

TEXAS A&M



Texas A&M

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 born 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 was 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 adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 05/21/2026

TABLE OF CONTENTS

Licensing

ATMO 321: Python for Atmospheric Sciences

- Licensing
- 1: Introduction
 - 1.1: Hello world!
 - 1.2: What is programming?
 - 1.3: Pseudocode
 - 1.4: Pseudocode, part 2
- 2: Variables, Expressions, and Statements
 - 2.1: Values and Types
 - 2.2: Variables
 - 2.3: Variable names and Keywords
 - 2.4: Statements
 - 2.5: Operators and Operands
 - 2.6: Expressions
 - 2.7: Order of Operations
 - 2.8: String Operations
 - 2.9: Asking the user for input
 - 2.10: Comments
 - 2.11: Choosing Mnemonic Variable Names
 - 2.12: Debugging
 - 2.E: Variables, Expressions, and Statements (Exercises)
 - 2.G: Glossary
- 3: Conditional Execution
 - 3.1: Boolean Expressions
 - 3.2: Logical Operators
 - 3.3: Conditional Execution
 - 3.4: Chained Conditionals
 - 3.5: Nested Conditionals
 - 3.6: Catching exceptions Using Try and Except
 - 3.7: Short-Circuit Evaluation of Logical Expressions
 - 3.8: Debugging
 - 3.E: Conditional Execution (Exercises)
 - 3.G: Conditional Execution (Glossary)
- 4: Functions
 - 4.1: Function Calls
 - 4.2: Built-in Functions
 - 4.3: Type Conversion Functions
 - 4.4: Random Numbers
 - 4.5: Math Functions
 - 4.6: Adding New Functions
 - 4.7: Definitions and Uses
 - 4.8: Flow of Execution

- 4.9: Parameters and Arguments
- 4.10: Fruitful functions and void functions
- 4.11: Scope
- 4.12: Why functions?
- 4.13: Debugging
- 4.E: Functions (Exercises)
- 4.G: Functions (Glossary)
- 5: Iterations
 - 5.1: Updating Variables
 - 5.2: The while Statement
 - 5.3: Infinite Loops
 - 5.4: "Infinite loops" and break
 - 5.5: continue, break, enumerate, zip
 - 5.6: Finishing iterations with continue
 - 5.7: Definite loops using for
 - 5.8: Loop patterns
 - 5.9: Counting and Summing Loops
 - 5.10: Maximum and Minimum Loops
 - 5.11: Debugging
 - 5.E: Iterations (Exercises)
 - 5.G: Iterations (Glossary)
- 6: Strings
 - 6.1: A string is a sequence
 - 6.2: Getting the length of a string using len
 - 6.3: Traversal through a string with a loop
 - 6.4: String Slices
 - 6.5: Looping and Counting
 - 6.6: The in operator
 - 6.7: String Comparison
 - 6.8: String Methods
 - 6.9: Parsing strings
 - 6.10: String formatting
 - 6.E: Strings (Exercises)
 - 6.G: Strings (Glossary)
- 7: Lists
 - 7.1: A list is a sequence
 - 7.2: Lists are mutable
 - 7.3: Traversing a List
 - 7.4: List operations
 - 7.5: List Slices
 - 7.6: List Methods
 - 7.7: Deleting Elements
 - 7.8: Lists and Functions
 - 7.9: Lists and Strings
 - 7.10: Parsing lines
 - 7.11: Objects and Values
 - 7.12: Aliasing
 - 7.13: List arguments
 - 7.14: List comprehensions

