign maximum = array[0]
    For index = 1 to Size(array) - 1
        If maximum < array[index]
            Assign maximum = array[index]
        End
    End
Return Integer maximum

Function min (Integer Array array)
    Declare Integer minimum
    Declare Integer index
```

```
30
31      Assign minimum = array[0]
32      For index = 1 to Size(array) - 1
33          If minimum > array[index]
34              Assign minimum = array[index]
35          End
36      End
37  Return Integer minimum
```

Output

```
01  Maximum age is: 49
02  Minimum age is: 16
```

Key Terms

linear search
: Using a loop to sequentially step through an array.

maximum
: Aka max or the largest member of an array.

minimum
: Aka min or the smallest member of an array.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

---

1. Wikipedia: Linear search ↩
2. Tony Gaddis, Judy Walters, and Godfrey Muganda, Starting Out with C++ Early Objects Sixth Edition (United States of America: Pearson – Addison Wesley, 2008) 559. ↩

This page titled 19.10: Searching Arrays is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

19.11: Sorting Arrays

Overview

A **sorting** algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order.[1] Most current programming languages include built-in or standard library functions for sorting arrays.

Discussion

Sorting is the process through which data are arranged according to their values. The following examples show standard library and/or built-in array sorting methods for different programming languages.

Language	Sort Example
C++	`#include <algorithm>` `sort(array, array + sizeof(array) / sizeof(int));`
C#	`System.Array.Sort(array)`
Java	`import java.util.Arrays;` `Arrays.sort(array);`
JavaScript	`array.sort();`
Python	`array.sort()`
Swift	`array.sort()`

Key Terms

sorting

Arranging data according to their values.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Sorting algorithm↵

This page titled 19.11: Sorting Arrays is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

19.12: Parallel Arrays

Overview

A group of **parallel arrays** is a form of implicit data structure that uses multiple arrays to represent a singular array of records. It keeps a separate, homogeneous data array for each field of the record, each having the same number of elements. Then, objects located at the same index in each array are implicitly the fields of a single record.[1]

Discussion

A data structure is a data organization and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. Data structure options include arrays, linked lists, records, and classes.[2]

Parallel arrays use two or more arrays to represent a collection of data where each corresponding array index is a matching field for a given record. For example, if there are two arrays, one for names and one for ages, the array elements at names[2] and ages[2] would describe the name and age of the third person.

Pseudocode

```
Function Main
    Declare String Array names[5]
    Declare Integer Array ages[5]

    Assign names = ["Lisa", "Michael", "Ashley", "Jacob", "Emily"]
    Assign ages = [49, 48, 26, 19, 16]

    DisplayArrays(names, ages)
End

Function DisplayArrays (String Array names, Integer Array ages)
    Declare Integer index

    For index = 0 to Size(array) - 1
        Output names[index] & " is " & ages[index] & " years old"
    End
End
```

Output

```
Lisa is 49 years old
Michael is 48 years old
Ashley is 26 years old
Jacob is 19 years old
Emily is 16 years old
```

Key Terms

parallel array
: An implicit data structure that uses multiple arrays to represent a singular array of records.

References

1. Wikipedia: Parallel array ↵
2. Wikipedia: Data structure ↵

This page titled 19.12: Parallel Arrays is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

19.13: Multidimensional Arrays

Overview

The number of indices needed to specify an element is called the dimension or dimensionality of the array. A two-dimensional array, or table, may be stored as a one-dimensional array of one-dimensional arrays (rows of columns) and accessed with double indexing (`array[row][column]` in typical notation).[1]

Discussion

An array is a sequenced collection of elements of the same data type with a single identifier name. As such, the array data type belongs to the “Complex” category or family of data types. Arrays can have multiple axes (more than one axis). Each axis is a **dimension**. Thus a single dimension array is also known as a **list**. A two-dimension array is commonly known as a **table** (a spreadsheet like Excel is a two dimension array). In real life, there are occasions to have data organized into multiple dimensioned arrays. Consider a theater ticket with section, row, and seat (three dimensions).

We refer to the individual values as members (or elements) of the array. Multidimensional arrays use one set of square brackets per dimension or axis of the array. For example, a table which has two dimensions would use two sets of square brackets to define the array variable and two sets of square brackets for the index operators to access the members of the array. Programming languages implement the details of arrays differently. The total number of dimensions allowed in an array is language-specific and also limited by available memory.

Pseudocode

```
01 Function Main
02     Declare String Array game[3][3]
03
04     Assign game = [ ["X", "O", "X"], ["O", "O", "O"], ["X", "O", "X"] ]
05
06     DisplayGame(game)
07 End
08
09 Function DisplayGame (String Array game)
10     Declare Integer row
11     Declare Integer column
12
13     Output "Tic-Tac-Toe"
14     For row = 0 to 2
15         For column = 0 to 2
16             Output game[row][column]
17             If column < 2 Then
18                 Output " | "
19             End
20         End
21     End
22 End
```

Output

```
01 Tic-Tac-Toe
02 X | O | X
```

```
0 | 0 | 0
X | 0 | X
```

Key Terms

array member
An element or value in an array.

dimension
An axis of an array.

index
An operator that allows us to reference a member of an array.

list
A single dimension array.

offset
The method of referencing array members by starting at zero.

table
A two-dimension array.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Array data type ↩

This page titled 19.13: Multidimensional Arrays is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

20: FUNCTION RETURN

20.1: Return Statement
20.2: Fruitful functions and void functions
20.3: Python Examples

This page titled 20: Function Return is shared under a license and was authored, remixed, and/or curated by LibreTexts.

20.1: Return Statement

Overview

A **return statement** causes execution to leave the current function and resume at the point in the code immediately after where the function was called. Return statements in many languages allow a function to specify a return value to be passed back to the code that called the function.[1]

Discussion

The return statement exits a function and returns to the statement where the function was called. Most programming languages support optionally returning a single value to the calling function. Consider the following pseudocode:

```
01 Function Main
02     ...
03     Assign fahrenheit = GetFahrenheit()
04     ...
05 End
06
07 Function GetFahrenheit
08     Declare Real fahrenheit
09
10     Output "Enter Fahrenheit temperature:"
11     Input fahrenheit
12 Return Real fahrenheit
```

In English, the Main function calls the GetFahrenheit function, passing in no parameters. The GetFahrenheit function retrieves input from the user and returns that input back to the main function, where it is assigned to the variable fahrenheit. In this example, the Main function has no return value.

Note that functions are independent, and each function must declare its own variables. While both functions have a variable named fahrenheit, they are not the same variable. Each variable refers to a different location in memory. Just as parameters by default are passed by position rather than by name, return values are also passed by position rather than by name. The following code would generate the same results.

```
01 Function Main
02     ...
03     Assign fahrenheit = GetTemperature()
04     ...
05 End
06
07 Function GetTemperature
08     Declare Real temperature
09
10     Output "Enter Fahrenheit temperature:"
11     Input temperature
12 Return Real temperature
```

Most programming languages support either zero or one return value from a function. There are some older programming languages where return values are not supported. In those languages, the modules are often referred to as subroutines rather than functions. There are also programming languages that support multiple return values in a single return statement, however, only single return values or no return value will be used in this book.

Key Terms

return
: A branching control structure that causes a function to jump back to the function that called it.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

---

1. Wikipedia: Return statement ↩

This page titled 20.1: Return Statement is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

20.2: Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them *fruitful functions*. Other functions, like `print_twice`, perform an action but don't return a value. They are called *void functions*.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.23606797749979
```

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value `None` is not the same as the string "None". It is a special value that has its own type:

```
>>> print(type(None))
<class 'NoneType'>
```

To return a result from a function, we use the `return` statement in our function. For example, we could make a very simple function called `addtwo` that adds two numbers together and returns a result.

Code 4.10.1 (Python)

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

```
07
08 # Code: http://www.py4e.com/code3/addtwo.py
09
```

When this script executes, the `print` statement will print out "8" because the `addtwo` function was called with 3 and 5 as arguments. Within the function, the parameters `a` and `b`were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named `added`. Then it used the `return` statement to send the computed value back to the calling code as the function result, which was assigned to the variable `x` and printed out.

This page titled 20.2: Fruitful functions and void functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.10: Fruitful functions and void functions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

20.3: Python Examples

Temperature

```
# This program asks the user for a Fahrenheit temperature,
# converts the given temperature to Celsius,
# and displays the results.
#
# References:
# https://www.mathsisfun.com/temperature-conversion.html
# https://en.wikibooks.org/wiki/Python_Programming


def get_fahrenheit():
    print("Enter Fahrenheit temperature:")
    fahrenheit = float(input())
    return fahrenheit


def calculate_celsius(fahrenheit):
    celsius = (fahrenheit - 32) * 5 / 9
    return celsius


def display_result(fahrenheit, celsius):
    print(str(fahrenheit) + "° Fahrenheit is " +
        str(celsius) + "° Celsius")


def main():
    fahrenheit = get_fahrenheit()
    celsius = calculate_celsius(fahrenheit)
    display_result(fahrenheit, celsius)


main()
```

Output

```
Enter Fahrenheit temperature:
 100
100.0° Fahrenheit is 37.77777777777778° Celsius
```

References

- Wikiversity: Computer Programming

This page titled 20.3: Python Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

21: FILE INPUT/OUTPUT

21.1: File Input and Output
21.2: Opening Files
21.3: Text files and Lines
21.4: Reading Files
21.5: Writing Files
21.6: Using try, except, and open
21.7: Loading an Array from a Text File
21.8: Letting the user choose the file name
21.9: Searching through a File
21.10: Python Examples

This page titled 21: File Input/Output is shared under a license and was authored, remixed, and/or curated by LibreTexts.

21.1: File Input and Output

Overview

A computer file is a computer resource for recording data discretely in a computer storage device. Just as words can be written to paper, so can information be written to a computer file.

There are different types of computer files, designed for different purposes. A file may be designed to store a picture, a written message, a video, a computer program, or a wide variety of other kinds of data. Some types of files can store several types of information at once.

By using computer programs, a person can open, read, change, and close a computer file. Computer files may be reopened, modified, and copied an arbitrary number of times.[1]

Discussion

In computer programming, standard **streams** are pre-connected input and output communication channels between a computer program and its environment when it begins execution. The three input/output (I/O) connections are called standard input (**stdin** – keyboard), standard output (**stdout** – originally a printer) and standard error (**stderr** – monitor). Streams may be redirected to other devices and/or files. In current environments, stdout is usually redirected to the monitor.[2]

Computer files are stored on secondary storage devices and used to maintain program data over time. Most programming languages have built-in functions or libraries to support processing files as text streams. We need to understand how to open, read, write and close text files. The following File Input/Output terms are explained:

Text File – A file consisting of characters from the ASCII character code set. Text files (also known as ASCII text files) contain character data. When we create a text file we usually think of it consisting of a series of lines. On each line are several characters (including spaces, punctuation, etc.) and we generally end the line with a return (a character within the ASCII character code set). The return is also known as the new line character. You are most likely already familiar with the escape code of \n which is used within many programming languages to indicate a return character when used within a literal string.

A typical text file consisting of lines can be created by text editors (Notepad) or word processing programs (Microsoft Word). When using a word processor you must usually specify the output file as text (.txt) when saving it. Most source code files are ASCII text files with a unique file extension; such as C++ using .cpp, C# using .cs, Python using .py, etc. Thus, most compiler/ Integrated Development Environment software packages can be used to create ASCII text files.

Filename – The name and its extension. Most operating systems have restrictions on which characters can be used in filenames. Example Lab_05.txt

Because some operating systems do not allow spaces, we suggest that you use the underscore where needed for spacing in a filename.

Path (Filespec) – The location of a file along with its filename. Filespec is short for file specification. Most operating systems have a set of rules on how to specify the drive and directory (or path through several directory levels) along with the filename. Example: C:\myfiles\cosc_1436\Lab_05.txt

Because some operating systems do not allow spaces, we suggest that you use the **underscore** where needed when creating folders or sub-directories.

Open – Your program requesting the operating system to let it have access to an existing file or to open a new file. In most current programming languages, a file data type exists and is used for file processing. A file variable will be used to store the device token that the operating system assigns to the file being opened. An open function or method is used to retrieve the device token, and typically requires at least two parameters: the path and the mode (read, write, append, or a combination thereof). Corresponding pseudocode would be:

```
Declare File datafile

datafile = open(filespec, mode)
```

The open function provides a return value of a **device token** from the operating system and it is stored in the variable named data.

It is considered good programming practice to determine if the file was opened properly. The reason the operating system usually can't open a file is because the filespec is wrong (misspelled or not typed case consistent in some operating systems) or the file is not stored in the location specified. Accessing files stored on a network or the Internet may fail due to a network error.

Verifying that a file was opened properly is processed with a condition control structure. That structure may be either be an if-then-else statement or a try-catch / try-except error handler, depending on the programming language used.

Read – Moving data from a device that has been opened into a memory location defined in your program. For example:
`text = read(datafile)`
or
`text = datafile.read()`

Write – Moving data from a memory location defined in your program to a device that has been opened. For example:
`write(datafile, text)`
or
`datafile.write(text)`

Close – Your program requesting the operating system to release a file that was previously opened. There are two reasons to close a file. First, it releases the file and frees up the associated operation system resources. Second, if closing a file that was opened for output; it will clear the out the operating system's buffer and ensure that all of the data is physically stored in the output file. For example:

`close(datafile)`
or
`datafile.close()`

Key Terms

close
: Your program requesting the operating system to release a file that was previously opened.

device token
: A key value provided by the operating system to associate a device to your program.

filename
: The name and its extension.

filespec
: The location of a file along with its filename.

open
: Your program requesting the operating system to let it have access to an existing file or to open a new file.

read
: Moving data from a device that has been opened into a memory location defined in your program.

stream
: A sequence of data elements made available over time.[3]

stdin
: Standard input stream, typically the keyboard. [4]

stderr
: Standard output error stream, typically the monitor.[5]

stdout
: Standard output stream, originally a printer, but now typically the monitor.[6]

text file
: A file consisting of characters from the ASCII character code set.

write
: Moving data from a memory location defined in your program to a device that has been opened.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Computer file ↵
2. Wikipedia: Standard streams ↵
3. Wikipedia: Stream (computing) ↵
4. Wikipedia: Standard streams ↵
5. Wikipedia: Standard streams ↵
6. Wikipedia: Standard streams ↵

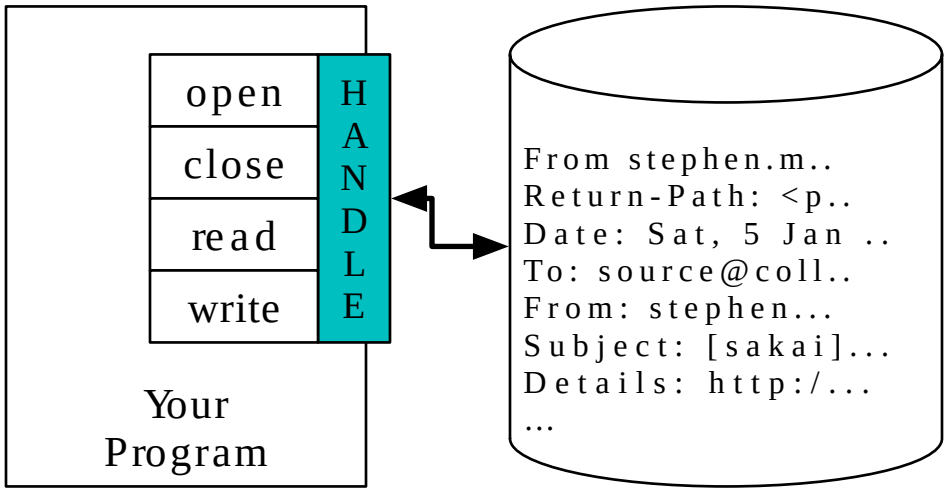
This page titled 21.1: File Input and Output is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

21.2: Opening Files

When we want to read or write a file (say on your hard drive), we first must *open* the file. Opening the file communicates with your operating system, which knows where the data for each file is stored. When you open a file, you are asking the operating system to find the file by name and make sure the file exists. In this example, we open the file `mbox.txt`, which should be stored in the same folder that you are in when you start Python. You can download this file from www.py4e.com/code3/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

If the `open` is successful, the operating system returns us a *file handle*. The file handle is not the actual data contained in the file, but instead it is a "handle" that we can use to read the data. You are given a handle if the requested file exists and you have the proper permissions to read the file.



A File Handle

If the file does not exist, `open` will fail with a traceback and you will not get a handle to access the contents of the file:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Later we will use `try` and `except` to deal more gracefully with the situation where we attempt to open a file that does not exist.

This page titled 21.2: Opening Files is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.2: Opening Files by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

21.3: Text files and Lines

A text file can be thought of as a sequence of lines, much like a Python string can be thought of as a sequence of characters. For example, this is a sample of a text file which records mail activity from various individuals in an open source project development team:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: source.sakaiproject.org/views...=rev&rev=39772
...
```

The entire file of mail interactions is available from

www.py4e.com/code3/mbox.txt

and a shortened version of the file is available from

www.py4e.com/code3/mbox-short.txt

These files are in a standard format for a file containing multiple mail messages. The lines which start with "From " separate the messages and the lines which start with "From:" are part of the messages. For more information about the mbox format, see en.Wikipedia.org/wiki/Mbox.

To break the file into lines, there is a special character that represents the "end of the line" called the *newline* character.

In Python, we represent the *newline* character as a backslash-n in string constants. Even though this looks like two characters, it is actually a single character. When we look at the variable by entering "stuff" in the interpreter, it shows us the `\n` in the string, but when we use `print` to show the string, we see the string broken into two lines by the newline character.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

You can also see that the length of the string `X\nY` is *three* characters because the newline character is a single character.

So when we look at the lines in a file, we need to *imagine* that there is a special invisible character called the newline at the end of each line that marks the end of the line.

So the newline character separates the characters in the file into lines.

This page titled 21.3: Text files and Lines is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.3: Text files and Lines by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

21.4: Reading Files

While the *file handle* does not contain the data for the file, it is quite easy to construct a `for` loop to read through and count each of the lines in a file:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

# Code: http://www.py4e.com/code3/open.py
```

We can use the file handle as the sequence in our `for` loop. Our `for` loop simply counts the number of lines in the file and prints them out. The rough translation of the `for` loop into English is, "for each line in the file represented by the file handle, add one to the `count` variable."

The reason that the `open` function does not read the entire file is that the file might be quite large with many gigabytes of data. The `open` statement takes the same amount of time regardless of the size of the file. The `for` loop actually causes the data to be read from the file.

When the file is read using a `for` loop in this manner, Python takes care of splitting the data in the file into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the `line` variable for each iteration of the `for` loop.

Because the `for` loop reads the data one line at a time, it can efficiently read and count the lines in very large files without running out of main memory to store the data. The above program can count the lines in any size file using very little memory since each line is read, counted, and then discarded.

If you know the file is relatively small compared to the size of your main memory, you can read the whole file into one string using the `read` method on the file handle.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

In this example, the entire contents (all 94,626 characters) of the file `mbox-short.txt` are read directly into the variable `inp`. We use string slicing to print out the first 20 characters of the string data stored in `inp`.

When the file is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable *inp*. Remember that this form of the `open`function should only be used if the file data will fit comfortably in the main memory of your computer.

If the file is too large to fit in main memory, you should write your program to read the file in chunks using a `for` or `while` loop.

This page titled 21.4: Reading Files is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.4: Reading Files by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

21.5: Writing Files

To write a file, you have to open it with mode "w" as a second parameter:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

The `write` method of the file handle object puts data into the file, returning the number of characters written. The default write mode is text for writing (and reading) strings.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

We must make sure to manage the ends of lines as we write to the file by explicitly inserting the newline character when we want to end a line. The `print` statement automatically appends a newline, but the `write` method does not add the newline automatically.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

When you are done writing, you have to close the file to make sure that the last bit of data is physically written to the disk so it will not be lost if the power goes off.

```
>>> fout.close()
```

We could close the files which we open for read as well, but we can be a little sloppy if we are only opening a few files since Python makes sure that all open files are closed when the program ends. When we are writing files, we want to explicitly close the files so as to leave nothing to chance.

This page titled 21.5: Writing Files is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.8: Writing Files by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

21.6: Using try, except, and open

I told you not to peek. This is your last chance.

What if our user types something that is not a file name?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'na na boo boo'
```

Do not laugh. Users will eventually do every possible thing they can do to break your programs, either on purpose or with malicious intent. As a matter of fact, an important part of any software development team is a person or group called *Quality Assurance* (or QA for short) whose very job it is to do the craziest things possible in an attempt to break the software that the programmer has created.

The QA team is responsible for finding the flaws in programs before we have delivered the program to the end users who may be purchasing the software or paying our salary to write the software. So the QA team is the programmer's best friend.

So now that we see the flaw in the program, we can elegantly fix it using the `try`/`except` structure. We need to assume that the `open` call might fail and add recovery code when the `open` fails as follows:

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search7.py
```

The `exit` function terminates the program. It is a function that we call that never returns. Now when our user (or QA team) types in silliness or bad file names, we "catch" them and recover gracefully:

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
```

```
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

Protecting the `open` call is a good example of the proper use of `try` and `except` in a Python program. We use the term "Pythonic" when we are doing something the "Python way". We might say that the above example is the Pythonic way to open a file.

Once you become more skilled in Python, you can engage in repartee with other Python programmers to decide which of two equivalent solutions to a problem is "more Pythonic". The goal to be "more Pythonic" captures the notion that programming is part engineering and part art. We are not always interested in just making something work, we also want our solution to be elegant and to be appreciated as elegant by our peers.

This page titled 21.6: Using try, except, and open is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.7: Using try, except, and open by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

21.7: Loading an Array from a Text File

Overview

Loading an array from a text file requires several steps, including: opening the file, reading the records, parsing (splitting) the records into fields, adding the fields to an array, and closing the file. The file may be read all at once and then parsed, or processed line by line. The array must either be at least as large as the number of records in the file, or must be generated dynamically.

Discussion

Loading an array from a file presents an interesting dilemma. The problem resolves around how many elements you should plan for in the array. Let's say 100, but what if the file has fewer or more than 100 values. How can the program handle it correctly?

Either:

1. Read the file and count the number of records.
2. Create a static array of that size.
3. Read the file again and add each record to the array.

Or:

1. Read the file and dynamically add the records to the array.

Processing Records

There are two options for file processing:

1. Read the entire file into memory, split the records and then process each record.
2. Read the file line by line and process one record at a time.

Which of these approaches is better will depend on the size of the file and the types of file and string processing supported by your programming language. Reading the entire file at once may be faster for small files. Very large files must be processed line by line.

Processing Fields

Processing fields requires splitting records based on the given file format. For example, a comma-separated-values file might be formatted as:

```
01 Celsius,Fahrenheit
02 0.0,32.0
03 1.0,33.8
04 2.0,35.6
```

The first line contains field names separated by commas. Following lines contain a value for each of the fields, separated by commas. Note that all text file input is strings. Each line must be split on the field separator (comma), and then numeric fields must be converted to integer or floating point values for processing.

Pseudocode

Static Array Processing

```
Open file
Read header
While Not End-Of-File
    Read line
    Increment record count
Close file

Declare array with length based on record count
Read Header
While Not End-Of-File
    Read line
    Split line into field(s)
    Convert numeric values to numeric data types
    Add field(s) to array or parallel arrays
Close file
```

Dynamic Array Processing

```
Declare empty array
Open file
Read Header
While Not End-Of-File
    Read line
    Split line into field(s)
    Convert numeric values to numeric data types
    Add field(s) to array or parallel arrays
Close file
```

Key Terms

dynamic memory
: Aka stack created memory associated with local scope.

static memory
: Aka data area memory associated with global scope.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 21.7: Loading an Array from a Text File is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

21.8: Letting the user choose the file name

We really do not want to have to edit our Python code every time we want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs so they can use our program on different files without changing the Python code.

This is quite simple to do by reading the file name from the user using `input` as follows:

```
fname = input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print('There were', count, 'subject lines in', fname)

# Code: http://www.py4e.com/code3/search6.py
```

We read the file name from the user and place it in a variable named `fname` and open that file. Now we can run the program repeatedly on different files.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

Before peeking at the next section, take a look at the above program and ask yourself, "What could go possibly wrong here?" or "What might our friendly user do that would cause our nice little program to ungracefully exit with a traceback, making us look not-so-cool in the eyes of our users?"

This page titled 21.8: Letting the user choose the file name is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.6: Letting the user choose the file name by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

21.9: Searching through a File

When you are searching through data in a file, it is a very common pattern to read through a file, ignoring most of the lines and only processing lines which meet a particular condition. We can combine the pattern for reading a file with string methods to build simple search mechanisms.

For example, if we wanted to read a file and only print out lines which started with the prefix "From:", we could use the string method *startswith* to select only those lines with the desired prefix:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
        print(line)

# Code: http://www.py4e.com/code3/search1.py
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

The output looks great since the only lines we are seeing are those which start with "From:", but why are we seeing the extra blank lines? This is due to that invisible *newline* character. Each of the lines ends with a newline, so the `print` statement prints the string in the variable *line* which includes a newline and then `print` adds *another* newline, resulting in the double spacing effect we see.

We could use line slicing to print all but the last character, but a simpler approach is to use the *rstrip* method which strips whitespace from the right side of a string as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)

# Code: http://www.py4e.com/code3/search2.py
```

When this program runs, we get the following output:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

```
From: cwen@iupui.edu
...
```

As your file processing programs get more complicated, you may want to structure your search loops using `continue`. The basic idea of the search loop is that you are looking for "interesting" lines and effectively skipping "uninteresting" lines. And then when we find an interesting line, we do something with that line.

We can structure the loop to follow the pattern of skipping uninteresting lines as follows:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)

# Code: http://www.py4e.com/code3/search3.py
```

The output of the program is the same. In English, the uninteresting lines are those which do not start with "From:", which we skip using `continue`. For the "interesting" lines (i.e., those that start with "From:") we perform the processing on those lines.

We can use the `find` string method to simulate a text editor search that finds lines where the search string is anywhere in the line. Since `find` looks for an occurrence of a string within another string and either returns the position of the string or -1 if the string was not found, we can write the following loop to show lines which contain the string "@uct.ac.za" (i.e., they come from the University of Cape Town in South Africa):

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Code: http://www.py4e.com/code3/search4.py
```

Which produces the following output:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

This page titled 21.9: Searching through a File is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.5: Searching through a File by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

21.10: Python Examples

Strings

```
# This program demonstrates string functions.


def main():
    string = "Hello"

    print("string: " + string)
    print("string.lower(): " + string.lower())
    print("string.upper(): " + string.upper())
    print("string.find('e'): " + str(string.find('e')))
    print("len(string): " + str(len(string)))
    print("string.replace('H', 'j'): " + string.replace('H', 'j'))
    print("string[::-1]: " + string[::-1])
    print("string[2:4]: " + string[2:4])
    print("string.strip('H'): " + string.strip('H'))

    name = "Bob"
    value = 123.456
    print("string.format(): {0} earned ${1:.2f}".format(name, value))


main()
```

Output

```
string: Hello
string.lower(): hello
string.upper(): HELLO
string.find('e'): 1
len(string): 5
string.replace('H', 'j'): jello
string[::-1]: olleH
string[2:4]: ll
string.strip('H'): ello
string.format(): Bob earned $123.46
```

Files

```
# This program creates a file, adds data to the file, displays the file,
# appends more data to the file, displays the file, and then deletes the file.
# It will not run if the file already exists.
#
# References:
#     https://www.mathsisfun.com/temperature-conversion.html
#     https://en.wikibooks.org/wiki/Python_Programming


```

```
import os
import sys


def calculate_fahrenheit(celsius):
    fahrenheit = celsius * 9 / 5 + 32
    return fahrenheit


def create_file(filename):
    try:
        file = open(filename, "w")
        file.write("Celsius,Fahrenheit\n")
        for celsius in range(0, 51):
            fahrenheit = calculate_fahrenheit(celsius)
            file.write("{:.1f},{:.1f}\n".format(celsius, fahrenheit))
        file.close()
    except:
        print("Error creating", filename)
        print(sys.exc_info()[1])


def read_file(filename):
    try:
        file = open(filename, "r")
        for line in file:
            line = line.strip()
            print(line)
        file.close()
        print()
    except:
        print("Error reading", filename)
        print(sys.exc_info()[1])


def append_file(filename):
    try:
        file = open(filename, "a")
        for celsius in range(51, 101):
            fahrenheit = calculate_fahrenheit(celsius)
            file.write("{:.1f},{:.1f}\n".format(celsius, fahrenheit))
        file.close()
    except:
        print("Error appending to", filename)
        print(sys.exc_info()[1])


def delete_file(filename):
    try:
        os.remove(filename)
    except:
        print("Error deleting", filename)
        print(sys.exc_info()[1])


def main():
    filename = "~file.txt"

```

```
    if os.path.isfile(filename):
        print("File already exists.")
    else:
        create_file(filename)
        read_file(filename)
        append_file(filename)
        read_file(filename)
        delete_file(filename)


main()
```

Output

```
Celsius,Fahrenheit
0.0,32.0
1.0,33.8
2.0,35.6
3.0,37.4
...
98.0,208.4
99.0,210.2
100.0,212.0
```

References

- Wikiversity: Computer Programming

This page titled 21.10: Python Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

22: SEARCHING AND SORTING ALGORITHMS

22.1: Activity 1 - Searching Algorithm
22.2: Activity 2 - Sorting Algorithm

This page titled 22: Searching and Sorting Algorithms is shared under a license and was authored, remixed, and/or curated by LibreTexts.

22.1: Activity 1 - Searching Algorithm

Introduction

This activity involves studying the searching process and how to write algorithms that can find a particular given item from a list of given items. The activity of searching will dedicate itself to providing the answer of a presence or no presence of the searched item. The section will dedicate itself to introducing the sequential and binary types of search.

Searching algorithm

In computer science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items.

Finding a Data Item - The Search Problem

Consider a collection of data items of same type e.g. integers, and imagine we wish to check if a particular data item is in the collection. Call the particular data item of interest to us, the key. The task is then to search among the records in the database to find out which one “matches” the key. This can first be done by arranging the data using a structure that can hold a collection of data items of the same type, that is, the arrays or linked lists. The searching can be done using either the sequential or binary search, where the former involves looking at each item in turn while the later involves items that are in order. The example below, is of an array consisting of seven elements long, containing numeric values. That is, the maximum number of comparisons is 7, and occurs when the key we are searching for is in A[6].

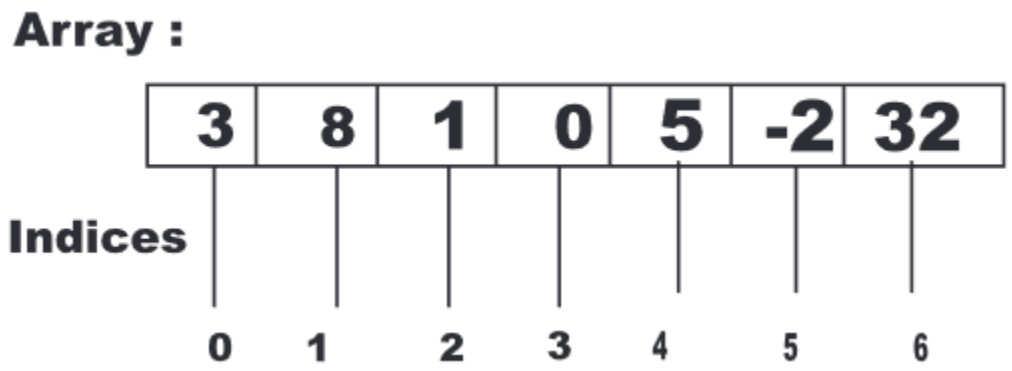


Figure 22.1.1: an array of numbers

Both sequential and binary search algorithms can be used to find the data item:

1. For a sequential search the following will do:

```
for (each item in list) {
compare search item to current item
if match,
save index of matching item
break
}
return index of matching item, or -1 if item not found
```

2. For a binary search, the following algorithm illustrates how it is carried out:

```
set first = 1, last = N, mid = N /2
while (item not found and first < last) {
compare search term to item at mid
```

```
04 if match
05 save index
06 break
07 else if search term is less than item at mid,
08 set last = mid-1
09 else
10 set first = mid+1
11 set mid = (first+last) /2
12 }
13 return index of matching item, or -1 if not found
```

Conclusion

In this activity, you learned two different search algorithms, the sequential and binary searches. Examples illustrating them were also given. Its application to an array was also introduced i.e. how a search can be applied in an array to search for a particular item in an array.

Assessment

Provide a short description of binary search algorithm.

This page titled 22.1: Activity 1 - Searching Algorithm is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 4.1: Activity 1 - Searching Algorithm is licensed CC BY-SA 4.0.

22.2: Activity 2 - Sorting Algorithm

Introduction

This activity involves studying the sorting process and how to write algorithms that can order particular items in a list to be in given order. The activity of sorting ensures items in a given list are arranged in a desired order, say ascending or descending. The section involves studying insertion sort, shell sort, and quicksort algorithms.

The sorting algorithm

A sorting algorithm is defined as an algorithm that puts elements of a list in a certain order. The most-used order is the alphabetical order, also known as the lexicographical order. Efficient sorting has the importance for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists. Examples of sorting algorithms include: insertion sort, shell sort and quick sort.

Insertion sort is the simplest method, and does not require any additional storage. Shell sort is a simple modification that improves performance significantly. Quicksort is the most efficient and popular method, and is the method of choice for large arrays.

Insertion Sort

Insertion sort is a simple sorting algorithm, and which is relatively efficient for small lists and mostly sorted lists. It can also be used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In an array structure, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The array is searched sequentially and unsorted items are moved and inserted into sorted sublist (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is no. of items.



Insertion sort compares the first two elements:



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



It swaps 33 with 27. Also it checks with all the elements of sorted sublist. Here we see that sorted sub-list has only one element 14 and 27 is greater than 14. Hence sorted sub-list remain sorted after swapping.



By now we have 14 and 27 in the sorted sublist. Next it compares 33 with 10.



These values are not in sorted order.



So we swap them.



But swapping makes 27 and 10 unsorted.



Again we find 14 and 10 in unsorted order.



And we swap them. By the end of third iteration we have a sorted sublist of 4 items.



This process goes until all the unsorted values are covered in sorted sublist. And now we shall see some programming aspects of insertion sort.

Example of the insertion algorithm

The following are simple steps by which we can achieve insertion sort.

Step 1 – If it is the first element, it is already sorted. return 1;

Step 2 – Pick next element

Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted

Shell Sort

Shell sort improves on the efficiency of insertion sort by quickly shifting values to their destination. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort if smaller value is very far right and has to move to far left. This algorithm uses insertion sort on widely spread elements first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This algorithm is quite efficient for medium sized data sets as its average and worst case complexity are of O(n) where n is no. of items.

How shell sort works

Consider the example below to have an idea how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding we take the interval of 4. And make a virtual sublist of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 14}.

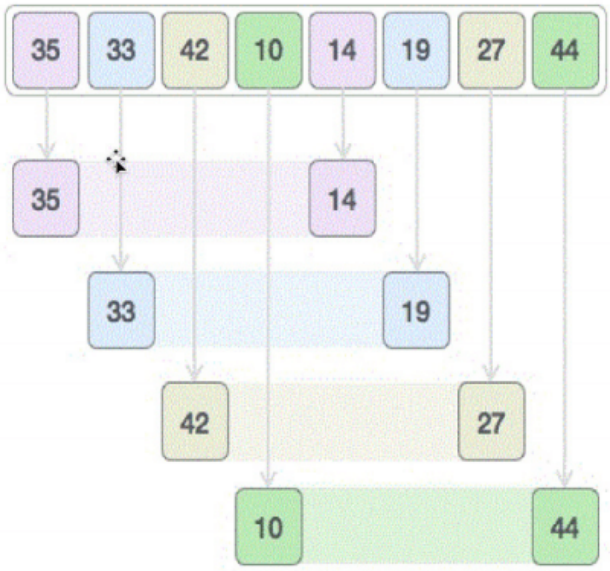
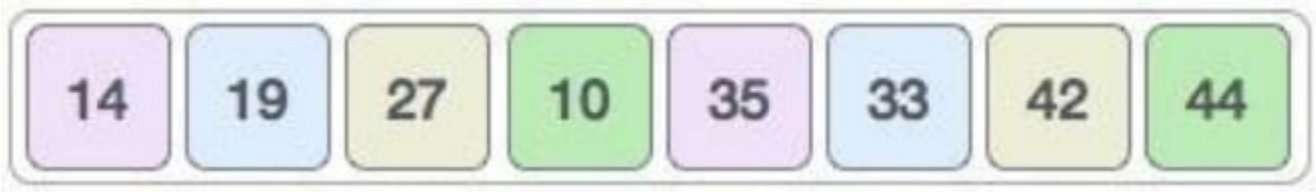


Figure 22.2.1: shell sort

We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this –



Then we take interval of 2 and this gap generates two sublists - {14, 27, 35, 42}, {19, 10, 33, 44}.

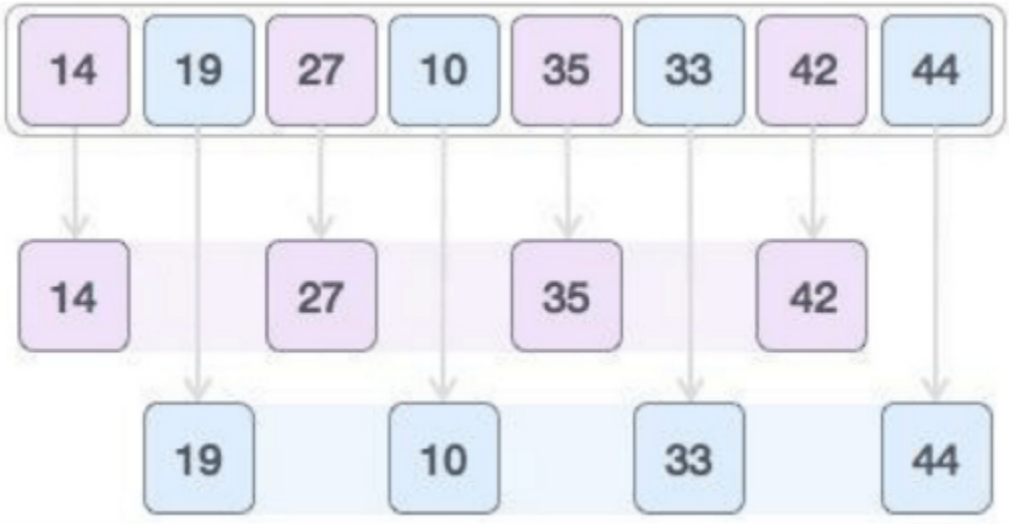


Figure 22.2.2

We compare and swap the values, if required, in the original array. After this step, this array should look like this –



And finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array. The step by step depiction is shown below –



Figure 22.2.3

It can be seen that it required only four swaps to sort the rest of the array.

The following is the algorithm for shell sort.

Step 1 – Initialize the value of h

Step 2 – Divide the list into smaller sub-list of equal interval h

Step 3 – Sort these sub-lists using insertion sort

Step 4 – Repeat until complete list is sorted

Quicksort

It is one of the most popular sorting algorithms. It is significantly a better algorithm than the insertion sort.

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays, one of which holds values smaller than specified value say pivot based on which the partition is made, and another array holds values greater than pivot value.

The quick sort partitions an array and then calls itself recursively twice to sort the resulting two subarrays. This algorithm is quite efficient for large sized data sets as its average and worst case complexity are of O(nlogn) where n is no. of items.

Below is the longer version of an algorithm for a quicksort:

Step 1 – Choose the highest index value as pivot

Step 2 – Take two variables to point left and right of the list excluding pivot

Step 3 – Left points to the low index

Step 4 – Right points to the high

Step 5 – While value at left is less than pivot move right

Step 6 – While value at right is greater than pivot move left

Step 7 – If both step 5 and step 6 does not match swap left and right

Step 8 – If left ≥ right, the point where they met is new pivot

It can be summarized into the following fewer steps; using pivot algorithm recursively, we end-up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as below –

Step 1 – Make the right-most index value pivot

Step 2 – Partition the array using pivot value

Step 3 – Quicksort left partition recursively

Step 4 – Quicksort right partition recursively

Conclusion

This activity introduced the learner to three different sort algorithms: the insertion sort, the shell sort and quicksort. A brief description of each was given.

Assessment

1. Describe briefly an insertion sorting algorithm.
2. What is the difference between selection and insertion sorting?

This page titled 22.2: Activity 2 - Sorting Algorithm is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 4.2: Activity 2 - Sorting Algorithm is licensed CC BY-SA 4.0.

23: OBJECT ORIENTED PROGRAMMING

23.1: Managing Larger Programs
23.2: Getting Started
23.3: Objects and Classes
23.4: Using Objects
23.5: Our First Python Object
23.6: Python Examples

This page titled 23: Object Oriented Programming is shared under a license and was authored, remixed, and/or curated by LibreTexts.

23.1: Managing Larger Programs

At the beginning of this book, we came up with four basic programming patterns which we use to construct programs:

- Sequential code
- Conditional code (if statements)
- Repetitive code (loops)
- Store and reuse (functions)

In later chapters, we explored simple variables as well as collection data structures like lists, tuples, and dictionaries.

As we build programs, we design data structures and write code to manipulate those data structures. There are many ways to write programs and by now, you probably have written some programs that are "not so elegant" and other programs that are "more elegant". Even though your programs may be small, you are starting to see how there is a bit of "art" and "aesthetic" to writing code.

As programs get to be millions of lines long, it becomes increasingly important to write code that is easy to understand. If you are working on a million line program, you can never keep the entire program in your mind at the same time. So we need ways to break the program into multiple smaller pieces so to solve a problem, fix a bug, or add a new feature we have less to look at.

In a way, object oriented programming is a way to arrange your code so that you can zoom into 500 lines of the code, and understand it while ignoring the other 999,500 lines of code for the moment.

This page titled 23.1: Managing Larger Programs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.1: Managing Larger Programs by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

23.2: Getting Started

Like many aspects of programming it is necessary to learn the concepts of object oriented programming before you can use them effectively. So approach this chapter as a way to learn some terms and concepts and work through a few simple examples to lay a foundation for future learning. Throughout the rest of the book we will be using objects in many of the programs but we won't be building our own new objects in the programs.

The key outcome of this chapter is to have a basic understanding of how objects are constructed and how they function and most importantly how we make use of the capabilities of objects that are provided to us by Python and Python libraries.

This page titled 23.2: Getting Started is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.2: Getting Started by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

23.3: Objects and Classes

Overview

Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects”, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object’s procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of “this” or “self”). There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.[1]

Discussion

Thus far, we have focused on procedural programming. Based on structured programming, procedures (routines, subroutines, or functions) contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program’s execution, including by other procedures or itself. The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines.[2] Small programs and scripts tend to be easier to develop using a simple procedural approach.

Object-oriented programming instead breaks down a programming task into objects that expose behavior (methods) and data (members or attributes) using interfaces. The most important distinction is that while procedural programming uses procedures to operate on separate data structures, object-oriented programming bundles the two together, so an “object”, which is an instance of a class, operates on its “own” data structure.[3] Larger programs benefit from better code and data isolation and reuse provided by an object-oriented approach.

Objects and classes are often designed to represent real-world objects. Consider a door as an example of a real-world object. Most doors have limited functionality. They may be opened and closed, and locked and unlocked. In procedural programming, we might design functions to open, close, lock, and unlock a door, such as:

```
Procedural Programming - Functions
OpenDoor(door)
CloseDoor(door)
LockDoor(door)
UnlockDoor(door)
```

Object-oriented programming combines code and data, so that, rather than having separate functions act on doors, we design doors that have methods that can act on themselves. Methods represent something the object can do, and are typically defined using verbs. Object-oriented door pseudocode might look like:

```
Object-Oriented Programming - Methods
door.Open()
door.Close()
door.Lock()
door.Unlock()
```

Objects may also have attributes, something the object is or has, and are typically defined using nouns or adjectives. Door attributes might include:

```
Object-Oriented Programming - Attributes
door.Height
door.Width
```

```
04 door.Color
05 door.Closed
06 door.Locked
```

When we write code to define a generic door, we would create a door class. The door class would contain all of the methods a door can perform and all of the attributes a door might have. We would then create instances of the class (objects) to represent specific doors, such as a front door, back door, or room door on a house, or a left door and right door on a car.

Key Terms

attribute
: A specification that defines a property of an object.[4]

class
: An extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).[5]

instance
: :A concrete occurrence of an object.[6]

method
: A specification that defines a procedure or behavior of an object.[7]

object
: A particular instance of a class where the object can be a combination of variables, functions, and data structures.[8]

this, self, or Me
: Keywords used in some computer programming languages to refer to the object, class, or other entity that the currently running code is part of.[9]

References

- Wikibooks: Object-Oriented Programming
- Wikipedia: Object-oriented programming
- Wikiversity: Computer Programming

---

1. Wikipedia: Object-oriented programming ↵
2. Wikipedia: Object-oriented programming ↵
3. Wikipedia: Object-oriented programming ↵
4. Wikipedia: Attribute (computing) ↵
5. Wikipedia: Class (computer programming) ↵
6. Wikipedia: Instance (computer science) ↵
7. Wikipedia: Method (computer programming) ↵
8. Wikipedia: Object (computer science) ↵
9. Wikipedia: this (computer programming) ↵

This page titled 23.3: Objects and Classes is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

23.4: Using Objects

It turns out we have been using objects all along in this class. Python provides us with many built-in objects. Here is some simple code where the first few lines should feel very simple and natural to you.

Code 14.3.1 (Python)

```
%%python3

stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])

print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: http://www.py4e.com/code3/party1.py

```

But instead of focusing on what these lines accomplish, lets look at what is really happening from the point of view of object-oriented programming. Don't worry if the following paragraphs don't make any sense the first time you read them because we have not yet defined all these terms.

The first line is *constructing* an object of type *list*, the second and third lines are calling the `append()` *method*, the fourth line is calling the `sort()` *method*, and the fifth line is retrieving the item at position `0`.

The sixth line is calling the `__getitem__()` method in the `stuff` list with a parameter of zero.

```
print (stuff.__getitem__(0))
```

The seventh line is an even more verbose way of retrieving the `0`th item in the list.