- 7.15: Debugging
- 7.E: Lists (Exercises)
- 7.G: Lists (Glossary)
- 8: Sets
 - 8.1: Introduction to sets
 - 8.2: set operations
 - 8.3: set order
- 9: Dictionaries
 - 9.1: Dictionaries
 - 9.2: Dictionary as a Set of Counters
 - 9.3: Dictionaries and Files
 - 9.4: Looping and Dictionaries
 - 9.5: Advanced Text Parsing
 - 9.6: Debugging
 - 9.E: Dictionaries (Exercises)
 - 9.G: Dictionaries (Glossary)
- 10: Packages
 - 10.1: Importing packages in python
 - 10.2: How to use functions in packages
 - 10.3: Aliasing packages
 - 10.4: Sub packages
- 11: Numpy, part 1
 - 11.1: Packages
 - 11.2: Mathematical Functions
 - 11.3: The NumPy 'ndarray'
- 12: Numpy (2 of 2)
 - 12.1: Accessing Individual Elements
 - 12.2: "Vectorized" Arithmetic Operators
 - 12.3: Broadcasting
 - 12.4: Copying – and Not Copying – Arrays
 - 12.5: Sorting Arrays
 - 12.6: More Exotic Array Modifications
 - 12.7: Summary
- 13: Matplotlib
 - 13.1: Loading matplotlib
 - 13.2: Matplotlib basics
 - 13.3: Matplotlib basics, 2
 - 13.4: Matplotlib tutorial
- 14: Pandas Series
 - 14.1: Introduction
 - 14.2: The Pandas Series
 - 14.3: More about Pandas series
 - 14.4: Vectorized Arithmetic Operators
 - 14.5: Sorting Serieses
 - 14.6: Queries
 - 14.7: Summary
- 15: Pandas Dataframes

- [15.1: Dataframe basics](#)
- [15.2: Accessing elements](#)
- [15.3: Groupby\(\)](#)
- [15.4: Math operations](#)
- [16: Date and time variables](#)
 - [16.1: np.datetime64](#)
 - [16.2: Date and time in Pandas](#)
 - [16.3: Extracting components of date and time](#)
 - [16.4: Setting the index to be date and time](#)
 - [16.5: Resampling](#)
 - [16.6: Other resources](#)
- [17: Xarray](#)
 - [17.1: Introduction](#)
 - [17.2: DataArrays](#)
 - [17.3: Selecting values](#)
 - [17.4: Datasets](#)
 - [17.5: Reading netCDF files](#)
 - [17.6: Groupby and Resample](#)
 - [17.7: Additional resources](#)
- [18: Regression](#)
 - [18.1: Introduction](#)
 - [18.2: scipy.stats.linregress\(\)](#)
 - [18.3: sklearn](#)
 - [18.4: multiple linear regression](#)
 - [18.5: More resources](#)
- [19: Cartopy](#)
 - [19.1: Abernathy cartopy chapter](#)
- [Index](#)
- [Glossary](#)
- [Detailed Licensing](#)
- [Detailed Licensing](#)

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

TABLE OF CONTENTS

ATMO 321: Python for Atmospheric Sciences

Licensing

1: Introduction

1.1: Hello world!

1.2: What is programming?

1.3: Pseudocode

1.4: Pseudocode, part 2

2: Variables, Expressions, and Statements

2.1: Values and Types

2.2: Variables

2.3: Variable names and Keywords

2.4: Statements

2.5: Operators and Operands

2.6: Expressions

2.7: Order of Operations

2.8: String Operations

2.9: Asking the user for input

2.10: Comments

2.11: Choosing Mnemonic Variable Names

2.12: Debugging

2.E: Variables, Expressions, and Statements (Exercises)

2.G: Glossary

3: Conditional Execution

3.1: Boolean Expressions

3.2: Logical Operators

3.3: Conditional Execution

3.4: Chained Conditionals

3.5: Nested Conditionals

3.6: Catching exceptions Using Try and Except

3.7: Short-Circuit Evaluation of Logical Expressions

3.8: Debugging

3.E: Conditional Execution (Exercises)

3.G: Conditional Execution (Glossary)

4: Functions

4.1: Function Calls

4.2: Built-in Functions

4.3: Type Conversion Functions

4.4: Random Numbers

4.5: Math Functions

4.6: Adding New Functions

- 4.7: Definitions and Uses
- 4.8: Flow of Execution
- 4.9: Parameters and Arguments
- 4.10: Fruitful functions and void functions
- 4.11: Scope
- 4.12: Why functions?
- 4.13: Debugging
- 4.E: Functions (Exercises)
- 4.G: Functions (Glossary)

5: Iterations

- 5.1: Updating Variables
- 5.2: The while Statement
- 5.3: Infinite Loops
- 5.4: "Infinite loops" and break
- 5.5: continue, break, enumerate, zip
- 5.6: Finishing iterations with continue
- 5.7: Definite loops using for
- 5.8: Loop patterns
- 5.9: Counting and Summing Loops
- 5.10: Maximum and Minimum Loops
- 5.11: Debugging
- 5.E: Iterations (Exercises)
- 5.G: Iterations (Glossary)

6: Strings

- 6.1: A string is a sequence
- 6.2: Getting the length of a string using len
- 6.3: Traversal through a string with a loop
- 6.4: String Slices
- 6.5: Looping and Counting
- 6.6: The in operator
- 6.7: String Comparison
- 6.8: String Methods
- 6.9: Parsing strings
- 6.10: String formatting
- 6.E: Strings (Exercises)
- 6.G: Strings (Glossary)

7: Lists

- 7.1: A list is a sequence
- 7.2: Lists are mutable
- 7.3: Traversing a List
- 7.4: List operations
- 7.5: List Slices
- 7.6: List Methods
- 7.7: Deleting Elements

7.8: Lists and Functions

7.9: Lists and Strings

7.10: Parsing lines

7.11: Objects and Values

7.12: Aliasing

7.13: List arguments

7.14: List comprehensions

7.15: Debugging

7.E: Lists (Exercises)

7.G: Lists (Glossary)

8: Sets

8.1: Introduction to sets

8.2: set operations

8.3: set order

9: Dictionaries

9.1: Dictionaries

9.2: Dictionary as a Set of Counters

9.3: Dictionaries and Files

9.4: Looping and Dictionaries

9.5: Advanced Text Parsing

9.6: Debugging

9.E: Dictionaries (Exercises)

9.G: Dictionaries (Glossary)

10: Packages

10.1: Importing packages in python

10.2: How to use functions in packages

10.3: Aliasing packages

10.4: Sub packages

11: Numpy, part 1

11.1: Packages

11.2: Mathematical Functions

11.3: The NumPy 'ndarray'

12: Numpy (2 of 2)

12.1: Accessing Individual Elements

12.2: “Vectorized” Arithmetic Operators

12.3: Broadcasting

12.4: Copying – and Not Copying – Arrays

12.5: Sorting Arrays

12.6: More Exotic Array Modifications

12.7: Summary

13: Matplotlib

13.1: Loading matplotlib

13.2: Matplotlib basics

[13.3: Matplotlib basics, 2](#)

[13.4: Matplotlib tutorial](#)

[14: Pandas Series](#)

[14.1: Introduction](#)

[14.2: The Pandas Series](#)

[14.3: More about Pandas series](#)

[14.4: Vectorized Arithmetic Operators](#)

[14.5: Sorting Serieses](#)

[14.6: Queries](#)

[14.7: Summary](#)

[15: Pandas Dataframes](#)

[15.1: Dataframe basics](#)

[15.2: Accessing elements](#)

[15.3: Groupby\(\)](#)

[15.4: Math operations](#)

[16: Date and time variables](#)

[16.1: np.datetime64](#)

[16.2: Date and time in Pandas](#)

[16.3: Extracting components of date and time](#)

[16.4: Setting the index to be date and time](#)

[16.5: Resampling](#)

[16.6: Other resources](#)

[17: Xarray](#)

[17.1: Introduction](#)

[17.2: DataArrays](#)

[17.3: Selecting values](#)

[17.4: Datasets](#)

[17.5: Reading netCDF files](#)

[17.6: Groupby and Resample](#)

[17.7: Additional resources](#)

[18: Regression](#)

[18.1: Introduction](#)

[18.2: scipy.stats.linregress\(\)](#)

[18.3: sklearn](#)

[18.4: multiple linear regression](#)

[18.5: More resources](#)

[19: Cartopy](#)

[19.1: Abernathy cartopy chapter](#)

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

[Detailed Licensing](#)

Texas A&M is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

Front Matter

[TitlePage](#)

[InfoPage](#)

[Table of Contents](#)

[Licensing](#)

Texas A&M

ATMO 321 python for atmospheric sciences

Andrew E. Dessler

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 born 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 was 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 adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 05/21/2026

TABLE OF CONTENTS

Licensing

1: Introduction

- 1.1: Hello world!
- 1.2: What is programming?
- 1.3: Pseudocode
- 1.4: Pseudocode, part 2

2: Variables, Expressions, and Statements

- 2.1: Values and Types
- 2.2: Variables
- 2.3: Variable names and Keywords
- 2.4: Statements
- 2.5: Operators and Operands
- 2.6: Expressions
- 2.7: Order of Operations
- 2.8: String Operations
- 2.9: Asking the user for input
- 2.10: Comments
- 2.11: Choosing Mnemonic Variable Names
- 2.12: Debugging
- 2.E: Variables, Expressions, and Statements (Exercises)
- 2.G: Glossary