```
print (list.__getitem__(stuff,0))
```

In this code, we care calling the `__getitem__` method in the `list` class and passing in the list (`stuff`) and the item we want retrieved from the list as parameters.

The last three lines of the program are completely equivalent, but it is more convenient to simply use the square bracket syntax to look up an item at a particular position in a list.

We can take a look into the capabilities of an object by looking at the output of the `dir()` function:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
```

```
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

The precise definition of `dir()` is that it lists the *method*s and *attribute*s of a Python object.

The rest of this chapter will define all of the above terms so make sure to come back after you finish the chapter and re-read the above paragraphs to check your understanding.

This page titled 23.4: Using Objects is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.3: Using Objects by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

23.5: Our First Python Object

At its simplest, an object is some code plus data structures that is smaller than a whole program. Defining a function allows us to store a bit of code and give it a name and then later invoke that code using the name of the function.

An object can contain a number of functions (which we call "methods") as well as data that is used by those functions. We call data items that are part of the object "attributes".

We use the `class` keyword to define the data and code that will make up each of the objects. The class keyword includes the name of the class and begins an indented block of code where we include the attributes (data) and methods (code).

Code 14.6.1 (Python)

```
%%python3

class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print(\"So far\",self.x)

an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Code: http://www.py4e.com/code3/party2.py

```

Each method looks like a function, starting with the `def` keyword and consisting of an indented block of code. This example has one attribute (x) and one method (party). The methods have a special first parameter that we name by convention `self`.

Much like the `def` keyword does not cause function code to be executed, the `class` keyword does not create an object. Instead, the `class` keyword defines a template indicating what data and code will be contained in each object of type `PartyAnimal`. The class is like a cookie cutter and the objects created using the class are the cookies[2]. You don't put frosting on the cookie cutter, you put frosting on the cookies - and you can put different frosting on each cookie.



A Class and Two Objects

If you continue through the example code, we see the first executable line of code:

```
01 an = PartyAnimal()
```

This is where we instruct Python to construct (e.g. create) an *object* or "instance of the class named PartyAnimal". It looks like a function call to the class itself and Python constructs the object with the right data and methods and returns the object which is then assigned to the variable `an`. In a way this is quite similar to the following line which we have been using all along:

```
01 counts = dict()
```

Here we are instructing Python to construct an object using the `dict` template (already present in Python), return the instance of dictionary and assign it to the variable `counts`.

When the PartyAnimal class is used to construct an object, the variable `an` is used to point to that object. We use `an` to access the code and data for that particular instance of a PartyAnimal object.

Each Partyanimal object/instance contains within it a variable `x` and a method/function named `party`. We call that `party` method in this line:

```
01 an.party()
```

When the `party` method is called, the first parameter (which we call by convention `self`) points to the particular instance of the PartyAnimal object that `party` is called from within. Within the `party` method, we see the line:

```
01 self.x = self.x + 1
```

This syntax using the 'dot' operator is saying 'the x within self'. So each time `party()` is called, the internal `x` value is incremented by 1 and the value is printed out.

To help make sense of the difference between a global function and a method within a class/object, the following line is another way to call the `party` method within the `an` object:

```
PartyAnimal.party(an)
```

In this variation, we are accessing the code from within the *class* and explicitly passing the object pointer `an` in as the first parameter (i.e. `self` within the method). You can think of `an.party()` as shorthand for the above line.

When the program executes, it produces the following output:

```
So far 1
So far 2
So far 3
So far 4
```

The object is constructed, and the `party` method is called four times, both incrementing and printing the value for `x` within the `an` object.

This page titled 23.5: Our First Python Object is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.6: Our First Python Object by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

23.6: Python Examples

Objects

```
# This class converts temperature between Celsius and Fahrenheit.
# It may be used by assigning a value to either Celsius or Fahrenheit
# and then retrieving the other value, or by calling the to_celsius or
# to_fahrenheit methods directly.
#
# References:
#     https://www.mathsisfun.com/temperature-conversion.html
#     https://en.wikibooks.org/wiki/Python_Programming

class Temperature:
    _celsius = None
    _fahrenheit = None

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        self._celsius = float(value)
        self._fahrenheit = self.to_fahrenheit(self._celsius)

    @property
    def fahrenheit(self):
        return self._fahrenheit

    @fahrenheit.setter
    def fahrenheit(self, value):
        self._fahrenheit = float(value)
        self._celsius = self.to_celsius(self._fahrenheit)

    def __init__(self, celsius=None, fahrenheit=None):
        if celsius != None:
            self._celsius = celsius
            self._fahrenheit = self.to_fahrenheit(celsius)
        if fahrenheit != None:
            self._fahrenheit = fahrenheit
            self._celsius = self.to_celsius(fahrenheit)

    def to_celsius(self, fahrenheit):
        return (fahrenheit - 32) * 5 / 9

    def to_fahrenheit(self, celsius):
        return celsius * 9 / 5 + 32


# This program creates instances of the Temperature class to convert Cesius
# and Fahrenheit temperatures.

def main():
```

```
    temp1 = Temperature(celsius=0)
    print("temp1.celsius =", temp1.celsius)
    print("temp1.fahrenheit =", temp1.fahrenheit)
    print("")

    temp1.celsius = 100
    print("temp1.celsius =", temp1.celsius)
    print("temp1.fahrenheit =", temp1.fahrenheit)
    print("")

    temp2 = Temperature(fahrenheit=0)
    print("temp2.fahrenheit =", temp2.fahrenheit)
    print("temp2.celsius =", temp2.celsius)
    print("")

    temp2.fahrenheit = 100
    print("temp2.fahrenheit =", temp2.fahrenheit)
    print("temp2.celsius =", temp2.celsius)

main()
```

Output

```
temp1.celsius = 0
temp1.fahrenheit = 32.0

temp1.celsius = 100.0
temp1.fahrenheit = 212.0

temp2.fahrenheit = 0
temp2.celsius = -17.77777777777778

temp2.fahrenheit = 100.0
temp2.celsius = 37.77777777777778
```

References

- Wikiversity: Computer Programming

This page titled 23.6: Python Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

24: NETWORKED PROGRAMS

While many of the examples in this book have focused on reading files and looking for data in those files, there are many different sources of information when one considers the Internet. In this chapter we will pretend to be a web browser and retrieve web pages using the HyperText Transfer Protocol (HTTP). Then we will read through the web page data and parse it.

24.1: HyperText Transfer Protocol - HTTP

24.2: The World's Simplest Web Browser

24.3: Retrieving an image over HTTP

24.4: Retrieving web pages with urllib

24.5: Parsing HTML and scraping the web

24.6: Parsing HTML using regular expressions

24.7: Parsing HTML using BeautifulSoup

24.8: Reading binary files using urllib

24.E: Networked Programs (Exercises)

24.G: Networked Programs (Glossary)

This page titled 24: Networked Programs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12: Networked Programs by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.1: HyperText Transfer Protocol - HTTP

The network protocol that powers the web is actually quite simple and there is built-in support in Python called `sockets` which makes it very easy to make network connections and retrieve data over those sockets in a Python program.

A *socket* is much like a file, except that a single socket provides a two-way connection between two programs. You can both read from and write to the same socket. If you write something to a socket, it is sent to the application at the other end of the socket. If you read from the socket, you are given the data which the other application has sent.

But if you try to read a socket when the program on the other end of the socket has not sent any data, you just sit and wait. If the programs on both ends of the socket simply wait for some data without sending anything, they will wait for a very long time.

So an important part of programs that communicate over the Internet is to have some sort of protocol. A protocol is a set of precise rules that determine who is to go first, what they are to do, and then what the responses are to that message, and who sends next, and so on. In a sense the two applications at either end of the socket are doing a dance and making sure not to step on each other's toes.

There are many documents which describe these network protocols. The HyperText Transfer Protocol is described in the following document:

http://www.w3.org/Protocols/rfc2616/rfc2616.txt

This is a long and complex 176-page document with a lot of detail. If you find it interesting, feel free to read it all. But if you take a look around page 36 of RFC2616 you will find the syntax for the GET request. To request a document from a web server, we make a connection to the `www.pr4e.org` server on port 80, and then send a line of the form

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

where the second parameter is the web page we are requesting, and then we also send a blank line. The web server will respond with some header information about the document and a blank line followed by the document content.

This page titled 24.1: HyperText Transfer Protocol - HTTP is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.1: HyperText Transfer Protocol - HTTP by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.2: The World's Simplest Web Browser

Perhaps the easiest way to show how the HTTP protocol works is to write a very simple Python program that makes a connection to a web server and follows the rules of the HTTP protocol to request a document and display what the server sends back.

Code 12.2.1 (Python)

```
%%python3

import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\\r\\n\\r\\n'.encode()
mysock.send(cmd)

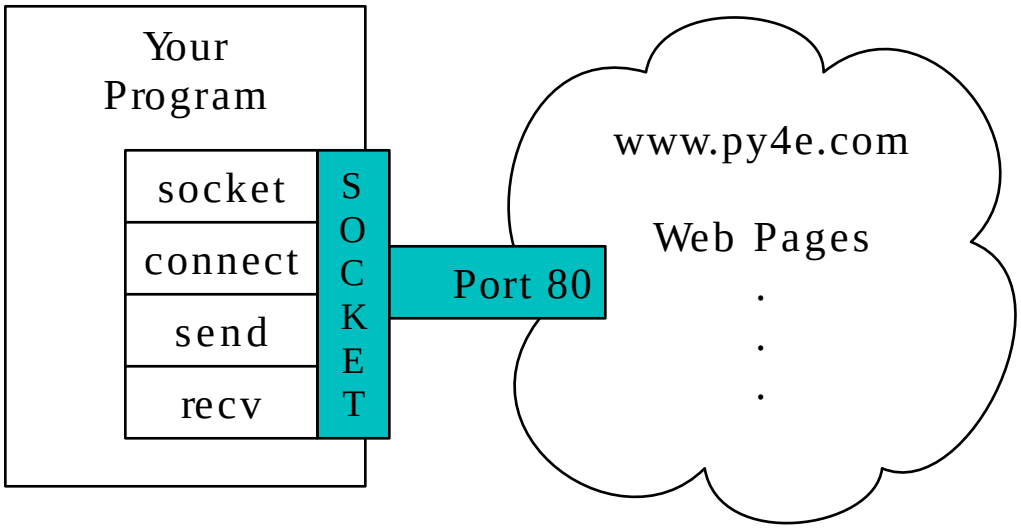
while True:
    data = mysock.recv(20)
    if (len(data) < 1):
        break
    print(data.decode(),end='')

mysock.close()

# Code: http://www.py4e.com/code3/socket1.py

```

First the program makes a connection to port 80 on the server www.py4e.com. Since our program is playing the role of the "web browser", the HTTP protocol says we must send the GET command followed by a blank line.



A Socket Connection

Once we send that blank line, we write a loop that receives data in 512-character chunks from the socket and prints the data out until there is no more data to read (i.e., the recv() returns an empty string).

The program produces the following output:

```
HTTP/1.1 200 OK
Date: Sun, 14 Mar 2010 23:52:41 GMT
Server: Apache
Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT
ETag: "143c1b33-a7-4b395bea"
Accept-Ranges: bytes
Content-Length: 167
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

The output starts with headers which the web server sends to describe the document. For example, the `Content-Type` header indicates that the document is a plain text document (`text/plain`).

After the server sends us the headers, it adds a blank line to indicate the end of the headers, and then sends the actual data of the file `romeo.txt`.

This example shows how to make a low-level network connection with sockets. Sockets can be used to communicate with a web server or with a mail server or many other kinds of servers. All that is needed is to find the document which describes the protocol and write the code to send and receive the data according to the protocol.

However, since the protocol that we use most commonly is the HTTP web protocol, Python has a special library specifically designed to support the HTTP protocol for the retrieval of documents and data over the web.

This page titled 24.2: The World's Simplest Web Browser is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.2: The World's Simplest Web Browser by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.3: Retrieving an image over HTTP

In the above example, we retrieved a plain text file which had newlines in the file and we simply copied the data to the screen as the program ran. We can use a similar program to retrieve an image across using HTTP. Instead of copying the data to the screen as the program runs, we accumulate the data in a string, trim off the headers, and then save the image data to a file as follows:

Code 12.3.1 (Python)

```
%%python3

import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\\r\\n\\r\\n')
count = 0
picture = b''

while True:
    data = mysock.recv(5120)
    if (len(data) < 1): break
    time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)
pos = picture.find(b\'\\r\\n\\r\\n\')
print('Header length', pos)
print(picture[:pos].decode())

# Skip past the header and save the picture data
picture = picture[pos+4:]
fhand = open('stuff.jpg', 'wb')
fhand.write(picture)
fhand.close()

# Code: http://www.py4e.com/code3/urljpeg.py

```

When the program runs it produces the following output:

```
$ python urljpeg.py
2920 2920
1460 4380
1460 5840
1460 7300
...
```

```
1460 62780
1460 64240
2920 67160
1460 68620
1681 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:15:07 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

You can see that for this url, the `Content-Type` header indicates that body of the document is an image (`image/jpeg`). Once the program completes, you can view the image data by opening the file `stuff.jpg` in an image viewer.

As the program runs, you can see that we don't get 5120 characters each time we call the `recv()` method. We get as many characters as have been transferred across the network to us by the web server at the moment we call `recv()`. In this example, we either get 1460 or 2920 characters each time we request up to 5120 characters of data.

Your results may be different depending on your network speed. Also note that on the last call to `recv()` we get 1681 bytes, which is the end of the stream, and in the next call to `recv()` we get a zero-length string that tells us that the server has called `close()` on its end of the socket and there is no more data forthcoming.

We can slow down our successive `recv()` calls by uncommenting the call to `time.sleep()`. This way, we wait a quarter of a second after each call so that the server can "get ahead" of us and send more data to us before we call `recv()` again. With the delay, in place the program executes as follows:

```
$ python urljpeg.py
1460 1460
5120 6580
5120 11700
...
5120 62900
5120 68020
2281 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:22:04 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

Now other than the first and last calls to `recv()`, we now get 5120 characters each time we ask for new data.

There is a buffer between the server making `send()` requests and our application making `recv()` requests. When we run the program with the delay in place, at some point the server might fill up the buffer in the socket and be forced to pause until our program starts to empty the buffer. The pausing of either the sending application or the receiving application is called "flow control".

This page titled 24.3: Retrieving an image over HTTP is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.3: Retrieving an image over HTTP by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.4: Retrieving web pages with urllib

While we can manually send and receive data over HTTP using the socket library, there is a much simpler way to perform this common task in Python by using the `urllib` library.

Using `urllib`, you can treat a web page much like a file. You simply indicate which web page you would like to retrieve and `urllib` handles all of the HTTP protocol and header details.

The equivalent code to read the `romeo.txt` file from the web using `urllib` is as follows:

Code 12.4.1 (Python)

```
%%python3

import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())

# Code: http://www.py4e.com/code3/urllib1.py

```

Once the web page has been opened with `urllib.urlopen`, we can treat it like a file and read through it using a `for` loop.

When the program runs, we only see the output of the contents of the file. The headers are still sent, but the `urllib` code consumes the headers and only returns the data to us.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

As an example, we can write a program to retrieve the data for `romeo.txt` and compute the frequency of each word in the file as follows:

Code 12.4.1 (Python)

```
%%python3

import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

```

```
# Code: http://www.py4e.com/code3/urlwords.py

```

Again, once we have opened the web page, we can read it like a local file.

This page titled 24.4: Retrieving web pages with urllib is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.4: Retrieving web pages with urllib by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.5: Parsing HTML and scraping the web

One of the common uses of the `urllib` capability in Python is to *scrape* the web. Web scraping is when we write a program that pretends to be a web browser and retrieves pages, then examines the data in those pages looking for patterns.

As an example, a search engine such as Google will look at the source of one web page and extract the links to other pages and retrieve those pages, extracting links, and so on. Using this technique, Google *spiders* its way through nearly all of the pages on the web.

Google also uses the frequency of links from pages it finds to a particular page as one measure of how "important" a page is and how high the page should appear in its search results.

This page titled 24.5: Parsing HTML and scraping the web is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.5: Parsing HTML and scraping the web by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.6: Parsing HTML using regular expressions

One simple way to parse HTML is to use regular expressions to repeatedly search for and extract substrings that match a particular pattern.

Here is a simple web page:

```
<h1>The First Page</h1>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

We can construct a well-formed regular expression to match and extract the link values from the above text as follows:

```
href="http://.+?"
```

Our regular expression looks for strings that start with "href="http://", followed by one or more characters (".+?"), followed by another double quote. The question mark added to the ".+?" indicates that the match is to be done in a "non-greedy" fashion instead of a "greedy" fashion. A non-greedy match tries to find the *smallest* possible matching string and a greedy match tries to find the *largest* possible matching string.

We add parentheses to our regular expression to indicate which part of our matched string we would like to extract, and produce the following program:

Code 12.6.1 (Python)

```
# Search for lines that start with From and have an at sign
import urllib.request, urllib.parse, urllib.error
import re

url = input('Enter - ')
html = urllib.request.urlopen(url).read()
links = re.findall(b'href=\"(http://.*?)\"', html)
for link in links:
    print(link.decode())

# Code: http://www.py4e.com/code3/urlregex.py

```

The `findall` regular expression method will give us a list of all of the strings that match our regular expression, returning only the link text between the double quotes.

When we run the program, we get the following output:

```
python urlregex.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
```

```
python urlregex.py
Enter -  http://allendowney.com/
```

```
03
04 http://gmpg.org/xfn/11
05 http://www.allendowney.com/wp/xmlrpc.php
06 http://www.allendowney.com/wp/feed/
07 http://www.allendowney.com/wp/comments/feed/
08 http://www.allendowney.com/wp/xmlrpc.php?rsd
09 http://www.allendowney.com/wp/wp-includes/wlwmanifest.xml
10 http://www.allendowney.com/wp/
11 http://www.allendowney.com/wp/wp-json/oembed/1.0/
   embed?url=http%3A%2F%2Fwww.allendowney.com%2Fwp%2F
12 http://www.allendowney.com/wp/wp-json/oembed/1.0/
   embed?url=http%3A%2F%2Fwww.allendowney.com%2Fwp%2F&#038;format=xml
13 http://www.allendowney.com/wp/
14 http://www.allendowney.com/wp/
15 http://www.allendowney.com/wp/books/
16 http://www.allendowney.com/wp/classes/
17 http://www.allendowney.com/wp/projects/
18 http://www.allendowney.com/wp/
19 http://www.olin.edu
20 http://allendowney.com/downeyCV.pdf
21 http://allendowney.blogspot.com/
22 http://www.oreilly.com/
23 http://greenteapress.com/
24 http://allendowney.com/research
25 http://www.allendowney.com/wp/public-service-announcement/
26 http://www.allendowney.com/wp/bayesian-statistics-for-undergrads/
27 http://www.allendowney.com/wp/epic-munchkin-win/
28 http://www.allendowney.com/wp/concurrent-programming-at-tufts/
29 http://www.allendowney.com/wp/think-python-2nd-edition/
30 http://www.allendowney.com/wp/
31 http://wordpress.org/
32 http://automattic.com/
```

Regular expressions work very nicely when your HTML is well formatted and predictable. But since there are a lot of "broken" HTML pages out there, a solution only using regular expressions might either miss some valid links or end up with bad data.

This can be solved by using a robust HTML parsing library.

This page titled 24.6: Parsing HTML using regular expressions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.6: Parsing HTML using regular expressions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.7: Parsing HTML using BeautifulSoup

There are a number of Python libraries which can help you parse HTML and extract data from the pages. Each of the libraries has its strengths and weaknesses and you can pick one based on your needs.

As an example, we will simply parse some HTML input and extract links using the *BeautifulSoup* library. You can download and install the BeautifulSoup code from:

http://www.crummy.com/software/

You can download and "install" BeautifulSoup or you can simply place the `BeautifulSoup.py` file in the same folder as your application.

Even though HTML looks like XML[1]i and some pages are carefully constructed to be XML, most HTML is generally broken in ways that cause an XML parser to reject the entire page of HTML as improperly formed. BeautifulSoup tolerates highly flawed HTML and still lets you easily extract the data you need.

We will use `urllib` to read the page and then use `BeautifulSoup` to extract the `href` attributes from the anchor (`a`) tags.

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# Or download the file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: http://www.py4e.com/code3/urllinks.py
```

The program prompts for a web address, then opens the web page, reads the data and passes the data to the BeautifulSoup parser, and then retrieves all of the anchor tags and prints out the `href` attribute for each tag.

When the program runs it looks as follows:

```
python urllinks.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
```

```
python urllinks.py
Enter - http://www.py4e.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.si502.com/
http://www.lib.umich.edu/espresso-book-machine
http://www.py4e.com/code
http://www.py4e.com/
```

You can use BeautifulSoup to pull out various parts of each tag as follows:

Code 12.7.1 (Python)

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# Or download the file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file


from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urlopen(url, context=ctx).read()

# html.parser is the HTML parser included in the standard Python 3 library.
# information on other HTML parsers is here:
# http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    # Look at the parts of a tag
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)

# Code: http://www.py4e.com/code3/urllink2.py

```

```
python urllink2.py
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
```

```
Content: ['\nSecond Page']
Attrs: [('href', 'http://www.dr-chuck.com/page2.htm')]
```

These examples only begin to show the power of BeautifulSoup when it comes to parsing HTML.

This page titled 24.7: Parsing HTML using BeautifulSoup is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.7: Parsing HTML using BeautifulSoup by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.8: Reading binary files using urllib

Sometimes you want to retrieve a non-text (or binary) file such as an image or video file. The data in these files is generally not useful to print out, but you can easily make a copy of a URL to a local file on your hard disk using `urllib`.

The pattern is to open the URL and use `read` to download the entire contents of the document into a string variable (`img`) then write that information to a local file as follows:

```
01 import urllib.request, urllib.parse, urllib.error
02
03 img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
04 fhand = open('cover3.jpg', 'wb')
05 fhand.write(img)
06 fhand.close()
07
08 # Code: http://www.py4e.com/code3/curl1.py
```

This program reads all of the data in at once across the network and stores it in the variable `img` in the main memory of your computer, then opens the file `cover.jpg` and writes the data out to your disk. This will work if the size of the file is less than the size of the memory of your computer.

However if this is a large audio or video file, this program may crash or at least run extremely slowly when your computer runs out of memory. In order to avoid running out of memory, we retrieve the data in blocks (or buffers) and then write each block to your disk before retrieving the next block. This way the program can read any size file without using up all of the memory you have in your computer.

Code 12.8.1 (Python)

```
01 %%python3
02
03 import urllib.request, urllib.parse, urllib.error
04
05 img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
06 fhand = open('cover3.jpg', 'wb')
07 size = 0
08 while True:
09     info = img.read(100000)
10     if len(info) < 1: break
11     size = size + len(info)
12     fhand.write(info)
13
14 print(size, 'characters copied.')
15 fhand.close()
16
17 # Code: http://www.py4e.com/code3/curl2.py
18
```

In this example, we read only 100,000 characters at a time and then write those characters to the `cover.jpg` file before retrieving the next 100,000 characters of data from the web.

This program runs as follows:

```
python curl2.py
568248 characters copied.
```

If you have a Unix or Macintosh computer, you probably have a command built in to your operating system that performs this operation as follows:

```
curl -O http://www.py4e.com/cover.jpg
```

The command `curl` is short for "copy URL" and so these two examples are cleverly named `curl1.py` and `curl2.py` on www.py4e.com/code3 as they implement similar functionality to the `curl` command. There is also a `curl3.py` sample program that does this task a little more effectively, in case you actually want to use this pattern in a program you are writing.

This page titled 24.8: Reading binary files using urllib is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.8: Reading binary files using urllib by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

24.E: Networked Programs (Exercises)

Exercise 1: Change the socket program `socket1.py` to prompt the user for the URL so it can read any web page. You can use `split('/')` to break the URL into its component parts so you can extract the host name for the socket `connect` call. Add error checking using `try` and `except` to handle the condition where the user enters an improperly formatted or non-existent URL.

Exercise 2: Change your socket program so that it counts the number of characters it has received and stops displaying any text after it has shown `3000` characters. The program should retrieve the entire document and count the total number of characters and display the count of the number of characters at the end of the document.

Exercise 3: Use `urllib` to replicate the previous exercise of (1) retrieving the document from a URL, (2) displaying up to `3000` characters, and (3) counting the overall number of characters in the document. Don't worry about the headers for this exercise, simply show the first `3000` characters of the document contents.

Exercise 4: Change the `urllinks.py` program to extract and count paragraph (p) tags from the retrieved HTML document and display the count of the paragraphs as the output of your program. Do not display the paragraph text, only count them. Test your program on several small web pages as well as some larger web pages.

Exercise 5: (Advanced) Change the socket program so that it only shows data after the headers and a blank line have been received. Remember that `recv` is receiving characters (newlines and all), not lines.

---

1. The XML format is described in the next chapter.↩

This page titled 24.E: Networked Programs (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 12.E: Networked Programs (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

BeautifulSoup

A Python library for parsing HTML documents and extracting data from HTML documents that compensates for most of the imperfections in the HTML that browsers generally ignore. You can download the BeautifulSoup code from www.crummy.com.

port

A number that generally indicates which application you are contacting when you make a socket connection to a server. As an example, web traffic usually uses port 80 while email traffic uses port 25.

scrape

When a program pretends to be a web browser and retrieves a web page, then looks at the web page content. Often programs are following the links in one page to find the next page so they can traverse a network of pages or a social network.

socket

A network connection between two applications where the applications can send and receive data in either direction.

spider

The act of a web search engine retrieving a page and then all the pages linked from a page and so on until they have nearly all of the pages on the Internet which they use to build their search index.

This page titled 24.G: Networked Programs (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

25: PYTHON AND WEB SERVICES

Once it became easy to retrieve documents and parse documents over HTTP using programs, it did not take long to develop an approach where we started producing documents that were specifically designed to be consumed by other programs (i.e., not HTML to be displayed in a browser). There are two common formats that we use when exchanging data across the web. The "eXtensible Markup Language" or XML has been in use for a very long time and is best suited for exchanging document-style data. When programs just want to exchange dictionaries, lists, or other internal information with each other, they use JavaScript Object Notation or JSON (see www.json.org). We will look at both formats.

25.1: eXtensible Markup Language - XML

25.2: Looping through Nodes

25.3: JavaScript Object Notation - JSON

25.4: Parsing JSON

25.5: Application Programming Interfaces

25.6: Google geocoding web service

25.7: Security and API usage

25.E: Python and Web Services (Exercises)

25.G: Python and Web Services (Glossary)

This page titled 25: Python and Web Services is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

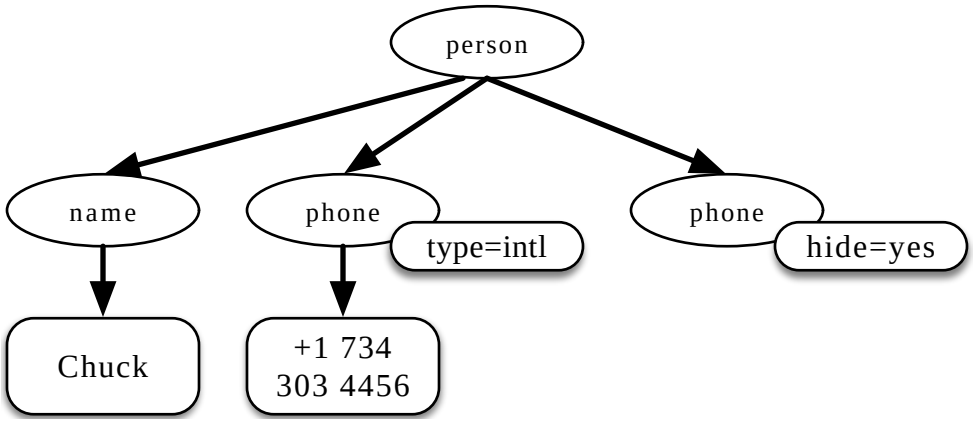
- 13: Python and Web Services by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

25.1: eXtensible Markup Language - XML

XML looks very similar to HTML, but XML is more structured than HTML. Here is a sample of an XML document:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>
```

Often it is helpful to think of an XML document as a tree structure where there is a top tag `person` and other tags such as `phone` are drawn as *children* of their parent nodes.



A Tree Representation of XML

Parsing XML

Here is a simple application that parses some XML and extracts some data elements from the XML:

Code 13.1.1 (Python)

```
%%python3

import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type=\"intl\">
     +1 734 303 4456
   </phone>
   <email hide=\"yes\"/>
</person>'''

tree = ET.fromstring(data)
```

```
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Code: http://www.py4e.com/code3/xml1.py

```

Calling `fromstring` converts the string representation of the XML into a "tree" of XML nodes. When the XML is in a tree, we have a series of methods we can call to extract portions of data from the XML.

The `find` function searches through the XML tree and retrieves a *node* that matches the specified tag. Each node can have some text, some attributes (like hide), and some "child" nodes. Each node can be the top of a tree of nodes.

```
Name: Chuck
Attr: yes
```

Using an XML parser such as `ElementTree` has the advantage that while the XML in this example is quite simple, it turns out there are many rules regarding valid XML and using `ElementTree` allows us to extract data from XML without worrying about the rules of XML syntax.

This page titled 25.1: eXtensible Markup Language - XML is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 13.1: eXtensible Markup Language - XML by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

25.2: Looping through Nodes

Often the XML has multiple nodes and we need to write a loop to process all of the nodes. In the following program, we loop through all of the `user` nodes:

Code 13.2.1 (Python)

```
%%python3

import xml.etree.ElementTree as ET

input = '''
<stuff>
    <users>
        <user x=\"2\">
            <id>001</id>
            <name>Chuck</name>
        </user>
        <user x=\"7\">
            <id>009</id>
            <name>Brent</name>
        </user>
    </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get(\"x\"))

# Code: http://www.py4e.com/code3/xml2.py

```

The `findall` method retrieves a Python list of subtrees that represent the `user` structures in the XML tree. Then we can write a `for` loop that looks at each of the user nodes, and prints the `name` and `id` text elements as well as the `x` attribute from the `user` node.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

This page titled 25.2: Looping through Nodes is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 13.2: Looping through Nodes by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

25.3: JavaScript Object Notation - JSON

The JSON format was inspired by the object and array format used in the JavaScript language. But since Python was invented before JavaScript, Python's syntax for dictionaries and lists influenced the syntax of JSON. So the format of JSON is nearly identical to a combination of Python lists and dictionaries.

Here is a JSON encoding that is roughly equivalent to the simple XML from above:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}
```

You will notice some differences. First, in XML, we can add attributes like "intl" to the "phone" tag. In JSON, we simply have key-value pairs. Also the XML "person" tag is gone, replaced by a set of outer curly braces.

In general, JSON structures are simpler than XML because JSON has fewer capabilities than XML. But JSON has the advantage that it maps *directly* to some combination of dictionaries and lists. And since nearly all programming languages have something equivalent to Python's dictionaries and lists, JSON is a very natural format to have two cooperating programs exchange data.

JSON is quickly becoming the format of choice for nearly all data exchange between applications because of its relative simplicity compared to XML.

This page titled 25.3: JavaScript Object Notation - JSON is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 13.3: JavaScript Object Notation - JSON by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

25.4: Parsing JSON

We construct our JSON by nesting dictionaries (objects) and lists as needed. In this example, we represent a list of users where each user is a set of key-value pairs (i.e., a dictionary). So we have a list of dictionaries.

In the following program, we use the built-in *json* library to parse the JSON and read through the data. Compare this closely to the equivalent XML data and code above. The JSON has less detail, so we must know in advance that we are getting a list and that the list is of users and each user is a set of key-value pairs. The JSON is more succinct (an advantage) but also is less self-describing (a disadvantage).

Code 13.4.1 (Python)

```
%%python3

import json

data = '''
[
  { \"id\" : \"001\",
    \"x\" : \"2\",
    \"name\" : \"Chuck\"
  } ,
  { \"id\" : \"009\",
    \"x\" : \"7\",
    \"name\" : \"Chuck\"
  }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: http://www.py4e.com/code3/json2.py

```

If you compare the code to extract data from the parsed JSON and XML you will see that what we get from *json.loads()* is a Python list which we traverse with a `for` loop, and each item within that list is a Python dictionary. Once the JSON has been parsed, we can use the Python index operator to extract the various bits of data for each user. We don't have to use the JSON library to dig through the parsed JSON, since the returned data is simply native Python structures.

The output of this program is exactly the same as the XML version above.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

In general, there is an industry trend away from XML and towards JSON for web services. Because the JSON is simpler and more directly maps to native data structures we already have in programming languages, the parsing and data extraction code is usually simpler and more direct when using JSON. But XML is more self-descriptive than JSON and so there are some applications where XML retains an advantage. For example, most word processors store documents internally using XML rather than JSON.

This page titled 25.4: Parsing JSON is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 13.4: Parsing JSON by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

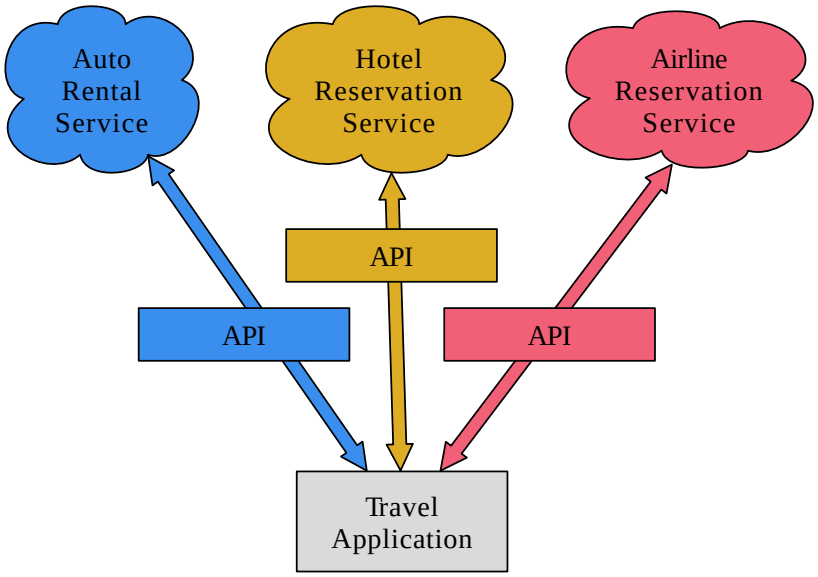
25.5: Application Programming Interfaces

We now have the ability to exchange data between applications using HyperText Transport Protocol (HTTP) and a way to represent complex data that we are sending back and forth between these applications using eXtensible Markup Language (XML) or JavaScript Object Notation (JSON).

The next step is to begin to define and document "contracts" between applications using these techniques. The general name for these application-to-application contracts is *Application Program Interfaces* or APIs. When we use an API, generally one program makes a set of *services* available for use by other applications and publishes the APIs (i.e., the "rules") that must be followed to access the services provided by the program.

When we begin to build our programs where the functionality of our program includes access to services provided by other programs, we call the approach a *Service-Oriented Architecture* or SOA. A SOA approach is one where our overall application makes use of the services of other applications. A non-SOA approach is where the application is a single standalone application which contains all of the code necessary to implement the application.

We see many examples of SOA when we use the web. We can go to a single web site and book air travel, hotels, and automobiles all from a single site. The data for hotels is not stored on the airline computers. Instead, the airline computers contact the services on the hotel computers and retrieve the hotel data and present it to the user. When the user agrees to make a hotel reservation using the airline site, the airline site uses another web service on the hotel systems to actually make the reservation. And when it comes time to charge your credit card for the whole transaction, still other computers become involved in the process.



Service Oriented Architecture

A Service-Oriented Architecture has many advantages including: (1) we always maintain only one copy of data (this is particularly important for things like hotel reservations where we do not want to over-commit) and (2) the owners of the data can set the rules about the use of their data. With these advantages, an SOA system must be carefully designed to have good performance and meet the user's needs.

When an application makes a set of services in its API available over the web, we call these *web services*.

This page titled 25.5: Application Programming Interfaces is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 13.5: Application Programming Interfaces by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

25.6: Google geocoding web service

Google has an excellent web service that allows us to make use of their large database of geographic information. We can submit a geographical search string like "Ann Arbor, MI" to their geocoding API and have Google return its best guess as to where on a map we might find our search string and tell us about the landmarks nearby.

The geocoding service is free but rate limited so you cannot make unlimited use of the API in a commercial application. But if you have some survey data where an end user has entered a location in a free-format input box, you can use this API to clean up your data quite nicely.

When you are using a free API like Google's geocoding API, you need to be respectful in your use of these resources. If too many people abuse the service, Google might drop or significantly curtail its free service.

You can read the online documentation for this service, but it is quite simple and you can even test it using a browser by typing the following URL into your browser:

http://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI

Make sure to unwrap the URL and remove any spaces from the URL before pasting it into your browser.

The following is a simple application to prompt the user for a search string, call the Google geocoding API, and extract information from the returned JSON.

Code 13.6.1 (Python)

```
import urllib.request, urllib.parse, urllib.error
import json

# Note that Google is increasingly requiring keys
# for this API
serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'

while True:
    address = input('Enter location: ')
    if len(address) < 1: break

    url = serviceurl + urllib.parse.urlencode(
        {'address': address})

    print('Retrieving', url)
    uh = urllib.request.urlopen(url)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')

    try:
        js = json.loads(data)
    except:
        js = None

    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue

```

```
    print(json.dumps(js, indent=4))

    lat = js[\"results\"][0][\"geometry\"][\"location\"][\"lat\"]
    lng = js[\"results\"][0][\"geometry\"][\"location\"][\"lng\"]
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)

# Code: http://www.py4e.com/code3/geojson.py
```

The program takes the search string and constructs a URL with the search string as a properly encoded parameter and then uses *urllib* to retrieve the text from the Google geocoding API. Unlike a fixed web page, the data we get depends on the parameters we send and the geographical data stored in Google's servers.

Once we retrieve the JSON data, we parse it with the *json* library and do a few checks to make sure that we received good data, then extract the information that we are looking for.

The output of the program is as follows (some of the returned JSON has been removed):

```
$ python3 geojson.py
Enter location: Ann Arbor, MI
Retrieving http://maps.googleapis.com/maps/api/
  geocode/json?address=Ann+Arbor%2C+MI
Retrieved 1669 characters
```

```
{
    "status": "OK",
    "results": [
        {
            "geometry": {
                "location_type": "APPROXIMATE",
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                }
            },
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ],
                    "short_name": "Ann Arbor"
                }
            ],
            "formatted_address": "Ann Arbor, MI, USA",
            "types": [
                "locality",
                "political"
            ]
        }
    ]
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA
```

```
Enter location:
```

You can download www.py4e.com/code3/geoxml.py to explore the XML variant of the Google geocoding API.

This page titled 25.6: Google geocoding web service is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 13.6: Google geocoding web service by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

25.7: Security and API usage

It is quite common that you need some kind of "API key" to make use of a vendor's API. The general idea is that they want to know who is using their services and how much each user is using. Perhaps they have free and pay tiers of their services or have a policy that limits the number of requests that a single individual can make during a particular time period.

Sometimes once you get your API key, you simply include the key as part of POST data or perhaps as a parameter on the URL when calling the API.

Other times, the vendor wants increased assurance of the source of the requests and so they add expect you to send cryptographically signed messages using shared keys and secrets. A very common technology that is used to sign requests over the Internet is called *OAuth*. You can read more about the OAuth protocol at www.oauth.net.

As the Twitter API became increasingly valuable, Twitter went from an open and public API to an API that required the use of OAuth signatures on each API request. Thankfully there are still a number of convenient and free OAuth libraries so you can avoid writing an OAuth implementation from scratch by reading the specification. These libraries are of varying complexity and have varying degrees of richness. The OAuth web site has information about various OAuth libraries.

For this next sample program we will download the files *twurl.py*, *hidden.py*, *oauth.py*, and *twitter1.py* from www.py4e.com/code and put them all in a folder on your computer.

To make use of these programs you will need to have a Twitter account, and authorize your Python code as an application, set up a key, secret, token and token secret. You will edit the file *hidden.py* and put these four strings into the appropriate variables in the file:

```
# Keep this file separate

# https://apps.twitter.com/
# Create new App and get the four strings

def oauth():
    return {"consumer_key": "h7Lu...Ng",
            "consumer_secret": "dNKenAC3New...mmn7Q",
            "token_key": "10185562-eibxCp9n2...P4GEQQOSGI",
            "token_secret": "H0ycCFemmC4wyf1...qoIpBo"}

# Code: http://www.py4e.com/code3/hidden.py
```

The Twitter web service are accessed using a URL like this:

api.twitter.com/1.1/statuses..._timeline.json

But once all of the security information has been added, the URL will look more like:

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

You can read the OAuth specification if you want to know more about the meaning of the various parameters that are added to meet the security requirements of OAuth.

For the programs we run with Twitter, we hide all the complexity in the files *oauth.py* and *twurl.py*. We simply set the secrets in

hidden.py and then send the desired URL to the *twurl.augment()* function and the library code adds all the necessary parameters to the URL for us.

This program retrieves the timeline for a particular Twitter user and returns it to us in JSON format in a string. We simply print the first 250 characters of the string:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    print(data[:250])
    headers = dict(connection.getheaders())
    # print headers
    print('Remaining', headers['x-rate-limit-remaining'])

# Code: http://www.py4e.com/code3/twitter1.py
```

When the program runs it produces the following output:

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178

Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

```

```
Enter Twitter Account:
```

Along with the returned timeline data, Twitter also returns metadata about the request in the HTTP response headers. One header in particular, *x-rate-limit-remaining*, informs us how many more requests we can make before we will be shut off for a short time period. You can see that our remaining retrievals drop by one each time we make a request to the API.

In the following example, we retrieve a user's Twitter friends, parse the returned JSON, and extract some of the information about the friends. We also dump the JSON after parsing and "pretty-print" it with an indent of four characters to allow us to pore through the data when we want to extract more fields.

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()

    js = json.loads(data)
    print(json.dumps(js, indent=2))

    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])

    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
            print('   * No status found')
            continue
        s = u['status']['text']
        print('  ', s[:50])

# Code: http://www.py4e.com/code3/twitter2.py
```

Since the JSON becomes a set of nested Python lists and dictionaries, we can use a combination of the index operation and `for` loops to wander through the returned data structures with very little Python code.

The output of the program looks as follows (some of the data items are shortened to fit on the page):

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14
```


```
{
    "next_cursor": 1444171224491980205,
    "users": [
        {
            "id": 662433,
            "followers_count": 28725,
            "status": {
                "text": "@jazzychad I just bought one .__.",
                "created_at": "Fri Sep 20 08:36:34 +0000 2013",
                "retweeted": false,
            },
            "location": "San Francisco, California",
            "screen_name": "leahculver",
            "name": "Leah Culver",
        },
        {
            "id": 40426722,
            "followers_count": 2635,
            "status": {
                "text": "RT @WSJ: Big employers like Google ...",
                "created_at": "Sat Sep 28 19:36:37 +0000 2013",
            },
            "location": "Victoria Canada",
            "screen_name": "_valeriei",
            "name": "Valerie Irvine",
    ],
    "next_cursor_str": "1444171224491980205"
}
```


```
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

The last bit of the output is where we see the for loop reading the five most recent "friends" of the *drchuck* Twitter account and printing the most recent status for each friend. There is a great deal more data available in the returned JSON. If you look in the output of the program, you can also see that the "find the friends" of a particular account has a different rate limitation than the number of timeline queries we are allowed to run per time period.

These secure API keys allow Twitter to have solid confidence that they know who is using their API and data and at what level. The rate-limiting approach allows us to do simple, personal data retrievals but does not allow us to build a product that pulls data from their API millions of times per day.

This page titled 25.7: Security and API usage is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 13.7: Security and API usage by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

25.E: Python and Web Services (Exercises)

This page titled 25.E: Python and Web Services (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

API

Application Program Interface - A contract between applications that defines the patterns of interaction between two application components.

ElementTree

A built-in Python library used to parse XML data.

JSON

JavaScript Object Notation. A format that allows for the markup of structured data based on the syntax of JavaScript Objects.

SOA

Service-Oriented Architecture. When an application is made of components connected across a network.

XML

eXtensible Markup Language. A format that allows for the markup of structured data.

This page titled 25.G: Python and Web Services (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

26: ALGORITHMS

26.1: Activity 1 - Overview of Algorithm Design and Analysis
26.2: Activity 1 - Introduction to Algorithms and Problem Solving
26.3: Activity 2 - The characteristics of an algorithm

This page titled 26: Algorithms is shared under a license and was authored, remixed, and/or curated by LibreTexts.

26.1: Activity 1 - Overview of Algorithm Design and Analysis

Introduction

There are compelling reasons to study algorithms. To put it simply, computer programs would not exist without algorithms. And with computer applications becoming indispensable in almost all aspects of our professional and personal lives, studying algorithms becomes a necessity for more and more people.

Another reason for studying algorithms is their usefulness in developing analytical skills. After all, algorithms can be seen as special kinds of solutions to problems, not answers but precisely defined procedures for getting answers. Consequently, specific algorithm design techniques can be interpreted as problem solving strategies that can be useful regardless of whether a computer is involved.

Activity Details

The term algorithm can be defined using any of the following statements:

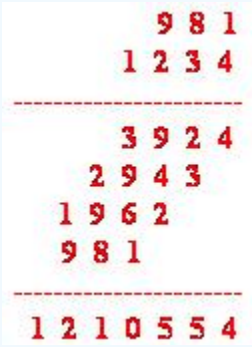
- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

The most famous algorithm in history dates well before the time of the ancient Greeks: this is the Euclid's algorithm for calculating the greatest common divisor of two integers.

The Classic Multiplication Algorithm

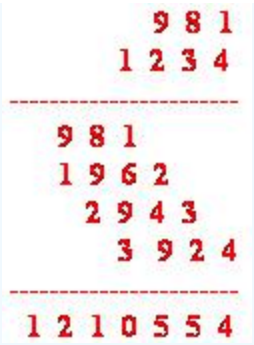
Multiplication, the American way:

Multiply the multiplicand one after another by each digit of the multiplier taken from right to left.



Multiplication, the English way:

Multiply the multiplicand one after another by each digit of the multiplier taken from left to right.



Algorithmics is a branch of computer science that consists of designing and analyzing computer algorithms.

The “design” concern to:

- The description of algorithm at an abstract level by means of a pseudo language, and
- Proof of correctness that is, the algorithm solves the given problem in all cases.

The “analysis” deals with performance evaluation (complexity analysis).

We start with defining the model of computation, which is usually the Random Access Machine (RAM) model, but other models of computations can be use such as PRAM. Once the model of computation has been defined, an algorithm can be describe using a simple language (or pseudo language) whose syntax is close to programming language such as C or java.

Algorithm’s Performance

Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns with determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat rough, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count.

Asymptotic analysis makes use of the O (Big Oh) notation. Two other notational constructs used by computer scientists in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation. The performance evaluation of an algorithm is obtained by tallying the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative constant.

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

Θ-Notation (Same order)

This notation bounds a function to within constant factors. We say f(n) = Θ(g(n)) if there exist positive constants n0, c1 and c2 such that to the right of n0 the value of f(n) always lies between c1 g(n) and c2 g(n) inclusive.

In the set notation, we write as follows:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c1, c2, \text{ and } n0 \text{ such that } 0 \le c1\, g(n) \le f(n) \le c2\, g(n) \text{ for all } n \ge n0\}$$

We say that g(n) is an asymptotically tight bound for f(n)

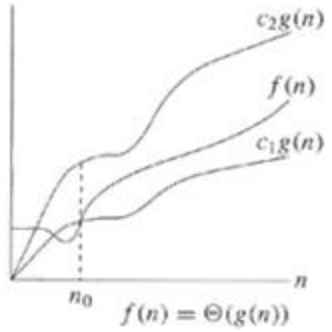


Figure 1.1.1 : Graphs to demonstrate Θ(g(n))

As shown in Figure 1.1.1 , graphically, for all values of n to the right of n0, the value of f(n) lies at or above c1 g(n) and at or below c2 g(n). In other words, for all n ≥ n0, the function f(n) is equal to g(n) to within a constant factor. We say that g(n) is an asymptotically tight bound for f(n).

In the set terminology, f(n) is said to be a member of the set Θ(g(n)) of functions. In other words, because O(g(n)) is a set, we could write

$$f(n) \in \Theta(g(n))$$

to indicate that f(n) is a member of Θ(g(n)). Instead, we write

$$f(n) = \Theta(g(n))$$

to express the same notation.

Historically, this notation is "f(n) = Θ(g(n))" although the idea that f(n) is equal to something called Θ(g(n)) is misleading.

Example: n2/2 − 2n = Θ(n2), with c1 = 1/4, c2 = 1/2, and n0 = 8.

O-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. We write f(n) = O(g(n)) if there are positive constants n0 and c such that to the right of n0, the value of f(n) always lies on or below c g(n).

In the set notation, we write as follows: For a given function g(n), the set of functions

$$O(g(n)) = \{f(n): \text{there exist positive constants c and } n_0 \text{ such that } 0 \le f(n) \le c\, g(n) \text{ for all } n \ge n_0\}$$

We say that the function g(n) is an asymptotic upper bound for the function f(n). We use O-notation to give an upper bound on a function, to within a constant factor.

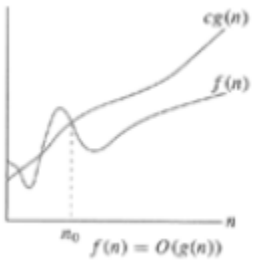


Figure 1.1.2 : Graphs to demonstrate O(g(n))

As shown in Figure 1.1.2 , graphically, for all values of n to the right of n0, the value of the function f(n) is on or below g(n). We write f(n) = O(g(n)) to indicate that a function f(n) is a member of the set O(g(n)) i.e.

$$f(n) \in O(g(n))$$

Note that f(n) = Θ(g(n)) implies f(n) = O(g(n)), since Θ-notation is a stronger notation than O-notation.

Example: 2n2 = O(n3), with c = 1 and n0 = 2.

Equivalently, we may also define f is of order g as follows:

If f(n) and g(n) are functions defined on the positive integers, then f(n) is O(g(n)) if and only if there is a c > 0 and an n0 > 0 such that

$$| f(n) | \leq | g(n) | \text{ for all } n \geq n_0$$

Ω-Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write f(n) = Ω(g(n)) if there are positive constants n0 and c such that to the right of n0, the value of f(n) always lies on or above c g(n).

In the set notation, we write as follows: For a given function g(n), the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants c and } n_0 \text{ such that } 0 \leq c\ g(n) \leq f(n) \text{ for all } n \geq n_0\}$$

We say that the function g(n) is an asymptotic lower bound for the function f(n).

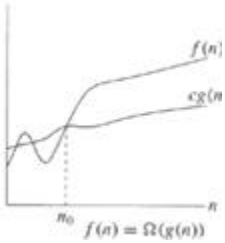


Figure 1.1.3 : Graphs to demonstrate Ω(g(n))

Figure 1.1.3 provides an insight behind Ω-notation. For example, √n = (lg n), with c = 1 and n0 = 16.

Algorithm's Analysis

The complexity of an algorithm is a function g(n) that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n.

There are two interpretations of upper bound.

- Worst-case Complexity: the running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.
- Average-case Complexity: the running time for any given size input will be the average number of operations over all problem instances for a given size.

Because it is quite difficult to estimate the statistical behaviour of the input, most of the time we content ourselves to a worst case behaviour. Most of the time, the complexity of g(n) is approximated by its family O(f(n)) where f(n) is one of the following functions: n (linear complexity), log n (logarithmic complexity), na where a ≥ 2 (polynomial complexity), an (exponential complexity).

Optimality

Once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the algorithms solving this problem. For example, any algorithm solving "the intersection of n segments" problem will execute at least n2 operations in the worst case even if it does nothing but print the output. This is abbreviated by saying that the problem has Ω(n2) complexity. If one finds an O(n2) algorithm that solves this problem, it will be optimal and of complexity Θ(n2).

Reduction

Another technique for estimating the complexity of a problem is the transformation of problems, also called problem reduction. As an example, suppose we know a lower bound for a problem A, and that we would like to estimate a lower bound for a problem B. If we can transform A into B by a transformation step whose cost is less than that for solving A, then B has the same bound as A.

The Convex hull problem nicely illustrates “reduction” technique. A lower bound of Convex-hull problem is established by reducing the sorting problem (complexity: Θ(n log n)) to the Convex hull problem.

Conclusion

This activity introduced the basic concepts of algorithm design and analysis. The need for analysing the time and space complexity of algorithms was articulated and the notations for describing algorithm complexity were presented.

Assessment

1. Distinguish between space and time complexity.
2. Briefly explain the commonly used notations in performance analysis.

This page titled 26.1: Activity 1 - Overview of Algorithm Design and Analysis is shared under a license and was authored, remixed, and/or curated by Godfry Justo (African Virtual University).

- 1.1: Activity 1 - Overview of Algorithm Design and Analysis by Godfry Justo has no license indicated.

26.2: Activity 1 - Introduction to Algorithms and Problem Solving

Introduction

In this learning activity section, the learner will be introduced to algorithms and how to write algorithms to solve tasks faced by learners or everyday problems. Examples of the algorithm are also provided with a specific application to everyday problems that the learner is familiar with. The learners will particularly learn what is an algorithm, the process of developing a solution for a given task, and finally examples of application of the algorithms are given.

Algorithm

An algorithm is a finite sequence of steps for accomplishing some computational task. It must

- Have steps that are simple and definite enough to be done by a computer, and
- Terminate after finitely many steps.

An algorithm can be considered as a computational procedure that consists of a set of instructions, that takes some value or set of values, as input, and produces some value or set of values, as output, as illustrated in Figure 1.3.1. It can also be described as a procedure that accepts data, manipulate them following the prescribed steps, so as to eventually fill the required unknown with the desired value(s). The concept of an algorithm is best illustrated by the example of a recipe, although many algorithms are much more complex; algorithms often have steps that repeat (iterate) or require decisions (such as logic or comparison) until the task is completed. Correctly performing an algorithm will not solve a problem if the algorithm is flawed or not appropriate to the problem. A recipe is a set of instructions that show how to prepare or make something, especially a culinary dish.

Different algorithms may complete the same task with a different set of instructions in more or less time, space, or effort than others. Algorithms are essential to the way computers process information, because a computer program is essentially an algorithm that tells the computer what specific steps to perform (in what specific order) in order to carry out a specified task.

Algorithmic problem solving comes in two phases. These include:

1. derivation of an algorithm that solves the problem, and
2. conversion of the algorithm into code.

It is worth noting that:

1. an algorithm is a sequence of steps, not a program.
2. same algorithm can be used in different programs, or the same algorithm can be expressed in different languages, because an algorithm is an entity that is abstracted from implementation details.

An algorithm can be expressed in the following ways:

a. human language
b. pseudo code
c. flow chart

Example 26.2.1

Problem: Given a list of positive numbers, return the largest number on the list.

Inputs: A list L of positive numbers. This list must contain at least one number.

Outputs: A number n, which will be the largest number of the list.

Algorithm

1. Set max to 0.
2. For each number x in the list L, compare it to max. If x is larger, set max to x.
3. max is now set to the largest number in the list.

Conclusion

The learner was introduced to the concept of algorithms and the various ways he/she can develop a solution to a task. In particular, the learner was introduced to the definition/s of an algorithm, the three main ways of developing or expressing an algorithm which are the human language, Pseudo code and the flow chart. An example was also given to reinforce the concept of the algorithm.

Assessment

1. Outline the algorithmic steps that can be used to add two given numbers.
2. By using an example, describe how the concept of algorithms can be well presented to a group of students being introduced to it.

This page titled 26.2: Activity 1 - Introduction to Algorithms and Problem Solving is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 1.1: Activity 1 - Introduction to Algorithms and Problem Solving is licensed CC BY-SA 4.0.

26.3: Activity 2 - The characteristics of an algorithm

Introduction

This section introduces the learners to the characteristics of algorithms. These characteristics make the learner become aware of what to ensure is basic, present and mandatory for any algorithm to qualify to be one. It also exposes the learner to what to expect from an algorithm to achieve or indicate. Key expectations are: the fact that an algorithm must be exact, terminate, effective, general among others.

Characteristics of an Algorithm

The following are some key characteristics of an algorithm.

1. Each step of an algorithm must be exact; this means that an algorithm must be precise and unambiguously described. This eliminates any uncertainty. It can also be said to the characteristic of precision, i.e. the steps are precisely stated (defined).
2. Algorithms must terminate; since the ultimate aim of an algorithm is to solve a problem, then it must terminate otherwise there won't be a solution for the problem. This leads to the fact that an algorithm must have a finite number of steps in its execution. The presence of endless (infinite) loops must be avoided.
3. An algorithm must be effective; this means that an algorithm must provide the correct answers at all times.
4. An algorithm must be general; this means that an algorithm must solve every instance of a problem.
5. Uniqueness: results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
6. Finiteness: the algorithm stops after a finite number of instructions are executed.
7. Output: the algorithm produces output.

Conclusion

This section has highlighted the properties of algorithms to the learner who is new to the concept of data structure and algorithm. It also alerts the learner on what to do to ensure the expected outcomes can be obtained when the algorithm is implemented in any computer programming language. This will ensure the learner comes up with very correct and accurate algorithms for solving any task at hand.

Assessment

1. With the aid of an appropriate example, explain the characteristics of algorithm.

This page titled 26.3: Activity 2 - The characteristics of an algorithm is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 1.2: Activity 2 - The characteristics of an algorithm is licensed CC BY-SA 4.0.

27: ADDITIONAL FUNDAMENTALS OF PROGRAMMING

27.1: Introduction

27.1.1: Computer Hardware Architecture

27.1.2: Writing a Program

27.1.E: Introduction (Exercises)

27.1.G: Introduction (Glossary)

27.2: Introduction to Programming

27.2.1: C++ Examples

27.2.2: C# Examples

27.2.3: Java Examples

27.2.4: JavaScript Examples

27.2.5: Swift Examples

27.2.6: Practice- Introduction to Programming

27.3: Data and Operators

27.3.1: JavaScript Examples

27.3.2: Swift Examples

27.3.3: Practice- Data and Operators

27.3.4: Nothing Data Type

27.3.5: Lvalue and Rvalue

27.3.6: Input-Process-Output Model

27.3.7: C++ Examples

27.3.8: C# Examples

27.3.9: Java Examples

27.4: Functions

27.4.1: Call by Value vs. Call by Reference

27.4.2: Void Data Type

27.4.3: C++ Examples

27.4.4: C# Examples

27.4.5: Java Examples

27.4.6: JavaScript Examples

27.4.7: Swift Examples

27.4.8: Practice- Functions

27.5: Conditions

27.5.1: Case Control Structure

27.5.2: C++ Examples

27.5.3: C# Examples

27.5.4: Java Examples

27.5.5: JavaScript Examples
27.5.6: Swift Examples
27.5.7: Practice- Conditions
27.5.8: Selection Control Structures

27.6: Loops

27.6.1: Do While Loop
27.6.2: Branching Statements
27.6.3: Integer Overflow
27.6.4: C++ Examples
27.6.5: C# Examples
27.6.6: Java Examples
27.6.7: JavaScript Examples
27.6.8: Swift Examples
27.6.9: Practice- Loops

27.7: Arrays and Lists

27.7.1: Arrays and Functions
27.7.2: Dynamic Arrays
27.7.3: C++ Examples
27.7.4: C# Examples
27.7.5: Java Examples
27.7.6: JavaScript Examples
27.7.7: Python Examples
27.7.8: Swift Examples
27.7.9: Practice- Arrays and Lists

27.8: Strings and Files

27.8.1: C++ Examples
27.8.2: C# Examples
27.8.3: Java Examples
27.8.4: JavaScript Examples
27.8.5: Swift Examples
27.8.6: Practice- Strings and Files

27.9: Object-Oriented Programming

27.9.1: Encapsulation
27.9.2: Inheritance and Polymorphism
27.9.3: C++ Examples
27.9.4: C# Examples
27.9.5: Java Examples
27.9.6: JavaScript Examples
27.9.7: Swift Examples

27.9.8: Practice

This page titled 27: Additional Fundamentals of Programming is shared under a license and was authored, remixed, and/or curated by LibreTexts.

27.1: INTRODUCTION

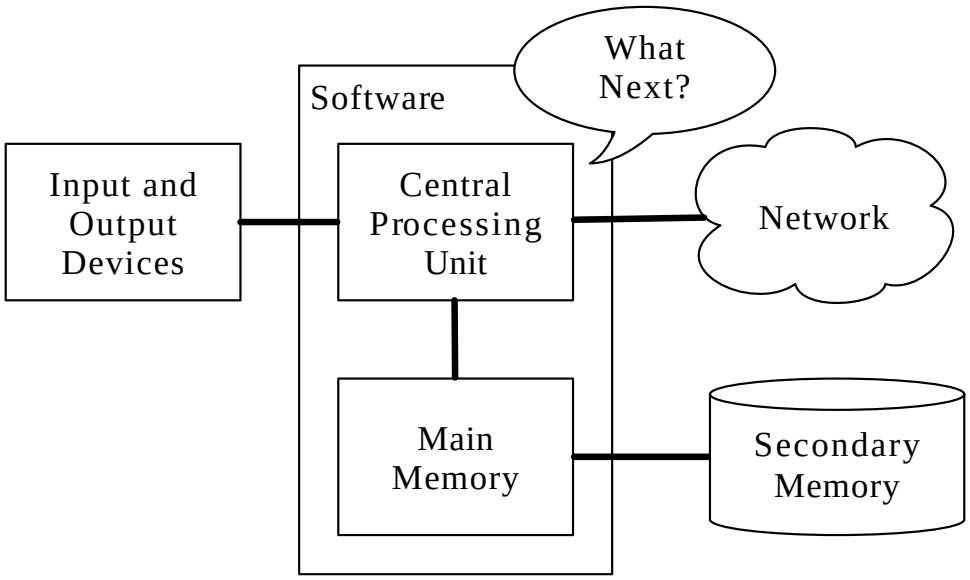
27.1.1: Computer Hardware Architecture
27.1.2: Writing a Program
27.1.E: Introduction (Exercises)
27.1.G: Introduction (Glossary)

This page titled 27.1: Introduction is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1: Introduction by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

27.1.1: Computer Hardware Architecture

Before we start learning the language we speak to give instructions to computers to develop software, we need to learn a small amount about how computers are built. If you were to take apart your computer or cell phone and look deep inside, you would find the following parts:



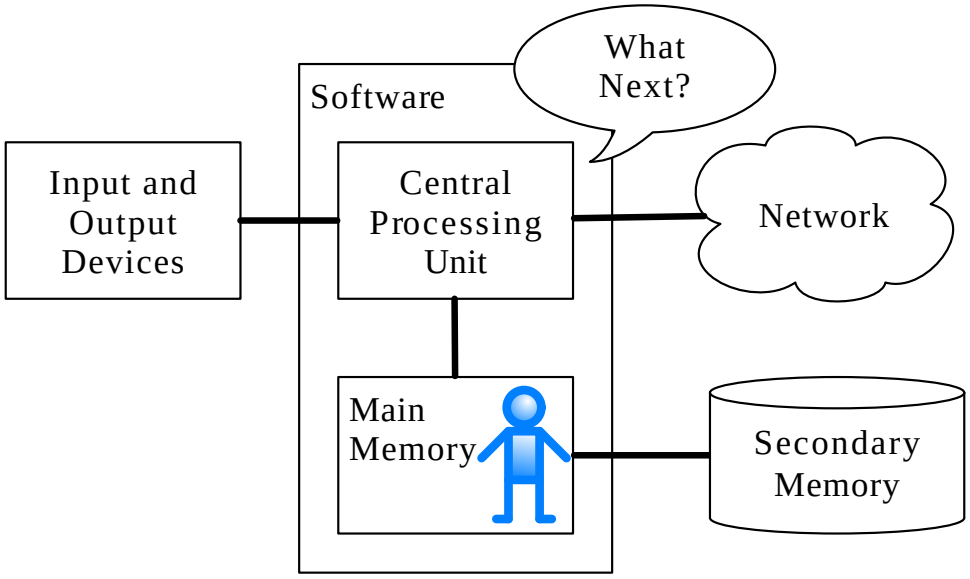
Hardware Architecture

The high-level definitions of these parts are as follows:

- The *Central Processing Unit* (or CPU) is the part of the computer that is built to be obsessed with "what is next?" If your computer is rated at 3.0 Gigahertz, it means that the CPU will ask "What next?" three billion times per second. You are going to have to learn how to talk fast to keep up with the CPU.
- The *Main Memory* is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. But the information stored in the main memory vanishes when the computer is turned off.
- The *Secondary Memory* is also used to store information, but it is much slower than the main memory. The advantage of the secondary memory is that it can store information even when there is no power to the computer. Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).
- The *Input and Output Devices* are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc. They are all of the ways we interact with the computer.
- These days, most computers also have a *Network Connection* to retrieve information over a network. We can think of the network as a very slow place to store and retrieve data that might not always be "up". So in a sense, the network is a slower and at times unreliable form of *Secondary Memory*.

While most of the detail of how these components work is best left to computer builders, it helps to have some terminology so we can talk about these different parts as we write our programs.

As a programmer, your job is to use and orchestrate each of these resources to solve the problem that you need to solve and analyze the data you get from the solution. As a programmer you will mostly be "talking" to the CPU and telling it what to do next. Sometimes you will tell the CPU to use the main memory, secondary memory, network, or the input/output devices.



Where Are You?

You need to be the person who answers the CPU's "What next?" question. But it would be very uncomfortable to shrink you down to 5mm tall and insert you into the computer just so you could issue a command three billion times per second. So instead, you must write down your instructions in advance. We call these stored instructions a *program* and the act of writing these instructions down and getting the instructions to be correct *programming*.

This page titled 27.1.1: Computer Hardware Architecture is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.3: Computer Hardware Architecture by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

27.1.2: Writing a Program

Typing commands into the Python interpreter is a great way to experiment with Python's features, but it is not recommended for solving more complex problems.

When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a *script*. By convention, Python scripts have names that end with `.py`.

To execute the script, you have to tell the Python interpreter the name of the file. In a Unix or Windows command window, you would type `python hello.py` as follows:

```
csev$ cat hello.py
print('Hello world!')
csev$ python hello.py
Hello world!
csev$
```

The "csev$" is the operating system prompt, and the "cat hello.py" is showing us that the file "hello.py" has a one-line Python program to print a string.

We call the Python interpreter and tell it to read its source code from the file "hello.py" instead of prompting us for lines of Python code interactively.

You will notice that there was no need to have *quit()* at the end of the Python program in the file. When Python is reading your source code from a file, it knows to stop when it reaches the end of the file.

This page titled 27.1.2: Writing a Program is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.8: Writing a Program by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

27.1.E: Introduction (Exercises)

Exercise 1: What is the function of the secondary memory in a computer?

a) Execute all of the computation and logic of the program
b) Retrieve web pages over the Internet
c) Store information for the long term, even beyond a power cycle
d) Take input from the user

Exercise 2: What is a program?

Exercise 3: What is the difference between a compiler and an interpreter?

Exercise 4: Which of the following contains "machine code"?

a) The Python interpreter
b) The keyboard
c) Python source file
d) A word processing document

Exercise 5: What is wrong with the following code:

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                        ^
SyntaxError: invalid syntax
>>>
```

Exercise 6: Where in the computer is a variable such as "x" stored after the following Python line finishes?

```
x = 123
```

a) Central processing unit
b) Main Memory
c) Secondary Memory
d) Input Devices
e) Output Devices

Exercise 7: What will the following program print out:

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) Error because x = x + 1 is not possible mathematically

Exercise 8: Explain each of the following using an example of a human capability: (1) Central processing unit, (2) Main Memory, (3) Secondary Memory, (4) Input Device, and (5) Output Device. For example, "What is the human equivalent to a Central Processing Unit"?

1. http://xkcd.com/231/↩

This page titled 27.1.E: Introduction (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 1.E: Introduction (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

bug
An error in a program.

central processing unit
The heart of any computer. It is what runs the software that we write; also called "CPU" or "the processor".

compile
To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

high-level language
A programming language like Python that is designed to be easy for humans to read and write.

interactive mode
A way of using the Python interpreter by typing commands and expressions at the prompt.

interpret
To execute a program in a high-level language by translating it one line at a time.

low-level language
A programming language that is designed to be easy for a computer to execute; also called "machine code" or "assembly language".

machine code
The lowest-level language for software, which is the language that is directly executed by the central processing unit (CPU).

main memory
Stores programs and data. Main memory loses its information when the power is turned off.

parse
To examine a program and analyze the syntactic structure.

portability
A property of a program that can run on more than one kind of computer.

print function
An instruction that causes the Python interpreter to display a value on the screen.

problem solving
The process of formulating a problem, finding a solution, and expressing the solution.

program
A set of instructions that specifies a computation.

prompt
When a program displays a message and pauses for the user to type some input to the program.

secondary memory
Stores programs and data and retains its information even when the power is turned off. Generally slower than main memory. Examples of secondary memory include disk drives and flash memory in USB sticks.

semantics
The meaning of a program.

semantic error
An error in a program that makes it do something other than what the programmer intended.

source code
A program in a high-level language.

This page titled 27.1.G: Introduction (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

27.2: INTRODUCTION TO PROGRAMMING

This chapter introduces programming, the software development process, tools and methods used to develop and test programs. These include integrated development environments (IDEs), version control, input and output, and a Hello World program in pseudocode and flowchart format. The programming languages C++, C#, Java, JavaScript, Python, and Swift are introduced with example code.

27.2.1: C++ Examples

27.2.2: C# Examples

27.2.3: Java Examples

27.2.4: JavaScript Examples

27.2.5: Swift Examples

27.2.6: Practice- Introduction to Programming

This page titled 27.2: Introduction to Programming is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 1: Introduction to Programming is licensed CC BY-SA 4.0.

27.2.1: C++ Examples



Overview

C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation. C++ was developed by Bjarne Stroustrup at Bell Labs starting in 1979 as an extension of the C language. The C++ programming language was initially standardized in 1998.[1]

C++ is one of the most popular current programming languages[2] and is often used in computer science courses.

Example

Hello World

```
01 // This program displays "Hello world!"
02 //
03 // References:
04 // http://www.cplusplus.com/doc/tutorial/program_structure/
05
06 #include <iostream>
07
08 int main()
09 {
10     std::cout << "Hello world!";
11 }
```

Output

```
01 Hello world!
```

Discussion

Each code element represents:[3]

- `//` begins a comment
- `#include <iostream>` includes standard input and output streams
- `int main()` begins the main function, which returns an integer value
- `{` begins a block of code
- `std::cout` is standard output
- `<<` directs the next element to standard output
- `"Hello world!"` is the literal string to be displayed

- ; ends each line of C++ code
- } ends a block of code

C++ IDEs

There are many free cloud-based and local IDEs available to begin coding in C++. Check with your instructor or do your own research for recommendations.

Cloud-Based IDEs

- CodeChef
- GDB Online
- Ideone
- paiza.IO
- PythonTutor
- repl.it
- TutorialsPoint

Local IDEs

- Code::Blocks
- Dev-C++
- Microsoft Visual Studio

References

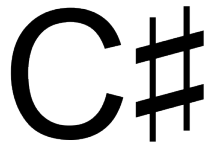
- Wikiversity: Computer Programming

---

1. Wikipedia: C++ ↩
2. TIOBE: Index ↩
3. Wikibooks: Programming Fundamentals/Hello World ↩

This page titled 27.2.1: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.2.2: C# Examples



Overview

C# is a general-purpose, object-oriented programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed around 2000 by Microsoft within its .NET initiative and later approved as a standard by Ecma (ECMA-334) and ISO (ISO/IEC 23270:2006). C# is one of the programming languages designed for the Common Language Infrastructure.

C# is one of the most popular current programming languages[1], is the primary language for Windows application development and is often used in computer science and gaming courses.

Example

Hello World

```
01 // This program displays "Hello world!"
02 //
03 // References:
04 // https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/hello-
   world-your-first-program
05
06 public class Hello
07 {
08     public static void Main()
09     {
10         System.Console.WriteLine("Hello world!");
11     }
12 }
```

Output

```
01 Hello world!
```

Discussion

Each code element represents:[2]

- `//` begins a comment
- `public class Hello` begins the Hello World program
- `{` begins a block of code
- `public static void Main()` begins the main function
- `System.Console.WriteLine()` calls the standard output write line function
- `"Hello world!"` is the literal string to be displayed
- `;` ends each line of C# code

- } ends a block of code

C# IDEs

There are many free cloud-based and local IDEs available to begin coding in C#. Check with your instructor or do your own research for recommendations.

Cloud-Based IDEs

- CodeChef
- C# Pad
- .NET Fiddle
- Ideone
- paiza.IO
- Rextester
- repl.it
- TutorialsPoint

Local IDEs

- Microsoft Visual Studio
- Visual Studio Code

References

- Wikiversity: Computer Programming

---

1. TIOBE: Index ↩
2. Wikibooks: Programming Fundamentals/Hello World ↩

This page titled 27.2.2: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.2.3: Java Examples



Overview

Java is a general-purpose computer-programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers “write once, run anywhere” (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java was originally developed by James Gosling at Sun Microsystems and released in 1995.

Java is one of the most popular current programming languages[1] and is often used in computer science courses.

Example

Hello World

```
01 // This program displays "Hello world!"
02 //
03 // References:
04 // https://introcs.cs.princeton.edu/java/11hello/HelloWorld.java.html
05
06 class Main {
07     public static void main(String[] args) {
08         System.out.println("Hello world!");
09     }
10 }
```

Output

```
01 Hello world!
```

Discussion

Each code element represents:[2]

- `//` begins a comment
- `class hello` begins the Hello World program
- `{` begins a block of code
- `public static void main(String[] args)` begins the main function
- `System.out.println()` calls the standard output print line function

- `"Hello world!"` is the literal string to be displayed
- `;` ends each line of Java code
- `}` ends a block of code

Java IDEs

There are many free cloud-based and local IDEs available to begin coding in Java. Check with your instructor or do your own research for recommendations.

Cloud-Based IDEs

- CodeChef
- GDB Online
- Ideone
- paiza.IO
- PythonTutor
- repl.it
- TutorialsPoint

Local IDEs

- BlueJ
- jEdit
- jGRASP

References

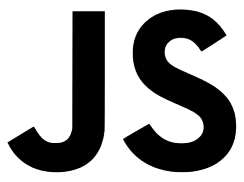
- Wikiversity: Computer Programming

---

1. TIOBE: Index ↩
2. Wikibooks: Programming Fundamentals/Hello World ↩

This page titled 27.2.3: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.2.4: JavaScript Examples



Overview

JavaScript, often abbreviated as JS, is a high-level, interpreted programming language. Alongside HTML and CSS, JavaScript is one of the three core technologies of the World Wide Web. JavaScript enables interactive web pages and therefore is an essential part of web applications. The vast majority of websites use it, and all major web browsers have a dedicated JavaScript engine to execute it.[1]

JavaScript is one of the most popular current programming languages[2], and is the primary programming language for front-end web development. JavaScript has been implemented in multiple platforms with different I/O commands. Several examples follow.

Example

Hello World – Console Log

```
01 // This script displays "Hello world!".
02 //
03 // References:
04 // https://www.digitalocean.com/community/tutorials/how-to-write-your-first-javascript-program
05
06 console.log("Hello world!")
```

Output

```
01 Hello world!
```

Discussion

Each code element represents:

- `//` begins a comment
- `console.log()` writes to the JavaScript console output log
- `"Hello world!"` is the literal string to be displayed

Hello World – Window Alert

```
01 // This script displays "Hello world!".
02 //
03 // References:
04 // https://www.digitalocean.com/community/tutorials/how-to-write-your-first-javascript-program
05
```

```
06 alert("Hello world!")
```

Output

```
01 Hello world!
```

Discussion

Each code element represents:

- `//` begins a comment
- `alert()` calls the window alert function to display a message
- `"Hello world!"` is the literal string to be displayed

Hello World – Document Write

```
01 // This script displays "Hello world!".
02 //
03 // References:
04 // https://www.w3schools.com/jsref/met_doc_write.asp
05 document.write("Hello world!")
```

Output

```
01 Hello world!
```

Discussion

Each code element represents:

- `//` begins a comment
- `document.write()` writes output to the current document
- `"Hello world!"` is the literal string to be displayed

JavaScript IDEs

There are many free cloud-based and local IDEs available to begin coding in JavaScript. Check with your instructor or do your own research for recommendations.

Cloud-Based IDEs

- Chapman.edu: Online JavaScript Interpreter
- CodeChef
- GDB Online
- Ideone
- paiza.IO
- PythonTutor
- repl.it

Local IDEs

- Brackets
- Visual Studio Code

References

- Wikiversity: Computer Programming

---

1. Wikipedia: JavaScript ↵
2. TIOBE: Index ↵

This page titled 27.2.4: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.2.5: Swift Examples

File:Swift logo with text.svg Overview

Swift is a general-purpose, multi-paradigm, compiled programming language developed by Apple Inc. for iOS, macOS, watchOS, tvOS, and Linux. Apple intended Swift to support many core concepts associated with Objective-C, but in a “safer” way, making it easier to catch software bugs. Swift was introduced in 2014.[1]

Swift is a popular programming language for the Apple platforms it supports, but it lacks support for Microsoft Windows environments.[2].

Example

Hello World

```
01 // This program displays "Hello world!"
02 //
03 // References:
04 //     https://developer.apple.com/library/content/documentation/Swift/Conceptual/
   Swift_Programming_Language/GuidedTour.html
05
06 print("Hello world!")
```

Output

```
01 Hello world!
```

Discussion

Each code element represents:[3]

- `//` begins a comment
- `print()` calls the print function
- `"Hello world!"` is the literal string to be displayed

Swift IDEs

There are several free cloud-based and local IDEs available to begin coding in Swift. Check with your instructor or do your own research for recommendations.

Cloud-Based IDEs

- GDB Online
- IBM Swift Sandbox
- Ideone
- iSwift
- paiza.IO

- repl.it

Local IDEs

- AppCode
- Atom
- Xcode

References

- Wikiversity: Computer Programming

---

1. Swift (programming language) ↵
2. TIOBE: Index ↵
3. Wikibooks: Programming Fundamentals/Hello World ↵

This page titled 27.2.5: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.2.6: Practice- Introduction to Programming

Review Questions

True / False:

1. Beginning programmers participate in all phases of the Systems Development Life Cycle.
2. Coding the program in a language like C++ is the first task of planning. You plan as you code.
3. Pseudocode is the only commonly used planning tool.
4. Pseudocode has a strict set of rules and is the same everywhere in the computer programming industry.
5. Test data is developed for testing the program once it is code into a language like C++.
6. The word pseudo means false and includes the concepts of fake or imitation.
7. Many programmers pick up the bad habit of not completing the planning step before starting to code the program.
8. IDE means Integer Division Expression.
9. Most modern compilers are really an IDE type of software, not just a compiler.
10. Programming errors are extremely easy to understand and fix.

Answers:

1. false
2. false
3. false
4. false
5. false
6. true
7. true
8. false
9. true
10. false

Short Answer:

1. List the steps of the Systems Development Life Cycle and indicate which step you are likely to work in as a new computer professional.
2. List and describe what might cause the four (4) types of errors encountered in a program using a compiler and an Integrated Development Environment software product.

Activities

Pseudocode and Flowcharts

The following activities focus on software planning and testing using pseudocode and / or flowcharts.

1. Search the Internet for pseudocode for making a peanut butter and jelly sandwich. Based on the examples you find, create pseudocode to make your own favorite sandwich or other prepared meal. Test your pseudocode by reading the instructions out loud as someone else follows your directions.
2. Search the Internet for a flowchart for making a peanut butter and jelly sandwich. Use a free online or downloadable flowchart tool to create a flowchart that describes how to make your favorite sandwich or other prepared meal. Test your flowchart by reading the instructions out loud while someone else follows your directions.
3. Create pseudocode or a flowchart for a program that would interact with bank customers and help them determine the value of a bag or jar of coins brought in for deposit. Include counts for pennies, nickels, dimes and quarters and calculate the total value of all of the coins deposited. Test your program by having someone else follow the instructions and guide them as they use your program.
4. Create pseudocode or a flowchart for a program that allows the user to enter gallons of gas and converts it to liters (metric

system). NOTE: One US gallon equals 3.7854 liters. Test your program by having someone else follow the instructions and guide them as they use your program.
5. A major restaurant sends a chef to purchase fruits and vegetables every day. Upon returning to the store the chef must enter two pieces of data for each item purchased: the quantity (Example: 2 cases) and the price paid (Example: $4.67). The program has a list of 20 items and after the chef enters the information, the program provides a total for the purchases for that day. Prepare test data for five (5) items: apples, oranges, bananas, lettuce, and tomatoes.

Programming Languages and Integrated Development Environments

The following activities focus on selecting a programming language and testing integrated development environments.

1. Research different programming languages and select a programming language to use with this textbook. Copy the Hello World example code for your selected programming language and use one of the free cloud-based IDEs to try running the Hello World program.
2. Modify the example Hello World program to instead display `Hello <name>!`, where `<name>` is your name. Include comments at the top of the program and test the program to verify that it works correctly.
3. Research free downloadable tools for your selected programming language (interpreter/compiler, IDE, etc.). Consider downloading and installing a development environment on your system. If you set up your own development environment, test the environment using your Hello Name program written above.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

This page titled 27.2.6: Practice- Introduction to Programming is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3: DATA AND OPERATORS

27.3.1: JavaScript Examples
27.3.2: Swift Examples
27.3.3: Practice- Data and Operators
27.3.4: Nothing Data Type
27.3.5: Lvalue and Rvalue
27.3.6: Input-Process-Output Model
27.3.7: C++ Examples
27.3.8: C# Examples
27.3.9: Java Examples

This page titled 27.3: Data and Operators is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 2: Data and Operators is licensed CC BY-SA 4.0.

27.3.1: JavaScript Examples

Overview

The following examples demonstrate data types, arithmetic operations, and input in JavaScript.

Data Types

```
01 // This program demonstrates variables, literal constants, and data types.
02
03 var n;
04 var s;
05 var b;
06
07 n = 1.23456789012345;
08 s = "string";
09 b = true;
10
11 output("Number n = " + n);
12 output("String s = " + s);
13 output("Boolean b = " + b);
14
15 // Check the JavaScript environment and write to the console,
16 // the current document, or standard output as appropriate.
17 // Reference: http://progopedia.com/example/hello-world/114/
18 function output(text) {
19   if (typeof console === 'object') {
20     console.log(text);
21   }
22   else if (typeof document === 'object') {
23     document.write(text);
24   }
25   else {
26     print(text);
27   }
28 }
```

Output

```
01 Number n = 1.23456789012345
02 String s = string
03 Boolean b = true
```

Discussion

Each code element represents:

- `//` begins a comment
- `var n, s, and b` define variables
- `;` ends each line of JavaScript code
- `i = , d = , s =, b =` assign literal values to the corresponding variables
- `output()` calls the output function

- `function output(text)` defines a output function that checks the JavaScript environment and writes to the console, the current document, or standard output as appropriate.

Arithmetic

```
01 // This program demonstrates arithmetic operations.
02
03 var a;
04 var b;
05
06 a = 3;
07 b = 2;
08 output("a = " + a);
09 output("b = " + b);
10 output("a + b = " + (a + b));
11 output("a - b = " + (a - b));
12 output("a * b = " + a * b);
13 output("a / b = " + a / b);
14 output("a % b = " + (a % b));
15
16 // Checks the JavaScript environment and writes to the console,
17 // the current document, or standard output as appropriate.
18 // Reference: http://progopedia.com/example/hello-world/114/
19 function output(text) {
20   if (typeof console === 'object') {
21     console.log(text);
22   }
23   else if (typeof document === 'object') {
24     document.write(text);
25   }
26   else {
27     print(text);
28   }
29 }
```

Output

```
01 a = 3
02 b = 2
03 a + b = 5
04 a - b = 1
05 a * b = 6
06 a / b = 1.5
07 a % b = 1
```

Discussion

Each new code element represents:

- `+, -, *, /, and %` represent addition, subtraction, multiplication, division, and modulus, respectively.

Temperature

```
01 // This program converts an input Fahrenheit temperature to Celsius.
```

```
02
03 var fahrenheit;
04 var celsius;
05
06 output("Enter Fahrenheit temperature:");
07 fahrenheit = input();
08
09 celsius = (fahrenheit - 32) * 5 / 9;
10
11 output(fahrenheit.toString() + "° Fahrenheit is " + celsius + "° Celsius");
12
13 // Checks the JavaScript environment and reads from the console,
14 // the current document, or standard input as appropriate.
15 function input(text) {
16   if (typeof console === 'object') {
17     return prompt(text)
18   }
19   else {
20     output(text);
21     var isr = new java.io.InputStreamReader(java.lang.System.in);
22     var br = new java.io.BufferedReader(isr);
23     var line = br.readLine();
24     return line.trim();
25   }
26 }
27
28 // Checks the JavaScript environment and writes to the console,
29 // the current document, or standard output as appropriate.
30 // Reference: http://progopedia.com/example/hello-world/114/
31 function output(text) {
32   if (typeof console === 'object') {
33     console.log(text);
34   }
35   else if (typeof document === 'object') {
36     document.write(text);
37   }
38   else {
39     print(text);
40   }
41 }
```

Output

```
01 Enter Fahrenheit temperature:
02  100
03 100° Fahrenheit is 37.7777777777778° Celsius
```

Discussion

Each new code element represents:

- `function input(text)` defines a function that checks the JavaScript environment and reads from the console, the current document, or standard input as appropriate.

References

- Wikiversity: Computer Programming

This page titled 27.3.1: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.2: Swift Examples

Overview

The following examples demonstrate data types, arithmetic operations, and input in Swift.

Data Types

```
01 // This program demonstrates variables, literal constants, and data types.
02
03 var i: Int
04 var d: Double
05 var s: String
06 var b: Bool
07
08 i = 1234567890
09 d = 1.23456789012345
10 s = "string"
11 b = true
12
13 print("Integer i =", i)
14 print("Double d =", d)
15 print("String s =", s)
16 print("Boolean b =", b)
```

Output

```
01 Integer i = 1234567890
02 Double d = 1.23456789012345
03 String s = string
04 Boolean b = true
```

Discussion

Each code element represents:

- `//` begins a comment
- `var i: Int` defines an integer variable named i
- `var d: Double` defines a double floating-point variable named d
- `var s: String` defines a string variable named s
- `var b: Bool` defines a Boolean variable named b
- `i = , d = , s =, b =` assign literal values to the corresponding variables
- `print()` calls the print function

Arithmetic

```
01 // This program demonstrates arithmetic operations.
02
03 var a: Int
04 var b: Int
```

```
05
06 a = 3
07 b = 2
08
09 print("a =", a)
10 print("b =", b)
11 print("a + b =", (a + b))
12 print("a - b =", (a - b))
13 print("a * b =", a * b)
14 print("a / b =", a / b)
15 print("a % b =", (a % b))
```

Output

```
01 a = 3
02 b = 2
03 a + b = 5
04 a - b = 1
05 a * b = 6
06 a / b = 1
07 a % b = 1
```

Discussion

Each new code element represents:

- `+, -, *, /, and %` represent addition, subtraction, multiplication, division, and modulus, respectively.

Temperature

```
01 // This program converts a Fahrenheit temperature to Celsius.
02 //
03 // References:
04 //     https://www.mathsisfun.com/temperature-conversion.html
05 //     https://developer.apple.com/library/content/documentation/Swift/Conceptual/
   Swift_Programming_Language/TheBasics.html
06
07 var fahrenheit: Double
08 var celsius: Double
09
10 print("Enter Fahrenheit temperature:")
11 fahrenheit = Double(readLine()!)!
12
13 celsius = (fahrenheit - 32) * 5 / 9
14
15 print(String(fahrenheit) + "° Fahrenheit is " + String(celsius) + "° Celsius")
```

Output

```
01 Enter Fahrenheit temperature:
02  100
03 100.0° Fahrenheit is 37.7777777777778° Celsius
```

Discussion

Each new code element represents:

- `readline()!` reads the next line from standard input
- `Double()!` converts the input to a double floating-point value
- `String()` converts the output numeric value to a string

References

- Wikiversity: Computer Programming

This page titled 27.3.2: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.3: Practice- Data and Operators

Review Questions

True or false:

1. A data type defines a set of values and the set of operations that can be applied to those values.
2. Reserved or key words can be used as identifier names.
3. The concept of precedence says that some operators (like multiplication and division) are to be executed before other operators (like addition and subtraction).
4. An operator that needs two operands, will promote one of the operands as needed to make both operands be of the same data type.
5. Parentheses change the precedence of operators.
6. Integer data types are stored with a mantissa and an exponent.
7. Strings are identified by single quote marks in most programming languages.
8. An operand is a value that receives the operator's action.
9. Arithmetic assignment is a shorter way to write some expressions.
10. Integer division is rarely used in computer programming.

Answers:

1. true
2. false
3. true
4. true
5. false – Parentheses change the order of evaluation in an expression.
6. false
7. false
8. true
9. true
10. false

Short Answer:

1. A men's clothing store that caters to the very rich wants to create a database for its customers that records clothing measurements. They need to record information for shoes, socks, pants, dress shirts and casual shirts. HINT: You may need more than 5 data items.
2. The sequence operator can be used when declaring multiple identifier names for variables or constants of the same data type. Is this a good or bad programming habit and why?

Activities

Complete the following activities using pseudocode, a flowcharting tool, or your selected programming language. Use appropriate data types for each variable, and include separate statements for input, processing, and output. Create test data to validate the accuracy of each program. Add comments at the top of the program and include references to any resources used.

1. Create a program to prompt the user for hours and rate per hour and then calculate and display their weekly, monthly, and annual gross pay (hours * rate).[1].
2. Create a program that asks the user how old they are in years, and then calculate and display their approximate age in months, days, hours, and seconds. For example, a person 1 year old is 12 months old, 365 days old, etc.
3. Review MathsIsFun: US Standard Lengths. Create a program that asks the user for a distance in miles, and then calculate and display the distance in yards, feet, and inches, or ask the user for a distance in miles, and then calculate and display the distance in kilometers, meters, and centimeters.
4. Review MathsIsFun: Area of Plane Shapes. Create a program that asks the user for the dimensions of different shapes

and then calculate and display the area of the shapes. Do not include shape choices. That will come later. For now, just include multiple shape calculations in sequence.

5. Create a program that calculates the area of a room to determine the amount of floor covering required. The room is rectangular with the dimensions measured in feet with decimal fractions. The output needs to be in square yards. There are 3 linear feet to a yard.
6. Create a program that helps the user determine how much paint is required to paint a room and how much it will cost. Ask the user for the length, width, and height of a room, the price of a gallon of paint, and the number of square feet that a gallon of paint will cover. Calculate the total area of the four walls as `2 * length * height + 2 * width * height` Calculate the number of gallons as: `total area / square feet per gallon` Note: You must round up to the next full gallon. To round up, add `0.9999` and then convert the resulting value to an integer. Calculate the total cost of the paint as: `gallons * price per gallon`.
7. Review MathsIsFun: Order of Operations. Create a program that demonstrates order of operations to the user. Include parentheses, exponents, multiplication, division, addition, and subtraction in your program. Use variables for the calculations and label the output. For example, part of the program might display:
   ```
   1 + 2 * 3 = 7
   (1 + 2) * 3 = 9
   ...
   ```
8. Review Wikipedia: Data type. Create a program that demonstrates integer, floating point, and character or string data, and demonstrate converting between data types. For example, user input is always a string, but adding string values of "1" + "1" is typically "11", whereas, adding numeric values of 1 + 1 is 2. Use variables for the calculations and label the output.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

---

1. PythonLearn: Variables, expressions, and statements ↵

This page titled 27.3.3: Practice- Data and Operators is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.4: Nothing Data Type

Overview

A **nothing data type** is a feature of some programming languages which allow the setting of a special value to indicate a missing or uninitialized value rather than using the value 0 (zero).[1]

Discussion

Most programming languages support the use of a reserved word or words to represent missing, uninitialized, or invalid values.

Language	Reserved Word	Meaning
C++	`null`	no value
C#	`null`	no value
Java	`null`	no value
JavaScript	`null`	no value
JavaScript	`NaN`	Not a Number
Python	`None`	no value
Swift	`nil`	no value

Key Terms

NaN
: Reserved word used to indicate a non-numeric value in a numeric variable.

null
: Reserved word used to represent a missing value or invalid value.

---

1. Wikipedia: Nullable type ↩

This page titled 27.3.4: Nothing Data Type is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.5: Lvalue and Rvalue

Overview

Some programming languages use the idea of **l-values** and **r-values**, deriving from the typical mode of evaluation on the left and right hand side of an assignment statement. An lvalue refers to an object that persists beyond a single expression. An rvalue is a temporary value that does not persist beyond the expression that uses it.[1].

Discussion

Lvalue and Rvalue refer to the left and right side of the assignment operator. The **Lvalue** (pronounced: L value) concept refers to the requirement that the operand on the left side of the assignment operator is modifiable, usually a variable. **Rvalue** concept pulls or fetches the value of the expression or operand on the right side of the assignment operator. Some examples:

```
01 age = 39
```

The value 39 is pulled or fetched (Rvalue) and stored into the variable named age (Lvalue); destroying the value previously stored in that variable.

```
01 voting_age = 18
02 age = voting_age
```

If the expression has a variable or named constant on the right side of the assignment operator, it would pull or fetch the value stored in the variable or constant. The value 18 is pulled or fetched from the variable named voting_age and stored into the variable named age.

```
01 age < 17
```

If the expression is a test expression or Boolean expression, the concept is still an Rvalue one. The value in the identifier named age is pulled or fetched and used in the relational comparison of less than.

```
01 JACK_BENNYS_AGE = 39
02 JACK_BENNYS_AGE = 65;
```

This is illegal because the identifier JACK_BENNYS_AGE does not have Lvalue properties. It is not a modifiable data object, because it is a constant.

Some uses of the Lvalue and Rvalue can be confusing in languages that support increment and decrement operators. Consider:

```
01 oldest = 55
02 age = oldest++
```

Postfix increment says to use my existing value then when you are done with the other operators; increment me. Thus, the first use of the oldest variable is an Rvalue context where the existing value of 55 is pulled or fetched and then assigned to the variable age; an Lvalue context. The second use of the oldest variable is an Lvalue context wherein the value of the oldest is incremented from 55 to 56.

Key Terms

Lvalue
: The requirement that the operand on the left side of the assignment operator is modifiable, usually a variable.

Rvalue
: Pulls or fetches the value stored in a variable or constant.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Value (computer science) ↩

This page titled 27.3.5: Lvalue and Rvalue is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.6: Input-Process-Output Model

Overview

The **input–process–output (IPO) model** is a widely used approach in systems analysis and software engineering for describing the structure of an information processing program or another process. Many introductory programming and systems analysis texts introduce this as the most basic structure for describing a process.[1]

Discussion

A computer program or any other sort of process using the input-process-output model receives inputs from a user or other source, does some computations on the inputs, and returns the results of the computations. The system divides the work into three categories:[2]

- A requirement from the environment (input)
- A computation based on the requirement (process)
- A provision for the environment (output)

For example, a program might be written to convert Fahrenheit temperatures into Celsius temperatures. Following the IPO model, the program must:

- Ask the user for the Fahrenheit temperature (input)
- Perform a calculation to convert the Fahrenheit temperature into the corresponding Celsius temperature (process)
- Display the Celsius temperature (output)

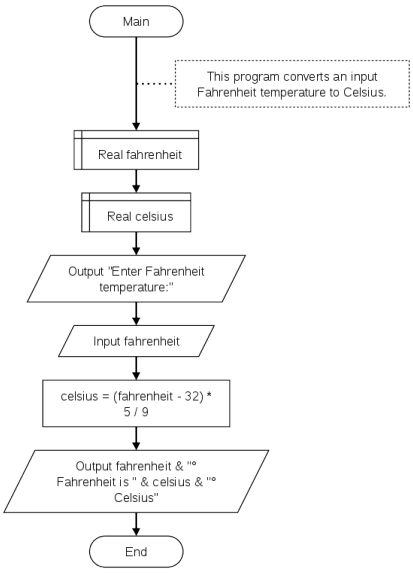
Pseudocode

```
01 Function Main
02     ... This program converts an input Fahrenheit temperature to Celsius.
03
04     Declare Real fahrenheit
05     Declare Real celsius
06
07     Output "Enter Fahrenheit temperature:"
08     Input fahrenheit
09
10     Assign celsius = (fahrenheit - 32) * 5 / 9
11
12     Output fahrenheit & "° Fahrenheit is " & celsius & "° Celsius"
13 End
```

Output

```
01 Enter Fahrenheit temperature:
02 100
03 100° Fahrenheit is 37.7777777777778° Celsius
```

Flowchart



References

- Wikiversity: Computer Programming
- Flowgorithm – Flowchart Programming Language

---

1. Wikipedia: IPO model ↵
2. Wikipedia: IPO model ↵

This page titled 27.3.6: Input-Process-Output Model is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.7: C++ Examples

Overview

The following examples demonstrate data types, arithmetic operations, and input in C++.

Data Types

```
01 // This program demonstrates variables, literal constants, and data types.
02
03 #include <iostream>
04 #include <sstream>
05
06 using namespace std;
07
08 int main() {
09     int i;
10     double d;
11     string s;
12     bool b;
13
14     i = 1234567890;
15     d = 1.23456789012345;
16     s = "string";
17     b = true;
18     cout << "Integer i = " << i << endl;
19     cout << "Double d = " << d << endl;
20     cout << "String s = " << s << endl;
21     cout << "Boolean b = " << b << endl;
22     return 0;
23 }
```

Output

```
01 Integer i = 1234567890
02 Real r = 1.23457
03 String s = string
04 Boolean b = 1
```

Discussion

Each code element represents:

- `//` begins a comment
- `#include <iostream>` includes standard input and output streams
- `#include <sstream>` includes standard string streams
- `using namespace std` allows reference to `string`, `cout`, and `endl` without writing `std::string`, `std::cout`, and `std::endl`.
- `int main()` begins the main function, which returns an integer value
- `{` begins a block of code
- `int i` defines an integer variable named i
- `;` ends each line of C++ code

- `double d` defines a double floating-point variable named d
- `string s` defines a string variable named s
- `bool b` defines a Boolean variable named b
- `i = , d = , s =, b =` assign literal values to the corresponding variables
- `cout` is standard output
- `<<` directs the next element to standard output
- `endl` ends the current line
- `return 0` returns the value 0 from main, indicating the main function completed successfully
- `}` ends a block of code

Arithmetic

```
01 // This program demonstrates arithmetic operations.
02
03 #include <iostream>
04 #include <sstream>
05
06 using namespace std;
07
08 int main() {
09     int a;
10     int b;
11
12     a = 3;
13     b = 2;
14
15     cout << "a = " << a << endl;
16     cout << "b = " << b << endl;
17     cout << "a + b = " << a + b << endl;
18     cout << "a - b = " << a - b << endl;
19     cout << "a * b = " << a * b << endl;
20     cout << "a / b = " << a / b << endl;
21     cout << "a % b = " << a + b << endl;
22     return 0;
23 }
```

Output

```
01 a = 3
02 b = 2
03 a + b = 5
04 a - b = 1
05 a * b = 6
06 a / b = 1
07 a % b = 5
```

Discussion

Each new code element represents:

- `+, -, *, /, and %` represent addition, subtraction, multiplication, division, and modulus, respectively.

Temperature

```
// This program converts an input Fahrenheit temperature to Celsius.
//
// References:
// https://www.mathsisfun.com/temperature-conversion.html
// https://en.wikibooks.org/wiki/C%2B%2B_Programming
#include <iostream>

using namespace std;

int main() {
    double fahrenheit;
    double celsius;

    cout << "Enter Fahrenheit temperature:" << endl;
    cin >> fahrenheit;

    celsius = (fahrenheit - 32) * 5 / 9;

    cout << fahrenheit << "° Fahrenheit is " << celsius << "° Celsius" << endl;

    return 0;
}
```

Output

```
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7778° Celsius
```

Discussion

Each new code element represents:

- `cin >> fahrenheit` reads the next integer from standard input and assigns the value to the fahrenheit variable

References

- Wikiversity: Computer Programming

This page titled 27.3.7: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.8: C# Examples

Overview

The following examples demonstrate data types, arithmetic operations, and input in C#.

Data Types

```
// This program demonstrates variables, literal constants, and data types.

using System;

public class DataTypes
{
    public static void Main(string[] args)
    {
        int i;
        double d;
        string s;
        Boolean b;

        i = 1234567890;
        d = 1.23456789012345;
        s = "string";
        b = true;

        Console.WriteLine("Integer i = " + i);
        Console.WriteLine("Double d = " + d);
        Console.WriteLine("String s = " + s);
        Console.WriteLine("Boolean b = " + b);
    }
}
```

Output

```
Integer i = 1234567890
Double d = 1.23456789012345
String s = string
Boolean b = True
```

Discussion

Each code element represents:

- `//` begins a comment
- `using System` allows references to `Boolean` and `Console` without writing `System.Boolean` and `System.Console`
- `public class DataTypes` begins the Data Types program
- `{` begins a block of code
- `public static void Main()` begins the main function
- `int i` defines an integer variable named i
- `;` ends each line of C# code
- `double d` defines a double floating-point variable named d

- `string s` defines a string variable named s
- `Boolean b` defines a Boolean variable named b
- `i = , d = , s =, b =` assign literal values to the corresponding variables
- `Console.WriteLine()` calls the standard output write line function
- `}` ends a block of code

Arithmetic

```
01 // This program demonstrates arithmetic operations.
02
03 using System;
04
05 public class Arithmetic
06 {
07     public static void Main(string[] args)
08     {
09         int a;
10         int b;
11
12         a = 3;
13         b = 2;
14
15         Console.WriteLine("a = " + a);
16         Console.WriteLine("b = " + b);
17         Console.WriteLine("a + b = " + (a + b));
18         Console.WriteLine("a - b = " + (a - b));
19         Console.WriteLine("a * b = " + a * b);
20         Console.WriteLine("a / b = " + a / b);
21         Console.WriteLine("a % b = " + (a + b));
22     }
23 }
```

Output

```
01 a = 3
02 b = 2
03 a + b = 5
04 a - b = 1
05 a * b = 6
06 a / b = 1
07 a % b = 5
```

Discussion

Each new code element represents:

- `+, -, *, /, and %` represent addition, subtraction, multiplication, division, and modulus, respectively.

Temperature

```
01 // This program converts an input Fahrenheit temperature to Celsius.
02
03 using System;
04
```

```
public class Temperature
{
    public static void Main(string[] args)
    {
        double fahrenheit;
        double celsius;

        Console.WriteLine("Enter Fahrenheit temperature:");
        fahrenheit = Convert.ToDouble(Console.ReadLine());

        celsius = (fahrenheit - 32) * 5 / 9;

        Console.WriteLine(
            fahrenheit.ToString() + "° Fahrenheit is " +
            celsius.ToString() + "° Celsius" + "\n");
    }
}
```

Output

```
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7777777777778° Celsius
```

Discussion

Each new code element represents:

- `Console.ReadLine()` reads the next line from standard input
- `Convert.ToDouble` converts the input to a double floating-point value

References

- Wikiversity: Computer Programming

This page titled 27.3.8: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.3.9: Java Examples

Overview

The following examples demonstrate data types, arithmetic operations, and input in Java.

Data Types

```
01 // This program demonstrates variables, literal constants, and data types.
02
03 public class Main {
04     public static void main(String[] args) {
05         int i;
06         double d;
07         String s;
08         boolean b;
09
10         i = 1234567890;
11         d = 1.23456789012345;
12         s = "string";
13         b = true;
14
15         System.out.println("Integer i = " + i);
16         System.out.println("Double d = " + d);
17         System.out.println("String s = " + s);
18         System.out.println("Boolean b = " + b);
19     }
20 }
```

Output