3: Conditional Execution

- 3.1: Boolean Expressions
- 3.2: Logical Operators
- 3.3: Conditional Execution
- 3.4: Chained Conditionals
- 3.5: Nested Conditionals
- 3.6: Catching exceptions Using Try and Except
- 3.7: Short-Circuit Evaluation of Logical Expressions
- 3.8: Debugging
- 3.E: Conditional Execution (Exercises)
- 3.G: Conditional Execution (Glossary)

4: Functions

- 4.1: Function Calls
- 4.2: Built-in Functions
- 4.3: Type Conversion Functions
- 4.4: Random Numbers
- 4.5: Math Functions
- 4.6: Adding New Functions
- 4.7: Definitions and Uses

- 4.8: Flow of Execution
- 4.9: Parameters and Arguments
- 4.10: Fruitful functions and void functions
- 4.11: Scope
- 4.12: Why functions?
- 4.13: Debugging
- 4.E: Functions (Exercises)
- 4.G: Functions (Glossary)

5: Iterations

- 5.1: Updating Variables
- 5.2: The while Statement
- 5.3: Infinite Loops
- 5.4: "Infinite loops" and break
- 5.5: continue, break, enumerate, zip
- 5.6: Finishing iterations with continue
- 5.7: Definite loops using for
- 5.8: Loop patterns
- 5.9: Counting and Summing Loops
- 5.10: Maximum and Minimum Loops
- 5.11: Debugging
- 5.E: Iterations (Exercises)
- 5.G: Iterations (Glossary)

6: Strings

- 6.1: A string is a sequence
- 6.2: Getting the length of a string using len
- 6.3: Traversal through a string with a loop
- 6.4: String Slices
- 6.5: Looping and Counting
- 6.6: The in operator
- 6.7: String Comparison
- 6.8: String Methods
- 6.9: Parsing strings
- 6.10: String formatting
- 6.E: Strings (Exercises)
- 6.G: Strings (Glossary)

7: Lists

- 7.1: A list is a sequence
- 7.2: Lists are mutable
- 7.3: Traversing a List
- 7.4: List operations
- 7.5: List Slices
- 7.6: List Methods
- 7.7: Deleting Elements
- 7.8: Lists and Functions
- 7.9: Lists and Strings

- 7.10: Parsing lines
- 7.11: Objects and Values
- 7.12: Aliasing
- 7.13: List arguments
- 7.14: List comprehensions
- 7.15: Debugging
- 7.E: Lists (Exercises)
- 7.G: Lists (Glossary)

8: Sets

- 8.1: Introduction to sets
- 8.2: set operations
- 8.3: set order

9: Dictionaries

- 9.1: Dictionaries
- 9.2: Dictionary as a Set of Counters
- 9.3: Dictionaries and Files
- 9.4: Looping and Dictionaries
- 9.5: Advanced Text Parsing
- 9.6: Debugging
- 9.E: Dictionaries (Exercises)
- 9.G: Dictionaries (Glossary)

10: Packages

- 10.1: Importing packages in python
- 10.2: How to use functions in packages
- 10.3: Aliasing packages
- 10.4: Sub packages

11: Numpy, part 1

- 11.1: Packages
- 11.2: Mathematical Functions
- 11.3: The NumPy 'ndarray'

12: Numpy (2 of 2)

- 12.1: Accessing Individual Elements
- 12.2: "Vectorized" Arithmetic Operators
- 12.3: Broadcasting
- 12.4: Copying – and Not Copying – Arrays
- 12.5: Sorting Arrays
- 12.6: More Exotic Array Modifications
- 12.7: Summary

13: Matplotlib

- 13.1: Loading matplotlib
- 13.2: Matplotlib basics

- 13.3: Matplotlib basics, 2
- 13.4: Matplotlib tutorial

14: Pandas Series

- 14.1: Introduction
- 14.2: The Pandas Series
- 14.3: More about Pandas series
- 14.4: Vectorized Arithmetic Operators
- 14.5: Sorting Serieses
- 14.6: Queries
- 14.7: Summary

15: Pandas Dataframes

- 15.1: Dataframe basics
- 15.2: Accessing elements
- 15.3: Groupby()
- 15.4: Math operations