```
01 Integer i = 1234567890
02 Double d = 1.23456789012345
03 String s = string
04 Boolean b = true
```

Discussion

Each code element represents:

- `//` begins a comment
- `public class DataTypes` begins the Data Types program
- `{` begins a block of code
- `public static void main(String[] args)` begins the main function
- `int i` defines an integer variable named i
- `;` ends each line of Java code
- `double d` defines a double floating-point variable named d
- `string s` defines a string variable named s
- `boolean b` defines a Boolean variable named b
- `i = , d = , s =, b =` assign literal values to the corresponding variables
- `System.out.println` calls the standard output print line function
- `}` ends a block of code

Arithmetic

```
01 // This program demonstrates arithmetic operations.
02
03 public class Main {
04     public static void main(String[] args) {
05         int a;
06         int b;
07
08         a = 3;
09         b = 2;
10
11         System.out.println("a = " + a);
12         System.out.println("b = " + b);
13         System.out.println("a + b = " + (a + b));
14         System.out.println("a - b = " + (a - b));
15         System.out.println("a * b = " + a * b);
16         System.out.println("a / b = " + a / b);
17         System.out.println("a % b = " + (a % b));
18     }
19 }
```

Output

```
01 a = 3
02 b = 2
03 a + b = 5
04 a - b = 1
05 a * b = 6
06 a / b = 1
07 a % b = 1
```

Discussion

Each new code element represents:

- `+, -, *, /, and %` represent addition, subtraction, multiplication, division, and modulus, respectively.

Temperature

```
01 // This program converts an input Fahrenheit temperature to Celsius.
02
03 import java.util.*;
04
05 public class Main {
06     private static Scanner input = new Scanner(System.in);
07
08     public static void main(String[] args) {
09         double fahrenheit;
10         double celsius;
11
12         System.out.println("Enter Fahrenheit temperature:");
13         fahrenheit = input.nextDouble();
14
```

```
        celsius = (fahrenheit - 32) * 5 / 9;

        System.out.println(Double.toString(fahrenheit) + "° Fahrenheit is " + celsius + "° Celsius");
    }
}
```

Output

```
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7777777777778° Celsius
```

Discussion

Each new code element represents:

- `private static Scanner input ...` defines an object to read from standard input
- `input.nextDouble()` reads input as a double floating-point value

References

- Wikiversity: Computer Programming

This page titled 27.3.9: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4: FUNCTIONS

27.4.1: Call by Value vs. Call by Reference
27.4.2: Void Data Type
27.4.3: C++ Examples
27.4.4: C# Examples
27.4.5: Java Examples
27.4.6: JavaScript Examples
27.4.7: Swift Examples
27.4.8: Practice- Functions

This page titled 27.4: Functions is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 3: Functions is licensed CC BY-SA 4.0.

27.4.1: Call by Value vs. Call by Reference

Overview

In **call by value**, a parameter acts within the function as a new local variable initialized to the value of the argument (a local (isolated) copy of the argument). In **call by reference**, the argument variable supplied by the caller can be affected by actions within the called function.[1]

Discussion

Call by Value

Within most current programming languages, parameters are passed by value by default, with the argument as a copy of the calling value. Arguments are isolated, and functions are free to make changes to parameter values without any risk of impact to the calling function. Consider the following pseudocode:

```
01 Function Main
02     Declare Real fahrenheit
03
04     Assign fahrenheit = 100
05     Output "Main fahrenheit = " & fahrenheit
06     Call ChangeFahrenheit(fahrenheit)
07     Output "Main fahrenheit = " & fahrenheit
08 End
09
10 Function ChangeFahrenheit (Real fahrenheit)
11     Output "ChangeFahrenheit fahrenheit = " & fahrenheit
12     Assign fahrenheit = 0
13     Output "ChangeFahrenheit fahrenheit = " & fahrenheit
14 End
```

Output

```
01 Main fahrenheit = 100
02 ChangeFahrenheit fahrenheit = 100
03 ChangeFahrenheit fahrenheit = 0
04 Main fahrenheit = 100
```

In English, the Main function assigns the value 100 to the variable fahrenheit, displays that value, and then calls ChangeFahrenheit passing a copy of that value. The called function displays the argument, changes it, and displays it again. Execution returns to the calling function, and Main displays the value of the original variable. With call by value, the variable fahrenheit in the calling function and the parameter fahrenheit in the called function refer to different memory addresses, and the called function cannot change the value of the variable in the calling function.

Call by Reference

If a programming language uses or supports call by reference, the variable in the calling function and the parameter in the called function refer to the same memory address, and the called function may change the value of the variable in the calling function. Using the same code example as above, call by reference output would change to:

```
Main fahrenheit = 100
ChangeFahrenheit fahrenheit = 100
ChangeFahrenheit fahrenheit = 0
Main fahrenheit = 0
```

Programming languages that support both call by value and call by reference use some type of key word or symbol to indicate which parameter passing method is being used.

Language	Call By Value	Call by Reference
C++	default	use `&parameter` in called function
C#	default	use `ref parameter` in calling and called functions
Java	default	applies to arrays and objects
JavaScript	default	applies to arrays and objects
Python	default	applies to arrays (lists) and mutable objects

Arrays and objects are covered in later chapters.

Key Terms

call by reference
: Parameters passed by calling functions may be modified by called functions.

call by value
: Parameters passed by calling functions cannot be modified by called functions.

References

- Wikiversity: Computer Programming

---

1. Wikipedia: Parameter (computer programming) ↵

This page titled 27.4.1: Call by Value vs. Call by Reference is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4.2: Void Data Type

Overview

The void data type, similar to the Nothing data type described earlier, is the data type for the result of a function that returns normally, but does not provide a result value to its caller.[1]

Discussion

The **void data type** has no values and no operations. It's a data type that represents the lack of a data type.

Language	Reserved Word
C++	`void`
C#	`void`
Java	`void`
JavaScript	`void`
Python	N/A
Swift	`Void`

Many programming languages need a data type to define the lack of return value to indicate that nothing is being returned. The void data type is typically used in the definition and prototyping of functions to indicate that either nothing is being passed in and/or nothing is being returned.

Key Terms

void data type
: A data type that has no values or operators and is used to represent nothing.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

1. Wikipedia: Void type ↩

This page titled 27.4.2: Void Data Type is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4.3: C++ Examples

Temperature

```
// This program asks the user for a Fahrenheit temperature,
// converts the given temperature to Celsius,
// and displays the results.
//
// References:
// https://www.mathsisfun.com/temperature-conversion.html
// https://en.wikibooks.org/wiki/C%2B%2B_Programming

#include <iostream>

using namespace std;

double getFahrenheit();
double calculateCelsius(double);
void displayResult(double, double);

int main() {
    double fahrenheit;
    double celsius;

    fahrenheit = getFahrenheit();
    celsius = calculateCelsius(fahrenheit);
    displayResult(fahrenheit, celsius);

    return 0;
}

double getFahrenheit() {
    double fahrenheit;

    cout << "Enter Fahrenheit temperature:" << endl;
    cin >> fahrenheit;

    return fahrenheit;
}

double calculateCelsius(double fahrenheit) {
    double celsius;

    celsius = (fahrenheit - 32) * 5 / 9;

    return celsius;
}

void displayResult(double fahrenheit, double celsius) {
    cout << fahrenheit << "° Fahrenheit is "
        << celsius << "° Celsius" << endl;
}
```

Output

```
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7778° Celsius
```

References

- Wikiversity: Computer Programming

This page titled 27.4.3: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4.4: C# Examples

Temperature

```
// This program asks the user for a Fahrenheit temperature,
// converts the given temperature to Celsius,
// and displays the results.
//
// References:
// https://www.mathsisfun.com/temperature-conversion.html
// https://en.wikibooks.org/wiki/C_Sharp_Programming

using System;

class Temperature
{
    public static void Main (string[] args)
    {
        double fahrenheit;
        double celsius;

        fahrenheit = GetFahrenheit();
        celsius = CalculateCelsius(fahrenheit);
        DisplayResult(fahrenheit, celsius);
    }

    private static double GetFahrenheit()
    {
        string input;
        double fahrenheit;

        Console.WriteLine("Enter Fahrenheit temperature:");
        input = Console.ReadLine();
        fahrenheit = Convert.ToDouble(input);

        return fahrenheit;
    }

    private static double CalculateCelsius(double fahrenheit)
    {
        double celsius;

        celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }

    private static void DisplayResult(double fahrenheit, double celsius)
    {
        Console.WriteLine(fahrenheit.ToString() + "° Fahrenheit is " +
            celsius.ToString() + "° Celsius");
    }
}
```

Output

```
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7777777777778° Celsius
```

References

- Wikiversity: Computer Programming

This page titled 27.4.4: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4.5: Java Examples

Temperature

```
// This program asks the user for a Fahrenheit temperature,
// converts the given temperature to Celsius,
// and displays the results.
//
// References:
// https://www.mathsisfun.com/temperature-conversion.html
// https://en.wikibooks.org/wiki/Java_Programming

import java.util.*;

class Main {
    private static Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        double fahrenheit;
        double celsius;

        fahrenheit = getFahrenheit();
        celsius = calculateCelsius(fahrenheit);
        displayResult(fahrenheit, celsius);
    }

    private static double getFahrenheit() {
        double fahrenheit;

        System.out.println("Enter Fahrenheit temperature:");
        fahrenheit = input.nextDouble();

        return fahrenheit;
    }

    private static double calculateCelsius(double fahrenheit) {
        double celsius;

        celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }

    private static void displayResult(double fahrenheit, double celsius) {
        System.out.println(fahrenheit + "° Fahrenheit is " +
            celsius + "° Celsius");
    }
}
```

Output

```
Enter Fahrenheit temperature:
 100
```

```
03 100° Fahrenheit is 37.7777777777778° Celsius
```

References

- Wikiversity: Computer Programming

This page titled 27.4.5: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4.6: JavaScript Examples

Temperature

```
// This program asks the user for a Fahrenheit temperature,
// converts the given temperature to Celsius,
// and displays the results.
//
// References:
// https://www.mathsisfun.com/temperature-conversion.html
// https://en.wikibooks.org/wiki/JavaScript

main();

function main() {
  var fahrenheit = getFahrenheit();
  var celisus = calculateCelsius(fahrenheit);
  displayResult(fahrenheit, celisus);
}

function getFahrenheit() {
  var fahrenheit = input("Enter Fahrenheit temperature:");
  return fahrenheit;
}

function calculateCelsius(fahrenheit) {
  var celisus = (fahrenheit - 32) * 5 / 9;
  return celisus;
}

function displayResult(fahrenheit, celisus) {
  output(fahrenheit + "° Fahrenheit is " +
    celisus + "° Celsius");
}

// Checks the JavaScript environment and reads from the console,
// the current document, or standard input as appropriate.
function input(text) {
  if (typeof console === "object") {
    return prompt(text);
  }
  else {
    output(text);
    var isr = new java.io.InputStreamReader(java.lang.System.in);
    var br = new java.io.BufferedReader(isr);
    var line = br.readLine();
    return line.trim();
  }
}

// Checks the JavaScript environment and writes to the console,
// the current document, or standard output as appropriate.
// Reference: http://progopedia.com/example/hello-world/114/
function output(text) {
```

```
  if (typeof console === "object") {
    console.log(text);
  }
  else if (typeof document === "object") {
    document.write(text);
  }
  else {
    print(text);
  }
}
```

Output

```
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7777777777778° Celsius
```

References

- Wikiversity: Computer Programming

This page titled 27.4.6: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4.7: Swift Examples

Temperature

```
01 // This program asks the user for a Fahrenheit temperature,
02 // converts the given temperature to Celsius,
03 // and displays the results.
04 //
05 // References:
06 //     https://www.mathsisfun.com/temperature-conversion.html
07 //     https://developer.apple.com/library/content/documentation/Swift/Conceptual/
   Swift_Programming_Language/TheBasics.html
08
09 func getFahrenheit() -> Double {
10     var fahrenheit: Double
11
12     print("Enter Fahrenheit temperature:")
13     fahrenheit = Double(readLine(strippingNewline: true)!)!
14
15     return fahrenheit
16 }
17
18 func calculateCelsius(fahrenheit: Double) -> Double {
19     var celsius: Double
20
21     celsius = (fahrenheit - 32) * 5 / 9
22
23     return celsius
24 }
25
26 func displayResult(fahrenheit: Double, celsius: Double) {
27     print(String(fahrenheit) + "° Fahrenheit is " + String(celsius) + "° Celsius")
28 }
29
30 func main() {
31     var fahrenheit: Double
32     var celsius: Double
33
34     fahrenheit = getFahrenheit()
35     celsius = calculateCelsius(fahrenheit:fahrenheit)
36     displayResult(fahrenheit:fahrenheit, celsius:celsius)
37 }
38
39 main()
```

Output

```
01 Enter Fahrenheit temperature:
02  100
03 100.0° Fahrenheit is 37.7777777777778° Celsius
```

References

- Wikiversity: Computer Programming

This page titled 27.4.7: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.4.8: Practice- Functions

Review Questions

True / False

1. In addition to the term function as the name of a subprogram, the computer industry also uses macro, procedure and module.
2. Generally, functions fall into two categories: Program Control and Specific Task.
3. Hierarchy Charts and Structure Charts are basically the same thing.
4. Program Control functions are used to simply subdivide and control the program.
5. The void data type is rarely used in C++.
6. Making source code readable is only used by beginning programmers.
7. Scope refers to a brand of mouthwash.
8. User-defined specific task functions are usually placed into a user-defined library.
9. Local and global data storage is associated with the concept of scope.
10. Creating a header file for user-defined specific task functions is a difficult task.
11. The stack is part of the computer's memory used for storage of data.
12. The standard library is a set of specific task functions that have been added to the programming language for universal use.
13. Programmers should not have confidence that standard library functions work properly.
14. It would be easier to write programs without using specific task functions.

Answers:

1. true
2. true
3. true
4. true
5. false
6. false
7. false – Although Scope is a brand of mouthwash; we are looking for the computer-related definition.
8. true
9. true
10. false – It may seem difficult at first, but with a little practice it is really quite easy.
11. true
12. true
13. false
14. false

Short Answer

1. Create a hierarchy chart for the function example program found in this chapter.
2. Review the programs you have already created for this course. Based on coding standards for your selected programming language, identify some problems that make your code "undocumented", "unreadable" or wrong in some other way.

Activities

Complete the following activities using pseudocode, a flowcharting tool, or your selected programming language. Use separate functions for input, each type of processing, and output. Avoid global variables by passing parameters and returning results. Create test data to validate the accuracy of each program. Add comments at the top of the program and include references to any resources used.

1. Create a program to prompt the user for hours and rate per hour and then compute gross pay (hours * rate).[1], or create a program that asks the user for their monthly income and then calculate and display their annual income (monthly income *

12).
2. Create a program that asks the user how old they are in years, and then calculate and display their approximate age in months, days, hours, and seconds. For example, a person 1 year old is 12 months old, 365 days old, etc.
3. Review MathsIsFun: US Standard Lengths. Create a program that asks the user for a distance in miles, and then calculate and display the distance in yards, feet, and inches, or ask the user for a distance in miles, and then calculate and display the distance in kilometers, meters, and centimeters.
4. Review MathsIsFun: Area of Plane Shapes. Create a program that asks the user for the dimensions of different shapes and then calculate and display the area of the shapes. Do not include shape choices. That will come later. For now, just include multiple shape calculations in sequence.
5. Create a program that calculates the area of a room to determine the amount of floor covering required. The room is rectangular with the dimensions measured in feet with decimal fractions. The output needs to be in square yards. There are 3 linear feet to a yard.
6. Create a program that helps the user determine how much paint is required to paint a room and how much it will cost. Ask the user for the length, width, and height of a room, the price of a gallon of paint, and the number of square feet that a gallon of paint will cover. Calculate the total area of the four walls as `2 * length * height + 2 * width * height` Calculate the number of gallons as: `total area / square feet per gallon` Note: You must round up to the next full gallon. To round up, add `0.9999` and then convert the resulting value to an integer. Calculate the total cost of the paint as: `gallons * price per gallon`.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

---

1. PythonLearn: Variables, expressions, and statements ↵

This page titled 27.4.8: Practice- Functions is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5: CONDITIONS

27.5.1: Case Control Structure
27.5.2: C++ Examples
27.5.3: C# Examples
27.5.4: Java Examples
27.5.5: JavaScript Examples
27.5.6: Swift Examples
27.5.7: Practice- Conditions
27.5.8: Selection Control Structures

This page titled 27.5: Conditions is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 4: Conditions is licensed CC BY-SA 4.0.

27.5.1: Case Control Structure

Overview

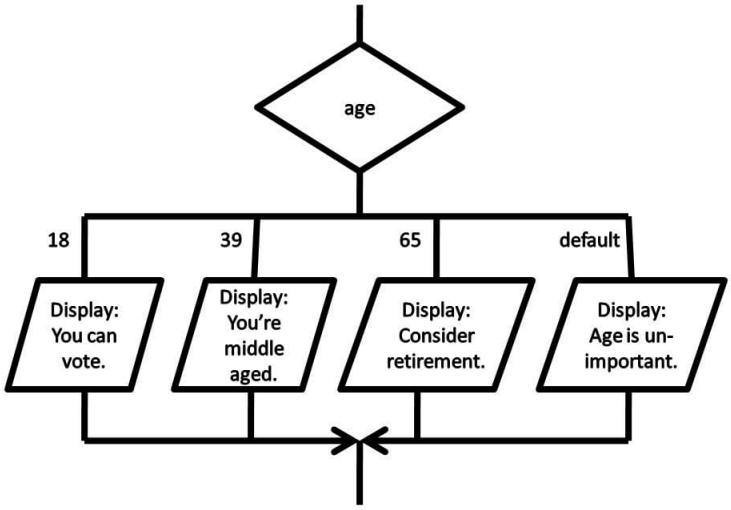
A **case** or switch statement is a type of selection control mechanism used to allow the value of a variable or expression to change the control flow of program execution via a multiway branch.[1]

Discussion

One of the drawbacks of two-way selection is that we can only consider two choices. But what do you do if you have more than two choices? Consider the following which has four choices:

```
01 if age equal to 18
02     you can vote
03 else if age equal to 39
04     you're middle-aged
05 else if age equal to 65
06     consider retirement
07 else
08     age is unimportant
```

You get an appropriate message depending on the value of age. The last item is referred to as the default. If the age is not equal to 18, 39 or 65 you get the default message. In some situations there is no default action. Consider this flowchart example:



This flowchart is of the case control structure and is used for multiway selection. The decision box holds the variable age. The logic of the case is one of equality wherein the value in the variable age is compared to the listed values in order from left to right. Thus, the value stored in age is compared to 18 or is “age equal to 18”. If it is true, the logic flows down through the action and drops out at the bottom of the case structure. If the value of the test expression is false, it moves to the next listed value to the right and makes another comparison. It works exactly the same as our nested if then else structure.

Code to Accomplish Multiway Selection

Python does not support a case control structure. But using the same example as above, here is C++ / C# / Java / JavaScript / Swift code to accomplish the case control structure.

```
switch (age)
{
    case 18:
        message = "You can vote.";
        break;
    case 39:
        message = "You're middle-aged.";
        break;
    case 65:
        message = "Consider retirement.";
        break;
    default:
        message = "Age is unimportant.";
        break;
}
```

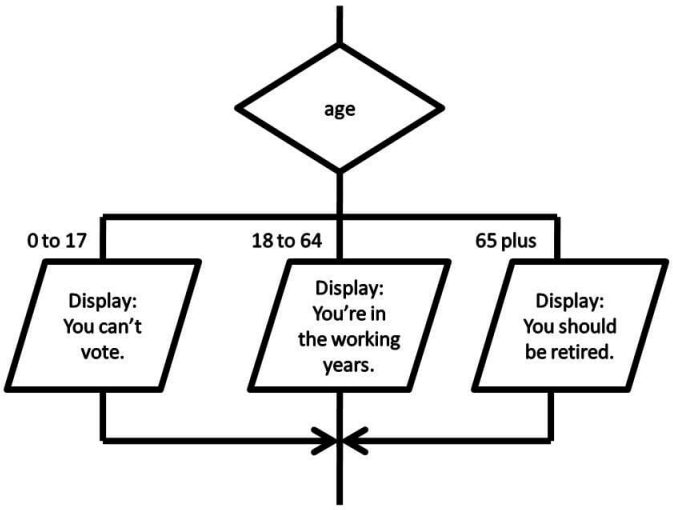
The value in the variable age is compared to the first “case”, which is the value 18 (also called the listed value) using an equality comparison or is “age equal to 18”. If it is true, the message is assigned the value “You can vote.” and the next line of code (the break) is done (which jumps us to the end of the control structure). If it is false, it moves on to the next case for comparison.

Many programming languages require the listed values for the case control structure be of the integer family of data types. This basically means either an integer or character data type. Consider this example that uses character data type (choice is a character variable):

```
switch (choice)
{
    case 'A':
        message = "You are an A student.";
        break;
    case 'B':
        message = "You are a B student.";
        break;
    case 'C':
        message = "You are a C student.";
        break;
    default:
        message = "Maybe you should study harder.";
        break;
}
```

Limitations of the Case Control Structure

Most programming languages do not allow ranges of values for case-like structures. Consider this flowcharting example that used ranges:



Consider also the following pseudocode for the same logic:

```
Case of age
  0 to 17    Display "You can't vote."
  18 to 64   Display "You're in your working years."
  65 +       Display "You should be retired."
End
```

Using the case control structure when using non-integer family or ranges of values is allowed when designing a program and documenting that design with pseudocode or flowcharting. However, the implementation in most languages would follow a nested if then else approach with complex Boolean expressions. The logic of the above examples would look like this:

```
if age > 0 and age <= to 17
    display You can't vote.
else if age is >= 18 and age <= 64
    display You're in your working years.
else
    display You should be retired.
```

Good Structured Programming Methods

Most textbook authors confirm that good structured programming techniques and habits are more important than concentrating on the technical possibilities and capabilities of the language that you are using to learn programming skills. Remember, this module is concentrating on programming fundamentals and concepts to build our initial programming skills. It is not a created with the intent to cover programming languages in detail, despite the fact that at times we have to cover language mechanics.

Key Terms

case

A control structure that does multiway selection.

switch

A control structure that can be made to act like a case control structure.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Switch statement ↩

This page titled 27.5.1: Case Control Structure is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5.2: C++ Examples

Temperature

The temperature program main function is listed twice, once using if-then-else and once using switch, followed by the supporting functions.

If Then Else

```
// This program asks the user to select Fahrenheit or Celsius conversion
// and input a given temperature. Then the program converts the given
// temperature and displays the result.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C%2B%2B_Programming

#include

using namespace std;

char getChoice();
void processCelsius();
void processFahrenheit();
double getTemperature(string scale);
double calculateCelsius(double fahrenheit);
double calculateFahrenheit(double celsius);
void displayResult(double temperature, string fromScale, double result, string toScale);

int main() {
    char choice;

    choice = getChoice();
    if (choice == 'C' || choice == 'c') {
        processCelsius();
    }
    else if (choice == 'F' || choice == 'f') {
        processFahrenheit();
    }
    else {
        cout << "You must enter C to convert to Celsius or F to convert to Fahrenheit." << endl;
    }
    return 0;
}
```

Switch

```
int main() {
    char choice;

    choice = getChoice();
```

```
    switch(choice) {
        case 'C':
        case 'c':
            processCelsius();
            break;
        case 'F':
        case 'f':
            processFahrenheit();
            break;
        default:
            cout << "You must enter C to convert to Celsius or F to convert to Fahrenheit." << endl;
    }
    return 0;
}
```

Supporting Functions

```
char getChoice() {
    char choice;

    cout << "Enter C to convert to Celsius or F to convert to Fahrenheit:" << endl; cin >> choice;

    return choice;
}

void processCelsius() {
    double temperature;
    double result;

    temperature = getTemperature("Fahrenheit");
    result = calculateCelsius(temperature);
    displayResult(temperature, "Fahrenheit", result, "Celsius");
}

void processFahrenheit() {
    double temperature;
    double result;

    temperature = getTemperature("Celsius");
    result = calculateFahrenheit(temperature);
    displayResult(temperature, "Celsius", result, "Fahrenheit");
}

double getTemperature(string scale) {
    double temperature;

    cout << "Enter " << scale << " temperature:" << endl; cin >> temperature;

    return temperature;
}

double calculateCelsius(double fahrenheit) {
    double celsius;

```

```
    celsius = (fahrenheit - 32) * 5 / 9;

    return celsius;
}

double calculateFahrenheit(double celsius) {
    double fahrenheit;

    fahrenheit = celsius * 9 / 5 + 32;

    return fahrenheit;
}

void displayResult(double temperature, string fromScale, double result, string toScale) {
    cout << temperature << "° " << fromScale << " is " << result << "° " << toScale << endl;
}
```

Output

```
Enter C to convert to Celsius or F to convert to Fahrenheit:
 c
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7778° Celsius

Enter C to convert to Celsius or F to convert to Fahrenheit:
 f
Enter Celsius temperature:
 100
100° Celsius is 212° Fahrenheit

Enter C to convert to Celsius or F to convert to Fahrenheit:
 x
You must enter C to convert to Celsius or F to convert to Fahrenheit.
```

References

- Wikiversity: Computer Programming

This page titled 27.5.2: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5.3: C# Examples

Temperature

The temperature program main function is listed twice, once using if-then-else and once using switch, followed by the supporting functions.

If Then Else

```
// This program asks the user to select Fahrenheit or Celsius conversion
// and input a given temperature. Then the program converts the given
// temperature and displays the result.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C_Sharp_Programming

using System;

public class Temperature
{
    public static void Main(string[] args)
    {
        string choice;

        choice = GetChoice();
        if (choice == "C" || choice == "c")
        {
            ProcessCelsius();
        }
        else if (choice == "F" || choice == "f")
        {
            ProcessFahrenheit();
        }
        else
        {
            Console.WriteLine("You must enter C to convert to Celsius or F to convert to Fahrenheit.");
        }
    }
```

Switch

```
    public static void Main(string[] args)
    {
        string choice;

        choice = GetChoice();
        switch (choice)
        {
            case "C":
            case "c":
```

```
                ProcessCelsius();
                break;
            case "F":
            case "f":
                ProcessFahrenheit();
                break;
            default:
                Console.WriteLine("You must enter C to convert to Celsius or F to convert to
Fahrenheit.");
                break;
        }
    }
```

Supporting Functions

```
    public static string GetChoice()
    {
        string choice;

        Console.WriteLine("Enter C to convert to Celsius or F to convert to Fahrenheit:");
        choice = Console.ReadLine();

        return choice;
    }

    public static void ProcessCelsius()
    {
        double temperature;
        double result;

        temperature = GetTemperature("Fahrenheit");
        result = CalculateCelsius(temperature);
        DisplayResult(temperature, "Fahrenheit", result, "Celsius");
    }

    public static void ProcessFahrenheit()
    {
        double temperature;
        double result;

        temperature = GetTemperature("Celsius");
        result = CalculateFahrenheit(temperature);
        DisplayResult(temperature, "Celsius", result, "Fahrenheit");
    }

    public static double GetTemperature(string scale)
    {
        double temperature;

        Console.WriteLine("Enter " + scale + " temperature:");
        temperature = Convert.ToDouble(Console.ReadLine());

        return temperature;
    }

    public static double CalculateCelsius(double fahrenheit)
```

```
    {
        double celsius;

        celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }

    public static double CalculateFahrenheit(double celsius)
    {
        double fahrenheit;

        fahrenheit = celsius * 9 / 5 + 32;

        return fahrenheit;
    }

    public static void DisplayResult(double temperature, string fromScale, double result,
string toScale)
    {
        Console.WriteLine(temperature + "° " + fromScale + " is " + result + "° " + toScale);
    }
}
```

Output

```
Enter C to convert to Celsius or F to convert to Fahrenheit:
 c
Enter Fahrenheit temperature:
 100
100° Fahrenheit is 37.7777777777778° Celsius

Enter C to convert to Celsius or F to convert to Fahrenheit:
 f
Enter Celsius temperature:
 100
100° Celsius is 212° Fahrenheit

Enter C to convert to Celsius or F to convert to Fahrenheit:
 x
You must enter C to convert to Celsius or F to convert to Fahrenheit.
```

References

- Wikiversity: Computer Programming

This page titled 27.5.3: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5.4: Java Examples

Temperature

The temperature program main function is listed twice, once using if-then-else and once using switch, followed by the supporting functions.

If Then Else

```
01 // This program asks the user to select Fahrenheit or Celsius conversion
02 // and input a given temperature. Then the program converts the given
03 // temperature and displays the result.
04 //
05 // References:
06 //     https://www.mathsisfun.com/temperature-conversion.html
07 //     https://en.wikibooks.org/wiki/Java_Programming
08
09 import java.util.*;
10
11 public class Main {
12     private static Scanner input = new Scanner(System.in);
13
14     public static void main(String[] args) {
15         String choice;
16
17         choice = getChoice();
18         if (choice.equals("C") || choice.equals("c")) {
19             processCelsius();
20         }
21         else if (choice.equals("F") || choice.equals("f")) {
22             processFahrenheit();
23         }
24         else {
25             System.out.println("You must enter C to convert to Celsius or F to convert to
   Fahrenheit.");
26         }
27     }
```

Switch

```
01     public static void main(String[] args) {
02         String choice;
03
04         choice = getChoice();
05         switch (choice) {
06             case "C":
07             case "c":
08                 processCelsius();
09                 break;
10             case "F":
11             case "f":
12                 processFahrenheit();
```

```
                break;
            default:
                System.out.println("You must enter C to convert to Celsius or F to convert to
Fahrenheit.");
        }
    }
```

Supporting Functions

```
    public static String getChoice() {
        String choice;

        System.out.println("Enter C to convert to Celsius or F to convert to Fahrenheit:");
        choice = input.nextLine();

        return choice;
    }

    public static void processCelsius() {
        double temperature;
        double result;

        temperature = getTemperature("Fahrenheit");
        result = calculateCelsius(temperature);
        displayResult(temperature, "Fahrenheit", result, "Celsius");
    }

    public static void processFahrenheit() {
        double temperature;
        double result;

        temperature = getTemperature("Celsius");
        result = calculateFahrenheit(temperature);
        displayResult(temperature, "Celsius", result, "Fahrenheit");
    }

    public static double getTemperature(String scale) {
        double temperature;

        System.out.println("Enter " + scale + " temperature:");
        temperature = input.nextDouble();

        return temperature;
    }

    public static double calculateCelsius(double fahrenheit) {
        double celsius;

        celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }

    public static double calculateFahrenheit(double celsius) {
        double fahrenheit;

```

```
        fahrenheit = celsius * 9 / 5 + 32;

        return fahrenheit;
    }

    public static void displayResult(double temperature, String fromScale, double result, String toScale) {
        System.out.println(Double.toString(temperature) + "° " + fromScale + " is " + result + "° " + toScale);
    }
}
```

Output

```
Enter C to convert to Celsius or F to convert to Fahrenheit:
 c
Enter Fahrenheit temperature:
 100
100.0° Fahrenheit is 37.77777777777778° Celsius

Enter C to convert to Celsius or F to convert to Fahrenheit:
 f
Enter Celsius temperature:
 100
100.0° Celsius is 212.0° Fahrenheit

Enter C to convert to Celsius or F to convert to Fahrenheit:
 x
You must enter C to convert to Celsius or F to convert to Fahrenheit.
```

References

- Wikiversity: Computer Programming

This page titled 27.5.4: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5.5: JavaScript Examples

Temperature

The temperature program main function is listed twice, once using if-then-else and once using switch, followed by the supporting functions.

If Then Else

```
// This program asks the user to select Fahrenheit or Celsius conversion
// and input a given temperature. Then the program converts the given
// temperature and displays the result.
//
// References:
//   https://www.mathsisfun.com/temperature-conversion.html
//   https://en.wikibooks.org/wiki/JavaScript

main();

function main() {
    var choice;

    choice = getChoice();
    if (choice == "C" || choice == "c") {
        processCelsius();
    }
    else if (choice == "F" || choice == "f") {
        processFahrenheit();
    }
    else {
        output("You must enter C to convert to Celsius or F to convert to Fahrenheit.");
    }
}
```

Switch

```
function main() {
    var choice;

    choice = getChoice();
    switch(choice)
    {
        case 'C':
        case 'c':
            processCelsius();
            break;
        case 'F':
        case 'f':
            processFahrenheit();
            break;
        default:
            output("You must enter C to convert to Celsius or F to convert to Fahrenheit.");
```

```
    }
}
```

Supporting Functions

```
function getChoice() {
    var choice;

    output("Enter C to convert to Celsius or F to convert to Fahrenheit:");
    choice = input();

    return choice;
}

function processCelsius() {
    var temperature;
    var result;

    temperature = getTemperature("Fahrenheit");
    result = calculateCelsius(temperature);
    displayResult(temperature, "Fahrenheit", result, "Celsius");
}

function processFahrenheit() {
    var temperature;
    var result;

    temperature = getTemperature("Celsius");
    result = calculateFahrenheit(temperature);
    displayResult(temperature, "Celsius", result, "Fahrenheit");
}

function getTemperature(scale) {
    var temperature;

    output("Enter " + scale + " temperature:");
    temperature = input();

    return temperature;
}

function calculateCelsius(fahrenheit) {
    var celsius;

    celsius = (fahrenheit - 32) * 5 / 9;

    return celsius;
}

function calculateFahrenheit(celsius) {
    var fahrenheit;

    fahrenheit = celsius * 9 / 5 + 32;

    return fahrenheit;
}
```

```
52
53 function displayResult(temperature, fromScale, result, toScale) {
54     output(temperature.toString() + "° " + fromScale + " is " + result + "° " + toScale);
55 }
56
57 // Checks the JavaScript environment and reads from the console,
58 // the current document, or standard input as appropriate.
59 function input(text) {
60   if (typeof console === "object") {
61     return prompt(text);
62   }
63   else {
64     output(text);
65     var isr = new java.io.InputStreamReader(java.lang.System.in);
66     var br = new java.io.BufferedReader(isr);
67     var line = br.readLine();
68     return line.trim();
69   }
70 }
71
72 // Checks the JavaScript environment and writes to the console,
73 // the current document, or standard output as appropriate.
74 // Reference: http://progopedia.com/example/hello-world/114/
75 function output(text) {
76   if (typeof console === "object") {
77     console.log(text);
78   }
79   else if (typeof document === "object") {
80     document.write(text);
81   }
82   else {
83     print(text);
84   }
85 }
```

Output

```
01 Enter C to convert to Celsius or F to convert to Fahrenheit:
02  c
03 Enter Fahrenheit temperature:
04  100
05 100° Fahrenheit is 37.77777777777778° Celsius
06
07 Enter C to convert to Celsius or F to convert to Fahrenheit:
08  f
09 Enter Celsius temperature:
10  100
11 100° Celsius is 212° Fahrenheit
12
13 Enter C to convert to Celsius or F to convert to Fahrenheit:
14  x
15 You must enter C to convert to Celsius or F to convert to Fahrenheit.
```

References

- Wikiversity: Computer Programming

This page titled 27.5.5: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5.6: Swift Examples

Temperature

The temperature program main function is listed twice, once using if-then-else and once using switch, followed by the supporting functions.

If Then Else

```
// This program asks the user for a Fahrenheit temperature,
// converts the given temperature to Celsius,
// and displays the results.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

func main() {
    var choice: String

    choice = getChoice()
    if choice == "C" || choice == "c" {
        processCelsius()
    }
    else if choice == "F" || choice == "f" {
        processFahrenheit()
    }
    else {
        print("You must enter C to convert to Celsius or F to convert to Fahrenheit.")
    }
}

main()
```

Switch

```
func main() {
    var choice: String

    choice = getChoice()
    switch choice {
        case "C", "c":
            processCelsius()
        case "F", "f":
            processFahrenheit()
        default:
            print("You must enter C to convert to Celsius or F to convert to Fahrenheit.")
    }
}
```

Supporting Functions

```
func getChoice() -> String {
    var choice: String

    print("Enter C to convert to Celsius or F to convert to Fahrenheit:")
    choice = readLine(strippingNewline: true)!

    return choice
}

func processCelsius() {
    var temperature: Double
    var result: Double

    temperature = getTemperature(scale:"Fahrenheit")
    result = calculateCelsius(fahrenheit:temperature)
    displayResult(temperature:temperature, fromScale:"Fahrenheit", result:result, toScale:"Celsius")
}

func processFahrenheit() {
    var temperature: Double
    var result: Double

    temperature = getTemperature(scale:"Celsius")
    result = calculateFahrenheit(celsius:temperature)
    displayResult(temperature:temperature, fromScale:"Celsius", result:result, toScale:"Fahrenheit")
}

func getTemperature(scale: String) -> Double {
    var temperature: Double

    print("Enter " + scale + " temperature:")
    temperature = Double(readLine(strippingNewline: true)!)!

    return temperature
}

func calculateCelsius(fahrenheit: Double) -> Double {
    var celsius: Double

    celsius = (fahrenheit - 32) * 5 / 9

    return celsius
}

func calculateFahrenheit(celsius: Double) -> Double {
    var fahrenheit: Double

    fahrenheit = celsius * 9 / 5 + 32

    return fahrenheit
}

func displayResult(temperature: Double, fromScale: String, result: Double, toScale: String) {
```

```
    print(String(temperature) + "° " + fromScale + " is " + String(result) + "° " +  toScale)
}
```

Output

```
Enter C to convert to Celsius or F to convert to Fahrenheit:
 c
Enter Fahrenheit temperature:
 100
100.0° Fahrenheit is 37.77777777777778° Celsius

Enter C to convert to Celsius or F to convert to Fahrenheit:
 f
Enter Celsius temperature:
 100
100.0° Celsius is 212.0° Fahrenheit

Enter C to convert to Celsius or F to convert to Fahrenheit:
 x
You must enter C to convert to Celsius or F to convert to Fahrenheit.
```

References

- Wikiversity: Computer Programming

This page titled 27.5.6: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5.7: Practice- Conditions

Review Questions

True / False

1. There are only two categories of control structures.
2. Branching control structures are rarely used in good structured programming.
3. If then else is a multiway selection control structure.
4. The while control structure is part of the branching category.
5. Pseudocode is better than flowcharting.

Answers:

1. false
2. true
3. false
4. false
5. false

Expressions

Evaluate the following Boolean expressions:

1. 25 < 7
2. 3 < 7
3. 14 > 7
4. 17 <= 7
5. 25 >= 7
6. 13 == 7
7. 9 != 7
8. 5 !> 7
9. 25 > 39 || 15 > 36
10. 19 > 26 || 13 < 17
11. 14 < 7 && 6 <= 6
12. 4 > 3 && 17 >= 7
13. ! true
14. ! (13 == 7)
15. 9 != 7 && ! 1
16. 6 < && 8

Answers:

1. 0
2. 1
3. 1
4. 0
5. 1
6. 0
7. 1
8. Error, the “not greater than” is not a valid operator.
9. 0
10. 1
11. 0

12. 1
13. 0
14. 1
15. 0
16. Error, there needs to be an operand between the operators < and &&.

Short Answer

1. List the four categories of control structures and provide a brief description of each category.
2. Create a table with the six relational operators and their meanings.

Activities

Complete the following activities using pseudocode, a flowcharting tool, or your selected programming language. Use separate functions for input, each type of processing, and output. Avoid global variables by passing parameters and returning results. Create test data to validate the accuracy of each program. Add comments at the top of the program and include references to any resources used.

1. Create a program to prompt the user for hours and rate per hour and then compute gross pay (hours * rate). Include a calculation to give 1.5 times the hourly rate for any overtime (hours worked above 40 hours).[1]
2. Create a program that asks the user how old they are in years. Then ask the user if they would like to know how old they are in (M)onths, (D)ays, (H)ours, or (S)econds. Use if/else conditional statements to calculate and display their approximate age in the selected timeframe. Do not perform any unnecessary calculations.
3. Review MathsIsFun: US Standard Lengths. Create a program that asks the user for a distance in miles, and then ask the user if they want the distance in US measurements (yards, feet, and inches) or in metric measurements (kilometers, meters, and centimeters). Use if/else conditional statements to determine their selection and then calculate and display the results.
4. Review MathsIsFun: Area of Plane Shapes. Create a program that asks the user what shape they would like to calculate the area for. Use if/else conditional statements to determine their selection and then gather the appropriate input and calculate and display the area of the shape.
5. Create a program that helps the user determine what sock size to order based on their shoe size:
   ```
   < 4 = extra small
   4 to 6 = small
   7 to 9 = medium
   10 to 12 = large
   13+ = extra large
   ```
   Use if/else conditional statements to determine their selection and then display the results. Round half-sizes up to the next whole size. One option for rounding is to add 0.5 and then convert to an integer.
6. If your programming language supports it, update one or more of the programs above to replace the if/else conditional statements with case/select conditional statements.
7. Review Wikipedia: Is functions. If your programming language supports it, update one or more of the programs above to include input validation for all numeric input.
8. If your programming language supports it, extend one or more of the programs above by adding structured exception handling statements (try-catch, try-except, etc.) to handle any runtime errors caused by the user entering invalid values for the input.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

---

1. PythonLearn: Variables, expressions, and statements ↩

This page titled 27.5.7: Practice- Conditions is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.5.8: Selection Control Structures

Overview

In **selection control structures**, conditional statements are features of a programming language which perform different computations or actions depending on whether a programmer-specified Boolean condition evaluates to true or false.[1]

Discussion

The basic attribute of a selection control structure is to be able to select between two or more alternate paths. This is described as either two-way selection or multi-way selection. A question using Boolean concepts usually controls which path is selected. All of the paths from a selection control structure join back up at the end of the control structure, before moving on to the next lines of code in a program.

If Then Else Control Structure

The **if then else** control structure is a two-way selection.

```
01 If age > 17
02     Output "You can vote."
03 False:
04     Output "You can't vote."
05 End
```

Language	Reserved Words
C++	`if`, `else`
C#	`if`, `else`
Java	`if`, `else`
JavaScript	`if`, `else`
Python	`if`, `elif`, `else`
Swift	`if`, `else`

Case Control Structure

The **case** control structure is a multi-way selection. Case control structures compare a given value with specified constants and take action according to the first expression to match.[2]

```
01 Case of age
02     0 to 17   Display "You can't vote."
03     18 to 64  Display "You're in your working years."
04     65 +      Display "You should be retired."
05 End
```

Language	Reserved Words
C++	`switch`, `case`, `break`, `default`

C#	`switch`, `case`, `break`, `default`
Java	`switch`, `case`, `break`, `default`
JavaScript	`switch`, `case`, `break`, `default`
Python	N/A
Swift	`switch`, `case`, `break` (optional), `default`

Python does not support a case control structure. There are workarounds, but they are beyond the scope of this book.

Key Terms

if then else
: A two-way selection control structure.

case
: A multi-way selection control structure.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Conditional (computer programming) ↵
2. Wikipedia: Conditional (computer programming) ↵

This page titled 27.5.8: Selection Control Structures is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6: LOOPS

27.6.1: Do While Loop
27.6.2: Branching Statements
27.6.3: Integer Overflow
27.6.4: C++ Examples
27.6.5: C# Examples
27.6.6: Java Examples
27.6.7: JavaScript Examples
27.6.8: Swift Examples
27.6.9: Practice- Loops

This page titled 27.6: Loops is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 5: Loops is licensed CC BY-SA 4.0.

27.6.1: Do While Loop

Overview

A **do while loop** is a control flow statement that executes a block of code at least once, and then repeatedly executes the block, or not, depending on a given boolean condition at the end of the block.[1]

Some languages may use a different naming convention for this type of loop. For example, the Pascal language has a **repeat until loop**, which continues to run until the control expression is true (and then terminates) — whereas a “while” loop runs while the control expression is true (and terminates once the expression becomes false).[2]

Discussion

Introduction to Test After Loops

There are two commonly used test after loops in the iteration (or repetition) category of control structures. They are: do while and repeat until. This module covers both.

Understanding Iteration in General – do while

The concept of iteration is connected to possibly wanting to repeat an action. Like all control structures, we ask a question to control the execution of the loop. The term loop comes from the circular looping motion that occurs when using flowcharting. The basic form of the do while loop is as follows:

```
01 do
02     some statements or action
03     some statements or action
04     some statements or action
05     update the flag
06 while the answer to the question is true
```

In most programming languages the question (called a test expression) is a Boolean expression. The Boolean data type has two values – true and false. Let’s rewrite the structure to consider this:

```
01 do
02     some statements or action
03     some statements or action
04     some statements or action
05     update the flag
06 while expression is true
```

Within the do while control structure there are three attributes of a properly working loop. They are:

- Action or actions
- Update of the flag
- Test expression

The English phrasing is, “You do the action while the expression is true”. This is looping on the true. When the test expression is false, you stop the loop and go on with the next item in the program. Notice, because this is a test after loop the action will always happen **at least once**. It is called a test after loop because the test comes after the action. It is also sometimes called a post-test loop, meaning the test is post (or Latin for after) the action and update.

Understanding Iteration in General – repeat until

The concept of iteration is connected to possibly wanting to repeat an action. Like all control structures, we ask a question to control the execution of the loop. The term loop comes from the circular looping motion that occurs when using flowcharting. The basic form of the repeat until loop is as follows:

```
repeat
    some statements or action
    some statements or action
    some statements or action
    update the flag
until the answer to the question becomes true
```

In most programming languages the question (called a test expression) is a Boolean expression. The Boolean data type has two values – true and false. Let's rewrite the structure to consider this:

```
repeat
    some statements or action
    some statements or action
    some statements or action
    update the flag
until expression becomes true
```

Within the repeat until control structure, there are three attributes of a properly working loop. They are:

- Action or actions
- Update of the flag
- Test expression

The English phrasing is, "You repeat the action until the expression becomes true". This is looping on the false. When the test expression becomes true, you stop the loop and go on with the next item in the program. Notice, because this is a test after loop the action will always happen **at least once**. It is called a "test after loop" because the test comes after the action. It is also sometimes called a post-test loop, meaning the test is post (or Latin for after) the action and update.

An Example

```
Do
    Output "What is your age? "
    Input user_age
    Output "What is your friend's age? "
    Input friend_age
    Output "Together your ages add up to: "
    Output age_user + friend_age
    Output "Do you want to try it again? y or n "
    Input loop_response
While loop_response == 'y'
```

The three attributes of a test after loop are present. The action part consists of the 6 lines that prompt for data and then displays the total of the two ages. The update of the flag is the displaying the question and getting the answer for the variable loop_response. The test is the equality relational comparison of the value in the flag variable to the lower case character of y.

This type of loop control is called an event controlled loop. The flag updating is an event where someone decides if they want the loop to execute again.

Using indentation with the alignment of the loop actions and flag update is the normal industry practice.

Infinite Loops

At this point, it is worth mentioning that good programming always provides for a method to ensure that the loop question will eventually be false so that the loop will stop executing and the program continues with the next line of code. However, if this does not happen, then the program is in an infinite loop. Infinite loops are a bad thing. Consider the following code:

```
loop_response = 'y'
Do
    Output "What is your age? "
    Input user_age
    Output "What is your friend's age? "
    Input friend_age
    Output "Together your ages add up to: "
    Output user_age + friend_age
While loop_response == 'y'
```

The programmer assigned a value to the flag before the loop and forgot to update the flag. Every time the test expression is asked it will always be true. Thus, an infinite loop because the programmer did not provide a way to exit the loop (he forgot to update the flag).

Consider the following code:

```
do
    Output "What is your age? "
    Input user_age
    Output "What is your friend's age? "
    Input friend_age
    Output "Together your ages add up to: "
    Output age_user + friend_age
    Output "Do you want to try it again? y or n "
    Input loop_response
While loop_response = 'y'
```

No matter what the user replies during the flag update, the test expression does not do a relational comparison but does an assignment. It assigns ‘y’ to the variable and asks if ‘y’ is true? Since all non-zero values are treated as representing true, the answer to the text question is true. Viola, you have an infinite loop.

Key Terms

action item
: An attribute of iteration control structures.

at least once
: Indicating that test after loops execute the action at least once.

do while
: A test after iteration control structure.

infinite loop
: A sequence of instructions which loops endlessly, either due to the loop having no terminating condition, having one that can never be met, or one that causes the loop to start over.[3]

repeat until
: A test after iteration control structure alternative available in some programming languages.

test item
: An attribute of iteration control structures.

update item
: An attribute of iteration control structures.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Do while loop ↵
2. Wikipedia: Do while loop ↵
3. Wikipedia: Infinite loop ↵

This page titled 27.6.1: Do While Loop is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.2: Branching Statements

Overview

A branch is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order.[1] Common branching statements include `break`, `continue`, `return`, and `goto`.

Discussion

Branching statements allow the flow of execution to jump to a different part of the program. The common branching statements used within other control structures include: `break`, `continue`, `return`, and `goto`. The goto is rarely used in modular structured programming. Additionally, we will add to our list of branching items a pre-defined function commonly used in programming languages of: `exit`.

Examples

break

The break is used in one of two ways; with a switch to make it act like a case structure or as part of a looping process to break out of the loop. The following gives the appearance that the loop will execute 8 times, but the break statement causes it to stop during the fifth iteration.

```
counter = 0;
While counter < 8
    Output counter
    If counter == 4
        break
    counter += 1
```

continue

The following gives the appearance that the loop will print to the monitor 8 times, but the continue statement causes it not to print number 4.

```
For counter = 0, counter < 8, counter += 1
    If counter == 4
        continue
    Output counter
```

return

The return statement exits a function and returns to the statement where the function was called.

```
Function DoSometing
    statements
Return <optional return value>
```

goto

The goto structure is typically not accepted in good structured programming. However, some programming languages allow you to create a label with an identifier name followed by a colon. You use the command word `goto` followed by the label.

```
some lines of code;
goto label;                // jumps to the label
some lines of code;
some lines of code;
some lines of code;
label: some statement;     // Declared label
some lines of code;
```

exit

Although exit is technically a pre-defined function, it is covered here because of its common usage in programming. A good example is the opening a file and then testing to see if the file was actually opened. If not, we have an error that usually indicates that we want to prematurely stop the execution of the program. The exit function terminates the running of the program and in the process returns an integer value back to the operating system. It fits the definition of branching which is to jump to some other place in the program.

Key Terms

branching statements
: Allow the flow of execution to jump to a different part of the program.

break
: A branching statement that terminates the existing structure.

continue
: A branching statement that causes a loop to stop its current iteration and begin the next one.

exit
: A predefined function used to prematurely stop a program and return to the operating system.

goto
: An unstructured branching statement that causes the logic to jump to a different place in the program.

return
: A branching statement that causes a function to jump back to the function that called it.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Branch (computer science) ↩

This page titled 27.6.2: Branching Statements is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.3: Integer Overflow

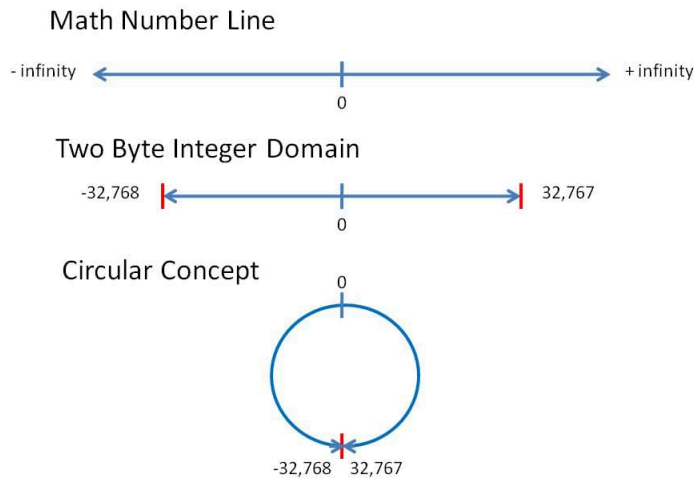
Overview

Integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value.[1]

The most common result of an overflow is that the least significant representable bits of the result are stored; the result is said to wrap around the maximum (i.e. modulo power of two). An overflow condition may give results leading to unintended behavior. In particular, if the possibility has not been anticipated, overflow can compromise a program's reliability and security.[2]

Discussion

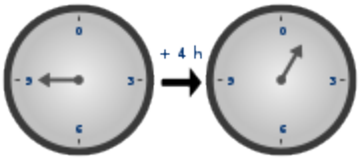
There are times when character and integer data types are lumped together because they both act the same (often called the integer family). Maybe we should say they act differently than the floating-point data types. The integer family values jump from one value to another. There is nothing between 6 and 7 nor between 'A' and 'B'. It could be asked why not make all your numbers floating-point data types. The reason is twofold. First, some things in the real world are not fractional. A dog, even with only 3 legs, is still one dog not three-fourths of a dog. Second, the integer data type is often used to control program flow by counting (counting loops). The integer family has a circular wrap-around feature. Using a two-byte integer, the next number bigger than 32767 is negative 32768 (character acts the same way going from 255 to 0. We could also reverse that to be the next smaller number than negative 32768 is positive 32767. This can be shown by using a normal math line, limiting the domain and then connecting the two ends to form a circle.



This circular nature of the integer family works for both integer and character data types. In theory, it should work for the Boolean data type as well; but in most programming languages it does not for various technical reasons.

"In mathematics, modular arithmetic (sometimes called clock arithmetic) is a system of arithmetic for integers where numbers "wrap around" after they reach a certain value — the modulus. ...

A familiar use of modular arithmetic is its use in the 12-hour clock the arithmetic of time-keeping in which the day is divided into two 12 hour periods. If the time is 7:00 now, then 8 hours later it will be 3:00. Regular addition would suggest that the later time should be 7 + 8 = 15, but this is not the answer because clock time "wraps around" every 12 hours; there is no "15 o'clock". Likewise, if the clock starts at 12:00 (noon) and 21 hours elapse, then the time will be 9:00 the next day, rather than 33:00. Since the hour number starts over when it reaches 12, this is arithmetic modulo 12.



Time-keeping on a clock gives an example of modular arithmetic." (Modular arithmetic from Wikipedia)

The use of the modulus operator in integer division is tied to the concepts used in modular arithmetic.

Implications When Executing Loops

If a programmer sets up a counting loop incorrectly, usually one of three things happen:

- Infinite loop – usually caused by missing update attribute.
- Loop never executes – usually, the text expression is wrong with the direction of the less than or greater than relationship needing to be switched.
- Loop executes more times than desired – update not properly handled. Usually, the direction of counting (increment or decrement) need to be switched.

Let's give an example of the loop executing for what appears to be for infinity (the third item on our list).

```
01 for int x = 0, x < 10, x--
02     Output x
```

The above code accidentally decrements and the value of x goes in a negative way towards -2147483648 (the largest negative value in a normal four-byte signed integer data type). It might take a while (thus it might appear to be in an infinite loop) for it to reach the negative 2 billion-plus value, before finally decrementing to positive 2147483647 which would, incidentally, stop the loop execution.

Key Terms

circular nature
: Connecting the negative and positive ends of the domain of an integer family data type.

loop control
: Making sure the attributes of a loop are properly handled.

modular arithmetic
: A system of arithmetic for integers where numbers "wrap around".

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

---

1. Wikipedia: Integer overflow ↵
2. Wikipedia: Integer overflow ↵

This page titled 27.6.3: Integer Overflow is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.4: C++ Examples

Temperature

```
// This program displays a temperature conversion table showing Fahrenheit
// temperatures from 0 to 100, in increments of 10, and the corresponding
// Celsius temperatures using While, Do While, and For loops.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C%2B%2B_Programming

#include

using namespace std;

void whileLoop();
void doLoop();
void forLoop();
void displayHeading();
double calculateCelsius(double);
void displayResult(double, double);

int main() {
    whileLoop();
    doLoop();
    forLoop();
}

void whileLoop() {
    double fahrenheit;
    double celsius;

    displayHeading();
    fahrenheit = 0;
    while (fahrenheit <= 100) {
        celsius = calculateCelsius(fahrenheit);
        displayResult(fahrenheit, celsius);
        fahrenheit += 10;
    }
    cout << endl;
}

void doLoop() {
    double fahrenheit;
    double celsius;

    displayHeading();
    fahrenheit = 0;
    do {
        celsius = calculateCelsius(fahrenheit);
        displayResult(fahrenheit, celsius);
        fahrenheit += 10;
    } while (fahrenheit <= 100);
```

```
    cout << endl;
}

void forLoop() {
    double fahrenheit;
    double celsius;

    displayHeading();
    for (fahrenheit = 0 ; fahrenheit <= 100 ; fahrenheit += 10) {
        celsius = calculateCelsius(fahrenheit);
        displayResult(fahrenheit, celsius);
    }
    cout << endl;
}

void displayHeading() {
    cout << "F°\tC°" << endl;
}

double calculateCelsius(double fahrenheit) {
    double celsius;

    celsius = (fahrenheit - 32) * 5 / 9;

    return celsius;
}

void displayResult(double fahrenheit, double celsius) {
    cout << fahrenheit << "\t" << celsius;
    cout << endl;
}
```

Output

```
F°      C°
0       -17.7778
10      -12.2222
20      -6.66667
30      -1.11111
40      4.44444
50      10
60      15.5556
70      21.1111
80      26.6667
90      32.2222
100     37.7778

F°      C°
0       -17.7778
10      -12.2222
20      -6.66667
30      -1.11111
40      4.44444
50      10
60      15.5556
70      21.1111
```

```
80      26.6667
90      32.2222
100     37.7778

F°      C°
0       -17.7778
10      -12.2222
20      -6.66667
30      -1.11111
40      4.44444
50      10
60      15.5556
70      21.1111
80      26.6667
90      32.2222
100     37.7778
```

References

- Wikiversity: Computer Programming

This page titled 27.6.4: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.5: C# Examples

Temperature

```
// This program displays a temperature conversion table showing Fahrenheit
// temperatures from 0 to 100, in increments of 10, and the corresponding
// Celsius temperatures using While, Do While, and For loops.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C_Sharp_Programming

using System;

public class Temperature
{
    public static void Main(String[] args)
    {
        WhileLoop();
        DoLoop();
        ForLoop();
    }

    private static void WhileLoop()
    {
        double fahrenheit;
        double celsius;

        DisplayHeading();
        fahrenheit = 0;
        while (fahrenheit <= 100)
        {
            celsius = CalculateCelsius(fahrenheit);
            DisplayResult(fahrenheit, celsius);
            fahrenheit += 10;
        }
        Console.WriteLine();
    }

    private static void DoLoop()
    {
        double fahrenheit;
        double celsius;

        DisplayHeading();
        fahrenheit = 0;
        do
        {
            celsius = CalculateCelsius(fahrenheit);
            DisplayResult(fahrenheit, celsius);
            fahrenheit += 10;
        } while (fahrenheit <= 100);
        Console.WriteLine();
    }
```

```
51
52      private static void ForLoop()
53      {
54          double fahrenheit;
55          double celsius;
56
57          DisplayHeading();
58          for (fahrenheit = 0 ; fahrenheit <= 100 ; fahrenheit += 10)
59          {
60              celsius = CalculateCelsius(fahrenheit);
61              DisplayResult(fahrenheit, celsius);
62          }
63          Console.WriteLine();
64      }
65
66      private static void DisplayHeading()
67      {
68          Console.WriteLine("F°\tC°");
69      }
70
71      public static double CalculateCelsius(double fahrenheit)
72      {
73          double celsius;
74
75          celsius = (fahrenheit - 32) * 5 / 9;
76
77          return celsius;
78      }
79
80      private static void DisplayResult(double fahrenheit, double celsius)
81      {
82          Console.WriteLine(fahrenheit.ToString() + "\t" + celsius.ToString());
83      }
84  }
```

Output

```
01  F°      C°
02  0       -17.7777777777778
03  10      -12.2222222222222
04  20      -6.66666666666667
05  30      -1.11111111111111
06  40      4.44444444444444
07  50      10
08  60      15.5555555555556
09  70      21.1111111111111
10  80      26.6666666666667
11  90      32.2222222222222
12  100     37.7777777777778
13
14  F°      C°
15  0       -17.7777777777778
16  10      -12.2222222222222
17  20      -6.66666666666667
18  30      -1.11111111111111
19  40      4.44444444444444
```

```
50      10
60      15.5555555555556
70      21.1111111111111
80      26.6666666666667
90      32.2222222222222
100     37.7777777777778

F°      C°
0       -17.7777777777778
10      -12.2222222222222
20      -6.66666666666667
30      -1.11111111111111
40      4.44444444444444
50      10
60      15.5555555555556
70      21.1111111111111
80      26.6666666666667
90      32.2222222222222
100     37.7777777777778
```

References

- Wikiversity: Computer Programming

This page titled 27.6.5: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.6: Java Examples

Temperature

```
// This program displays a temperature conversion table showing Fahrenheit
// temperatures from 0 to 100, in increments of 10, and the corresponding
// Celsius temperatures using While, Do While, and For loops.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/Java_Programming

import java.util.*;

class Main
{
    public static void main(String[] args) {
        whileLoop();
        doLoop();
        forLoop();
    }

    private static void whileLoop() {
        double fahrenheit;
        double celsius;

        displayHeading();
        fahrenheit = 0;
        while (fahrenheit <= 100) {
            celsius = calculateCelsius(fahrenheit);
            displayResult(fahrenheit, celsius);
            fahrenheit += 10;
        }
        System.out.println();
    }

    private static void doLoop() {
        double fahrenheit;
        double celsius;

        displayHeading();
        fahrenheit = 0;
        do {
            celsius = calculateCelsius(fahrenheit);
            displayResult(fahrenheit, celsius);
            fahrenheit += 10;
        } while (fahrenheit <= 100);
        System.out.println();
    }

    private static void forLoop() {
        double fahrenheit;
        double celsius;

```

```
        displayHeading();
        for (fahrenheit = 0 ; fahrenheit <= 100 ; fahrenheit += 10) {
            celsius = calculateCelsius(fahrenheit);
            displayResult(fahrenheit, celsius);
        }
        System.out.println();
    }

    private static void displayHeading() {
        System.out.println("F°\tC°");
    }

    private static double calculateCelsius(double fahrenheit) {
        double celsius;

        celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }

    private static void displayResult(double fahrenheit, double celsius) {
            System.out.println(String.valueOf(fahrenheit) + "\t" + String.valueOf(celsius));
    }
}
```

Output

```
F°      C°
0.0     -17.77777777777778
10.0    -12.222222222222221
20.0    -6.666666666666667
30.0    -1.1111111111111112
40.0    4.444444444444445
50.0    10.0
60.0    15.555555555555555
70.0    21.11111111111111
80.0    26.666666666666668
90.0    32.22222222222222
100.0   37.77777777777778

F°      C°
0.0     -17.77777777777778
10.0    -12.222222222222221
20.0    -6.666666666666667
30.0    -1.1111111111111112
40.0    4.444444444444445
50.0    10.0
60.0    15.555555555555555
70.0    21.11111111111111
80.0    26.666666666666668
90.0    32.22222222222222
100.0   37.77777777777778

F°      C°
0.0     -17.77777777777778
10.0    -12.222222222222221
```

```
20.0    -6.666666666666667
30.0    -1.1111111111111112
40.0    4.444444444444445
50.0    10.0
60.0    15.555555555555555
70.0    21.11111111111111
80.0    26.666666666666668
90.0    32.22222222222222
100.0   37.77777777777778
```

This page titled 27.6.6: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.7: JavaScript Examples

Temperature

```
// This program displays a temperature conversion table showing Fahrenheit
// temperatures from 0 to 100, in increments of 10, and the corresponding
// Celsius temperatures using While, Do While, and For loops.
//
// References:
//   https://www.mathsisfun.com/temperature-conversion.html
//   https://en.wikibooks.org/wiki/JavaScript

main();

function main() {
    whileLoop();
    doLoop();
    forLoop();
}

function whileLoop() {
    var fahrenheit;
    var celsius;

    displayHeading();
    fahrenheit = 0;
    while (fahrenheit <= 100) {
        celsius = calculateCelsius(fahrenheit);
        displayResult(fahrenheit, celsius);
        fahrenheit += 10;
    }
    output("");
}

function doLoop() {
    var fahrenheit;
    var celsius;

    displayHeading();
    fahrenheit = 0;
    do {
        celsius = calculateCelsius(fahrenheit);
        displayResult(fahrenheit, celsius);
        fahrenheit += 10;
    } while (fahrenheit <= 100);
    output("");
}

function forLoop() {
    var fahrenheit;
    var celsius;

    displayHeading();
    for (fahrenheit = 0 ; fahrenheit <= 100 ; fahrenheit += 10) {
```

```
        celsius = calculateCelsius(fahrenheit);
        displayResult(fahrenheit, celsius);
    }
    output("");
}

function displayHeading() {
    output("F°\tC°");
}

function calculateCelsius(fahrenheit) {
    var celsius;

    celsius = (fahrenheit - 32) * 5 / 9;

    return celsius;
}

function displayResult(fahrenheit, celsius) {
    output(fahrenheit.toString() + "\t" + celsius.toString());
}

// Checks the JavaScript environment and writes to the console,
// the current document, or standard output as appropriate.
// Reference: http://progopedia.com/example/hello-world/114/
function output(text) {
    if (typeof console === 'object') {
        console.log(text);
    }
    else if (typeof document === 'object') {
        document.write(text);
    }
    else {
        print(text);
    }
}
```

Output

```
F°      C°
0       -17.77777777777778
10      -12.222222222222221
20      -6.666666666666667
30      -1.1111111111111112
40      4.444444444444445
50      10
60      15.555555555555555
70      21.11111111111111
80      26.666666666666668
90      32.22222222222222
100     37.77777777777778

F°      C°
0       -17.77777777777778
10      -12.222222222222221
20      -6.666666666666667
```

```
30      -1.1111111111111112
40      4.444444444444445
50      10
60      15.555555555555555
70      21.11111111111111
80      26.666666666666668
90      32.22222222222222
100     37.77777777777778

F°      C°
0       -17.77777777777778
10      -12.222222222222221
20      -6.666666666666667
30      -1.1111111111111112
40      4.444444444444445
50      10
60      15.555555555555555
70      21.11111111111111
80      26.666666666666668
90      32.22222222222222
100     37.77777777777778
```

References

- Wikiversity: Computer Programming

This page titled 27.6.7: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.8: Swift Examples

Temperature

```
// This program displays a temperature conversion table showing Fahrenheit
// temperatures from 0 to 100, in increments of 10, and the corresponding
// Celsius temperatures using While, For, and Do loops.
//
// References:
// https://www.mathsisfun.com/temperature-conversion.html
// https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

func whileLoop() {
    var fahrenheit: Double
    var celsius: Double

    displayHeading()
    fahrenheit = Double(0)
    while fahrenheit <= 100 {
        celsius = calculateCelsius(fahrenheit:fahrenheit)
        displayResult(fahrenheit:fahrenheit, celsius:celsius)
        fahrenheit += 10
    }
    print()
}

func doLoop()
{
    var fahrenheit: Double
    var celsius: Double

    displayHeading()
    fahrenheit = Double(0)
    repeat
    {
        celsius = calculateCelsius(fahrenheit:fahrenheit)
        displayResult(fahrenheit:fahrenheit, celsius:celsius)
        fahrenheit += 10
    } while fahrenheit <= 100
    print()
}

func forLoop() {
    var celsius: Double

    displayHeading()
    for fahrenheit in stride(from: 0.0, through: 100.0, by: 10.0) {
        celsius = calculateCelsius(fahrenheit:fahrenheit)
        displayResult(fahrenheit:fahrenheit, celsius:celsius)
    }
    print()
}

```

```
func displayHeading() {
    print("F°\tC°")
}

func calculateCelsius(fahrenheit: Double) -> Double {
    var celsius: Double

    celsius = (fahrenheit - 32) * 5 / 9

    return celsius
}

func displayResult(fahrenheit: Double, celsius: Double) {
    print(String(fahrenheit) + "\t" + String(celsius))
}

func main()
{
    whileLoop()
    doLoop()
    forLoop()
}

main()
```

Output

```
F°      C°
0       -17.77777777777778
10      -12.222222222222221
20      -6.666666666666667
30      -1.1111111111111112
40      4.444444444444445
50      10.0
60      15.555555555555555
70      21.11111111111111
80      26.666666666666668
90      32.22222222222222
100     37.77777777777778

F°      C°
0       -17.77777777777778
10      -12.222222222222221
20      -6.666666666666667
30      -1.1111111111111112
40      4.444444444444445
50      10.0
60      15.555555555555555
70      21.11111111111111
80      26.666666666666668
90      32.22222222222222
100     37.77777777777778

F°      C°
0       -17.77777777777778
10      -12.222222222222221
```

```
20      -6.666666666666667
30      -1.1111111111111112
40      4.444444444444445
50      10.0
60      15.555555555555555
70      21.11111111111111
80      26.666666666666668
90      32.22222222222222
100     37.77777777777778
```

References

- Wikiversity: Computer Programming

This page titled 27.6.8: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.6.9: Practice- Loops

Review Questions

True / False

1. The do while and repeat until structure act exactly the same.
2. Students sometimes confuse assignment and equality.
3. The repeat until looping control structure is available in all programming languages.
4. Because flags are often used, they are usually a special data type.
5. The do while is a test before loop.
6. Only for loops can be counting loops.
7. The integer data type has modular arithmetic attributes.
8. The escape code of \n is part of formatting output.
9. Nested for loops is not allowed in the C++ programming language.
10. Counting loops use all four of the loop attributes.

Answers:

1. false
2. true
3. false
4. false
5. false
6. false
7. true
8. true
9. false
10. true

Activities

Complete the following activities using pseudocode, a flowcharting tool, or your selected programming language. Use separate functions for input, each type of processing, and output. Avoid global variables by passing parameters and returning results. Create test data to validate the accuracy of each program. Add comments at the top of the program and include references to any resources used.

While Loops

Complete the following using a while loop structure.

1. Create a program that uses a loop to generate a list of multiplication expressions for a given value. Ask the user to enter the value and the number of expressions to be displayed. For example, a list of three expressions for the value 1 would be:
   ```
   1 * 1 = 1
   1 * 2 = 2
   1 * 3 = 3
   ```
   A list of five expressions for the value 3 would be:
   ```
   3 * 1 = 3
   3 * 2 = 6
   3 * 3 = 9
   3 * 4 = 12
   3 * 5 = 15
   ```
2. Review MathsIsFun: Definition of Average. Create a program that asks the user to enter grade scores. Start by asking the

user how many scores they would like to enter. Then use a loop to request each score and add it to a total. Finally, calculate and display the average for the entered scores.
3. Review MathsIsFun: Pi. Write a program that uses the Nilakantha series to calculate Pi based on a given number of iterations entered by the user.

Do While / Repeat Until Loops

Complete the following using a do while / repeat until loop structure.

1. Review MathsIsFun: Definition of Average. Create a program that asks the user to enter grade scores. Use a loop to request each score and add it to a total. Continue accepting scores until the user enters no value (empty input). Finally, calculate and display the average for the entered scores.
2. Review Khan Academy: A guessing game. Write a program that allows the user to think of a number between 0 and 100, inclusive. Then have the program try to guess the user's number. Start at the midpoint (50) and ask the user if their number is (h)igher, (l)ower, or (e)qual to the guess. If they indicate lower, guess the new midpoint (25). If they indicate higher, guess the new midpoint (75). Continue efficiently guessing higher or lower until they indicate equal, then print the number of guesses required to guess their number and end the program.
3. Add a do while / repeat until loop to any activity from a previous chapter. Continue running the program while the user wants to continue or until the user wants to stop.
4. Add an input validation loop to any activity from a previous chapter. Verify that the input is valid before returning the value. Ask the user to input the value again while the input is invalid.

For Loops

Complete the following using a for loop structure.

1. Create a program that uses a loop to generate a list of multiplication expressions for a given value. Ask the user to enter the value and the number of expressions to be displayed. For example, a list of three expressions for the value 1 would be:
   ```
   1 * 1 = 1
   1 * 2 = 2
   1 * 3 = 3
   ```
   A list of five expressions for the value 3 would be:
   ```
   3 * 1 = 3
   3 * 2 = 6
   3 * 3 = 9
   3 * 4 = 12
   3 * 5 = 15
   ```
2. Review MathsIsFun: Definition of Average. Create a program that asks the user to enter grade scores. Start by asking the user how many scores they would like to enter. Then use a loop to request each score and add it to a total. Finally, calculate and display the average for the entered scores.
3. Review MathsIsFun: Pi. Write a program that uses the Nilakantha series to calculate Pi based on a given number of iterations entered by the user.

Nested Loops

Complete the following using a nested loop structure.

1. Review MathsIsFun: 10x Printable Multiplication Table. Create a program that uses nested loops to generate a multiplication table. Rather than simply creating a 10 by 10 table, ask the user to enter the starting and ending values. Include row and column labels. For example, the output from 1 to 3 might look like:
   ```
   1 2 3
   1 1 2 3
   2 2 4 6
   3 3 6 9
   ```
   The output from 3 to 5 might look like:
   ```
   3 4 5
   3 9 12 15
   ```

```
4 12 16 20
5 15 20 25
```

2. Add a do while / repeat until loop to any activity from this chapter. Continue running the program while the user wants to continue or until the user wants to stop.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

This page titled 27.6.9: Practice- Loops is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7: ARRAYS AND LISTS

27.7.1: Arrays and Functions
27.7.2: Dynamic Arrays
27.7.3: C++ Examples
27.7.4: C# Examples
27.7.5: Java Examples
27.7.6: JavaScript Examples
27.7.7: Python Examples
27.7.8: Swift Examples
27.7.9: Practice- Arrays and Lists

This page titled 27.7: Arrays and Lists is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 6: Arrays and Lists is licensed CC BY-SA 4.0.

27.7.1: Arrays and Functions

Overview

In modular programming, specific task functions are often created and used or reused for array processing. Array processing functions are usually passed the array and any data necessary to process the array for the given task.

It should be noted that arrays are passed by reference in most current programming languages. Array processing functions must take care not to alter the array unless intended.

Discussion

Arrays are an important complex data type used in almost all programming. We continue to concentrate on simple one dimension arrays also called a list. Most programmers develop a series of user-defined specific task functions that can be used with an array for normal processing. These functions are usually passed the array along with the number of elements within the array. Some functions also pass another piece of data needed for that particular function's task.

This module covers the displaying the array members on the monitor via calling an **array function** dedicated to that task.

Pseudocode

```
01 Function Main
02     Declare Integer Array ages[5]
03
04     Assign ages = [49, 48, 26, 19, 16]
05     Call DisplayArray(ages)
06 End
07
08 Function DisplayArray (Integer Array array)
09     Declare Integer index
10
11     For index = 0 to Size(array) - 1
12         Output array[index]
13     End
14 End
```

Output

```
01 49
02 48
03 26
04 19
05 16
```

Key Terms

array function
: A user-defined specific task function designed to process an array.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

This page titled 27.7.1: Arrays and Functions is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.2: Dynamic Arrays

Overview

A **dynamic array** is a random access, variable-size list data structure that allows elements to be added or removed. It is supplied with standard libraries in many modern programming languages. Dynamic arrays overcome a limit of static arrays, which have a fixed capacity that needs to be specified at allocation.[1]

Discussion

A limitation of the static arrays discussed so far is that the array size is determined when the array is created and/or allocated. Dynamic arrays allow elements to be added and removed at runtime. Most current programming languages include built-in or standard library functions for creating and managing dynamic arrays.

Language	Class	Add	Remove
C++	`#include <list>` `std::list`	`insert`	`erase`
C#	`System.Collections.Generic.List`	`Add`	`Remove`
Java	`java.util.ArrayList`	`add`	`remove`
JavaScript	`Array`	`push`, `splice`	`pop`, `splice`
Python	`list`	`append`	`remove`
Swift	`Array`	`append`	`remove`

Key Terms

dynamic array
: A data structure consisting of a collection of elements that allows individual elements to be added or removed.

References

1. Wikipedia: Dynamic array ↩

This page titled 27.7.2: Dynamic Arrays is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.3: C++ Examples

Arrays

```
// This program demonstrates array processing, including:
// display, total, max, min, parallel arrays, sort
// multidimensional arrays, and dynamic arrays.

#include
#include
#include
#include
#include

using namespace std;

void displayArray(int [], int);
int sum(int [], int);
int max(int [], int);
int min(int [], int);
void displayParallel(string [], int [], int);
void displayMultidimensional();
void dynamicArray();

int main() {
    string names[] = {"Lisa", "Michael", "Ashley", "Jacob", "Emily"};
    int ages[] = {49, 48, 26, 19, 16};

    displayArray(ages, sizeof(ages) / sizeof(int));

    int total = sum(ages, sizeof(ages) / sizeof(int));
    int maximum = max(ages, sizeof(ages) / sizeof(int));
    int minimum = min(ages, sizeof(ages) / sizeof(int));

    cout << "total: " << total << endl;
    cout << "maximum: " << maximum << endl;
    cout << "minimum: " << minimum << endl;

    displayParallel(names, ages, sizeof(ages) / sizeof(int));

    sort(ages, ages + sizeof(ages) / sizeof(int));
    displayArray(ages, sizeof(ages) / sizeof(int));

    displayMultidimensional();
    dynamicArray();

    return 0;
}

void displayArray(int arry[], int size) {
    for (int index = 0; index < size; index++) {
        cout << "array[" << index << "] = " << arry[index] << endl;
    }
}
```

```
51
52  int sum(int arry[], int size) {
53      int total = 0;
54      for (int index = 0; index < size; index++) {
55          total += arry[index];
56      }
57      return total;
58  }
59
60  int max(int arry[], int size) {
61      int maximum = arry[0];
62      for (int index = 1; index < size; index++) {
63          if (maximum < arry[index]) {
64              maximum = arry[index];
65          }
66      }
67      return maximum;
68  }
69
70  int min(int arry[], int size) {
71      int minimum = arry[0];
72      for (int index = 1; index < size; index++) { if (minimum > arry[index]) {
73              minimum = arry[index];
74          }
75      }
76      return minimum;
77  }
78
79  void displayParallel(string names[], int ages[], int size) {
80      for (int index = 0; index < size; index++) {
81          cout << names[index] << " is " << ages[index] << " years old" << endl;
82      }
83  }
84
85  void displayMultidimensional() {
86      string game[3][3] = {
87          {"X", "O", "X"},
88          {"O", "O", "O"},
89          {"X", "O", "X"} };
90
91      for (int row = 0; row < 3; row++) {
92          for (int column = 0; column < 3; column++) {
93              cout << (game[row][column]);
94              if (column < 2) {
95                  cout << " | ";
96              }
97          }
98          cout << endl;
99      }
100 }
101
102 void dynamicArray() {
103     list<int> arry;
104     srand (time(NULL));
105     for (int index = 0; index < 5; index++) {
106         int number = rand() % 100;
107         arry.push_back(number);
108     }
```

```
109
110     for (list<int>::iterator it = arry.begin(); it != arry.end(); it++) {
111         cout << *it << endl;
112     }
113 }
```

Output

```
01 array[0] = 49
02 array[1] = 48
03 array[2] = 26
04 array[3] = 19
05 array[4] = 16
06 total: 158
07 maximum: 49
08 minimum: 16
09 Lisa is 49 years old
10 Michael is 48 years old
11 Ashley is 26 years old
12 Jacob is 19 years old
13 Emily is 16 years old
14 array[0] = 16
15 array[1] = 19
16 array[2] = 26
17 array[3] = 48
18 array[4] = 49
19 X | O | X
20 O | O | O
21 X | O | X
22 38
23 65
24 20
25 8
26 30
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 27.7.3: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.4: C# Examples

Arrays

```
// This program demonstrates array processing, including:
// display, total, max, min, parallel arrays, sort
// multidimensional arrays, and dynamic arrays.

using System;
using System.Collections.Generic;

class Arrays {
    public static void Main (string[] args)
    {
        String[] names = {"Lisa", "Michael", "Ashley", "Jacob", "Emily"};
        int[] ages = {49, 48, 26, 19, 16};

        DisplayArray(ages);

        int total = sum(ages);
        int maximum = max(ages);
        int minimum = min(ages);

        Console.WriteLine("total: " + total);
        Console.WriteLine("maximum: " + maximum);
        Console.WriteLine("minimum: " + minimum);

        DisplayParallel(names, ages);

        System.Array.Sort(ages);
        DisplayArray(ages);

        DisplayMultidimensional();
        DynamicArray();
    }

    public static void DisplayArray(int[] array)
    {
        for (int index = 0; index < array.Length; index++)
        {
            Console.WriteLine("array[" + index + "] = " + array[index]);
        }
    }

    public static int sum(int[] array)
    {
        int total = 0;
        for (int index = 0; index < array.Length; index++)
        {
            total += array[index];
        }
        return total;
    }

```

```
    public static int max(int[] array)
    {
        int maximum = array[0];
        for (int index = 1; index < array.Length; index++)
        {
            if (maximum < array[index])
            {
                maximum = array[index];
            }
        }
        return maximum;
    }

    public static int min(int[] array)
    {
        int minimum = array[0];
        for (int index = 1; index < array.Length; index++) { if (minimum > array[index])
            {
                minimum = array[index];
            }
        }
        return minimum;
    }

    public static void DisplayParallel(String[] names, int[] ages)
    {
        for (int index = 0; index < names.Length; index++)
        {
            Console.WriteLine(names[index] + " is " +
                ages[index] + " years old");
        }
    }

    public static void DisplayMultidimensional()
    {
        String[,] game = new String[,]
        {
            {"X", "O", "X"},
            {"O", "O", "O"},
            {"X", "O", "X"}
        };

        for (int row = 0; row < 3; row++)
        {
            for (int column = 0; column < 3; column++)
            {
                Console.Write(game[row, column]);
                if (column < 2)
                {
                    Console.Write(" | ");
                }
            }
            Console.WriteLine();
        }
    }

    public static void DynamicArray()
    {
```

```
        List<int> array = new List<int>();

        Random random = new Random();
        for (int index = 0; index < 5; index++)
        {
            int number = random.Next(0, 100);
            array.Add(number);
        }
        for (int index = 0; index < array.Count; index++)
        {
            Console.WriteLine("array[" + index + "] = " + array[index]);
        }
    }
}
```

Output

```
array[0] = 49
array[1] = 48
array[2] = 26
array[3] = 19
array[4] = 16
total: 158
maximum: 49
minimum: 16
Lisa is 49 years old
Michael is 48 years old
Ashley is 26 years old
Jacob is 19 years old
Emily is 16 years old
array[0] = 16
array[1] = 19
array[2] = 26
array[3] = 48
array[4] = 49
X | O | X
O | O | O
X | O | X
array[0] = 85
array[1] = 75
array[2] = 54
array[3] = 66
array[4] = 36
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 27.7.4: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.5: Java Examples

Arrays

```
// This program demonstrates array processing, including:
// display, total, max, min, parallel arrays, sort
// multidimensional arrays, and dynamic arrays.

import java.util.*;

class Main {
    public static void main(String[] args) {
        String[] names = {"Lisa", "Michael", "Ashley", "Jacob", "Emily"};
        int[] ages = {49, 48, 26, 19, 16};

        displayArray(ages);

        int total = sum(ages);
        int maximum = max(ages);
        int minimum = min(ages);

        System.out.println("total: " + total);
        System.out.println("maximum: " + maximum);
        System.out.println("minimum: " + minimum);

        displayParallel(names, ages);

        Arrays.sort(ages);
        displayArray(ages);

        displayMultidimensional();
        dynamicArray();
    }

    public static void displayArray(int[] array) {
        for (int index = 0; index < array.length; index++) {
            System.out.println("array[" + index + "] = " + array[index]);
        }
    }

    public static int sum(int[] array) {
        int total = 0;
        for (int index = 0; index < array.length; index++) {
            total += array[index];
        }
        return total;
    }

    public static int max(int[] array) {
        int maximum = array[0];
        for (int index = 1; index < array.length; index++) {
            if (maximum < array[index]) {
                maximum = array[index];
            }
```

```
        }
        return maximum;
    }

    public static int min(int[] array) {
        int minimum = array[0];
        for (int index = 1; index < array.length; index++) { if (minimum > array[index]) {
                minimum = array[index];
            }
        }
        return minimum;
    }

    public static void displayParallel(String[] names, int[] ages) {
        for (int index = 0; index < names.length; index++) {
            System.out.println(names[index] + " is " +
                ages[index] + " years old");
        }
    }

    public static void displayMultidimensional() {
        String[][] game = {
            {"X", "O", "X"},
            {"O", "O", "O"},
            {"X", "O", "X"} };

        for (int row = 0; row < 3; row++) {
            for (int column = 0; column < 3; column++) {
                System.out.print(game[row][column]);
                if (column < 2) {
                    System.out.print(" | ");
                }
            }
            System.out.println();
        }
    }

    public static void dynamicArray() {
        ArrayList<Integer> array = new ArrayList<Integer>();

        for (int index = 0; index < 5; index++) {
            int number = (int) Math.floor(Math.random() * 100);
            array.add(number);
        }
        System.out.println(array);
    }
}
```

Output

```
array[0] = 49
array[1] = 48
array[2] = 26
array[3] = 19
array[4] = 16
total: 158
```

```
maximum: 49
minimum: 16
Lisa is 49 years old
Michael is 48 years old
Ashley is 26 years old
Jacob is 19 years old
Emily is 16 years old
array[0] = 16
array[1] = 19
array[2] = 26
array[3] = 48
array[4] = 49
X | O | X
O | O | O
X | O | X
[49, 83, 75, 13, 17]
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 27.7.5: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.6: JavaScript Examples

Arrays

```
// This program demonstrates array processing, including:
// display, total, max, min, parallel arrays, sort
// multidimensional arrays, and dynamic arrays.

main()

function main() {
    var names = ['Lisa', 'Michael', 'Ashley', 'Jacob', 'Emily'];
    var ages = [49, 48, 26, 19, 16];

    displayArray(names);
    displayArray(ages);

    var total = sum(ages);
    var maximum = max(ages);
    var minimum = min(ages);

    output('total: ' + total);
    output('maximum: ' + maximum);
    output('minimum: ' + minimum);

    displayParallel(names, ages);

    ages.sort();
    displayArray(ages);

    displayMultidimensional();
    dynamicArray();
}

function displayArray(array) {
    for (var index = 0; index < array.length; index++) {
        output('array[' + index + '] = ' + array[index]);
    }
}

function sum(array) {
    var total = 0;
    for (var index = 0; index < array.length; index++) {
        total += array[index];
    }
    return total;
}

function max(array) {
    var maximum = array[0];
    for (var index = 1; index < array.length; index++) {
        if (maximum < array[index]) {
            maximum = array[index];
        }
```

```
    }
    return maximum;
}

function min(array) {
    var minimum = array[0];
    for (var index = 1; index < array.length; index++) { if (minimum > array[index]) {
            minimum = array[index];
        }
    }
    return minimum;
}

function displayParallel(names, ages) {
    for (var index = 0; index < names.length; index++) {
        output(names[index] + ' is ' + ages[index] + ' years old');
    }
}

function displayMultidimensional() {
    var game = [
        ['X', 'O', 'X'],
        ['O', 'O', 'O'],
        ['X', 'O', 'X'] ];

    for (var row = 0; row < 3; row++) {
        var line = '';
        for (var column = 0; column < 3; column++) {
            line += game[row][column];
            if (column < 2) {
                line += ' | ';
            }
        }
        output(line);
    }
}

function dynamicArray() {
    var array = []

    for (var index = 0; index < 5; index++) {
        var number = Math.floor(Math.random() * 100);
        array.push(number);
    }
    displayArray(array);
}

// Checks the JavaScript environment and writes to the console,
// the current document, or standard output as appropriate.
// Reference: http://progopedia.com/example/hello-world/114/
function output(text) {
    if (typeof console === 'object') {
        console.log(text);
    }
    else if (typeof document === 'object') {
        document.write(text);
    }
    else {
```

```
        print(text);
    }
}
```

Output

```
array[0] = Lisa
array[1] = Michael
array[2] = Ashley
array[3] = Jacob
array[4] = Emily
array[0] = 49
array[1] = 48
array[2] = 26
array[3] = 19
array[4] = 16
total: 158
maximum: 49
minimum: 16
Lisa is 49 years old
Michael is 48 years old
Ashley is 26 years old
Jacob is 19 years old
Emily is 16 years old
array[0] = 16
array[1] = 19
array[2] = 26
array[3] = 48
array[4] = 49
X | O | X
O | O | O
X | O | X
array[0] = 30
array[1] = 97
array[2] = 87
array[3] = 42
array[4] = 40
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 27.7.6: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.7: Python Examples

Arrays

```
# This program demonstrates array processing, including:
# display, total, max, min, parallel arrays, sort
# multidimensional arrays, and dynamic arrays.


def display_array(array):
    for index in range(len(array)):
        print('array[' + str(index) + '] = ' +
            str(array[index]))


def sum(array):
    total = 0
    for index in range(len(array)):
        total += array[index]
    return total


def max(array):
    maximum = array[0]
    for index in range(1, len(array)):
        if maximum < array[index]:
            maximum = array[index]
    return maximum


def min(array):
    minimum = array[0]
    for index in range(1, len(array)):
        if minimum > array[index]:
            minimum = array[index]
    return minimum


def display_parallel(names, ages):
    for index in range(len(names)):
        print(names[index] + ' is ' +
            str(ages[index]) + ' years old')


def display_multidimensional():
    game = [
        ['X', 'O', 'X'],
        ['O', 'O', 'O'],
        ['X', 'O', 'X'] ]

    for row in range (0, 3):
        for column in range(0, 3):
            print(game[row][column], end='')
            if column < 2:
```

```
                print(' | ', end='')
        print()


def dynamic_array():
    import random

    array = []
    for index in range(5):
        array.append(random.randint(0, 100))
    display_array(array)


def main():
    names = ['Lisa', 'Michael', 'Ashley', 'Jacob', 'Emily']
    ages = [49, 48, 26, 19, 16]

    display_array(names)
    display_array(ages)

    total = sum(ages)
    maximum = max(ages)
    minimum = min(ages)

    print('total: ' + str(total))
    print('maximum: ' + str(maximum))
    print('minimum: ' + str(minimum))

    display_parallel(names, ages)

    ages.sort()
    display_array(ages)

    display_multidimensional()
    dynamic_array()


main()
```

Output

```
array[0] = Lisa
array[1] = Michael
array[2] = Ashley
array[3] = Jacob
array[4] = Emily
array[0] = 49
array[1] = 48
array[2] = 26
array[3] = 19
array[4] = 16
total: 158
nmaximum: 49
minimum: 16
Lisa is 49 years old
Michael is 48 years old
```

```
Ashley is 26 years old
Jacob is 19 years old
Emily is 16 years old
array[0] = 16
array[1] = 19
array[2] = 26
array[3] = 48
array[4] = 49
X | O | X
O | O | O
X | O | X
array[0] = 71
array[1] = 39
array[2] = 80
array[3] = 35
array[4] = 40
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++

This page titled 27.7.7: Python Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.8: Swift Examples

Arrays

```
// This program demonstrates array processing, including:
// display, total, max, min, parallel arrays, sort
// multidimensional arrays, and dynamic arrays.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

import Foundation

func displayArray(array: [Int]) {
    for index in 0...array.count - 1 {
        print("array[" + String(index) + "] = " + String(array[index]))
    }
}

func displayParallel(names:[String], ages:[Int]) {
    for index in 0...names.count - 1 {
        print(names[index] + " is " + String(ages[index]))
    }
}

func displayMultidimensional() {
    var game: [[String]]

    game = [
        ["X", "O", "X"],
        ["O", "X", "O"],
        ["X", "O", "X"]
    ]

    for row in 0...2 {
        for column in 0...2 {
            print(game[row][column], terminator:"")
            if column < 2 {
                print(" | ", terminator:"")
            }
        }
        print()
    }
}

func dynamicArray() {
    var array: [Int] = []

    srand(UInt32(time(nil)))
    for _ in 0...4 {
        array.append(random() % 100)
    }
```

```
    print(array)
}

func main() {
    var names: [String]
    var ages: [Int]
    var total: Int
    var maximum: Int
    var minimum: Int

    names = ["Lisa", "Michael", "Ashley", "Jacob", "Emily"]
    ages = [49, 48, 26, 19, 16]

    displayArray(array:ages)
    total = ages.reduce(0, +)
    maximum = ages.max()!
    minimum = ages.min()!

    print("total:", total)
    print("maximum:", maximum)
    print("minimum:", minimum)

    displayParallel(names:names, ages:ages)

    ages.sort()
    displayArray(array:ages)

    displayMultidimensional()
    dynamicArray()
}

main()
```

Output

```
array[0] = 49
array[1] = 48
array[2] = 26
array[3] = 19
array[4] = 16
total: 158
maximum: 49
minimum: 16
Lisa is 49
Michael is 48
Ashley is 26
Jacob is 19
Emily is 16
array[0] = 16
array[1] = 19
array[2] = 26
array[3] = 48
array[4] = 49
X | O | X
O | X | O
X | O | X
```

```
22 [93, 3, 7, 35, 59]
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

This page titled 27.7.8: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.7.9: Practice- Arrays and Lists

Review Questions

True / False

1. The array data type is one of the standard data types in C++.
2. Arrays can have more than one dimension.
3. For loops are often used to display the members of an array.
4. When defining an array, it is preferable to specify how many members are in the array.
5. Arrays are rarely used to represent data.
6. Linear searches require complex algorithms.
7. Functions are often created for searching for the max and min values in an array.
8. The bubble sort is an easy way to arrange data an array.
9. There is only one method of bubble sorting.
10. Sorting an array is frequently done.

Answers:

1. false
2. true
3. true
4. false
5. false
6. false
7. true
8. true
9. false
10. true

Short Answer

1. Briefly explain what an array is and list the two common operators used with arrays.
2. Give a short explanation of bubble sorting.

Activities

Complete the following activities using pseudocode, a flowcharting tool, or your selected programming language. Use separate functions for input, each type of processing, and output. Avoid global variables by passing parameters and returning results. Create test data to validate the accuracy of each program. Add comments at the top of the program and include references to any resources used.

1. Review MathsIsFun: Definition of Average. Create a program that asks the user to enter grade scores. Start by asking the user how many scores they would like to enter. Then use a loop to request each score and add it to a static (fixed-size) array. After the scores are entered, calculate and display the high, low, and average for the entered scores.
2. Review MathsIsFun: Leap Years. Create a program that asks the user for a year, and then calculate whether or not the given year is a leap year. Build an array where each entry is the number of days in the corresponding month (January = 31, February = 28 or 29 depending on year, March = 31, etc.). Build a parallel string array with the names of each month. Then ask the user to enter a month number, and look up the corresponding month name and number of days and display the information. Continue accepting input and displaying results until the user enters a number less than 1 or greater than 12.
3. Review Wikipedia: Zeller's congruence. Create a program that asks the user for their birthday (year, month, and day) and then calculates and displays the day of the week on which they were born. Use an array lookup to convert the numeric day of the week to the correct string representation (Monday, Tuesday, Wednesday, etc.).

4. Review Khan Academy: A guessing game. Write a program that allows the user to think of a number between 0 and 100, inclusive. Then have the program try to guess their number. Start at the midpoint (50) and ask the user if their number is (h)igher, (l)ower, or (e)qual to the guess. If they indicate lower, guess the new midpoint (25). If they indicate higher, guess the new midpoint (75). Record each guess in an an array and continue efficiently guessing higher or lower until they indicate equal, then display the list of guesses required to guess their number and end the program.
5. Review Wikipedia: Monty Hall problem. Create a program that uses an array to simulate the three doors. Use 0 (zero) to indicate goats and 1 (one) to indicate the car. Clear each "door" and then use a random number function to put the number 1 in one of the array elements. Then use the random number function to randomly select one of the three elements. Run the simulation in a loop 100 times to confirm a 1/3 chance of winning. Then run the simulation again, this time switching the selection after a 0 (goat) is removed from the remaining choices. Run the simulation in a loop 100 times to confirm a 2/3 chance of winning by switching.
6. If your programming language supports it, use a built-in sort function to sort the grade scores from the activity above and display the array in order from highest score to lowest score.
7. If your programming language supports it, update one or more of the programs above to replace the static array with a dynamic array, and extend the array as each item is added to the array. Continue accepting scores until the user enters no value (empty input).

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

This page titled 27.7.9: Practice- Arrays and Lists is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.8: STRINGS AND FILES

27.8.1: C++ Examples
27.8.2: C# Examples
27.8.3: Java Examples
27.8.4: JavaScript Examples
27.8.5: Swift Examples
27.8.6: Practice- Strings and Files

This page titled 27.8: Strings and Files is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 7: Strings and Files is licensed CC BY-SA 4.0.

27.8.1: C++ Examples

Strings

```
#include <algorithm>
#include <iostream>
#include <string>

using namespace std;

string toLower(string);
string toUpper(string);

int main() {
    string str = "Hello";

    cout << "string: " << str << endl;
    cout << "tolower: " << toLower(str) << endl;
    cout << "toupper: " << toUpper(str) << endl;
    cout << "string.find('e'): " << str.find('e') << endl;
    cout << "string.length(): " << str.length() << endl;
    cout << "string.replace(0, 1, \"j\"): " << str.replace(0, 1, "j") << endl;
    cout << "string.substr(2, 2): " << str.substr(2, 2) << endl;

    string name = "Bob";
    double value = 123.456;
    cout << name << " earned $" << fixed << setprecision (2) << value << endl;

}

string toLower(string str) {
    transform(str.begin(), str.end(), str.begin(), ::tolower);

    return str;
}

string toUpper(string str) {
    transform(str.begin(), str.end(), str.begin(), ::toupper);

    return str;
}
```

Output

```
string: Hello
tolower: hello
toupper: HELLO
string.find('e'): 1
string.length(): 5
string.replace(0, 1, "j"): jello
string.substr(2, 2): ll
Bob earned $123.46
```

Files

```
// This program creates a file, adds data to the file, displays the file,
// appends more data to the file, displays the file, and then deletes the file.
// It will not run if the file already exists.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C%2B%2B_Programming

#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>

using namespace std;

double calculateFahrenheit(double celsius);
void createFile(string);
void readFile(string);
void appendFile(string);
void deleteFile(string);
int fileExists(string);

int main() {
    string FILENAME = "~file.txt";

    if(fileExists(FILENAME)) {
        cout << "File already exists." << endl;
    } else {
        createFile(FILENAME);
        readFile(FILENAME);
        appendFile(FILENAME);
        readFile(FILENAME);
        deleteFile(FILENAME);
    }
}

double calculateFahrenheit(double celsius) {
    double fahrenheit;

    fahrenheit = celsius * 9 / 5 + 32;
    return fahrenheit;
}

void createFile(string filename) {
    fstream file;
    float celsius;
    float fahrenheit;

    file.open(filename, fstream::out);
    if (file.is_open()) {
        file << "Celsius,Fahrenheit\n";
        for(celsius = 0; celsius <= 50; celsius++) {
            fahrenheit = calculateFahrenheit(celsius);
            file << fixed << setprecision (1) << celsius << "," << fahrenheit << endl;
        }
```

```
        file.close();
    } else {
        cout << "Error creating " << filename << endl;
    }
}

void readFile(string filename)
{
    fstream file;
    string line;

    file.open(filename, fstream::in);
    if (file.is_open()) {
        while (getline(file, line))
        {
            cout << line << endl;
        }
        file.close();
        cout << endl;
    } else {
        cout << "Error reading " << filename << endl;
    }
}

void appendFile(string filename)
{
    fstream file;
    float celsius;
    float fahrenheit;

    file.open(filename, fstream::out | fstream::app);
    if (file.is_open()) {
        for(celsius = 51; celsius <= 100; celsius++) {
            fahrenheit = calculateFahrenheit(celsius);
            file << fixed << setprecision (1) << celsius << "," << fahrenheit << endl;
        }
        file.close();
    } else {
        cout << "Error appending to " << filename << endl;
    }
}

void deleteFile(string filename)
{
    remove(filename.c_str());
}

int fileExists(string filename)
{
    FILE *file;

    file = fopen (filename.c_str(), "r");
    if (file != NULL)
    {
        fclose (file);
    }
    return (file != NULL);
}
```

Output

```
Celsius,Fahrenheit
0.0,32.0
1.0,33.8
2.0,35.6
...
98.0,208.4
99.0,210.2
100.0,212.0
```

References

- Wikiversity: Computer Programming

This page titled 27.8.1: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.8.2: C# Examples

Strings

```
// This program demonstrates string functions.

using System;

class Strings
{
    public static void Main (string[] args)
    {
        String str = "Hello";