16: Date and time variables

- 16.1: np.datetime64
- 16.2: Date and time in Pandas
- 16.3: Extracting components of date and time
- 16.4: Setting the index to be date and time
- 16.5: Resampling
- 16.6: Other resources

17: Xarray

- 17.1: Introduction
- 17.2: DataArrays
- 17.3: Selecting values
- 17.4: Datasets
- 17.5: Reading netCDF files
- 17.6: Groupby and Resample
- 17.7: Additional resources

18: Regression

- 18.1: Introduction
- 18.2: scipy.stats.linregress()
- 18.3: sklearn
- 18.4: multiple linear regression
- 18.5: More resources

19: Cartopy

- 19.1: Abernathy cartopy chapter

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

[Detailed Licensing](#)

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

CHAPTER OVERVIEW

1: Introduction

- 1.1: Hello world!
- 1.2: What is programming?
- 1.3: Pseudocode
- 1.4: Pseudocode, part 2

1: [Introduction](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

1.1: Hello world!

Welcome to "Python for Atmospheric Sciences," where you will embark on a journey to unravel the mysteries of our atmosphere through the power of Python programming. The atmospheric sciences encompass a wide range of topics, from weather forecasting and climate modeling to air pollution studies and beyond. These fields are data-driven and computationally intensive, demanding a powerful tool to analyze complex datasets and generate meaningful insights.

Enter Python, one of the most popular programming languages today, particularly within the scientific community. Python's simplicity, flexibility, and robust collection of scientific libraries—such as NumPy for numerical computations, Matplotlib for data visualization, and Pandas for data analysis—make it an ideal language for atmospheric scientists.

With Python, you can perform tasks as simple as calculating the mean temperature of a dataset or as complex as simulating the dynamics of our atmosphere using physical equations. It enables you to manipulate large climate datasets, visualize atmospheric processes, and even create your own climate models.

Learning Python not only equips you with a valuable skill in the scientific field, but it also fosters problem-solving and computational thinking—skills that are vital for today's atmospheric scientists. Moreover, the Python community is enormous and ever-growing, which means extensive support, readily available libraries, and constant updates.

Through this book, you'll learn not only Python but also how to apply it to real-world atmospheric problems. By the end of this book, you should be able to read in data, manipulate it, and plot it up. Are you ready to harness the power of Python for atmospheric sciences? Let's dive in!

1.1: Hello world! is shared under a [not declared](#) license and was authored, remixed, and/or curated by Andy Dessler.

1.2: What is programming?

Programming, at its core, is the act of instructing a computer to perform specific tasks or solve problems. Despite the astonishing computational prowess of modern computers, they are fundamentally rule-followers. They rely on precise, step-by-step instructions provided by humans—programmers—to carry out their tasks.

This process is analogous to cooking using a recipe. To make a dish, you don't just throw all the ingredients into a pot at once. Instead, you follow a sequence of steps, each with a specific order and set of actions—like chopping the vegetables, sautéing them, adding spices, and so on.

In programming, the problem you want to solve is the dish you want to cook, and your program is the recipe. Just as a recipe breaks down the complex process of cooking a dish into a series of manageable steps, programming involves breaking down a problem into smaller, logical steps that a computer can understand and execute.

For example, if you're programming a weather forecasting model, you don't tell the computer to "predict tomorrow's weather." Instead, you might instruct it to first gather data from various weather sensors, then clean and organize that data, perform specific calculations using that data (like calculating pressure gradients, humidity levels, temperature changes), apply certain algorithms or models, and finally output a prediction for the next day's weather. Each of these steps involves its own series of sub-steps, which are coded into the program.

Therefore, the art of programming isn't just about writing code—it's about logical thinking, problem-solving, and breaking complex problems into smaller, manageable tasks that a computer can execute. It's a vital skill in many fields, including atmospheric sciences, where complex data analysis and computational modeling are key to understanding and predicting weather and climate patterns.

As an example, imagine that you're calculating the number of heads you get when flipping a coin 100 times. Here's how we can break this task down into manageable programming steps:

1. **Define the Process:** A single coin flip can result in either heads or tails. We'll need to define this process in our program, which usually involves generating a random number that represents either heads or tails. In Python, we can use the ``random`` module for this.
2. **Simulate the Flips:** Once we have the coin flip defined, we need to repeat this process 100 times. This is typically done with a loop that runs the coin flip process as many times as we want.
3. **Count the Heads:** Each time we flip the coin, we need to check if the result was heads and, if so, increment a count of heads. This typically involves an ``if`` statement to check the result of each flip, and a variable to store the count.
4. **Output the Result:** After the loop has run 100 times, we should have the total number of heads. The final step is to output this result, which might be as simple as printing the value to the console.