        Console.WriteLine("string: " + str);
        Console.WriteLine("string.ToLower(): " + str.ToLower());
        Console.WriteLine("string.ToUpper(): " + str.ToUpper());
        Console.WriteLine("string.IndexOf('e'): " + str.IndexOf('e'));
        Console.WriteLine("string.Length: " + str.Length);
        Console.WriteLine("string.Replace('H', 'j'): " + str.Replace('H', 'j'));
        Console.WriteLine("string(Substring(2, 2)): " + str.Substring(2, 2));
        Console.WriteLine("string.Trim(): " + str.Trim());

        String name = "Bob";
        double value = 123.456;
        Console.WriteLine(String.Format("{0} earned {1:$0.00}", name, value));
    }
}
```

Output

```
string: Hello
string.ToLower(): hello
string.ToUpper(): HELLO
string.IndexOf('e'): 1
string.Length: 5
string.Replace('H', 'j'): jello
string(Substring(2, 2)): ll
string.Trim(): Hello
Bob earned $123.46
```

Files

```
// This program creates a file, adds data to the file, displays the file,
// appends more data to the file, displays the file, and then deletes the file.
// It will not run if the file already exists.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C_Sharp_Programming

```

```
using System;

public class Files
{
    public static void Main(String[] args)
    {
        string FILENAME = "~file.txt";

        if(System.IO.File.Exists(FILENAME))
        {
            System.Console.WriteLine("File already exists.\n");
        }
        else
        {
            CreateFile(FILENAME);
            ReadFile(FILENAME);
            AppendFile(FILENAME);
            ReadFile(FILENAME);
            DeleteFile(FILENAME);
        }
    }

    private static double CalculateFahrenheit(double celsius)
    {
        double fahrenheit;

        fahrenheit = celsius * 9 / 5 + 32;
        return fahrenheit;
    }

    private static void CreateFile(string filename)
    {
        System.IO.StreamWriter file;
        double celsius;
        double fahrenheit;

        try
        {
            file = System.IO.File.CreateText(filename);
            file.WriteLine("Celsius,Fahrenheit");
            for(celsius = 0; celsius <= 50; celsius++)
            {
                fahrenheit = CalculateFahrenheit(celsius);
                file.WriteLine(String.Format("{0:F1},{1:F1}", celsius, fahrenheit));
            }
            file.Close();
        }
        catch(Exception exception)
        {
            Console.WriteLine("Error creating " + filename);
            Console.WriteLine(exception.Message);
        }
    }

    private static void ReadFile(string filename)
    {
        System.IO.StreamReader file;
        string line;
```

```
67
68          try
69          {
70              file = System.IO.File.OpenText(filename);
71              while (true)
72              {
73                  line = file.ReadLine();
74                  if (line == null)
75                  {
76                      break;
77                  }
78                  Console.WriteLine(line);
79              }
80              file.Close();
81              Console.WriteLine("");
82          }
83          catch(Exception exception)
84          {
85              Console.WriteLine("Error reading " + filename);
86              Console.WriteLine(exception.Message);
87          }
88      }
89
90      private static void AppendFile(string filename)
91      {
92          System.IO.StreamWriter file;
93          double celsius;
94          double fahrenheit;
95
96          try
97          {
98              file = System.IO.File.AppendText(filename);
99              for(celsius = 51; celsius <= 100; celsius++)
100             {
101                 fahrenheit = CalculateFahrenheit(celsius);
102                 file.WriteLine(String.Format("{0:F1},{1:F1}", celsius, fahrenheit));
103             }
104             file.Close();
105         }
106         catch(Exception exception)
107         {
108             Console.WriteLine("Error appending to " + filename);
109             Console.WriteLine(exception.Message);
110         }
111     }
112
113     private static void DeleteFile(string filename)
114     {
115         try
116         {
117             System.IO.File.Delete(filename);
118         }
119         catch(Exception exception)
120         {
121             Console.WriteLine("Error deleting " + filename);
122             Console.WriteLine(exception.Message);
123         }
124     }
```

```
125 }
```

Output

```
01 Celsius,Fahrenheit
02 0.0,32.0
03 1.0,33.8
04 2.0,35.6
05 ...
06 98.0,208.4
07 99.0,210.2
08 100.0,212.0
```

References

- Wikiversity: Computer Programming

This page titled 27.8.2: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.8.3: Java Examples

Strings

```
// This program demonstrates string functions.

class Main {
    public static void main(String[] args) {
        String str = "Hello";

        System.out.println("string: " + str);
        System.out.println("string.toLowerCase(): " + str.toLowerCase());
        System.out.println("string.toUpperCase(): " + str.toUpperCase());
        System.out.println("string.indexOf('e'): " + str.indexOf('e'));
        System.out.println("string.length(): " + str.length());
        System.out.println("string.replace('H', 'j'): " + str.replace('H', 'j'));
        System.out.println("string(substring(2,4): " + str.substring(2, 4));
        System.out.println("string.trim(): " + str.trim());

        String name = "Bob";
        double value = 123.456;
        System.out.println(String.format("%s earned $%.2f", name, value));
    }
}
```

Output

```
string: Hello
string..toLowerCase(): hello
string.toUpperCase(): HELLO
string.indexOf('e'): 1
string.length(): 5
string.replace('H', 'j'): jello
string(substring(2,4): ll
string.trim(): Hello
Bob earned $123.46
```

Files

```
// This program creates a file, adds data to the file, displays the file,
// appends more data to the file, displays the file, and then deletes the file.
// It will not run if the file already exists.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/Java_Programming

import java.util.*;

class Main {
    public static void main(String[] args) {
```

```
        String FILENAME = "~file.txt";

        if(fileExists(FILENAME)) {
            System.out.println("File already exists.\n");
        } else {
            createFile(FILENAME);
            readFile(FILENAME);
            appendFile(FILENAME);
            readFile(FILENAME);
            deleteFile(FILENAME);
        }
    }

    private static double calculateFahrenheit(double celsius) {
        double fahrenheit;

        fahrenheit = celsius * 9 / 5 + 32;
        return fahrenheit;
    }

    private static void createFile(String filename) {
        try {
            java.io.File file = new java.io.File(filename);
            java.io.BufferedWriter writer =
                new java.io.BufferedWriter(new java.io.FileWriter(file));
            double celsius;
            double fahrenheit;

            writer.write("Celsius,Fahrenheit\n");
            for(celsius = 0; celsius <= 50; celsius++) {
                fahrenheit = calculateFahrenheit(celsius);
                writer.write(celsius + "," + fahrenheit + "\n");
            }
            writer.close();
        } catch(Exception exception) {
            System.out.println("Error creating " + filename);
            exception.printStackTrace();
        }
    }

    private static void readFile(String filename) {
        try {
            java.io.File file = new java.io.File(filename);
            java.io.BufferedReader reader =
                new java.io.BufferedReader(new java.io.FileReader(file));
            String line;

            while(true) {
                line = reader.readLine();
                if (line == null) {
                    break;
                }
                System.out.println(line);
            }
            reader.close();
            System.out.println("");
        } catch(Exception exception) {
            System.out.println("Error reading " + filename);
```

```
            exception.printStackTrace();
        }
    }

    private static void appendFile(String filename)
    {
        try {
            java.io.File file = new java.io.File(filename);
            java.io.BufferedWriter writer =
                new java.io.BufferedWriter(new java.io.FileWriter(file, true));
            double celsius;
            double fahrenheit;

            for(celsius = 51; celsius <= 100; celsius++) {
                fahrenheit = calculateFahrenheit(celsius);
                writer.write(celsius + "," + fahrenheit + "\n");
            }
            writer.close();
        } catch(Exception exception) {
            System.out.println("Error appending to " + filename);
            exception.printStackTrace();
        }
    }

    private static void deleteFile(String filename) {
        java.io.File file;

        try {
            file = new java.io.File(filename);
            file.delete();
        } catch(Exception exception) {
            System.out.println("Error deleting " + filename);
            exception.printStackTrace();
        }

    }

    private static boolean fileExists(String filename) {
        java.io.File file;

        file = new java.io.File(filename);
        return file.exists();
    }
}
```

Output

```
Celsius,Fahrenheit
0.0,32.0
1.0,33.8
2.0,35.6
...
98.0,208.4
99.0,210.2
100.0,212.0
```

References

- Wikiversity: Computer Programming

This page titled 27.8.3: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.8.4: JavaScript Examples

Strings

```
// This program demonstrates string functions.

main();

function main()
{
    var str = "Hello";

    output("string: " + str);
    output("string.toLowerCase(): " + str.toLowerCase());
    output("string.toUpperCase(): " + str.toUpperCase());
    output("string.indexOf('e'): " + str.indexOf('e'));
    output("string.length: " + str.length);
    output("string.replace('H', 'j'): " + str.replace('H', 'j'));
    output("string(substring(2,4): " + str.substring(2, 4));
    output("string.trim(): " + str.trim());

    var name = "Bob";
    var value = 123.456;
    output(`string.format(): ${name} earned $${value.toFixed(2)}`);
}

// Checks the JavaScript environment and writes to the console,
// the current document, or standard output as appropriate.
// Reference: http://progopedia.com/example/hello-world/114/
function output(text) {
    if (typeof console === "object") {
      console.log(text);
    }
    else if (typeof document === "object") {
      document.write(text);
    }
    else {
      print(text);
    }
}
```

Output

```
string: Hello
string..toLowerCase(): hello
string.toUpperCase(): HELLO
string.indexOf('e'): 1
string.length: 5
string.replace('H', 'j'): jello
string(substring(2,4): ll
string.trim(): Hello
string.format(): Bob earned $123.46
```

Files

Note: For security reasons, JavaScript in a browser requires the user to select the file to be processed. This example is based on node.js rather than browser-based JavaScript.

```
// This program creates a file, adds data to the file, displays the file,
// appends more data to the file, displays the file, and then deletes the file.
// It will not run if the file already exists.

function calculateFahrenheit(celsius) {
    fahrenheit = celsius * 9 / 5 + 32
    return fahrenheit
}

function createFile(filename) {
    var fs = require('fs')

    fs.writeFile(filename, "Celsius,Fahrenheit\n",  function(err) {
        if (err) return console.error(err);
    });

    for(var celsius = 0; celsius <= 50; celsius++) {
        var fahrenheit = calculateFahrenheit(celsius);
        fs.appendFile(filename, celsius.toFixed(1) + "," +
            fahrenheit.toFixed(1) + "\n", function (err) {
            if (err) {
                return console.error(err);
            }
        });
    }
}

function readFile(filename) {
    var file = require('readline').createInterface( {
      input: require('fs').createReadStream(filename)
    });

    file.on('line', function (line) {
        console.log(line);
    });
}

function appendFile(filename) {
    var fs = require('fs')

    for(var celsius = 51; celsius <= 100; celsius++) {
        var fahrenheit = calculateFahrenheit(celsius);
        fs.appendFile(filename, celsius.toFixed(1) + "," +
            fahrenheit.toFixed(1) + "\n", function (err) {
            if (err) {
                return console.error(err);
            }
        });
    }
}

function deleteFile(filename) {
```

```
    var fs = require("fs");

    fs.unlink(filename, function(err) {
        if (err) {
            return console.error(err);
        }
    });
}

function fileExists(filename) {
    var fs = require('fs');
    return fs.existsSync(filename);
}

function main() {
    var filename = "~file.txt";

    if(fileExists(filename)) {
        console.log("File already exists.")
    } else {
        createFile(filename);
        readFile(filename);
        appendFile(filename);
        readFile(filename);
        deleteFile(filename);
    }
}

main();
```

Output

```
Celsius,Fahrenheit
0.0,32.0
1.0,33.8
2.0,35.6
...
98.0,208.4
99.0,210.2
100.0,212.0
```

References

- Wikiversity: Computer Programming

This page titled 27.8.4: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.8.5: Swift Examples

Strings

```
// This program demonstrates string functions.

import Foundation

func main() {
    let string:String = "Hello"

    print("string: " + string)
    print("string.lowercased(): " + string.lowercased())
    print("string.uppercased(): " + string.uppercased())
    print("find(string, \"e\"): " + String(find(string:string, character:"e")))
    print("string.count: " + String(string.count))
    print("string.replacingOccurrences(of:\"H\", with:\"j\"): " +
string.replacingOccurrences(of:"H", with:"j"))
    print("string.reversed(): " + String(string.reversed()))
    print("substring(2, 2): " + substring(string:string, start:2, length:2))
    print("string.trimmingCharacters(\"H\"): " +
string.trimmingCharacters(in:CharacterSet.init(charactersIn: "H")))

    let name:String = "Bob"
    let value:Double = 123.456
    print("\(name) earned $" + String(format:"%.2f", value))
}

func find(string:String, character:Character) -> Int {
    var result: Int

    if let index = string.firstIndex(of:character) {
        result = string.distance(from: string.startIndex, to: index)
    } else {
        result = -1
    }
    return result
}

func substring(string:String, start:Int, length:Int) -> String {
    let startIndex = string.index(string.startIndex, offsetBy: start)
    let endIndex = string.index(string.startIndex, offsetBy: start + length - 1)
    return String(string[startIndex...endIndex])
}

main()
```

Output

```
string: Hello
string.lowercased(): hello
string.uppercased(): HELLO
find(string, "e"): 1
```

```
string.count: 5
string.replacingOccurrences(of:"H", with:"j"): jello
string.reversed(): olleH
substring(2, 2): ll
string.trimmingCharacters("H"): ello
Bob earned $123.46
```

Files

```
// This program creates a file, adds data to the file, displays the file,
// appends more data to the file, displays the file, and then deletes the file.
// It will not run if the file already exists.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

import Foundation

func fileExists(filename:String) -> Bool {
    let fileManager = FileManager.default
    return fileManager.fileExists(atPath:filename)
}

func calculateFahrenheit(celsius:Double) -> Double {
    var fahrenheit: Double
    fahrenheit = celsius * 9 / 5 + 32
    return fahrenheit
}

func createFile(filename:String) {
    var text: String
    var fahrenheit: Double

    text = "Celsius,Fahrenheit\n"
    for celsius in stride(from: 0.0, through: 50.0, by: 1.0) {
        fahrenheit = calculateFahrenheit(celsius:celsius)
        text += String(celsius) + "," + String(fahrenheit) + "\n"
    }

    do {
        try text.write(toFile: filename, atomically: true, encoding: .utf8)
    } catch {
        print("Error creating ", filename)
        print(error.localizedDescription)
    }
}

func readFile(filename:String) {
    var text = ""

    do {
        text = try String(contentsOfFile: filename, encoding: .utf8)

        let lines = text.components(separatedBy:"\n")
```

```
        for line in lines {
            print(line)
        }
    } catch {
        print("Error reading " + filename)
        print(error.localizedDescription)
    }
}

func appendFile(filename:String) {
    var text: String
    var fahrenheit: Double

    do {
        text = try String(contentsOfFile: filename, encoding: .utf8)

        for celsius in stride(from: 51.0, through: 100.0, by: 1.0) {
            fahrenheit = calculateFahrenheit(celsius:celsius)
            text += String(celsius) + "," + String(fahrenheit) + "\n"
        }

        try text.write(toFile: filename, atomically: true, encoding: .utf8)
    } catch {
        print("Error appending to ", filename)
        print(error.localizedDescription)
    }
}

func deleteFile(filename:String) {
    do {
        let fileManager = FileManager.default
        try fileManager.removeItem(atPath:filename)
    } catch {
        print("Error deleting", filename)
        print(error.localizedDescription)
    }
}

func main() {
    let filename:String = "~file.txt"

    if (fileExists(filename:filename)) {
        print("File already exists.")
    }
    else {
        createFile(filename:filename)
        readFile(filename:filename)
        appendFile(filename:filename)
        readFile(filename:filename)
        deleteFile(filename:filename)
    }
}

main()
```

Output

```
Celsius,Fahrenheit
0.0,32.0
1.0,33.8
2.0,35.6
3.0,37.4
...
98.0,208.4
99.0,210.2
100.0,212.0
```

References

- Wikiversity: Computer Programming

This page titled 27.8.5: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.8.6: Practice- Strings and Files

Review Questions

True / False

1. The character data type in C++ uses the double quote marks, like: char grade = “A”;
2. Sizeof is an operator that tells you how many bytes a data type occupies in storage.
3. Typedef helps people who can’t hear and is one of the standard accommodation features of a programming language for people with a learning disability.
4. The sequence operator should be used when defining variables in order to save space.
5. Programming can be both enjoyable and frustrating.
6. Text files are hard to create.
7. A filespec refers to a very small (like a spec dust) file.
8. A device token is a special non zero value the operating system gives your program and is associated with the file that you requested to be opened.
9. Programmers should not worry about closing a file.
10. Where you define an item, that is global or local scope, is rarely important.

Answers:

1. false
2. true
3. false
4. false
5. true
6. false
7. false
8. true
9. false
10. false

Short Answer

1. Describe the normal operations allowed with the string data type.
2. Describe why unary positive is worthless.
3. Describe how unary negative works.

Activities

Complete the following activities using pseudocode, a flowcharting tool, or your selected programming language. Use separate functions for input, each type of processing, and output. Avoid global variables by passing parameters and returning results. Create test data to validate the accuracy of each program. Add comments at the top of the program and include references to any resources used.

String Activities

1. Create a program that asks the user for a line of text containing a first name and last name, such as `Firstname Lastname`. Use string functions/methods to parse the line and print out the name in the form last name, first initial, such as `Lastname, F.` Include a trailing period after the first initial. Handle invalid input errors, such as extra spaces or missing name parts.
2. Create a program that asks the user for a line of text. Use string functions/methods to delete leading and trailing spaces, and then print the line of text backwards. For example:

```
the cat in the hat
tah eht ni tac eht
```

3. Create a program that asks the user for a line of comma-separated-values. It could be a sequence of test scores, names, or any other values. Use string functions/methods to parse the line and print out each item on a separate line. Remove commas and any leading or trailing spaces from each item when printed.
4. Create a program that asks the user for a line of text. Then ask the user for the number of characters to print in each line, the number of lines to be printed, and a scroll direction, right or left. Using the given line of text, duplicate the text as needed to fill the given number of characters per line. Then print the requested number of lines, shifting the entire line's content by one character, left or right, each time the line is printed. The first or last character will be shifted / appended to the other end of the string. For example:

```
Repeat this. Repeat this.
epeat this. Repeat this. R
peat this. Repeat this. Re
```

File Activities

1. Using a text editor or IDE, create a text file of names and grade scores to use for testing based on the following format:

```
Larry Fine: 80
Curly Howard: 70
Moe Howard: 90
```

Using the file above, create a program that displays high, low, and average scores based on input from the file. Verify that the file exists and then use string functions/methods to parse the file content and display the high, low, and average score. Note that the program must work for any given number of scores in the file. Do not assume there will only be three scores.

2. Using the file above, create a program that displays high, low, and average scores based on input from the file. Verify that the file exists and then use string functions/methods to parse the file content and add each score to an array. Display the array contents and then calculate and display the high, low, and average score. Note that the program must work for any given number of scores in the file. Do not assume there will only be three scores.
3. Using the file above, create a program that displays high, low, and average scores based on input from the file. Verify that the file exists and then use string functions/methods to parse the file content and add each score to an array. Display the array contents and then calculate and display the high, low, and average score. Include error handling in case the file is formatted incorrectly. Note that the program must work for any given number of scores in the file. Do not assume there will only be three scores.
4. Create a program that asks the user for the name of a text/HTML file that contains HTML tags, such as:

```
<strong>This is a bold paragraph.</strong></p>
```

Verify that the file exists and then use string methods to search for and remove all HTML tags from the text, saving each removed tag in an array. Display the untagged text and then display the array of removed tags. For example:

```
This is a bold paragraph.

<strong>
</strong>
</p>
```

5. Using a text editor or IDE, create a text file of names and addresses to use for testing based on the following format:

```
Firstname Lastname
123 Any Street
City, State/Province/Region PostalCode
```

Include a blank line between addresses, and include at least three addresses in the file. Create a program that verifies that the file exists, and then processes the file and displays each address as a single line of comma-separated values in the form:

```
Lastname, Firstname, Address, City, State/Province/Region, PostalCode
```

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

This page titled 27.8.6: Practice- Strings and Files is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9: OBJECT-ORIENTED PROGRAMMING

27.9.1: Encapsulation
27.9.2: Inheritance and Polymorphism
27.9.3: C++ Examples
27.9.4: C# Examples
27.9.5: Java Examples
27.9.6: JavaScript Examples
27.9.7: Swift Examples
27.9.8: Practice

This page titled 27.9: Object-Oriented Programming is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

- 8: Object-Oriented Programming is licensed CC BY-SA 4.0.

27.9.1: Encapsulation

Overview

Encapsulation is one of the fundamentals of OOP (object-oriented programming). It refers to the bundling of data with the methods that operate on that data. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them. Publicly accessible methods are generally provided in the class (so-called getters and setters) to access the values, and other client classes call these methods to retrieve and modify the values within the object.[1]

Discussion

The most important principle of object orientation is *encapsulation*: the idea that data inside the object should only be accessed through a public *interface* – that is, the object's methods.

If we want to use the data stored in an object to perform an action or calculate a derived value, we define a method associated with the object which does this. Then whenever we want to perform this action we call the method on the object. We consider it bad practice to retrieve the information from inside the object and write separate code to perform the action outside of the object.

Encapsulation is a good idea for several reasons:

- the functionality is defined *in one place* and not in multiple places.
- it is defined in a logical place – the place where the data is kept.
- data inside our object is not modified unexpectedly by external code in a completely different part of our program.
- when we use a method, we only need to know what result the method will produce – we don't need to know details about the object's internals in order to use it. We could switch to using another object which is completely different on the inside, and not have to change any code because both objects have the same interface.

We can say that the object "knows how" to do things with its own data, and it's a bad idea for us to access its internals and do things with the data ourselves. If an object doesn't have an interface method which does what we want to do, we should add a new method or update an existing one.

Some languages have features which allow us to enforce encapsulation strictly. In Java or C++, we can define access permissions on object attributes, and make it illegal for them to be accessed from outside the object's methods. In Java it is also considered good practice to write setters and getters for all attributes, even if the getter simply retrieves the attribute and the setter just assigns it the value of the parameter which you pass in.

In Python, encapsulation is not enforced by the language, but there is a convention that we can use to indicate that a property is intended to be private and is not part of the object's public interface: we begin its name with an underscore. Python also supports the use of a property decorator to replace a simple attribute with a method without changing the object's interface.

Key Terms

abstraction
: A technique for arranging complexity of computer systems so that functionality may be separated from specific implementation details.[2][3]

accessor
: A method used to return the value of a private member variable, also known as a getter method.[4]

encapsulation
: A language mechanism for restricting direct access to some of an object's components.[5]

information hiding
: The principle of segregation of the design decisions in a computer program from other parts of the program. See encapsulation.[6]

mutator
: A method used to control changes to a private member variable, also known as a setter method.[7]

private
: An access modifier that restricts visibility of a property or method to the class in which it is defined.[8]

public
: An access modifier that opens visibility of a property or method to all other classes.[9]

References

- Read the Docs: Object-Oriented Programming in Python
- Wikiversity: Computer Programming

---

1. Wikipedia: Encapsulation (computer programming) ↩
2. Wikipedia: Object-oriented programming ↩
3. Wikipedia: Abstraction (computer science) ↩
4. Wikipedia: Mutator method ↩
5. Wikipedia: Encapsulation (computer programming) ↩
6. Wikipedia: Information hiding ↩
7. Wikipedia: Mutator method ↩
8. Wikipedia: Access modifiers ↩
9. Wikipedia: Access modifiers ↩

This page titled 27.9.1: Encapsulation is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9.2: Inheritance and Polymorphism

Overview

In object-oriented programming, **inheritance** is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation. In most class-based object-oriented languages, an object created through inheritance (a “child object”) acquires all the properties and behaviors of the parent object (except: constructors, destructor, overloaded operators and friend functions of the base class). Inheritance allows programmers to create classes that are built upon existing classes, to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces.[1]

Discussion

Inheritance is a way of arranging objects in a hierarchy from the most general to the most specific. An object which *inherits* from another object is considered to be a *subtype* of that object. An example might include Instructor and Student, each of which inherit from Person. When we can describe the relationship between two objects using the phrase *is-a*, that relationship is inheritance.

We also often say that a class is a *subclass* or *child class* of a class from which it inherits, or that the other class is its *superclass* or *parent class*. We can refer to the most generic class at the base of a hierarchy as a *base class*.

Inheritance can help us to represent objects which have some differences and some similarities in the way they work. We can put all the functionality that the objects have in common in a base class, and then define one or more subclasses with their own custom functionality.

Inheritance is also a way of reusing existing code easily. If we already have a class which does *almost* what we want, we can create a subclass in which we partially override some of its behavior, or perhaps add some new functionality.

In some statically typed languages inheritance is very popular because it allows the programmer to work around some of the restrictions of static typing. If an instructor and a student are both a kind of person, we can write a function which accepts a parameter of type Person and have it work on both instructor and student objects because they both inherit from Person. This is known as *polymorphism*.

Key Terms

inheritance
: An object or class being based on another object or class, using the same implementation or specifying a new implementation to maintain the same behavior.[2]

polymorphism
: The provision of a single interface to entities of different types.[3]

References

- Read the Docs: Object-Oriented Programming in Python
- Wikiversity: Computer Programming

---

1. Wikipedia: Inheritance (object-oriented programming) ↩
2. Wikipedia: Inheritance (object-oriented programming) ↩
3. Wikipedia: Polymorphism (computer science) ↩

This page titled 27.9.2: Inheritance and Polymorphism is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9.3: C++ Examples

Objects

```
// This class converts temperature between Celsius and Fahrenheit.
// It may be used by assigning a value to either Celsius or Fahrenheit
// and then retrieving the other value, or by calling the ToCelsius or
// ToFahrenheit methods directly.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C%2B%2B_Programming

#include <iostream>

using namespace std;

class Temperature {
    public:
        double getCelsius(void);
        void setCelsius(double value);
        double getFahrenheit(void);
        void setFahrenheit(double value);
        double toCelsius(double fahrenheit);
        double toFahrenheit(double celsius);

    private:
        double celsius;
        double fahrenheit;
};

double Temperature::getCelsius(void) {
    return celsius;
}

void Temperature::setCelsius(double value) {
    celsius = value;
    fahrenheit = toFahrenheit(celsius);
}

double Temperature::getFahrenheit(void) {
    return fahrenheit;
}

void Temperature::setFahrenheit(double value) {
    fahrenheit = value;
    celsius = toCelsius(fahrenheit);
}

double Temperature::toCelsius(double fahrenheit) {
    return (fahrenheit - 32) * 5 / 9;
}

double Temperature::toFahrenheit(double celsius) {
```

```
    return celsius * 9 / 5 + 32;
}

// This program creates instances of the Temperature class to convert Celsius
// and Fahrenheit temperatures.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C%2B%2B_Programming

int main() {
    Temperature temp1;
    temp1.setCelsius(100.0);
    cout << "temp1.celsius = " << temp1.getCelsius() << endl;
    cout << "temp1.fahrenheit = " << temp1.getFahrenheit() << endl;
    cout << endl;

    Temperature temp2;
    temp2.setFahrenheit(100.0);
    cout << "temp2.fahrenheit = " << temp2.getFahrenheit() << endl;
    cout << "temp2.celsius = " << temp2.getCelsius() << endl;
}
```

Output

```
temp1.celsius = 100
temp1.fahrenheit = 212

temp2.fahrenheit = 100
temp2.celsius = 37.7778
```

References

- Wikiversity: Computer Programming

This page titled 27.9.3: C++ Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9.4: C# Examples

Objects

```
// This program creates instances of the Temperature class to convert Celsius
// and Fahrenheit temperatures.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/C_Sharp_Programming

using System;

public class Objects
{
    public static void Main(String[] args)
    {
        Temperature temp1 = new Temperature(celsius: 0);
        Console.WriteLine("temp1.Celsius = " + temp1.Celsius.ToString());
        Console.WriteLine("temp1.Fahrenheit = " + temp1.Fahrenheit.ToString());
        Console.WriteLine("");

        temp1.Celsius = 100;
        Console.WriteLine("temp1.Celsius = " + temp1.Celsius.ToString());
        Console.WriteLine("temp1.Fahrenheit = " + temp1.Fahrenheit.ToString());
        Console.WriteLine("");

        Temperature temp2 = new Temperature(fahrenheit: 0);
        Console.WriteLine("temp2.Fahrenheit = " + temp2.Fahrenheit.ToString());
        Console.WriteLine("temp2.Celsius = " + temp2.Celsius.ToString());
        Console.WriteLine("");

        temp2.Fahrenheit = 100;
        Console.WriteLine("temp2.Fahrenheit = " + temp2.Fahrenheit.ToString());
        Console.WriteLine("temp2.Celsius = " + temp2.Celsius.ToString());
    }
}

// This class converts temperature between Celsius and Fahrenheit.
// It may be used by assigning a value to either Celsius or Fahrenheit
// and then retrieving the other value, or by calling the ToCelsius or
// ToFahrenheit methods directly.

public class Temperature
{
    double _celsius;
    double _fahrenheit;

    public double Celsius
    {
        get
        {
            return _celsius;
        }
```

```
51
52          set
53          {
54              _celsius = value;
55              _fahrenheit = ToFahrenheit(value);
56          }
57      }
58
59      public double Fahrenheit
60      {
61          get
62          {
63              return _fahrenheit;
64          }
65
66          set
67          {
68              _fahrenheit = value;
69              _celsius = ToCelsius(value);
70          }
71      }
72
73      public Temperature(double? celsius = null, double? fahrenheit = null)
74      {
75          if (celsius.HasValue)
76          {
77              this.Celsius = Convert.ToDouble(celsius);
78          }
79
80          if (fahrenheit.HasValue)
81          {
82              this.Fahrenheit = Convert.ToDouble(fahrenheit);
83          }
84      }
85
86      public double ToCelsius(double fahrenheit)
87      {
88          return (fahrenheit - 32) * 5 / 9;
89      }
90
91      public double ToFahrenheit(double celsius)
92      {
93          return celsius * 9 / 5 + 32;
94      }
95  }
```

Output

```
01  temp1.Celsius = 0
02  temp1.Fahrenheit = 32
03
04  temp1.Celsius = 100
05  temp1.Fahrenheit = 212
06
07  temp2.Fahrenheit = 0
08  temp2.Celsius = -17.7777777777778
```

```
09
10 temp2.Fahrenheit = 100
11 temp2.Celsius = 37.7777777777778
```

References

- Wikiversity: Computer Programming

This page titled 27.9.4: C# Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9.5: Java Examples

Objects

```java
// This program creates instances of the Temperature class to convert Celsius
// and Fahrenheit temperatures.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/Java_Programming

import java.util.*;

class Main {
    public static void main(String[] args) {
        Temperature temp1 = new Temperature();
        temp1.setCelsius(100.0);
        System.out.println("temp1.celsius = " + temp1.getCelsius().toString());
        System.out.println("temp1.fahrenheit = " + temp1.getFahrenheit().toString());
        System.out.println("");

        Temperature temp2 = new Temperature();
        temp2.setFahrenheit(100.0);
        System.out.println("temp2.fahrenheit = " + temp2.getFahrenheit().toString());
        System.out.println("temp2.celsius = " + temp2.getCelsius().toString());
    }
}

// This class converts temperature between Celsius and Fahrenheit.
// It may be used by assigning a value to either Celsius or Fahrenheit
// and then retrieving the other value, or by calling the ToCelsius or
// ToFahrenheit methods directly.

class Temperature {
    Double celsius;
    Double fahrenheit;

    public Double getCelsius() {
        return celsius;
    }

    public void setCelsius(Double value) {
        celsius = value;
        fahrenheit = toFahrenheit(celsius);
    }

    public Double getFahrenheit() {
        return fahrenheit;
    }

    public void setFahrenheit(Double value) {
        fahrenheit = value;
        celsius = toCelsius(fahrenheit);
    }
```

```
51
52      public Double toCelsius(Double fahrenheit) {
53          return (fahrenheit - 32) * 5 / 9;
54      }
55
56      public Double toFahrenheit(Double celsius) {
57          return celsius * 9 / 5 + 32;
58      }
59  }
```

Output

```
01  temp1.celsius = 100.0
02  temp1.fahrenheit = 212.0
03
04  temp2.fahrenheit = 100.0
05  temp2.celsius = 37.77777777777778
```

References

- Wikiversity: Computer Programming

This page titled 27.9.5: Java Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9.6: JavaScript Examples

Objects

```
// This class converts temperature between Celsius and Fahrenheit.
// It may be used by assigning a value to either Celsius or Fahrenheit
// and then retrieving the other value, or by calling the ToCelsius or
// ToFahrenheit methods directly.

class Temperature {
    constructor() {
        this._celsius = 0;
        this._fahrenheit = 32;
    }

    get celsius() {
        return this._celsius;
    }

    set celsius(value) {
        this._celsius = value;
        this._fahrenheit = this.toFahrenheit(value);
    }

    get fahrenheit() {
        return this._fahrenheit;
    }

    set fahrenheit(value) {
        this._fahrenheit = value;
        this._celsius = this.toCelsius(value);
    }

    toCelsius(fahrenheit) {
        return (fahrenheit - 32) * 5 / 9
    }

    toFahrenheit(celsius) {
        return celsius * 9 / 5 + 32
    }
}

// This program creates instances of the Temperature class to convert Celsius
// and Fahrenheit temperatures.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://en.wikibooks.org/wiki/JavaScript

main()

function main() {
    var temp1 =  new Temperature();
    temp1.celsius = 0
```

```
    output("temp1.celsius = " + temp1.celsius);
    output("temp1.fahrenheit = " + temp1.fahrenheit);
    output("");

    temp1.celsius = 100;
    output("temp1.celsius = " + temp1.celsius);
    output("temp1.fahrenheit = " + temp1.fahrenheit);
    output("");

    var temp2 = new Temperature();
    temp2.fahrenheit = 0
    output("temp2.fahrenheit = " + temp2.fahrenheit);
    output("temp2.celsius = " + temp2.celsius);
    output("");

    temp2.fahrenheit = 100;
    output("temp2.fahrenheit = " + temp2.fahrenheit);
    output("temp2.celsius = " + temp2.celsius);
}

// Checks the JavaScript environment and writes to the console,
// the current document, or standard output as appropriate.
// Reference: http://progopedia.com/example/hello-world/114/
function output(text) {
    if (typeof console === 'object') {
        console.log(text);
    }
    else if (typeof document === 'object') {
        document.write(text);
    }
    else {
        print(text);
    }
}
```

Output

```
temp1.celsius = 0
temp1.fahrenheit = 32

temp1.celsius = 100
temp1.fahrenheit = 212

temp2.fahrenheit = 0
temp2.celsius = -17.77777777777778

temp2.fahrenheit = 100
temp2.celsius = 37.77777777777778
```

References

- Wikiversity: Computer Programming

This page titled 27.9.6: JavaScript Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9.7: Swift Examples

Objects

```
// This class converts temperature between Celsius and Fahrenheit.
// It may be used by assigning a value to either Celsius or Fahrenheit
// and then retrieving the other value, or by calling the ToCelsius or
// ToFahrenheit methods directly.

class Temperature {
    var _celsius:Double = 0
    var _fahrenheit:Double = 32

    init(celsius:Double?=nil, fahrenheit:Double?=nil) {
        if celsius != nil {
            self.celsius = celsius!
        }

        if fahrenheit != nil {
            self.fahrenheit = fahrenheit!
        }
    }

    var celsius: Double {
        get {
            return self._celsius
        }
        set {
            self._celsius = newValue
            self._fahrenheit = toFahrenheit(celsius:self._celsius)
        }
    }

    var fahrenheit: Double {
        get {
            return self._fahrenheit
        }
        set {
            self._fahrenheit = newValue
            self._celsius = toCelsius(fahrenheit:self._fahrenheit)
        }
    }

    func getCelsius() -> Double {
        return self.celsius
    }

    func setCelsius(celsius:Double) {
        self.celsius = celsius
        self.fahrenheit = toFahrenheit(celsius:celsius)
    }

    func getFahrenheit() -> Double {
        return self.fahrenheit
```

```
    }

    func setFahrenheit(fahrenheit:Double) {
        self.fahrenheit = fahrenheit
        self.celsius = toCelsius(fahrenheit:fahrenheit)
    }

    func toCelsius(fahrenheit:Double) -> Double {
        return (fahrenheit - 32) * 5 / 9
    }

    func toFahrenheit(celsius:Double) -> Double {
        return celsius * 9 / 5 + 32
    }
}

// This program creates instances of the Temperature class to convert Celsius
// and Fahrenheit temperatures.
//
// References:
//     https://www.mathsisfun.com/temperature-conversion.html
//     https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

func main() {
    let temp1 =  Temperature(celsius:0);
    print("temp1.celsius = " + String(temp1.celsius));
    print("temp1.fahrenheit = " + String(temp1.fahrenheit));
    print("");

    temp1.celsius = 100;
    print("temp1.celsius = " + String(temp1.celsius));
    print("temp1.fahrenheit = " + String(temp1.fahrenheit));
    print("");

    let temp2 = Temperature(fahrenheit:0);
    print("temp2.fahrenheit = " + String(temp2.fahrenheit));
    print("temp2.celsius = " + String(temp2.celsius));
    print("");

    temp2.fahrenheit = 100;
    print("temp2.fahrenheit = " + String(temp2.fahrenheit));
    print("temp2.celsius = " + String(temp2.celsius));
}

main()
```

Output

```
temp1.celsius = 0.0
temp1.fahrenheit = 32.0

temp1.celsius = 100.0
temp1.fahrenheit = 212.0

temp2.fahrenheit = 0.0
```

```
temp2.celsius = -17.7777777777778

temp2.fahrenheit = 100.0
temp2.celsius = 37.7777777777778
```

References

- Wikiversity: Computer Programming

This page titled 27.9.7: Swift Examples is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

27.9.8: Practice

Review Questions

Answer the following statements as either true or false:

1. Procedural programming and object-oriented programming cannot be done with the same compiler/IDE.
2. Object-oriented programming encapsulates data and functions.

Answers:

1. false
2. true

Short Answer

1. Describe the fundamental differences between procedural (modular structured) programming and object-oriented programming.

Activities

Complete the following activities using your selected programming language. Use separate functions for input, each type of processing, and output. Avoid global variables by passing parameters and returning results. Create test data to validate the accuracy of each program. Add comments at the top of the program and include references to any resources used.

1. Review MathsIsFun: Area of Plane Shapes. Create a program that asks the user what shape they would like to calculate the area for. Use if/else conditional statements to determine their selection and then gather the appropriate input and calculate and display the area of the shape. Perform all area calculations using a ShapeArea class that has separate methods to calculate and return the area for different shapes. Include data validation in the class and error handling in the main program.
2. Create a program that asks the user how old they are in years. Then ask the user if they would like to know how old they are in months, days, hours, or seconds. Use if/else conditional statements to display their approximate age in the selected timeframe. Perform all calculations using an AgeConverter class that accepts the age in years during initialization and has separate properties and methods to calculate and return the age in months, days, hours, and seconds. Include data validation in the class and error handling in the main program.
3. Review Wikipedia: Zeller's congruence. Create a program that asks the user for their birthday (year, month, and day) and then calculate and display the day of the week on which they were born. Use if/else conditional statements to convert the numeric day of the week to the correct string representation (Monday, Tuesday, Wednesday, etc.). Perform all calculations using a DayOfWeek class that accepts the year, month, and day during initialization and has separate properties and methods to calculate and return the day of week as a number, as an abbreviated string (Mon, Tue, etc.), and as a full string (Monday, Tuesday, etc.). Include data validation in the class and error handling in the main program.

References

- cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
- Wikiversity: Computer Programming

This page titled 27.9.8: Practice is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by LibreTexts.

28: ADDITIONAL PYTHON 4 EVERYONE

28.1: A trip to Jupyter

28.1.1: Jupyter Notebooks

28.1.2: Code and Output

28.2: Variables, Expressions, and Statements

28.2.1: Variable names and Keywords

28.2.2: Operators and Operands

28.2.3: Expressions

28.2.4: Order of Operations

28.2.5: Choosing Mnemonic Variable Names

28.2.6: Debugging

28.2.E: Variables, Expressions, and Statements (Exercises)

28.2.G: Glossary

28.3: Conditional Execution

28.3.1: Short-Circuit Evaluation of Logical Expressions

28.3.2: Debugging

28.3.E: Conditional Execution (Exercises)

28.3.G: Conditional Execution (Glossary)

28.4: Functions

28.4.1: Function Calls

28.4.2: Built-in Functions

28.4.3: Type Conversion Functions

28.4.4: Parameters and Arguments

28.4.5: Debugging

28.4.E: Functions (Exercises)

28.4.G: Functions (Glossary)

28.5: Iterations

28.5.1: "Infinite loops" and break

28.5.2: Finishing iterations with continue

28.5.3: Debugging

28.5.E: Iterations (Exercises)

28.5.G: Iterations (Glossary)

28.6: Strings

28.6.1: Debugging

28.6.E: Strings (Exercises)

28.6.G: Strings (Glossary)

28.7: Files

28.7.1: Persistence
28.7.2: Debugging
28.7.E: Files (Exercises)
28.7.G: Files (Glossary)

28.8: Lists

28.8.1: Lists and Functions
28.8.2: Lists and Strings
28.8.3: Parsing lines
28.8.4: Objects and Values
28.8.5: Aliasing
28.8.6: List arguments
28.8.7: Debugging
28.8.E: Lists (Exercises)
28.8.G: Lists (Glossary)

28.9: Object-Oriented Programming

28.9.1: Starting with Programs
28.9.2: Subdividing a Problem - Encapsulation
28.9.3: Classes as Types
28.9.4: Many Instances
28.9.5: Object Lifecycle
28.9.6: Inheritance
28.9.G: Object-Oriented Programming (Glossary)
28.9.S: Object-Oriented Programming (Summary)

This page titled 28: Additional Python 4 Everyone is shared under a license and was authored, remixed, and/or curated by LibreTexts.

28.1: A TRIP TO JUPYTER

Python is a ridiculously popular **language** for programming and data science (currently the third most widely used in the world[1]) which is one of many reasons we're using it for this course. The language itself is different from the **programming environment** used to write code in it, just as "English" is different from "Microsoft Word" and "Google Docs." A programming environment is just a fancy name for a tool or application used to write programs. At a minimum, it must include a way to edit (write and revise) code, and a way to execute (run) it.

There are many different programming environments data scientists use to write Python code, just as there are many different word processing apps people use to write English. The choice largely comes down to personal preference. Some use full-blown **IDE**s ("integrated development environments") like Spyder or Atom; some use text-based tools like Notepad++ or **vim**. In this class, we're going to use the friendly and minimalistic "**Jupyter Notebooks**" environment since it's appropriate for an intro experience.

28.1.1: Jupyter Notebooks

28.1.2: Code and Output

[1]See https://www.tiobe.com/tiobe-index/.

This page titled 28.1: A trip to Jupyter is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Stephen Davies (allthemath.org) via source content that was edited to the style and standards of the LibreTexts platform.

- 2: A trip to Jupyter by Stephen Davies is licensed CC BY-SA 4.0. Original source: https://eng.libretexts.org/@api/deki/files/58148/crystal1.pdf.

28.1.1: Jupyter Notebooks

The concept of a Jupyter Notebook is simple: it's a Web page with editable "cells." Each cell is a little text window you can type in.

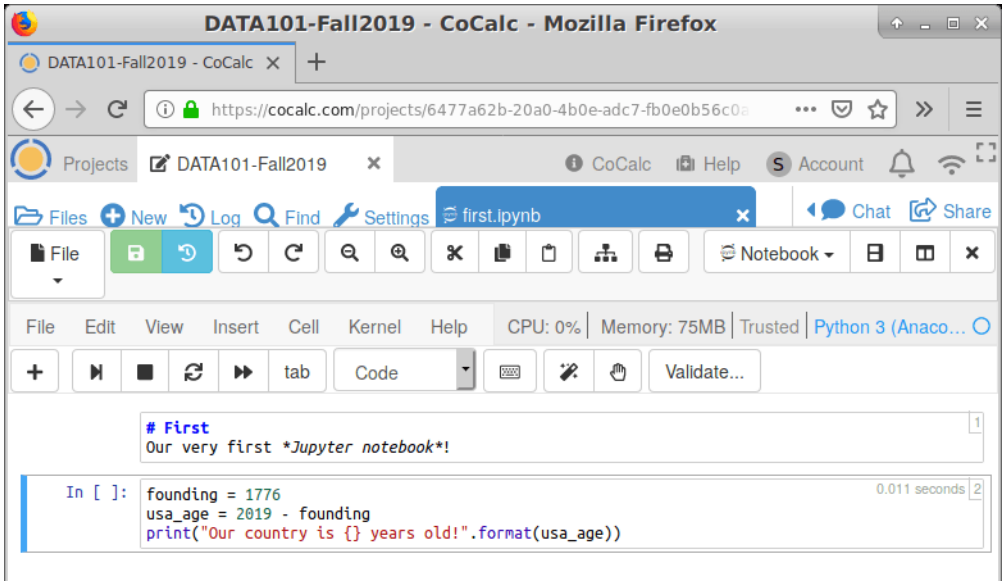
There are three kinds of cells in Jupyter Notebooks:

- **Raw**
  - "Raw" is dumb. Never use it.
- **Markdown**
  - "Markdown" cells are for English text, not Python code. They're mostly used to describe and annotate what you're doing in the code cells, like a running commentary. You can type plain-ol' text in a Markdown cell, plus various cutesy formatting adornments like boldface (putting double-splats (**) around a word or phrase), italics (single splats), outline headings (prefacing a line with one or more hashtags (like # or ###), and so forth.[2] When you type in a Markdown cell, you see the raw text and formatting; to actually get Jupyter to render your cells and make them pretty, you choose "Run All" from the "Cell" menu.
- **Code**
  - The most important cells are "Code" cells which contain (duh) code. When executed (again, by choosing "Run All" from the "Cell" menu) they actually carry out the Python instructions you have typed in that cell, and display any results.

By the way, a common snafu is to somehow accidentally click in a way that changes the type of a cell from "Code" to one of the other types. If you do this, the Python code in that cell won't execute until you change the type back to "Code" (more on this below).

Figure 2.1.1 shows a Jupyter Notebook hosted by the **CoCalc** cloud computing platform, which we'll use this semester. It has two cells, one Markdown and one Code. Note carefully the **cell-type dropdown** which is kind of hidden in the middle of the page: it currently reads "Code" because the second cell is the one that's highlighted. (If we clicked to highlight and edit the top cell, that dropdown would change to "Markdown.")

The top figure shows the two cells before the user has done a "Run All" from the "Cell" menu: all the Markdown is unrendered (see the literal splats and hashtags) and the code is just sitting there. After "Run All," the picture changes: you see the formatted message in the top cell, and the **output** of the Python code snippet after it runs. (The latter is easy to miss; stare at that bottom picture and find the "**Our country is 245 years old!**" message. That's the "output.") We haven't yet covered what that Python code means (that's the main subject of this book) but you can probably guess some of what it's doing.



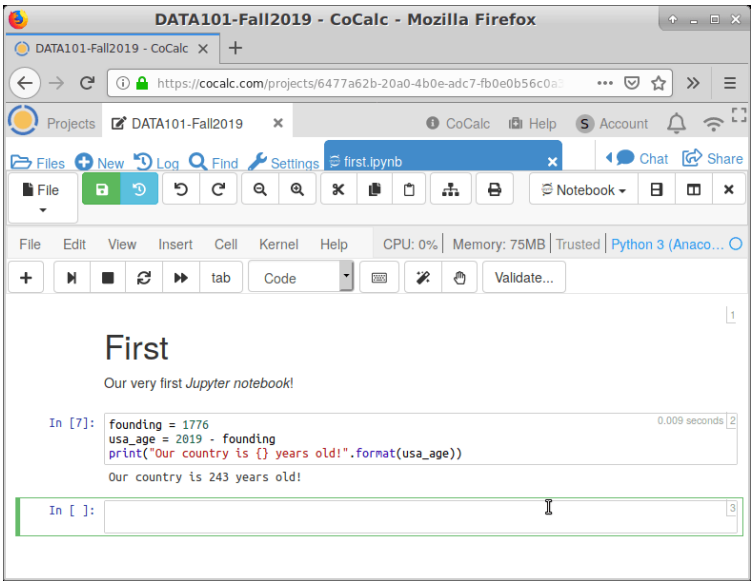


Figure 28.1.1.1: A Jupyter Notebook with one Markdown cell and one Code cell. In the top image, the two cells have been edited but not yet "run" – hence the Markdown formatting is unrendered and the code has not been executed. The bottom pane shows both cells after the use has chosen "Run All" from the "Cell" menu.

[2]For a complete list of formatting options, see https://github.com/ adam-p/markdown-here/wiki/Markdown-Cheatsheet.

This page titled 28.1.1: Jupyter Notebooks is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Stephen Davies (allthemath.org) via source content that was edited to the style and standards of the LibreTexts platform.

- 2.1: Jupyter Notebooks by Stephen Davies is licensed CC BY-SA 4.0. Original source: https://eng.libretexts.org/@api/deki/files/58148/crystal1.pdf.

28.1.2: Code and Output

Incredibly, that's about it. Everything else in this book is going to concern what to type in those Code cells and how to interpret its output.

From now on, whenever I give example Python code in this book, I'll write it like this

```
founding = 1776

usa_age = 2021 - founding

print("Our country is {} years old!".format(usa_age))
```

That means "this stuff goes in a Code cell of a Notebook." When I write the corresponding output (i.e., what gets printed on the page immediately below the code cell when "Run All" is chosen from the "Cell" menu), I'll write it like this:

```
| Our country is 245 years old!
```

That vertical bar means "this stuff is the printed result of executing the code cell." Easy enough. Onward!

This page titled 28.1.2: Code and Output is shared under a CC BY-SA 4.0 license and was authored, remixed, and/or curated by Stephen Davies (allthemath.org) via source content that was edited to the style and standards of the LibreTexts platform.

- 2.2: Code and Output by Stephen Davies is licensed CC BY-SA 4.0. Original source: https://eng.libretexts.org/@api/deki/files/58148/crystal1.pdf.

28.2: VARIABLES, EXPRESSIONS, AND STATEMENTS

28.2.1: Variable names and Keywords

28.2.2: Operators and Operands

28.2.3: Expressions

28.2.4: Order of Operations

28.2.5: Choosing Mnemonic Variable Names

28.2.6: Debugging

28.2.E: Variables, Expressions, and Statements (Exercises)

28.2.G: Glossary

This page titled 28.2: Variables, Expressions, and Statements is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2: Variables, Expressions, and Statements by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.2.1: Variable names and Keywords

Programmers generally choose names for their variables that are meaningful and document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they cannot start with a number. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter (you'll see why later).

The underscore character (`_`) can appear in a name. It is often used in names with multiple words, such as `my_name` or `airspeed_of_unladen_swallow`. Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` is illegal because it begins with a number. `more@` is illegal because it contains an illegal character, `@`. But what's wrong with `class`?

It turns out that `class` is one of Python's *keywords*. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 33 keywords:

```
and       del       from      None      True
as        elif      global    nonlocal  try
assert    else      if        not       while
break     except    import    or        with
class     False     in        pass      yield
continue  finally   is        raise
def       for       lambda    return
```

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

This page titled 28.2.1: Variable names and Keywords is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.3: Variable names and Keywords by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.2.2: Operators and Operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called *operands*.

The operators `+`, `-`, `*`, `/`, and `**` perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

There has been a change in the division operator between Python 2.x and Python 3.x. In Python 3.x, the result of this division is a floating point result:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

The division operator in Python 2.0 would divide two integers and truncate the result to an integer:

```
>>> minute = 59
>>> minute/60
0
```

To obtain the same answer in Python 3.0 use floored (// integer) division.

```
>>> minute = 59
>>> minute//60
0
```

In Python 3.0 integer division functions much more as you would expect if you entered the expression on a calculator.

This page titled 28.2.2: Operators and Operands is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.5: Operators and Operands by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.2.3: Expressions

An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

```
17
x
x + 17
```

If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:

```
>>> 1 + 1
2
```

But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Exercise 1: Type the following statements in the Python interpreter to see what they do:

```
5
x = 5
x + 1
```

This page titled 28.2.3: Expressions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.6: Expressions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.2.4: Order of Operations

When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*. For mathematical operators, Python follows mathematical convention. The acronym *PEMDAS* is a useful way to remember the rules:

- *P*arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * (3-1)` is 4, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.
- *E*xponentiation has the next highest precedence, so `2**1+1` is 3, not 4, and `3*1**3` is 3, not 27.
- *M*ultiplication and *D*ivision have the same precedence, which is higher than *A*ddition and Subtraction, which also have the same precedence. So `2*3-1` is 5, not 4, and `6+4/2` is 8.0, not 5.
- Operators with the same precedence are evaluated from left to right. So the expression `5-3-1` is 1, not 3, because the `5-3` happens first and then `1` is subtracted from `2`.

When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

This page titled 28.2.4: Order of Operations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.7: Order of Operations by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.2.5: Choosing Mnemonic Variable Names

As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables. In the beginning, this choice can be confusing both when you read a program and when you write your own programs. For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
01 a = 35.0
02 b = 12.50
03 c = a * b
04 print(c)
05
```

```
01 hours = 35.0
02 rate = 12.50
03 pay = hours * rate
04 print(pay)
05
```

```
01 x1q3z9ahd = 35.0
02 x1q3z9afd = 12.50
03 x1q3p9afd = x1q3z9ahd * x1q3z9afd
04 print(x1q3p9afd)
05
```

The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently. Humans will most quickly understand the *intent* of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.

We call these wisely chosen variable names "mnemonic variable names". The word *mnemonic*[2] means "memory aid". We choose mnemonic variable names to help us remember why we created the variable in the first place.

While this all sounds great, and it is a very good idea to use mnemonic variable names, mnemonic variable names can get in the way of a beginning programmer's ability to parse and understand code. This is because beginning programmers have not yet memorized the reserved words (there are only 33 of them) and sometimes variables with names that are too descriptive start to look like part of the language and not just well-chosen variable names.

Take a quick look at the following Python sample code which loops through some data. We will cover loops soon, but for now try to just puzzle through what this means:

```
01 for word in words:
02     print(word)
```

What is happening here? Which of the tokens (for, word, in, etc.) are reserved words and which are just variable names? Does Python understand at a fundamental level the notion of words? Beginning programmers have trouble separating what parts of the code *must* be the same as this example and what parts of the code are simply choices made by the programmer.

The following code is equivalent to the above code:

```
01 for slice in pizza:
```

```
    print(slice)
```

It is easier for the beginning programmer to look at this code and know which parts are reserved words defined by Python and which parts are simply variable names chosen by the programmer. It is pretty clear that Python has no fundamental understanding of pizza and slices and the fact that a pizza consists of a set of one or more slices.

But if our program is truly about reading data and looking for words in the data, `pizza` and `slice` are very un-mnemonic variable names. Choosing them as variable names distracts from the meaning of the program.

After a pretty short period of time, you will know the most common reserved words and you will start to see the reserved words jumping out at you:

```
for word in words:
    print word
```

The parts of the code that are defined by Python (`for`, `in`, `print`, and `:`) are in bold and the programmer-chosen variables (`word` and `words`) are not in bold. Many text editors are aware of Python syntax and will color reserved words differently to give you clues to keep your variables and reserved words separate. After a while you will begin to read Python and quickly determine what is a variable and what is a reserved word.

This page titled 28.2.5: Choosing Mnemonic Variable Names is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.12: Choosing Mnemonic Variable Names by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.2.6: Debugging

At this point, the syntax error you are most likely to make is an illegal variable name, like `class` and `yield`, which are keywords, or `odd~job` and `US$`, which contain illegal characters.

If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

```
>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

For syntax errors, the error messages don't help much. The most common messages are `SyntaxError: invalid syntax` and `SyntaxError: invalid token`, neither of which is very informative.

The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value. This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Variables names are case sensitive, so `LaTeX` is not the same as `latex`.

At this point, the most likely cause of a semantic error is the order of operations. For example, to evaluate 1/2π, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

But the division happens first, so you would get π/2, which is not the same thing! There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

This page titled 28.2.6: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.13: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.2.E: Variables, Expressions, and Statements (Exercises)

Exercise 2: Write a program that uses `input` to prompt a user for their name and then welcomes them.

```
Enter your name: Chuck
Hello Chuck
```

```
%%python3
#Use this space to write and run your code


```

Exercise 3: Write a program to prompt the user for hours and rate per hour to compute gross pay.

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

We won't worry about making sure our pay has exactly two digits after the decimal place for now. If you want, you can play with the built-in Python `round` function to properly round the resulting pay to two decimal places.

```
%%python3
#Use this space to write and run your code
#Save this code for a later exercise!

```

Exercise 4: Assume that we execute the following assignment statements:

```
width = 17
height = 12.0
```

For each of the following expressions, write the value of the expression and the type (of the value of the expression).

1. `width//2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`

```
%%python3
#Use this space to check your answers


```

Exercise 5: Write a program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature.

```
%%python3
#Use this space to write and run your code


```

1. In Python 2.0, this function was named `raw_input`.↩
2. See http://en.Wikipedia.org/wiki/Mnemonic for an extended description of the word "mnemonic".↩

This page titled 28.2.E: Variables, Expressions, and Statements (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 2.E: Variables, Expressions, and Statements (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

assignment
A statement that assigns a value to a variable.

concatenate
To join two operands end to end.

comment
Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

evaluate
To simplify an expression by performing the operations in order to yield a single value.

expression
A combination of variables, operators, and values that represents a single result value.

floating point
A type that represents numbers with fractional parts.

integer
A type that represents whole numbers.

keyword
A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

mnemonic
A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.

modulus operator
An operator, denoted with a percent sign (`%`), that works on integers and yields the remainder when one number is divided by another.

operand
One of the values on which an operator operates.

operator
A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

rules of precedence
The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

statement
A section of code that represents a command or action. So far, the statements we have seen are assignments and print expression statement.

string
A type that represents sequences of characters.

type
A category of values. The types we have seen so far are integers (type `int`), floating-point numbers (type `float`), and strings (type `str`).

value
One of the basic units of data, like a number or string, that a program manipulates.

variable
A name that refers to a value.

This page titled 28.2.G: Glossary is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

28.3: CONDITIONAL EXECUTION

28.3.1: Short-Circuit Evaluation of Logical Expressions
28.3.2: Debugging
28.3.E: Conditional Execution (Exercises)
28.3.G: Conditional Execution (Glossary)

This page titled 28.3: Conditional Execution is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3: Conditional Execution by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.3.1: Short-Circuit Evaluation of Logical Expressions

When Python is processing a logical expression such as `x >= 2 and (x/y) > 2`, it evaluates the expression from left to right. Because of the definition of `and`, if `x` is less than 2, the expression `x >= 2` is `False` and so the whole expression is `False` regardless of whether `(x/y) > 2` evaluates to `True` or `False`.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called *short-circuiting* the evaluation.

While this may seem like a fine point, the short-circuit behavior leads to a clever technique called the *guardian pattern*. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

The third calculation failed because Python was evaluating `(x/y)` and `y` was zero, which causes a runtime error. But the second example did *not* fail because the first part of the expression `x >= 2` evaluated to `False` so the `(x/y)` was not ever executed due to the *short-circuit* rule and there was no error.

We can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

In the first logical expression, `x >= 2` is `False` so the evaluation stops at the `and`. In the second logical expression, `x >= 2` is `True` but `y != 0` is `False` so we never reach `(x/y)`.

In the third logical expression, the `y != 0` is *after* the `(x/y)` calculation so the expression fails with an error.

In the second expression, we say that `y != 0` acts as a *guard* to insure that we only execute `(x/y)` if `y` is non-zero.

This page titled 28.3.1: Short-Circuit Evaluation of Logical Expressions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.8: Short-Circuit Evaluation of Logical Expressions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.3.2: Debugging

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

In general, error messages tell you where the problem was discovered, but that is often not where it was caused.

This page titled 28.3.2: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.9: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.3.E: Conditional Execution (Exercises)

Exercise 1: Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

```
%%python3
#Use this space to write and run your code

```

Exercise 2: Rewrite your pay program using `try` and `except` so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows two executions of the program:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input
```

```
Enter Hours: forty
Error, please enter numeric input
```

```
%%python3
#Use this space to write and run your code

```

Exercise 3: Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

```
Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
< 0.6    F
```

```
Enter score: 0.95
A
```

```
Enter score: perfect
Bad score
```

```
Enter score: 10.0
Bad score
```

```
Enter score: 0.75
C
```

```
Enter score: 0.5
F
```

```
%%python3
#Use this space to write and run your code
```

Run the program repeatedly as shown above to test the various different values for input.

---

1. We will learn about functions in Chapter 4 and loops in Chapter 5.↩

This page titled 28.3.E: Conditional Execution (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 3.E: Conditional Execution (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

body
The sequence of statements within a compound statement.

boolean expression
An expression whose value is either `True` or `False`.

branch
One of the alternative sequences of statements in a conditional statement.

chained conditional
A conditional statement with a series of alternative branches.

comparison operator
One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

conditional statement
A statement that controls the flow of execution depending on some condition.

condition
The boolean expression in a conditional statement that determines which branch is executed.

compound statement
A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

guardian pattern
Where we construct a logical expression with additional comparisons to take advantage of the short-circuit behavior.

logical operator
One of the operators that combines boolean expressions: `and`, `or`, and `not`.

nested conditional
A conditional statement that appears in one of the branches of another conditional statement.

traceback
A list of the functions that are executing, printed when an exception occurs.

short circuit
When Python is part-way through evaluating a logical expression and stops the evaluation because Python knows the final value for the expression without needing to evaluate the rest of the expression.

This page titled 28.3.G: Conditional Execution (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

28.4: FUNCTIONS

28.4.1: Function Calls
28.4.2: Built-in Functions
28.4.3: Type Conversion Functions
28.4.4: Parameters and Arguments
28.4.5: Debugging
28.4.E: Functions (Exercises)
28.4.G: Functions (Glossary)

This page titled 28.4: Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4: Functions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.4.1: Function Calls

In the context of programming, a *function* is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a *function call*:

```
>>> type(32)
<class 'int'>
```

The name of the function is `type`. The expression in parentheses is called the *argument* of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the `type` function, is the type of the argument.

It is common to say that a function "takes" an argument and "returns" a result. The result is called the *return value*.

This page titled 28.4.1: Function Calls is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.1: Function Calls by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.4.2: Built-in Functions

Python provides a number of important built-in functions that we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

The `max` and `min` functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

The `max` function tells us the "largest character" in the string (which turns out to be the letter "w") and the `min` function shows us the smallest character (which turns out to be a space).

Another very common built-in function is the `len` function which tells us how many items are in its argument. If the argument to `len` is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
>>>
```

These functions are not limited to looking at strings. They can operate on any set of values, as we will see in later chapters.

You should treat the names of built-in functions as reserved words (i.e., avoid using "max" as a variable name).

This page titled 28.4.2: Built-in Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.2: Built-in Functions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.4.3: Type Conversion Functions

Python also provides built-in functions that convert values from one type to another. The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, `str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

This page titled 28.4.3: Type Conversion Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.3: Type Conversion Functions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.4.4: Parameters and Arguments

Some of the built-in functions we have seen require arguments. For example, when you call `math.sin` you pass a number as an argument. Some functions take more than one argument: `math.pow` takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called *parameters*. Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

This function assigns the argument to a parameter named `bruce`. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions `Spam  '*4` and `math.cos(math.pi)` are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (`michael`) has nothing to do with the name of the parameter (`bruce`). It doesn't matter what the value was called back home (in the caller); here in `print_twice`, we call everybody `bruce`.

This page titled 28.4.4: Parameters and Arguments is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.9: Parameters and Arguments by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.4.5: Debugging

If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don't.

Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.

Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case, the program you are looking at in the text editor is not the same as the program you are running.

Debugging can take a long time if you keep running the same incorrect program over and over!

Make sure that the code you are looking at is the code you are running. If you're not sure, put something like `print("hello")` at the beginning of the program and run it again. If you don't see `hello`, you're not running the right program!

This page titled 28.4.5: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.12: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.4.E: Functions (Exercises)

Exercise 4: What is the purpose of the "def" keyword in Python?

a) It is slang that means "the following code is really cool"
b) It indicates the start of a function
c) It indicates that the following indented section of code is to be stored for later
d) b and c are both true
e) None of the above

Exercise 5: What will the following Python program print out?

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Exercise 6: Rewrite your pay computation with time-and-a-half for overtime and create a function called `computepay` which takes two parameters (`hours` and `rate`).

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

Exercise 7: Rewrite the grade program from the previous chapter using a function called `computegrade` that takes a score as its parameter and returns a grade as a string.

```
Score   Grade
> 0.9     A
> 0.8     B
> 0.7     C
> 0.6     D
<= 0.6    F
```

```
Program Execution:
```

```
Enter score: 0.95
A
```

```
Enter score: perfect
```

```
Bad score
```

```
Enter score: 10.0
Bad score
```

```
Enter score: 0.75
C
```

```
Enter score: 0.5
F
```

Run the program repeatedly to test the various different values for input.

This page titled 28.4.E: Functions (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 4.E: Functions (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

algorithm
A general process for solving a category of problems.

argument
A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

body
The sequence of statements inside a function definition.

composition
Using an expression as part of a larger expression, or a statement as part of a larger statement.

deterministic
Pertaining to a program that does the same thing each time it runs, given the same inputs.

dot notation
The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

flow of execution
The order in which statements are executed during a program run.

fruitful function
A function that returns a value.

function
A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call
A statement that executes a function. It consists of the function name followed by an argument list.

function definition
A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object
A value created by a function definition. The name of the function is a variable that refers to a function object.

header
The first line of a function definition.

import statement
A statement that reads a module file and creates a module object.

module object
A value created by an `import` statement that provides access to the data and code defined in a module.

parameter
A name used inside a function to refer to the value passed as an argument.

pseudorandom
Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.

return value
The result of a function. If a function call is used as an expression, the return value is the value of the expression.

void function
A function that does not return a value.

This page titled 28.4.G: Functions (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

28.5: ITERATIONS

28.5.1: "Infinite loops" and break
28.5.2: Finishing iterations with continue
28.5.3: Debugging
28.5.E: Iterations (Exercises)
28.5.G: Iterations (Glossary)

This page titled 28.5: Iterations is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5: Iterations by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.5.1: "Infinite loops" and break

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can write an infinite loop on purpose and then use the `break` statement to jump out of the loop.

This loop is obviously an *infinite loop* because the logical expression on the `while` statement is simply the logical constant `True`:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true by virtue of the fact that the expression is the constant value `True`.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using `break`when we have reached the exit condition.

For example, suppose you want to take input from the user until they type `done`. You could write:

Code 5.4.1 (Python)

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone1.py
```

The loop condition is `True`, which is always true, so the loop runs repeatedly until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> hello there
hello there
> finished
finished
> done
Done!
```

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

This page titled 28.5.1: "Infinite loops" and break is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.4: "Infinite loops" and break by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.5.2: Finishing iterations with continue

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the `continue` statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types "done", but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

Code 5.5.1 (Python)

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

Here is a sample run of this new program with `continue` added.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the `continue` is executed, it ends the current iteration and jumps back to the `while` statement to start the next iteration, thus skipping the `print` statement.

This page titled 28.5.2: Finishing iterations with continue is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.5: Finishing iterations with continue by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.5.3: Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide. One way to cut your debugging time is "debugging by bisection." For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

This page titled 28.5.3: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.10: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.5.E: Iterations (Exercises)

Exercise 1: Write a program which repeatedly reads numbers until the user enters "done". Once "done" is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect their mistake using `try` and `except` and print an error message and skip to the next number.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.333333333333333
```

Exercise 2: Write another program that prompts for a list of numbers as above and at the end prints out both the maximum and minimum of the numbers instead of the average.

---

1. We will examine lists in more detail in a later chapter.↩

This page titled 28.5.E: Iterations (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 5.E: Iterations (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

accumulator
A variable used in a loop to add up or accumulate a result.

counter
A variable used in a loop to count the number of times something happened. We initialize a counter to zero and then increment the counter each time we want to "count" something.

decrement
An update that decreases the value of a variable.

initialize
An assignment that gives an initial value to a variable that will be updated.

increment
An update that increases the value of a variable (often by one).

infinite loop
A loop in which the terminating condition is never satisfied or for which there is no terminating condition.

iteration
Repeated execution of a set of statements using either a function that calls itself or a loop.

This page titled 28.5.G: Iterations (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

28.6: STRINGS

28.6.1: Debugging
28.6.E: Strings (Exercises)
28.6.G: Strings (Glossary)

This page titled 28.6: Strings is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6: Strings by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.6.1: Debugging

A skill that you should cultivate as you program is always asking yourself, "What could go wrong here?" or alternatively, "What crazy thing might our user do to crash our (seemingly) perfect program?"

For example, look at the program which we used to demonstrate the `while` loop in the chapter on iteration:

Code 6.12.1 (Python)

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

Look what happens when the user enters an empty line of input:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

The code works fine until it is presented an empty line. Then there is no zero-th character, so we get a traceback. There are two solutions to this to make line three "safe" even if the line is empty.

One possibility is to simply use the `startswith` method which returns `False` if the string is empty.

```
    if line.startswith('#'):
```

Another way is to safely write the `if` statement using the *guardian* pattern and make sure the second logical expression is evaluated only where there is at least one character in the string.:

```
    if len(line) > 0 and line[0] == '#':
```

This page titled 28.6.1: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.12: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.6.E: Strings (Exercises)

Exercise 5: Take the following Python code that stores a string:`

```
str = 'X-DSPAM-Confidence:0.8475'
```

Use `find` and string slicing to extract the portion of the string after the colon character and then use the `float` function to convert the extracted string into a floating point number.

Exercise 6:

Read the documentation of the string methods at

https://docs.python.org/3.5/library/...string-methods

You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful.

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional.

---

1. A tuple is a sequence of comma-separated values inside a pair of parenthesis. We will cover tuples in Chapter 10↩

This page titled 28.6.E: Strings (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 6.E: Strings (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

counter
A variable used to count something, usually initialized to zero and then incremented.

empty string
A string with no characters and length 0, represented by two quotation marks.

format operator
An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format sequence
A sequence of characters in a format string, like %d, that specifies how a value should be formatted.

format string
A string, used with the format operator, that contains format sequences.

flag
A boolean variable used to indicate whether a condition is true or false.

invocation
A statement that calls a method.

immutable
The property of a sequence whose items cannot be assigned.

index
An integer value used to select an item in a sequence, such as a character in a string.

item
One of the values in a sequence.

method
A function that is associated with an object and called using dot notation.

object
Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

search
A pattern of traversal that stops when it finds what it is looking for.

sequence
An ordered set; that is, a set of values where each value is identified by an integer index.

slice
A part of a string specified by a range of indices.

traverse
To iterate through the items in a sequence, performing a similar operation on each.

This page titled 28.6.G: Strings (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

28.7: FILES

In this chapter, we start to work with *Secondary Memory* (or files). Secondary memory is not erased when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

28.7.1: Persistence

28.7.2: Debugging

28.7.E: Files (Exercises)

28.7.G: Files (Glossary)

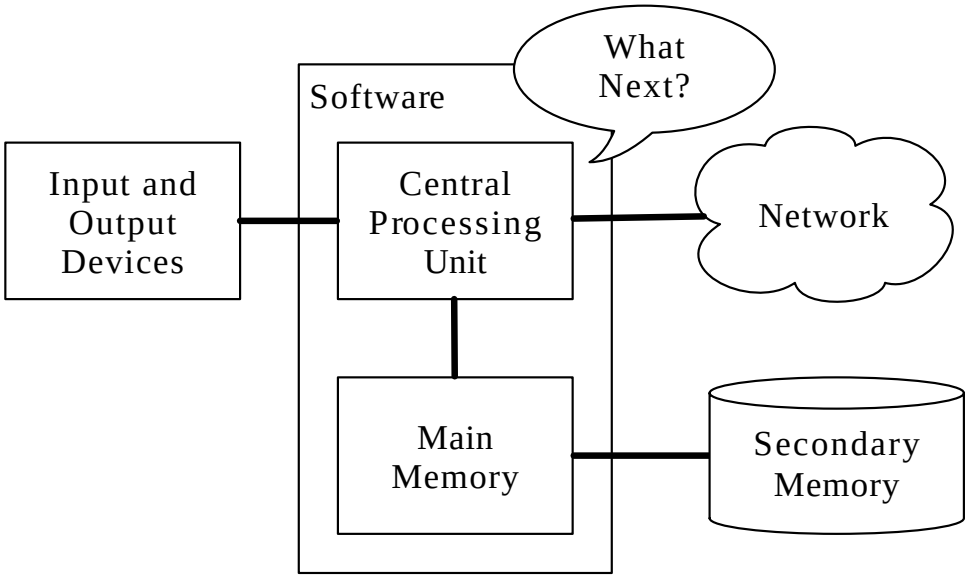
This page titled 28.7: Files is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7: Files by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.7.1: Persistence

So far, we have learned how to write programs and communicate our intentions to the *Central Processing Unit* using conditional execution, functions, and iterations. We have learned how to create and use data structures in the *Main Memory*. The CPU and memory are where our software works and runs. It is where all of the "thinking" happens.

But if you recall from our hardware architecture discussions, once the power is turned off, anything stored in either the CPU or main memory is erased. So up to now, our programs have just been transient fun exercises to learn Python.



Secondary Memory

In this chapter, we start to work with *Secondary Memory* (or files). Secondary memory is not erased when the power is turned off. Or in the case of a USB flash drive, the data we write from our programs can be removed from the system and transported to another system.

We will primarily focus on reading and writing text files such as those we create in a text editor. Later we will see how to work with database files which are binary files, specifically designed to be read and written through database software.

This page titled 28.7.1: Persistence is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.1: Persistence by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.7.2: Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs, and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies might cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at Wikipedia.org/wiki/Newline. Or, of course, you could write one yourself.

This page titled 28.7.2: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.9: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.7.E: Files (Exercises)

Exercise 1: Write a program to read through a file and print the contents of the file (line by line) all in upper case. Executing the program will look as follows:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

You can download the file from

www.py4e.com/code3/mbox-short.txt

Exercise 2: Write a program to prompt for a file name, and then read through the file and look for lines of the form:

`X-DSPAM-Confidence:0.8475`

When you encounter a line that starts with "X-DSPAM-Confidence:" pull apart the line to extract the floating-point number on the line. Count these lines and then compute the total of the spam confidence values from these lines. When you reach the end of the file, print out the average spam confidence.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

Test your file on the `mbox.txt` and `mbox-short.txt` files.

Exercise 3: Sometimes when programmers get bored or want to have a bit of fun, they add a harmless *Easter Egg* to their program Modify the program that prompts the user for the file name so that it prints a funny message when the user types in the exact file name "na na boo boo". The program should behave normally for all other files which exist and don't exist. Here is a sample execution of the program:

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

We are not encouraging you to put Easter Eggs in your programs; this is just an exercise.

This page titled 28.7.E: Files (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 7.E: Files (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

catch
: To prevent an exception from terminating a program using the `try` and `except` statements.

newline
: A special character used in files and strings to indicate the end of a line.

Pythonic
: A technique that works elegantly in Python. "Using try and except is the *Pythonic* way to recover from missing files".

Quality Assurance
: A person or team focused on insuring the overall quality of a software product. QA is often involved in testing a product and identifying problems before the product is released.

text file
: A sequence of characters stored in permanent storage like a hard drive.

This page titled 28.7.G: Files (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

28.8: LISTS

28.8.1: Lists and Functions
28.8.2: Lists and Strings
28.8.3: Parsing lines
28.8.4: Objects and Values
28.8.5: Aliasing
28.8.6: List arguments
28.8.7: Debugging
28.8.E: Lists (Exercises)
28.8.G: Lists (Glossary)

This page titled 28.8: Lists is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8: Lists by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.1: Lists and Functions

There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

The `sum()` function only works when the list elements are numbers. The other functions (`max()`, `len()`, etc.) work with lists of strings and other types that can be comparable.

We could rewrite an earlier program that computed the average of a list of numbers entered by the user using a list.

First, the program to compute an average without a list:

Code 8.8.1 (Python)

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Average:', average)

# Code: http://www.py4e.com/code3/avenum.py

```

In this program, we have `count` and `total` variables to keep the number and running total of the user's numbers as we repeatedly prompt the user for a number.

We could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

Code 8.8.1 (Python)

```
numlist = list()
while (True):
```

```
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Average:', average)

# Code: http://www.py4e.com/code3/avelist.py

```

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

This page titled 28.8.1: Lists and Functions is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.8: Lists and Functions by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.2: Lists and Strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. I also avoid the letter `l` because it looks too much like the number `1`. So that's why I use `t`.

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

Once you have used `split` to break the string into a list of words, you can use the index operator (square bracket) to look at a particular word in the list.

You can call `split` with an optional argument called a *delimiter* that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` is the inverse of `split`. It takes a list of strings and concatenates the elements. `join` is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so `join` puts a space between words. To concatenate strings without spaces, you can use the empty string, "", as a delimiter.

This page titled 28.8.2: Lists and Strings is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.9: Lists and Strings by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.3: Parsing lines

Usually when we are reading a file we want to do something to the lines other than just printing the whole line. Often we want to find the "interesting lines" and then *parse* the line to find some interesting *part* of the line. What if we wanted to print out the day of the week from those lines that start with "From "?

```
From stephen.marquard@uct.ac.zaSatJan 5 09:14:16 2008
```

The `split` method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with "From ", `split` those lines, and then print out the third word in the line:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Code: http://www.py4e.com/code3/search5.py
```

Here we also use the contracted form of the `if` statement where we put the `continue` on the same line as the `if`. This contracted form of the `if` functions the same as if the `continue` were on the next line and indented.

The program produces the following output:

```
Sat
Fri
Fri
Fri
    ...
```

Later, we will learn increasingly sophisticated techniques for picking the lines to work on and how we pull those lines apart to find the exact bit of information we are looking for.

This page titled 28.8.3: Parsing lines is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.10: Parsing lines by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.4: Objects and Values

If we execute these assignment statements:

```
a = 'banana'
b = 'banana'
```

we know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:



Variables and Objects

In one case, `a` and `b` refer to two different objects that have the same value. In the second case, they refer to the same object.

To check whether two variables refer to the same object, you can use the `is` operator.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

In this example, Python only created one string object, and both `a` and `b` refer to it.

But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

In this case we would say that the two lists are *equivalent*, because they have the same elements, but not *identical*, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you execute `a = [1,2,3]`, `a` refers to a list object whose value is a particular sequence of elements. If another list has the same elements, we would say it has the same value.

This page titled 28.8.4: Objects and Values is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.11: Objects and Values by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.5: Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The association of a variable with an object is called a *reference*. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is *aliased*.

If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

it almost never makes a difference whether a and b refer to the same string or not.

This page titled 28.8.5: Aliasing is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.12: Aliasing by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.6: List arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

The parameter `t` and the variable `letters` are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the `append` method modifies a list, but the `+` operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

This difference is important when you write functions that are supposed to modify lists. For example, this function *does not* delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]              # WRONG!
```

The slice operator creates a new list and the assignment makes `t` refer to it, but none of that has any effect on the list that was passed as an argument.

An alternative is to write a function that creates and returns a new list. For example, `tail` returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
```

```
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

Exercise 1:

Write a function called `chop` that takes a list and modifies it, removing the first and last elements, and returns `None`.

Then write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements.

This page titled 28.8.6: List arguments is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.13: List arguments by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.7: Debugging

Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

1. Don't forget that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

   If you are used to writing string code like this:

   ```
   word = word.strip()
   ```

   It is tempting to write list code like this: ~~ {.python} t = t.sort() # WRONG! ~~

   Because `sort` returns `None`, the next operation you perform with `t` is likely to fail.

   Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode. The methods and operators that lists share with other sequences (like strings) are documented at https://docs.python.org/3.5/library/...nce-operations. The methods and operators that only apply to mutable sequences are documented at https://docs.python.org/3.5/library/...sequence-types.

2. Pick an idiom and stick with it.

   Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use `pop`, `remove`, `del`, or even a slice assignment.

   To add an element, you can use the `append` method or the `+` operator. But don't forget that these are right:

   ```
   t.append(x)
   t = t + [x]
   ```

   And these are wrong:

   ```
   t.append([x])          # WRONG!
   t = t.append(x)        # WRONG!
   t + [x]                # WRONG!
   t = t + x              # WRONG!
   ```

   Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

3. Make copies to avoid aliasing.

   If you want to use a method like `sort` that modifies the argument, but you need to keep the original list as well, you can make a copy.

   ```
   orig = t[:]
   t.sort()
   ```

   In this example you could also use the built-in function `sorted`, which returns a new, sorted list and leaves the original alone. But in that case you should avoid using `sorted` as a variable name!

4. Lists, `split`, and files

When we read and parse files, there are many opportunities to encounter input that can crash our program so it is a good idea to revisit the *guardian* pattern when it comes writing programs that read through a file and look for a "needle in the haystack".

Let's revisit our program that is looking for the day of the week on the from lines of our file:

```
From stephen.marquard@uct.ac.zaSatJan 5 09:14:16 2008
```

Since we are breaking this line into words, we could dispense with the use of `startswith` and simply look at the first word of the line to determine if we are interested in the line at all. We can use `continue` to skip lines that don't have "From" as the first word as follows:

```
01 fhand = open('mbox-short.txt')
02 for line in fhand:
03     words = line.split()
04     if words[0] != 'From' : continue
05     print(words[2])
```

This looks much simpler and we don't even need to do the `rstrip` to remove the newline at the end of the file. But is it better?

```
01         python search8.py
02         Sat
03         Traceback (most recent call last):
04           File "search8.py", line 5, in <module>
05             if words[0] != 'From' : continue
06         IndexError: list index out of range
```

It kind of works and we see the day from the first line (Sat), but then the program fails with a traceback error. What went wrong? What messed-up data caused our elegant, clever, and very Pythonic program to fail?

You could stare at it for a long time and puzzle through it or ask someone for help, but the quicker and smarter approach is to add a `print` statement. The best place to add the print statement is right before the line where the program failed and print out the data that seems to be causing the failure.

Now this approach may generate a lot of lines of output, but at least you will immediately have some clue as to the problem at hand. So we add a print of the variable `words` right before line five. We even add a prefix "Debug:" to the line so we can keep our regular output separate from our debug output.

```
01     for line in fhand:
02         words = line.split()
03         print('Debug:', words)
04         if words[0] != 'From' : continue
05         print(words[2])
```

When we run the program, a lot of output scrolls off the screen but at the end, we see our debug output and the traceback so we know what happened just before the traceback.

```
01         Debug: ['X-DSPAM-Confidence:', '0.8475']
02         Debug: ['X-DSPAM-Probability:', '0.0000']
03         Debug: []
04         Traceback (most recent call last):
05           File "search9.py", line 6, in <module>
06             if words[0] != 'From' : continue
07         IndexError: list index out of range
```

Each debug line is printing the list of words which we get when we `split` the line into words. When the program fails, the list of words is empty `[]`. If we open the file in a text editor and look at the file, at that point it looks as follows:

```
        X-DSPAM-Result: Innocent
        X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
        X-DSPAM-Confidence: 0.8475
        X-DSPAM-Probability: 0.0000

        Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

The error occurs when our program encounters a blank line! Of course there are "zero words" on a blank line. Why didn't we think of that when we were writing the code? When the code looks for the first word (`word[0]`) to check to see if it matches "From", we get an "index out of range" error.

This of course is the perfect place to add some *guardian* code to avoid checking the first word if the first word is not there. There are many ways to protect this code; we will choose to check the number of words we have before we look at the first word:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print 'Debug:', words
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print(words[2])
```

First we commented out the debug print statement instead of removing it, in case our modification fails and we need to debug again. Then we added a guardian statement that checks to see if we have zero words, and if so, we use `continue` to skip to the next line in the file.

We can think of the two `continue` statements as helping us refine the set of lines which are "interesting" to us and which we want to process some more. A line which has no words is "uninteresting" to us so we skip to the next line. A line which does not have "From" as its first word is uninteresting to us so we skip it.

The program as modified runs successfully, so perhaps it is correct. Our guardian statement does make sure that the `words[0]` will never fail, but perhaps it is not enough. When we are programming, we must always be thinking, "What might go wrong?"

Exercise 2: Figure out which line of the above program is still not properly guarded. See if you can construct a text file which causes the program to fail and then modify the program so that the line is properly guarded and test it to make sure it handles your new text file.

Exercise 3: Rewrite the guardian code in the above example without two `if` statements. Instead, use a compound logical expression using the `and` logical operator with a single `if`statement.

This page titled 28.8.7: Debugging is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.14: Debugging by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.8.E: Lists (Exercises)

Exercise 4: Download a copy of the file from www.py4e.com/code3/romeo.txt

Write a program to open the file `romeo.txt` and read it line by line. For each line, split the line into a list of words using the `split` function.

For each word, check to see if the word is already in a list. If the word is not in the list, add it to the list.

When the program completes, sort and print the resulting words in alphabetical order.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Exercise 5: Write a program to read through the mail box data and when you find line that starts with "From", you will split the line into words using the `split` function. We are interested in who sent the message, which is the second word on the From line.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

You will parse the From line and print out the second word for each From line, then you will also count the number of From (not From:) lines and print out a count at the end.

This is a good sample output with a few lines removed:

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

Exercise 6: Rewrite the program that prompts the user for a list of numbers and prints out the maximum and minimum of the numbers at the end when the user enters "done". Write the program to store the numbers the user enters in a list and use the `max()` and `min()` functions to compute the maximum and minimum numbers after the loop completes.

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
```

```
08 Minimum: 2.0
```

This page titled 28.8.E: Lists (Exercises) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 8.E: Lists (Exercises) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

aliasing
A circumstance where two or more variables refer to the same object.

delimiter
A character or string used to indicate where a string should be split.

element
One of the values in a list (or other sequence); also called items.

equivalent
Having the same value.

index
An integer value that indicates an element in a list.

identical
Being the same object (which implies equivalence).

list
A sequence of values.

list traversal
The sequential accessing of each element in a list.

nested list
A list that is an element of another list.

object
Something a variable can refer to. An object has a type and a value.

reference
The association between a variable and its value.

This page titled 28.8.G: Lists (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

28.9: OBJECT-ORIENTED PROGRAMMING

As programs get to be millions of lines long, it becomes increasingly important to write code that is easy to understand. If you are working on a million line program, you can never keep the entire program in your mind at the same time. So we need ways to break the program into multiple smaller pieces so to solve a problem, fix a bug, or add a new feature we have less to look at. In a way, object oriented programming is a way to arrange your code so that you can zoom into 500 lines of the code, and understand it while ignoring the other 999,500 lines of code for the moment.

28.9.1: Starting with Programs

28.9.2: Subdividing a Problem - Encapsulation

28.9.3: Classes as Types

28.9.4: Many Instances

28.9.5: Object Lifecycle

28.9.6: Inheritance

28.9.G: Object-Oriented Programming (Glossary)

28.9.S: Object-Oriented Programming (Summary)

This page titled 28.9: Object-Oriented Programming is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14: Object-Oriented Programming by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

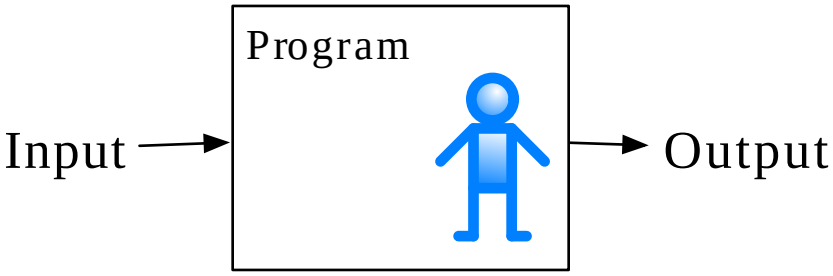
28.9.1: Starting with Programs

A program in its most basic form takes some input, does some processing, and produces some output. Our elevator conversion program demonstrates a very short but complete program showing all three of these steps.

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)

# Code: http://www.py4e.com/code3/elev.py
```

If we think a bit more about this program, there is the "outside world" and the program. The input and output aspects are where the program interacts with the outside world. Within the program we have code and data to accomplish the task the program is designed to solve.



A Program

When we are "in" the program, we have some defined interactions with the "outside" world, but those interactions are well defined and generally not something we focus on. While we are coding we worry only about the details "inside the program".

One way to think about object oriented programming is that we are separating our program into multiple "zones". Each "zone" contains some code and data (like a program) and has well defined interactions with the outside world and the other zones within the program.

If we look back at the link extraction application where we used the BeautifulSoup library, we can see a program that is constructed by connecting different objects together to accomplish a task:

Code 14.4.1 (Python)

```
# To run this, you can install BeautifulSoup
# https://pypi.python.org/pypi/beautifulsoup4

# Or download the file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
```

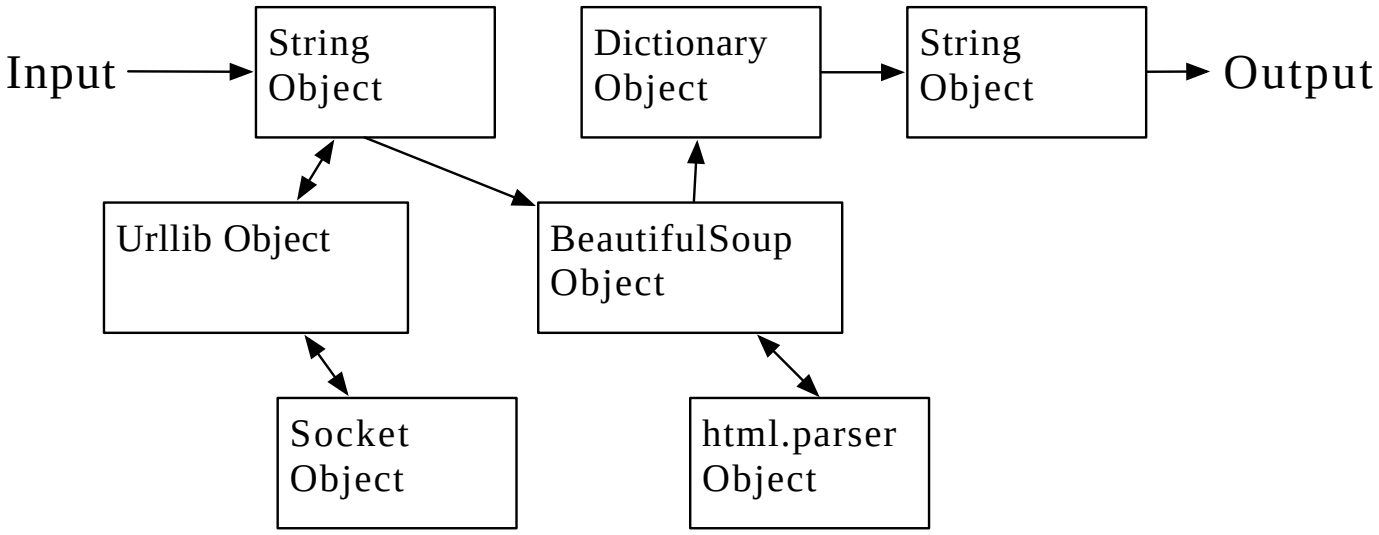
```
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: http://www.py4e.com/code3/urllinks.py
```

We read the URL into a string, and then pass that into `urllib` to retrieve the data from the web. The `urllib` library uses the `socket` library to make the actual network connection to retrieve the data. We take the string that we get back from `urllib` and hand it to BeautifulSoup for parsing. BeautifulSoup makes use of another object called `html.parser`[1] and returns an object. We call the `tags()` method in the returned object and then get a dictionary of tag objects, and loop through the tags and call the `get()` method for each tag to print out the 'href' attribute.



A Program as Network of Objects

We can draw a picture of this program and how the objects work together.

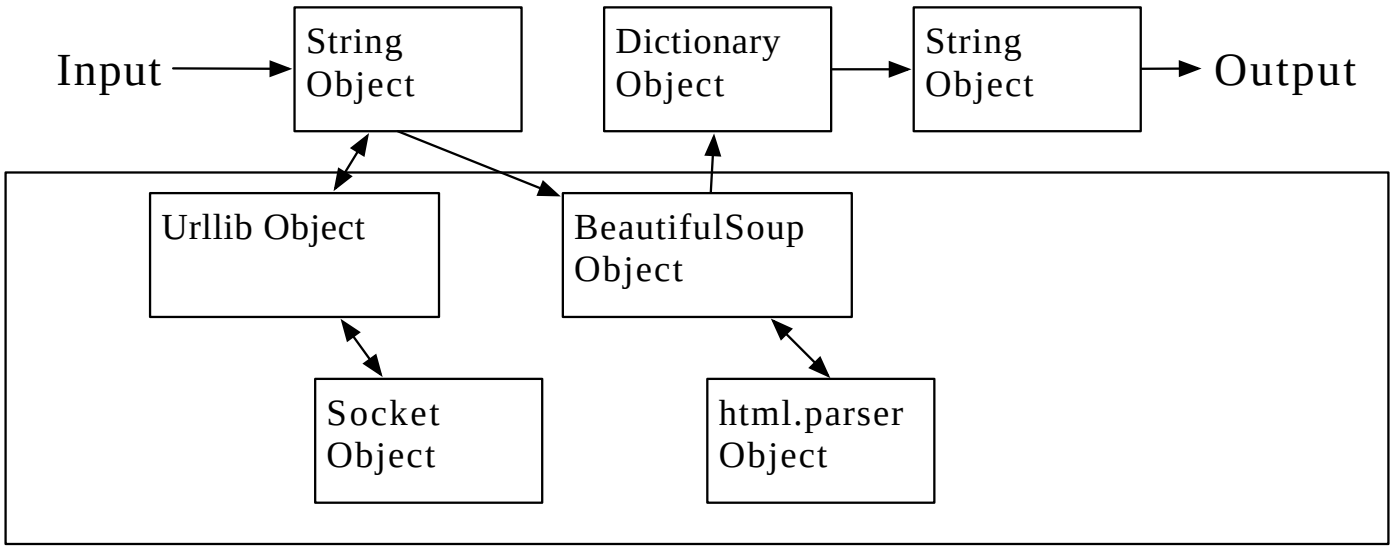
The key here is not to fully understand how this program works but to see how we build a network of interacting objects and orchestrate the movement of information between the objects to create a program. It is also important to note that when you looked at that program several chapters back, you could fully understand what was going on in the program without even realizing that the program was "orchestrating the movement of data between objects". Back then it was just lines of code that got the job done.

This page titled 28.9.1: Starting with Programs is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.4: Starting with Programs by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

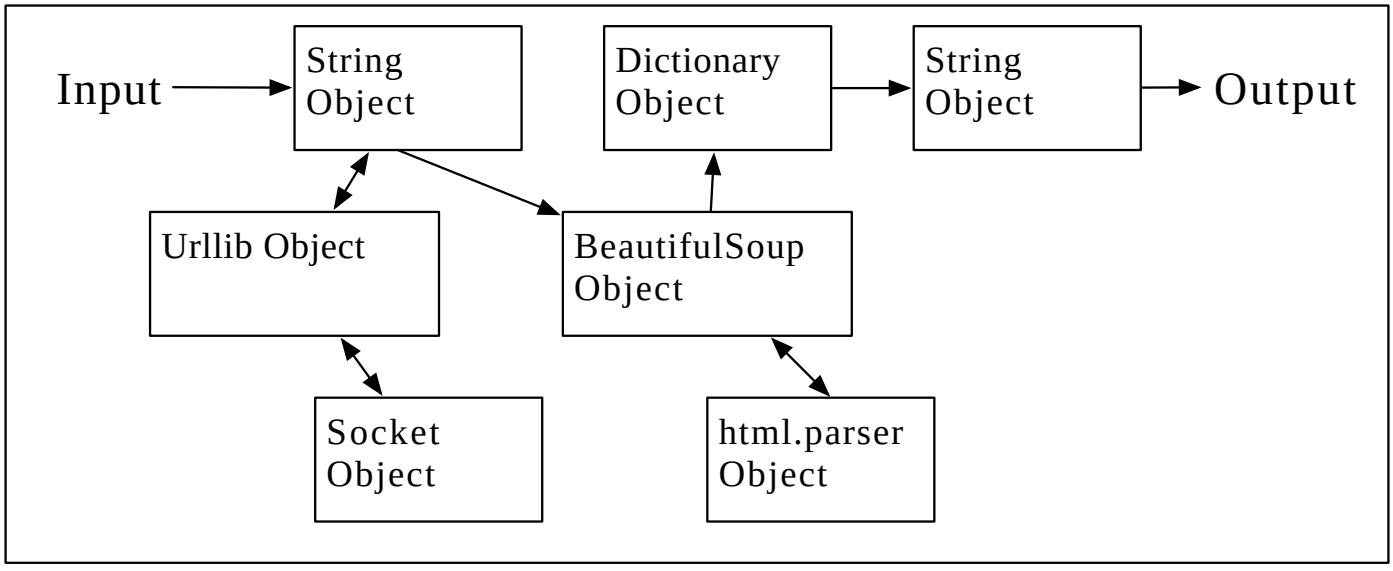
28.9.2: Subdividing a Problem - Encapsulation

One of the advantages of the object oriented approach is that it can hide complexity. For example, while we need to know how to use the `urllib` and BeautifulSoup code, we do not need to know how those libraries work internally. It allows us to focus on the part of the problem we need to solve and ignore the other parts of the program.



Ignoring Detail When Using an Object

This ability to focus on a part of a program that we care about and ignore the rest of the program is also helpful to the developers of the objects. For example the programmers developing BeautifulSoup do not need to know or care about how we retrieve our HTML page, what parts we want to read or what we plan to do with the data we extract from the web page.



Ignoring Detail When Building an Object

Another word we use to capture this idea that we ignore the internal detail of objects we use is "encapsulation". This means that we can know how to use an object without knowing how it internally accomplishes what we need done.

This page titled 28.9.2: Subdividing a Problem - Encapsulation is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.5: Subdividing a Problem - Encapsulation by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.9.3: Classes as Types

As we have seen, in Python, all variables have a type. And we can use the built-in `dir` function to examine the capabilities of a variable. We can use `type` and `dir` with the classes that we create.

Code 14.7.1 (Python)

```
%%python3

class PartyAnimal:
   x = 0

   def party(self) :
     self.x = self.x + 1
     print(\"So far\",self.x)

an = PartyAnimal()
print (\"Type\", type(an))
print (\"Dir \", dir(an))
print (\"Type\", type(an.x))
print (\"Type\", type(an.party))

# Code: http://www.py4e.com/code3/party3.py

```

When this program executes, it produces the following output:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party', 'x']
Type <class 'int'>
Type <class 'method'>
```

You can see that using the `class` keyword, we have created a new type. From the `dir` output, you can see both the `x` integer attribute and the `party` method are available in the object.

This page titled 28.9.3: Classes as Types is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.7: Classes as Types by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.9.4: Many Instances

So far, we have been defining a class, making a single object, using that object, and then throwing the object away. But the real power in object oriented happens when we make many instances of our class.

When we are making multiple objects from our class, we might want to set up different initial values for each of the objects. We can pass data into the constructors to give each object a different initial value:

Code 14.8.1 (Python)

```
%%python3

class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name,'constructed')

   def party(self) :
     self.x = self.x + 1
     print(self.name,'party count',self.x)

s = PartyAnimal('Sally')
j = PartyAnimal('Jim')

s.party()
j.party()
s.party()

# Code: http://www.py4e.com/code3/party5.py

```

The constructor has both a `self` parameter that points to the object instance and then additional parameters that are passed into the constructor as the object is being constructed:

```
s = PartyAnimal('Sally')
```

Within the constructor, the line:

```
self.name = nam
```

Copies the parameter that is passed in (`nam`) into the `name` attribute within the object instance.

The output of the program shows that each of the objects (`s` and `j`) contain their own independent copies of `x` and `nam`:

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Sally party count 2
```

This page titled 28.9.4: Many Instances is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.8: Many Instances by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.9.5: Object Lifecycle

In the previous examples, we are defining a class (template) and using that class to create an instance of that class (object) and then using the instance. When the program finishes, all the variables are discarded. Usually we don't think much about the creation and destruction of variables, but often as our objects become more complex, we need to take some action within the object to set things up as the object is being constructed and possibly clean things up as the object is being discarded.

If we want our object to be aware of these moments of construction and destruction, we add specially named methods to our object:

Code 14.9.1 (Python)

```
%%python3

class PartyAnimal:
   x = 0

   def __init__(self):
     print('I am constructed')

   def party(self) :
     self.x = self.x + 1
     print('So far',self.x)

   def __del__(self):
     print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)

# Code: http://www.py4e.com/code3/party4.py

```

When this program executes, it produces the following output:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

As Python is constructing our object, it calls our `__init__` method to give us a chance to set up some default or initial values for the object. When Python encounters the line:

```
an = 42
```

It actually 'thows our object away' so it can reuse the `an` variable to store the value `42`. Just at the moment when our `an` object is being 'destroyed' our destructor code (`__del__`) is called. We cannot stop our variable from being destroyed, but we can do any necessary cleanup right before our object no longer exists.

When developing objects, it is quite common to add a constructor to an object to set in initial values in the object, it is relatively rare to need a destructor for an object.

This page titled 28.9.5: Object Lifecycle is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.9: Object Lifecycle by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.9.6: Inheritance

Another powerful feature of object oriented programming is the ability to create a new class by extending an existing class. When extending a class, we call the original class the 'parent class' and the new class as the 'child class'.

For this example, we will move our `PartyAnimal` class into its own file:

```
class PartyAnimal:
   x = 0
   name = ''
   def __init__(self, nam):
     self.name = nam
     print(self.name,'constructed')

   def party(self) :
     self.x = self.x + 1
     print(self.name,'party count',self.x)

# Code: http://www.py4e.com/code3/party.py
```

Then, we can 'import' the `PartyAnimal` class in a new file and extend it as follows:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):
   points = 0
   def six(self):
      self.points = self.points + 6
      self.party()
      print(self.name,"points",self.points)

s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
j.party()
j.six()
print(dir(j))

# Code: http://www.py4e.com/code3/party6.py
```

When we are defining the `CricketFan` object, we indicate that we are extending the `PartyAnimal` class. This means that all of the variables (`x`) and methods (`party`) from the `PartyAnimal` class are inherited by the `CricketFan` class.

You can see that within the `six` method in the `CricketFan` class, we can call the `party` method from the `PartyAnimal` class. The variables and methods from the parent class are *merged* into the child class.

As the program executes, we can see that the `s` and `j` are independent instances of `PartyAnimal` and `CricketFan`. The `j` object has additional capabilities beyond the `s` object.

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
```

```
Jim party count 2
Jim points 6
['__class__', '__delattr__', ... '__weakref__',
'name', 'party', 'points', 'six', 'x']
```

In the `dir` output for the `j` object (instance of the `CricketFan` class) you can see that it both has the attributes and methods of the parent class as well as the attributes and methods that were added when the class was extended to create the `CricketFan` class.

This page titled 28.9.6: Inheritance is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.10: Inheritance by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

attribute
A variable that is part of a class.

class
A template that can be used to construct an object. Defines the attributes and methods that will make up the object.

child class
A new class created when a parent class is extended. The child class inherits all of the attributes and methods of the parent class.

constructor
An optional specially named method (`__init__`) that is called at the moment when a class is being used to construct an object. Usually this is used to set up initial values for the object.

destructor
An optional specially named method (`__del__`) that is called at the moment just before an object is destroyed. Destructors are rarely used.

inheritance
When we create a new class (child) by extending an existing class (parent). The child class has all the attributes and methods of the parent class plus additional attributes and methods defined by the child class.

method
A function that is contained within a class and the objects that are constructed from the class. Some object-oriented patterns use 'message' instead of 'method' to describe this concept.

object
A constructed instance of a class. An object contains all of the attributes and methods that were defined by the class. Some object-oriented documentation uses the term 'instance' interchangeably with 'object'.

parent class
The class which is being extended to create a new child class. The parent class contributes all of its methods and attributes to the new child class.

---

1. https://docs.python.org/3/library/html.parser.html↩
2. Cookie image copyright CC-BY www.flickr.com/photos/dinner...es/23570475099↩
3. If you are curious about where the list class is defined, take a look at (hopefully the URL won't change) github.com/python/cpython/bl...s/listobject.c - the list class is written in a language called "C". If you take a look at that source code and find it curious you might want to explore a few Computer Science courses.↩

This page titled 28.9.G: Object-Oriented Programming (Glossary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.G: Object-Oriented Programming (Glossary) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

28.9.S: Object-Oriented Programming (Summary)

This is a very quick introduction to object-oriented programming that focuses mainly on terminology and the syntax of defining and using objects. Let's quickly review the code that we looked at in the beginning of the chapter. At this point you should fully understand what is going on.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])

print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: http://www.py4e.com/code3/party1.py
```

The first line constructs a `list` *object*. When Python creates the `list` object, it calls the *constructor* method (named `__init__`) to set up the internal data attributes that will be used to store the list data. Due to *encapsulation* we neither need to know, nor need to care about how these internal data attributes are arranged.

We are not passing any parameters to the *constructor* and when the constructor returns, we use the variable `stuff` to point to the returned instance of the `list` class.

The second and third lines are calling the `append` method with one parameter to add a new item at the end of the list by updating the attributes within `stuff`. Then in the fourth line, we call the `sort` method with no parameters to sort the data within the `stuff` object.

Then we print out the first item in the list using the square brackets which are a shortcut to calling the `__getitem__` method within the `stuff` *object*. And this is equivalent to calling the `__getitem__` method in the `list` *class* passing the `stuff` object in as the first parameter and the position we are looking for as the second parameter.

At the end of the program the `stuff` object is discarded but not before calling the *destructor* (named `__del__`) so the object can clean up any loose ends as necessary.

Those are the basics and terminology of object oriented programming. There are many additional details as to how to best use object oriented approaches when developing large applications and libraries that are beyond the scope of this chapter.[3]

This page titled 28.9.S: Object-Oriented Programming (Summary) is shared under a CC BY-NC-SA 4.0 license and was authored, remixed, and/or curated by Chuck Severance via source content that was edited to the style and standards of the LibreTexts platform.

- 14.S: Object-Oriented Programming (Summary) by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: https://www.py4e.com/html3.

INDEX

A

algorithm
- 26.1: Activity 1 - Overview of Algorithm Design and Analysis
- 26.2: Activity 1 - Introduction to Algorithms and Problem Solving
- 26.3: Activity 2 - The characteristics of an algorithm

algorithmics
- 26.1: Activity 1 - Overview of Algorithm Design and Analysis

API
- 25.5: Application Programming Interfaces

API key
- 25.7: Security and API usage

API token
- 25.7: Security and API usage

arguments
- 28.4.4: Parameters and Arguments
- 4.7: Parameters and Arguments

B

binary search
- 22.1: Activity 1 - Searching Algorithm

C

catching exceptions
- 10.7: Catching exceptions Using Try and Except

D

debugging (python)
- 28.5.3: Debugging

dot notation
- 14.5: Math Functions

E

except (python)
- 10.7: Catching exceptions Using Try and Except

eXtensible Markup Language
- 25.1: eXtensible Markup Language - XML

F

flowchart
- 7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms

I

infinite loops
- 28.5.1: "Infinite loops" and break

insertion sort
- 22.2: Activity 2 - Sorting Algorithm

iterations
- 28.5: Iterations

J

jupyter:python
- 10.7: Catching exceptions Using Try and Except
- 13.3: Definite loops using for
- 13.4: Counting and Summing Loops
- 14.2: Modulus Operator
- 14.4: Random Numbers
- 16.9: Asking the user for input
- 17.4: String Operations
- 17.9: Looping and Counting
- 18.2: The while Statement
- 18.6: Maximum and Minimum Loops
- 19.4: Traversing a List
- 2.2: Words and Sentences
- 2.3: Conversing with Python
- 20.2: Fruitful functions and void functions
- 23.4: Using Objects
- 23.5: Our First Python Object
- 24.2: The World's Simplest Web Browser
- 24.3: Retrieving an image over HTTP
- 24.4: Retrieving web pages with urllib
- 24.6: Parsing HTML using regular expressions
- 24.7: Parsing HTML using BeautifulSoup
- 24.8: Reading binary files using urllib
- 25.1: eXtensible Markup Language - XML
- 25.2: Looping through Nodes
- 25.4: Parsing JSON
- 25.6: Google geocoding web service
- 25.7: Security and API usage
- 28.2.5: Choosing Mnemonic Variable Names
- 28.2.E: Variables, Expressions, and Statements (Exercises)
- 28.3.E: Conditional Execution (Exercises)
- 28.5.1: "Infinite loops" and break
- 28.5.2: Finishing iterations with continue
- 28.6.1: Debugging
- 28.8.1: Lists and Functions
- 28.9.3: Classes as Types
- 28.9.4: Many Instances
- 28.9.5: Object Lifecycle
- 4.4: Definitions and Uses
- 6.6: Choosing Mnemonic Variable Names
- 6.7: Statements
- 8.8: Chained Conditionals
- 9.2: Definitions and Uses

M

modulus operator
- 14.2: Modulus Operator

P

parameters
- 28.4.4: Parameters and Arguments
- 4.7: Parameters and Arguments

Parsing JSON
- 25.4: Parsing JSON

Pseudocode
- 7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms

python (language)
1.1: What is a program?
1.3: Terminology - Interpreter and Compiler
1.4: Why should you learn to write programs?
1.5: Creativity and Motivation
1.6: Understanding Programming
1.7: The Learning Journey
10.2: Logical Operators
10.6: Nested Conditionals
10.7: Catching exceptions Using Try and Except
13.3: Definite loops using for
13.4: Counting and Summing Loops
14.2: Modulus Operator
14.4: Random Numbers
14.5: Math Functions
16.6: Values and Types
16.9: Asking the user for input
17.10: The in operator
17.11: String Comparison
17.12: String Methods
17.13: Parsing strings
17.15: Format operator
17.2: A string is a sequence
17.4: String Operations
17.5: Getting the length of a string using len
17.6: Traversal through a string with a loop
17.7: String Slices
17.8: Strings are immutable
17.9: Looping and Counting
18.2: The while Statement
18.4: Infinite Loops
18.5: Loop patterns
18.6: Maximum and Minimum Loops
19.4: Traversing a List
19.5: List operations
19.6: List Slices
19.7: List Methods
19.8: Deleting Elements
2.2: Words and Sentences
2.3: Conversing with Python
2.4: What could possibly go wrong?
2.5: Comments
2.6: Comments
20.2: Fruitful functions and void functions
21.2: Opening Files
21.3: Text files and Lines
21.4: Reading Files
21.5: Writing Files
21.6: Using try, except, and open
21.8: Letting the user choose the file name
21.9: Searching through a File
23.1: Managing Larger Programs
23.2: Getting Started
23.4: Using Objects
23.5: Our First Python Object
24: Networked Programs
24.1: HyperText Transfer Protocol - HTTP
24.2: The World's Simplest Web Browser
24.3: Retrieving an image over HTTP
24.4: Retrieving web pages with urllib
24.5: Parsing HTML and scraping the web
24.6: Parsing HTML using regular expressions
24.7: Parsing HTML using BeautifulSoup
24.8: Reading binary files using urllib
24.E: Networked Programs (Exercises)
24.G: Networked Programs (Glossary)
25: Python and Web Services
25.1: eXtensible Markup Language - XML
25.2: Looping through Nodes
25.3: JavaScript Object Notation - JSON
25.4: Parsing JSON
25.5: Application Programming Interfaces
25.6: Google geocoding web service
25.7: Security and API usage
25.E: Python and Web Services (Exercises)
25.G: Python and Web Services (Glossary)
27.1: Introduction
27.1.1: Computer Hardware Architecture
27.1.2: Writing a Program
27.1.E: Introduction (Exercises)
27.1.G: Introduction (Glossary)
28.2: Variables, Expressions, and Statements
28.2.1: Variable names and Keywords
28.2.2: Operators and Operands
28.2.3: Expressions
28.2.4: Order of Operations
28.2.5: Choosing Mnemonic Variable Names
28.2.6: Debugging
28.2.E: Variables, Expressions, and Statements (Exercises)
28.2.G: Glossary
28.3: Conditional Execution
28.3.1: Short-Circuit Evaluation of Logical Expressions
28.3.2: Debugging
28.3.E: Conditional Execution (Exercises)
28.3.G: Conditional Execution (Glossary)
28.4: Functions
28.4.1: Function Calls
28.4.2: Built-in Functions
28.4.3: Type Conversion Functions
28.4.4: Parameters and Arguments
28.4.5: Debugging
28.4.E: Functions (Exercises)
28.4.G: Functions (Glossary)
28.5: Iterations
28.5.1: "Infinite loops" and break
28.5.2: Finishing iterations with continue
28.5.3: Debugging
28.5.E: Iterations (Exercises)
28.5.G: Iterations (Glossary)
28.6: Strings
28.6.1: Debugging
28.6.E: Strings (Exercises)
28.6.G: Strings (Glossary)
28.7: Files
28.7.1: Persistence
28.7.2: Debugging
28.7.E: Files (Exercises)
28.7.G: Files (Glossary)
28.8: Lists
28.8.1: Lists and Functions
28.8.2: Lists and Strings
28.8.3: Parsing lines
28.8.4: Objects and Values
28.8.5: Aliasing
28.8.6: List arguments
28.8.7: Debugging
28.8.E: Lists (Exercises)
28.8.G: Lists (Glossary)
28.9: Object-Oriented Programming
28.9.1: Starting with Programs
28.9.2: Subdividing a Problem - Encapsulation
28.9.3: Classes as Types
28.9.4: Many Instances
28.9.5: Object Lifecycle
28.9.6: Inheritance
28.9.G: Object-Oriented Programming (Glossary)
28.9.S: Object-Oriented Programming (Summary)
3.1: The Building Blocks of Programs
4.2: Why functions?
4.3: Function Calls
4.4: Definitions and Uses
4.5: Flow of Execution
4.7: Parameters and Arguments
6.10: Updating Variables
6.14: A list is a sequence
6.15: Lists are mutable
6.16: Debugging
6.5: Variable names and Keywords
6.6: Choosing Mnemonic Variable Names
6.7: Statements
6.9: Variables
8.2: Boolean Expressions
8.3: Conditional Execution
8.5: Alternative Execution
8.8: Chained Conditionals
9.1: Adding New Functions
9.2: Definitions and Uses
9.3: Why functions?
9.4: Flow of Execution

Q

quicksort
22.2: Activity 2 - Sorting Algorithm

S

searching
22.1: Activity 1 - Searching Algorithm
sequential search
22.1: Activity 1 - Searching Algorithm
shell sort
22.2: Activity 2 - Sorting Algorithm
sorting
22.2: Activity 2 - Sorting Algorithm

T

time complexity
26.1: Activity 1 - Overview of Algorithm Design and Analysis
try (python)
10.7: Catching exceptions Using Try and Except

V

Variables (python)
6.9: Variables
void functions
20.2: Fruitful functions and void functions

W

www.py4e.com/code3/geojson.py
25.E: Python and Web Services (Exercises)
www.py4e.com/code3/geoxml.py
25.E: Python and Web Services (Exercises)

X

XML
25.1: eXtensible Markup Language - XML

This page titled Index is shared under a license and was authored, remixed, and/or curated by LibreTexts.

INDEX

A

algorithm
- 26.1: Activity 1 - Overview of Algorithm Design and Analysis
- 26.2: Activity 1 - Introduction to Algorithms and Problem Solving
- 26.3: Activity 2 - The characteristics of an algorithm

algorithmics
- 26.1: Activity 1 - Overview of Algorithm Design and Analysis

API
- 25.5: Application Programming Interfaces

API key
- 25.7: Security and API usage

API token
- 25.7: Security and API usage

arguments
- 28.4.4: Parameters and Arguments
- 4.7: Parameters and Arguments

B

binary search
- 22.1: Activity 1 - Searching Algorithm

C

catching exceptions
- 10.7: Catching exceptions Using Try and Except

D

debugging (python)
- 28.5.3: Debugging

dot notation
- 14.5: Math Functions

E

except (python)
- 10.7: Catching exceptions Using Try and Except

eXtensible Markup Language
- 25.1: eXtensible Markup Language - XML

F

flowchart
- 7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms

I

infinite loops
- 28.5.1: "Infinite loops" and break

insertion sort
- 22.2: Activity 2 - Sorting Algorithm

iterations
- 28.5: Iterations

J

jupyter:python
- 10.7: Catching exceptions Using Try and Except
- 13.3: Definite loops using for
- 13.4: Counting and Summing Loops
- 14.2: Modulus Operator
- 14.4: Random Numbers
- 16.9: Asking the user for input
- 17.4: String Operations
- 17.9: Looping and Counting
- 18.2: The while Statement
- 18.6: Maximum and Minimum Loops
- 19.4: Traversing a List
- 2.2: Words and Sentences
- 2.3: Conversing with Python
- 20.2: Fruitful functions and void functions
- 23.4: Using Objects
- 23.5: Our First Python Object
- 24.2: The World's Simplest Web Browser
- 24.3: Retrieving an image over HTTP
- 24.4: Retrieving web pages with urllib
- 24.6: Parsing HTML using regular expressions
- 24.7: Parsing HTML using BeautifulSoup
- 24.8: Reading binary files using urllib
- 25.1: eXtensible Markup Language - XML
- 25.2: Looping through Nodes
- 25.4: Parsing JSON
- 25.6: Google geocoding web service
- 25.7: Security and API usage
- 28.2.5: Choosing Mnemonic Variable Names
- 28.2.E: Variables, Expressions, and Statements (Exercises)
- 28.3.E: Conditional Execution (Exercises)
- 28.5.1: "Infinite loops" and break
- 28.5.2: Finishing iterations with continue
- 28.6.1: Debugging
- 28.8.1: Lists and Functions
- 28.9.3: Classes as Types
- 28.9.4: Many Instances
- 28.9.5: Object Lifecycle
- 4.4: Definitions and Uses
- 6.6: Choosing Mnemonic Variable Names
- 6.7: Statements
- 8.8: Chained Conditionals
- 9.2: Definitions and Uses

M

modulus operator
- 14.2: Modulus Operator

P

parameters
- 28.4.4: Parameters and Arguments
- 4.7: Parameters and Arguments

Parsing JSON
- 25.4: Parsing JSON

Pseudocode
- 7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms

python (language)
1.1: What is a program?
1.3: Terminology - Interpreter and Compiler
1.4: Why should you learn to write programs?
1.5: Creativity and Motivation
1.6: Understanding Programming
1.7: The Learning Journey
10.2: Logical Operators
10.6: Nested Conditionals
10.7: Catching exceptions Using Try and Except
13.3: Definite loops using for
13.4: Counting and Summing Loops
14.2: Modulus Operator
14.4: Random Numbers
14.5: Math Functions
16.6: Values and Types
16.9: Asking the user for input
17.10: The in operator
17.11: String Comparison
17.12: String Methods
17.13: Parsing strings
17.15: Format operator
17.2: A string is a sequence
17.4: String Operations
17.5: Getting the length of a string using len
17.6: Traversal through a string with a loop
17.7: String Slices
17.8: Strings are immutable
17.9: Looping and Counting
18.2: The while Statement
18.4: Infinite Loops
18.5: Loop patterns
18.6: Maximum and Minimum Loops
19.4: Traversing a List
19.5: List operations
19.6: List Slices
19.7: List Methods
19.8: Deleting Elements
2.2: Words and Sentences
2.3: Conversing with Python
2.4: What could possibly go wrong?
2.5: Comments
2.6: Comments
20.2: Fruitful functions and void functions
21.2: Opening Files
21.3: Text files and Lines
21.4: Reading Files
21.5: Writing Files
21.6: Using try, except, and open
21.8: Letting the user choose the file name
21.9: Searching through a File
23.1: Managing Larger Programs
23.2: Getting Started
23.4: Using Objects
23.5: Our First Python Object
24: Networked Programs
24.1: HyperText Transfer Protocol - HTTP
24.2: The World's Simplest Web Browser
24.3: Retrieving an image over HTTP
24.4: Retrieving web pages with urllib
24.5: Parsing HTML and scraping the web
24.6: Parsing HTML using regular expressions
24.7: Parsing HTML using BeautifulSoup
24.8: Reading binary files using urllib
24.E: Networked Programs (Exercises)
24.G: Networked Programs (Glossary)
25: Python and Web Services
25.1: eXtensible Markup Language - XML
25.2: Looping through Nodes
25.3: JavaScript Object Notation - JSON
25.4: Parsing JSON
25.5: Application Programming Interfaces
25.6: Google geocoding web service
25.7: Security and API usage
25.E: Python and Web Services (Exercises)
25.G: Python and Web Services (Glossary)
27.1: Introduction
27.1.1: Computer Hardware Architecture
27.1.2: Writing a Program
27.1.E: Introduction (Exercises)
27.1.G: Introduction (Glossary)
28.2: Variables, Expressions, and Statements
28.2.1: Variable names and Keywords
28.2.2: Operators and Operands
28.2.3: Expressions
28.2.4: Order of Operations
28.2.5: Choosing Mnemonic Variable Names
28.2.6: Debugging
28.2.E: Variables, Expressions, and Statements (Exercises)
28.2.G: Glossary
28.3: Conditional Execution
28.3.1: Short-Circuit Evaluation of Logical Expressions
28.3.2: Debugging
28.3.E: Conditional Execution (Exercises)
28.3.G: Conditional Execution (Glossary)
28.4: Functions
28.4.1: Function Calls
28.4.2: Built-in Functions
28.4.3: Type Conversion Functions
28.4.4: Parameters and Arguments
28.4.5: Debugging
28.4.E: Functions (Exercises)
28.4.G: Functions (Glossary)
28.5: Iterations
28.5.1: "Infinite loops" and break
28.5.2: Finishing iterations with continue
28.5.3: Debugging
28.5.E: Iterations (Exercises)
28.5.G: Iterations (Glossary)
28.6: Strings
28.6.1: Debugging
28.6.E: Strings (Exercises)
28.6.G: Strings (Glossary)
28.7: Files
28.7.1: Persistence
28.7.2: Debugging
28.7.E: Files (Exercises)
28.7.G: Files (Glossary)
28.8: Lists
28.8.1: Lists and Functions
28.8.2: Lists and Strings
28.8.3: Parsing lines
28.8.4: Objects and Values
28.8.5: Aliasing
28.8.6: List arguments
28.8.7: Debugging
28.8.E: Lists (Exercises)
28.8.G: Lists (Glossary)
28.9: Object-Oriented Programming
28.9.1: Starting with Programs
28.9.2: Subdividing a Problem - Encapsulation
28.9.3: Classes as Types
28.9.4: Many Instances
28.9.5: Object Lifecycle
28.9.6: Inheritance
28.9.G: Object-Oriented Programming (Glossary)
28.9.S: Object-Oriented Programming (Summary)
3.1: The Building Blocks of Programs
4.2: Why functions?
4.3: Function Calls
4.4: Definitions and Uses
4.5: Flow of Execution
4.7: Parameters and Arguments
6.10: Updating Variables
6.14: A list is a sequence
6.15: Lists are mutable
6.16: Debugging
6.5: Variable names and Keywords
6.6: Choosing Mnemonic Variable Names
6.7: Statements
6.9: Variables
8.2: Boolean Expressions
8.3: Conditional Execution
8.5: Alternative Execution
8.8: Chained Conditionals
9.1: Adding New Functions
9.2: Definitions and Uses
9.3: Why functions?
9.4: Flow of Execution

Q

quicksort
22.2: Activity 2 - Sorting Algorithm

S

searching
22.1: Activity 1 - Searching Algorithm
sequential search
22.1: Activity 1 - Searching Algorithm
shell sort
22.2: Activity 2 - Sorting Algorithm
sorting
22.2: Activity 2 - Sorting Algorithm

T

time complexity
26.1: Activity 1 - Overview of Algorithm Design and Analysis
try (python)
10.7: Catching exceptions Using Try and Except

V

Variables (python)
6.9: Variables
void functions
20.2: Fruitful functions and void functions

W

www.py4e.com/code3/geojson.py
25.E: Python and Web Services (Exercises)
www.py4e.com/code3/geoxml.py
25.E: Python and Web Services (Exercises)

X

XML
25.1: eXtensible Markup Language - XML

This page titled Index is shared under a license and was authored, remixed, and/or curated by LibreTexts.

GLOSSARY

S

Sample Word 1 | Sample Definition 1

This page titled Glossary is shared under a license and was authored, remixed, and/or curated by LibreTexts.

DETAILED LICENSING

Detailed licensing information for this resource is not available at this time.

This page titled Intro to Programming with Programming Fundamentals and Python for Everyone is shared under a license and was authored, remixed, and/or curated by LibreTexts.

DETAILED LICENSING

Overview

Title: Intro to Programming with Programming Fundamentals and Python for Everyone

Webpages: 344

Applicable Restrictions: Noncommercial

All licenses found:

- CC BY-NC-SA 4.0: 45.9% (158 pages)
- CC BY-SA 4.0: 42.2% (145 pages)
- Undeclared: 11.6% (40 pages)
- CC BY-NC 4.0: 0.3% (1 page)

By Page

- Intro to Programming with Programming Fundamentals and Python for Everyone - *Undeclared*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
  - 1: Welcome to Programming - *Undeclared*
    - 1.1: What is a program? - *CC BY-NC-SA 4.0*
    - 1.2: Integrated Development Environment - *CC BY-SA 4.0*
    - 1.3: Terminology - Interpreter and Compiler - *CC BY-NC-SA 4.0*
    - 1.4: Why should you learn to write programs? - *CC BY-NC-SA 4.0*
    - 1.5: Creativity and Motivation - *CC BY-NC-SA 4.0*
    - 1.6: Understanding Programming - *CC BY-NC-SA 4.0*
    - 1.7: The Learning Journey - *CC BY-NC-SA 4.0*
  - 2: Commands - *Undeclared*
    - 2.1: Hello World - *CC BY-SA 4.0*
    - 2.2: Words and Sentences - *CC BY-NC-SA 4.0*
    - 2.3: Conversing with Python - *CC BY-NC-SA 4.0*
    - 2.4: What could possibly go wrong? - *CC BY-NC-SA 4.0*
    - 2.5: Comments - *CC BY-NC-SA 4.0*
    - 2.6: Comments - *CC BY-NC-SA 4.0*
  - 3: Principles of Structured Programming - *Undeclared*
    - 3.1: The Building Blocks of Programs - *CC BY-NC-SA 4.0*
    - 3.2: Structured Programming - *CC BY-SA 4.0*
  - 4: Functions and Events - *Undeclared*
    - 4.1: Modular Programming - *CC BY-SA 4.0*
    - 4.2: Why functions? - *CC BY-NC-SA 4.0*
    - 4.3: Function Calls - *CC BY-NC-SA 4.0*
    - 4.4: Definitions and Uses - *CC BY-NC-SA 4.0*
    - 4.5: Flow of Execution - *CC BY-NC-SA 4.0*
    - 4.6: Parameters and Arguments - *CC BY-SA 4.0*
    - 4.7: Parameters and Arguments - *CC BY-NC-SA 4.0*
  - 5: Algorithms and Design - *Undeclared*
    - 5.1: Thinking Algorithmically - *CC BY-NC 4.0*
    - 5.2: Program Design - *CC BY-SA 4.0*
    - 5.3: Program Quality - *CC BY-SA 4.0*
  - 6: Math and Variables - *Undeclared*
    - 6.1: Arithmetic Operators - *CC BY-SA 4.0*
    - 6.2: Order of Operations - *CC BY-SA 4.0*
    - 6.3: Constants and Variables - *CC BY-SA 4.0*
    - 6.4: Identifier Names - *CC BY-SA 4.0*
    - 6.5: Variable names and Keywords - *CC BY-NC-SA 4.0*
    - 6.6: Choosing Mnemonic Variable Names - *CC BY-NC-SA 4.0*
    - 6.7: Statements - *CC BY-NC-SA 4.0*
    - 6.8: Assignment - *CC BY-SA 4.0*
    - 6.9: Variables - *CC BY-NC-SA 4.0*
    - 6.10: Updating Variables - *CC BY-NC-SA 4.0*
    - 6.11: Python Examples - *CC BY-SA 4.0*
    - 6.12: Arrays and Lists - *CC BY-SA 4.0*
    - 6.13: Index Notation - *CC BY-SA 4.0*
    - 6.14: A list is a sequence - *CC BY-NC-SA 4.0*
    - 6.15: Lists are mutable - *CC BY-NC-SA 4.0*
    - 6.16: Debugging - *CC BY-NC-SA 4.0*
  - 7: Problem Solving and Design - *Undeclared*
    - 7.1: Pseudocode - *CC BY-SA 4.0*
    - 7.2: Flowcharts - *CC BY-SA 4.0*
    - 7.3: Activity 3 - Using pseudo-codes and flowcharts to represent algorithms - *CC BY-SA 4.0*
  - 8: Decisions - *Undeclared*

- 8.1: Relational Operators - *CC BY-SA 4.0*
- 8.2: Boolean Expressions - *CC BY-NC-SA 4.0*
- 8.3: Conditional Execution - *CC BY-NC-SA 4.0*
- 8.4: If Then Else - *CC BY-SA 4.0*
- 8.5: Alternative Execution - *CC BY-NC-SA 4.0*
- 8.6: Assignment vs Equality - *CC BY-SA 4.0*
- 8.7: Code Blocks - *CC BY-SA 4.0*
- 8.8: Chained Conditionals - *CC BY-NC-SA 4.0*

◦ 9: Helper Functions - *Undeclared*
- 9.1: Adding New Functions - *CC BY-NC-SA 4.0*
- 9.2: Definitions and Uses - *CC BY-NC-SA 4.0*
- 9.3: Why functions? - *CC BY-NC-SA 4.0*
- 9.4: Flow of Execution - *CC BY-NC-SA 4.0*
- 9.5: Function Examples - *CC BY-SA 4.0*
- 9.6: Hierarchy or Structure Chart - *CC BY-SA 4.0*

◦ 10: Nested Decision - *Undeclared*
- 10.1: Logical Operators - *CC BY-SA 4.0*
- 10.2: Logical Operators - *CC BY-NC-SA 4.0*
- 10.3: Nested If Then Else - *CC BY-SA 4.0*
- 10.4: Condition Examples - *CC BY-SA 4.0*
- 10.5: Python Examples - *CC BY-SA 4.0*
- 10.6: Nested Conditionals - *CC BY-NC-SA 4.0*
- 10.7: Catching exceptions Using Try and Except - *CC BY-NC-SA 4.0*

◦ 11: Software Development Life Cycle - *Undeclared*
- 11.1: Systems Development Life Cycle - *CC BY-SA 4.0*
- 11.2: Software Testing - *CC BY-SA 4.0*
- 11.3: Version Control - *CC BY-SA 4.0*
- 11.4: Programming Style - *CC BY-SA 4.0*

◦ 12: Variable Scope - *Undeclared*
- 12.1: Scope - *CC BY-SA 4.0*

◦ 13: For Loops - *Undeclared*
- 13.1: Iteration Control Structures - *CC BY-SA 4.0*
- 13.2: For Loop - *CC BY-SA 4.0*
- 13.3: Definite loops using for - *CC BY-NC-SA 4.0*
- 13.4: Counting and Summing Loops - *CC BY-NC-SA 4.0*

◦ 14: More Math and Random Numbers - *Undeclared*
- 14.1: Integer Division and Modulus - *CC BY-SA 4.0*
- 14.2: Modulus Operator - *CC BY-NC-SA 4.0*
- 14.3: Standard Libraries - *CC BY-SA 4.0*
- 14.4: Random Numbers - *CC BY-NC-SA 4.0*
- 14.5: Math Functions - *CC BY-NC-SA 4.0*

◦ 15: Nested For Loops - *Undeclared*
- 15.1: Nested For Loops - *CC BY-SA 4.0*

◦ 16: Data Types and User Input - *Undeclared*
- 16.1: Data Types - *CC BY-SA 4.0*
- 16.2: Integer Data Type - *CC BY-SA 4.0*
- 16.3: Floating-Point Data Type - *CC BY-SA 4.0*
- 16.4: String Data Type - *CC BY-SA 4.0*
- 16.5: Boolean Data Type - *CC BY-SA 4.0*
- 16.6: Values and Types - *CC BY-NC-SA 4.0*
- 16.7: Data Type Conversions - *CC BY-SA 4.0*
- 16.8: Input and Output - *CC BY-SA 4.0*
- 16.9: Asking the user for input - *CC BY-NC-SA 4.0*

◦ 17: Strings - *Undeclared*
- 17.1: Strings - *CC BY-SA 4.0*
- 17.2: A string is a sequence - *CC BY-NC-SA 4.0*
- 17.3: String Functions - *CC BY-SA 4.0*
- 17.4: String Operations - *CC BY-NC-SA 4.0*
- 17.5: Getting the length of a string using len - *CC BY-NC-SA 4.0*
- 17.6: Traversal through a string with a loop - *CC BY-NC-SA 4.0*
- 17.7: String Slices - *CC BY-NC-SA 4.0*
- 17.8: Strings are immutable - *CC BY-NC-SA 4.0*
- 17.9: Looping and Counting - *CC BY-NC-SA 4.0*
- 17.10: The in operator - *CC BY-NC-SA 4.0*
- 17.11: String Comparison - *CC BY-NC-SA 4.0*
- 17.12: String Methods - *CC BY-NC-SA 4.0*
- 17.13: Parsing strings - *CC BY-NC-SA 4.0*
- 17.14: String Formatting - *CC BY-SA 4.0*
- 17.15: Format operator - *CC BY-NC-SA 4.0*

◦ 18: While Loops - *Undeclared*
- 18.1: While Loop - *CC BY-SA 4.0*
- 18.2: The while Statement - *CC BY-NC-SA 4.0*
- 18.3: Flag Concept - *CC BY-SA 4.0*
- 18.4: Infinite Loops - *CC BY-NC-SA 4.0*
- 18.5: Loop patterns - *CC BY-NC-SA 4.0*
- 18.6: Maximum and Minimum Loops - *CC BY-NC-SA 4.0*
- 18.7: Loop Examples - *CC BY-SA 4.0*
- 18.8: Python Examples - *CC BY-SA 4.0*

◦ 19: Lists - *Undeclared*
- 19.1: Arrays and Lists - *CC BY-SA 4.0*
- 19.2: Index Notation - *CC BY-SA 4.0*
- 19.3: Displaying Array Members - *CC BY-SA 4.0*
- 19.4: Traversing a List - *CC BY-NC-SA 4.0*
- 19.5: List operations - *CC BY-NC-SA 4.0*
- 19.6: List Slices - *CC BY-NC-SA 4.0*
- 19.7: List Methods - *CC BY-NC-SA 4.0*
- 19.8: Deleting Elements - *CC BY-NC-SA 4.0*
- 19.9: Math Statistics with Arrays - *CC BY-SA 4.0*
- 19.10: Searching Arrays - *CC BY-SA 4.0*
- 19.11: Sorting Arrays - *CC BY-SA 4.0*
- 19.12: Parallel Arrays - *CC BY-SA 4.0*
- 19.13: Multidimensional Arrays - *CC BY-SA 4.0*

◦ 20: Function Return - *Undeclared*
- 20.1: Return Statement - *CC BY-SA 4.0*
- 20.2: Fruitful functions and void functions - *CC BY-NC-SA 4.0*
- 20.3: Python Examples - *CC BY-SA 4.0*

◦ 21: File Input/Output - *Undeclared*
- 21.1: File Input and Output - *CC BY-SA 4.0*
- 21.2: Opening Files - *CC BY-NC-SA 4.0*
- 21.3: Text files and Lines - *CC BY-NC-SA 4.0*
- 21.4: Reading Files - *CC BY-NC-SA 4.0*
- 21.5: Writing Files - *CC BY-NC-SA 4.0*
- 21.6: Using try, except, and open - *CC BY-NC-SA 4.0*

- 21.7: Loading an Array from a Text File - *CC BY-SA 4.0*
- 21.8: Letting the user choose the file name - *CC BY-NC-SA 4.0*
- 21.9: Searching through a File - *CC BY-NC-SA 4.0*
- 21.10: Python Examples - *CC BY-SA 4.0*

◦ 22: Searching and Sorting Algorithms - *Undeclared*
- 22.1: Activity 1 - Searching Algorithm - *CC BY-SA 4.0*
- 22.2: Activity 2 - Sorting Algorithm - *CC BY-SA 4.0*

◦ 23: Object Oriented Programming - *Undeclared*
- 23.1: Managing Larger Programs - *CC BY-NC-SA 4.0*
- 23.2: Getting Started - *CC BY-NC-SA 4.0*
- 23.3: Objects and Classes - *CC BY-SA 4.0*
- 23.4: Using Objects - *CC BY-NC-SA 4.0*
- 23.5: Our First Python Object - *CC BY-NC-SA 4.0*
- 23.6: Python Examples - *CC BY-SA 4.0*

◦ 24: Networked Programs - *CC BY-NC-SA 4.0*
- 24.1: HyperText Transfer Protocol - HTTP - *CC BY-NC-SA 4.0*
- 24.2: The World's Simplest Web Browser - *CC BY-NC-SA 4.0*
- 24.3: Retrieving an image over HTTP - *CC BY-NC-SA 4.0*
- 24.4: Retrieving web pages with urllib - *CC BY-NC-SA 4.0*
- 24.5: Parsing HTML and scraping the web - *CC BY-NC-SA 4.0*
- 24.6: Parsing HTML using regular expressions - *CC BY-NC-SA 4.0*
- 24.7: Parsing HTML using BeautifulSoup - *CC BY-NC-SA 4.0*
- 24.8: Reading binary files using urllib - *CC BY-NC-SA 4.0*
- 24.E: Networked Programs (Exercises) - *CC BY-NC-SA 4.0*
- 24.G: Networked Programs (Glossary) - *CC BY-NC-SA 4.0*

◦ 25: Python and Web Services - *CC BY-NC-SA 4.0*
- 25.1: eXtensible Markup Language - XML - *CC BY-NC-SA 4.0*
- 25.2: Looping through Nodes - *CC BY-NC-SA 4.0*
- 25.3: JavaScript Object Notation - JSON - *CC BY-NC-SA 4.0*
- 25.4: Parsing JSON - *CC BY-NC-SA 4.0*
- 25.5: Application Programming Interfaces - *CC BY-NC-SA 4.0*
- 25.6: Google geocoding web service - *CC BY-NC-SA 4.0*
- 25.7: Security and API usage - *CC BY-NC-SA 4.0*
- 25.E: Python and Web Services (Exercises) - *CC BY-NC-SA 4.0*
- 25.G: Python and Web Services (Glossary) - *CC BY-NC-SA 4.0*

◦ 26: Algorithms - *Undeclared*
- 26.1: Activity 1 - Overview of Algorithm Design and Analysis - *Undeclared*
- 26.2: Activity 1 - Introduction to Algorithms and Problem Solving - *CC BY-SA 4.0*
- 26.3: Activity 2 - The characteristics of an algorithm - *CC BY-SA 4.0*

◦ 27: Additional Fundamentals of Programming - *Undeclared*
- 27.1: Introduction - *CC BY-NC-SA 4.0*
  - 27.1.1: Computer Hardware Architecture - *CC BY-NC-SA 4.0*
  - 27.1.2: Writing a Program - *CC BY-NC-SA 4.0*
  - 27.1.E: Introduction (Exercises) - *CC BY-NC-SA 4.0*
  - 27.1.G: Introduction (Glossary) - *CC BY-NC-SA 4.0*
- 27.2: Introduction to Programming - *CC BY-SA 4.0*
  - 27.2.1: C++ Examples - *CC BY-SA 4.0*
  - 27.2.2: C# Examples - *CC BY-SA 4.0*
  - 27.2.3: Java Examples - *CC BY-SA 4.0*
  - 27.2.4: JavaScript Examples - *CC BY-SA 4.0*
  - 27.2.5: Swift Examples - *CC BY-SA 4.0*
  - 27.2.6: Practice- Introduction to Programming - *CC BY-SA 4.0*
- 27.3: Data and Operators - *CC BY-SA 4.0*
  - 27.3.1: JavaScript Examples - *CC BY-SA 4.0*
  - 27.3.2: Swift Examples - *CC BY-SA 4.0*
  - 27.3.3: Practice- Data and Operators - *CC BY-SA 4.0*
  - 27.3.4: Nothing Data Type - *CC BY-SA 4.0*
  - 27.3.5: Lvalue and Rvalue - *CC BY-SA 4.0*
  - 27.3.6: Input-Process-Output Model - *CC BY-SA 4.0*
  - 27.3.7: C++ Examples - *CC BY-SA 4.0*
  - 27.3.8: C# Examples - *CC BY-SA 4.0*
  - 27.3.9: Java Examples - *CC BY-SA 4.0*
- 27.4: Functions - *CC BY-SA 4.0*
  - 27.4.1: Call by Value vs. Call by Reference - *CC BY-SA 4.0*
  - 27.4.2: Void Data Type - *CC BY-SA 4.0*
  - 27.4.3: C++ Examples - *CC BY-SA 4.0*
  - 27.4.4: C# Examples - *CC BY-SA 4.0*
  - 27.4.5: Java Examples - *CC BY-SA 4.0*
  - 27.4.6: JavaScript Examples - *CC BY-SA 4.0*
  - 27.4.7: Swift Examples - *CC BY-SA 4.0*
  - 27.4.8: Practice- Functions - *CC BY-SA 4.0*
- 27.5: Conditions - *CC BY-SA 4.0*
  - 27.5.1: Case Control Structure - *CC BY-SA 4.0*
  - 27.5.2: C++ Examples - *CC BY-SA 4.0*
  - 27.5.3: C# Examples - *CC BY-SA 4.0*

- 27.5.4: Java Examples - *CC BY-SA 4.0*
- 27.5.5: JavaScript Examples - *CC BY-SA 4.0*
- 27.5.6: Swift Examples - *CC BY-SA 4.0*
- 27.5.7: Practice- Conditions - *CC BY-SA 4.0*
- 27.5.8: Selection Control Structures - *CC BY-SA 4.0*

- 27.6: Loops - *CC BY-SA 4.0*
  - 27.6.1: Do While Loop - *CC BY-SA 4.0*
  - 27.6.2: Branching Statements - *CC BY-SA 4.0*
  - 27.6.3: Integer Overflow - *CC BY-SA 4.0*
  - 27.6.4: C++ Examples - *CC BY-SA 4.0*
  - 27.6.5: C# Examples - *CC BY-SA 4.0*
  - 27.6.6: Java Examples - *CC BY-SA 4.0*
  - 27.6.7: JavaScript Examples - *CC BY-SA 4.0*
  - 27.6.8: Swift Examples - *CC BY-SA 4.0*
  - 27.6.9: Practice- Loops - *CC BY-SA 4.0*
- 27.7: Arrays and Lists - *CC BY-SA 4.0*
  - 27.7.1: Arrays and Functions - *CC BY-SA 4.0*
  - 27.7.2: Dynamic Arrays - *CC BY-SA 4.0*
  - 27.7.3: C++ Examples - *CC BY-SA 4.0*
  - 27.7.4: C# Examples - *CC BY-SA 4.0*
  - 27.7.5: Java Examples - *CC BY-SA 4.0*
  - 27.7.6: JavaScript Examples - *CC BY-SA 4.0*
  - 27.7.7: Python Examples - *CC BY-SA 4.0*
  - 27.7.8: Swift Examples - *CC BY-SA 4.0*
  - 27.7.9: Practice- Arrays and Lists - *CC BY-SA 4.0*
- 27.8: Strings and Files - *CC BY-SA 4.0*
  - 27.8.1: C++ Examples - *CC BY-SA 4.0*
  - 27.8.2: C# Examples - *CC BY-SA 4.0*
  - 27.8.3: Java Examples - *CC BY-SA 4.0*
  - 27.8.4: JavaScript Examples - *CC BY-SA 4.0*
  - 27.8.5: Swift Examples - *CC BY-SA 4.0*
  - 27.8.6: Practice- Strings and Files - *CC BY-SA 4.0*
- 27.9: Object-Oriented Programming - *CC BY-SA 4.0*
  - 27.9.1: Encapsulation - *CC BY-SA 4.0*
  - 27.9.2: Inheritance and Polymorphism - *CC BY-SA 4.0*
  - 27.9.3: C++ Examples - *CC BY-SA 4.0*
  - 27.9.4: C# Examples - *CC BY-SA 4.0*
  - 27.9.5: Java Examples - *CC BY-SA 4.0*
  - 27.9.6: JavaScript Examples - *CC BY-SA 4.0*
  - 27.9.7: Swift Examples - *CC BY-SA 4.0*
  - 27.9.8: Practice - *CC BY-SA 4.0*

- 28: Additional Python 4 Everyone - *Undeclared*
  - 28.1: A trip to Jupyter - *CC BY-SA 4.0*
    - 28.1.1: Jupyter Notebooks - *CC BY-SA 4.0*
    - 28.1.2: Code and Output - *CC BY-SA 4.0*
  - 28.2: Variables, Expressions, and Statements - *CC BY-NC-SA 4.0*
    - 28.2.1: Variable names and Keywords - *CC BY-NC-SA 4.0*
    - 28.2.2: Operators and Operands - *CC BY-NC-SA 4.0*
    - 28.2.3: Expressions - *CC BY-NC-SA 4.0*
    - 28.2.4: Order of Operations - *CC BY-NC-SA 4.0*
    - 28.2.5: Choosing Mnemonic Variable Names - *CC BY-NC-SA 4.0*
    - 28.2.6: Debugging - *CC BY-NC-SA 4.0*
    - 28.2.E: Variables, Expressions, and Statements (Exercises) - *CC BY-NC-SA 4.0*
    - 28.2.G: Glossary - *CC BY-NC-SA 4.0*
  - 28.3: Conditional Execution - *CC BY-NC-SA 4.0*
    - 28.3.1: Short-Circuit Evaluation of Logical Expressions - *CC BY-NC-SA 4.0*
    - 28.3.2: Debugging - *CC BY-NC-SA 4.0*
    - 28.3.E: Conditional Execution (Exercises) - *CC BY-NC-SA 4.0*
    - 28.3.G: Conditional Execution (Glossary) - *CC BY-NC-SA 4.0*
  - 28.4: Functions - *CC BY-NC-SA 4.0*
    - 28.4.1: Function Calls - *CC BY-NC-SA 4.0*
    - 28.4.2: Built-in Functions - *CC BY-NC-SA 4.0*
    - 28.4.3: Type Conversion Functions - *CC BY-NC-SA 4.0*
    - 28.4.4: Parameters and Arguments - *CC BY-NC-SA 4.0*
    - 28.4.5: Debugging - *CC BY-NC-SA 4.0*
    - 28.4.E: Functions (Exercises) - *CC BY-NC-SA 4.0*
    - 28.4.G: Functions (Glossary) - *CC BY-NC-SA 4.0*
  - 28.5: Iterations - *CC BY-NC-SA 4.0*
    - 28.5.1: "Infinite loops" and break - *CC BY-NC-SA 4.0*
    - 28.5.2: Finishing iterations with continue - *CC BY-NC-SA 4.0*
    - 28.5.3: Debugging - *CC BY-NC-SA 4.0*
    - 28.5.E: Iterations (Exercises) - *CC BY-NC-SA 4.0*
    - 28.5.G: Iterations (Glossary) - *CC BY-NC-SA 4.0*
  - 28.6: Strings - *CC BY-NC-SA 4.0*
    - 28.6.1: Debugging - *CC BY-NC-SA 4.0*
    - 28.6.E: Strings (Exercises) - *CC BY-NC-SA 4.0*
    - 28.6.G: Strings (Glossary) - *CC BY-NC-SA 4.0*
  - 28.7: Files - *CC BY-NC-SA 4.0*
    - 28.7.1: Persistence - *CC BY-NC-SA 4.0*
    - 28.7.2: Debugging - *CC BY-NC-SA 4.0*
    - 28.7.E: Files (Exercises) - *CC BY-NC-SA 4.0*
    - 28.7.G: Files (Glossary) - *CC BY-NC-SA 4.0*
  - 28.8: Lists - *CC BY-NC-SA 4.0*
    - 28.8.1: Lists and Functions - *CC BY-NC-SA 4.0*

        - 28.8.2: Lists and Strings - *CC BY-NC-SA 4.0*
        - 28.8.3: Parsing lines - *CC BY-NC-SA 4.0*
        - 28.8.4: Objects and Values - *CC BY-NC-SA 4.0*
        - 28.8.5: Aliasing - *CC BY-NC-SA 4.0*
        - 28.8.6: List arguments - *CC BY-NC-SA 4.0*
        - 28.8.7: Debugging - *CC BY-NC-SA 4.0*
        - 28.8.E: Lists (Exercises) - *CC BY-NC-SA 4.0*
        - 28.8.G: Lists (Glossary) - *CC BY-NC-SA 4.0*
    - 28.9: Object-Oriented Programming - *CC BY-NC-SA 4.0*
        - 28.9.1: Starting with Programs - *CC BY-NC-SA 4.0*
        - 28.9.2: Subdividing a Problem - Encapsulation - *CC BY-NC-SA 4.0*
        - 28.9.3: Classes as Types - *CC BY-NC-SA 4.0*
        - 28.9.4: Many Instances - *CC BY-NC-SA 4.0*
        - 28.9.5: Object Lifecycle - *CC BY-NC-SA 4.0*
        - 28.9.6: Inheritance - *CC BY-NC-SA 4.0*
        - 28.9.G: Object-Oriented Programming (Glossary) - *CC BY-NC-SA 4.0*
        - 28.9.S: Object-Oriented Programming (Summary) - *CC BY-NC-SA 4.0*
- Back Matter - *Undeclared*
    - Index - *Undeclared*
    - Index - *Undeclared*
    - Glossary - *Undeclared*
    - Detailed Licensing - *Undeclared*
    - Detailed Licensing - *Undeclared*
    - Detailed Licensing - *Undeclared*

This page titled Intro to Programming with Programming Fundamentals and Python for Everyone is shared under a license and was authored, remixed, and/or curated by LibreTexts.

DETAILED LICENSING

Detailed licensing information for this resource is not available at this time.

This page titled Intro to Programming with Programming Fundamentals and Python for Everyone is shared under a license and was authored, remixed, and/or curated by LibreTexts.