In summary, the problem can be broken down into defining the random process (a coin flip), simulating this process a given number of times (100 coin flips), counting the desired outcomes (number of heads), and outputting the result. Each of these steps can be translated into specific programming commands or code blocks.

At its core, being a proficient programmer requires the ability to take complex problems and decompose them into simpler, more manageable steps. This skill, often called computational thinking, is the backbone of efficient problem solving in the realm of programming. It involves taking the full scope of a problem, understanding its intricacies, and methodically breaking it down into sub-problems that are easier to handle. This step-by-step process allows you to tackle each part individually, and when all the parts are completed, the overall problem is solved. Therefore, honing your ability to segment problems is fundamental to successful programming.

1.2: What is programming? is shared under a [not declared](#) license and was authored, remixed, and/or curated by Andy Dessler.

SECTION OVERVIEW

1.3: Pseudocode

As a first step to learning programming, before you start writing code, you need to be able to write *pseudocode*. Pseudocode serves as an intermediary step between the conception of a computer program and its actual implementation in a specific programming language. It is a high-level, human-readable representation of the algorithmic steps required to solve a problem or perform a task. Unlike actual code, pseudocode is halfway between plain English and the strict syntax of programming languages. This allows for a more intuitive and flexible expression of ideas.

In essence, pseudocode serves as a bridge between the conceptualization of a solution and the detailed coding required for execution. It helps programmers outline the logic and structure of their programs without being constrained by the precise rules of a programming language. By using plain language constructs and informal notation, pseudocode enables a clear and concise expression of algorithmic thinking.

As an example, let's consider a simple pseudocode example making a peanut butter and jelly sandwich.

BEGIN

```
Open the bread package
Take out two slices of bread
Place the slices side by side on a clean surface

Open the peanut butter jar
Take a knife
Use the knife to scoop out some peanut butter
Spread the peanut butter evenly on one slice of bread
Clean the knife (if necessary)
```

```
Open the jelly jar
Use the knife to scoop out some jelly
Spread the jelly evenly on the other slice of bread
```

```
Place the slice with peanut butter on top of the slice with jelly, peanut butter
side facing down
```

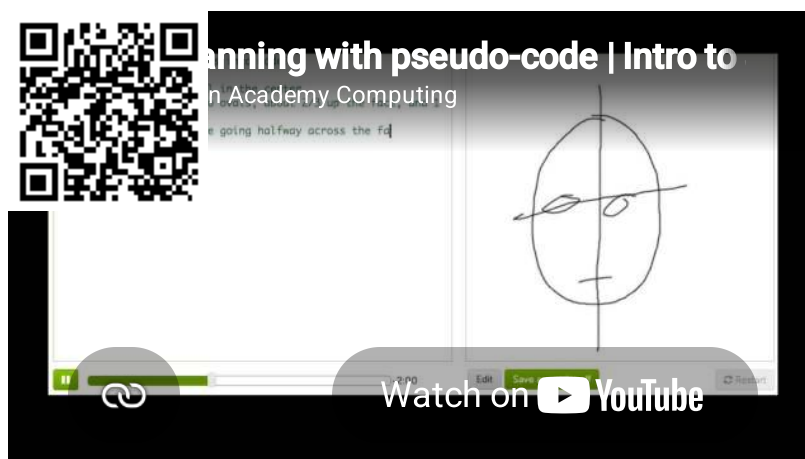
```
Press lightly to make sure the slices stick together
```

```
Cut the sandwich diagonally (optional)
```

```
Serve the sandwich on a plate
```

END

Here's a video that goes over this:



1.3: Pseudocode is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

1.4: Pseudocode, part 2

As an example, let's consider a simple pseudocode example for finding the maximum element in an array. In this algorithm, we iterate through each element of the array, keeping track of the maximum value encountered so far. Here's a straightforward representation in pseudocode:

Input: Array of numbers A

1. Set `max_value` to the first element of the array A (`A[0]`)
2. For each element `num` in A, starting from the second element (`A[1]`):
 - a. If `num` is greater than `max_value`, update `max_value` to `num`
3. Output `max_value` as the maximum element in the array

This pseudocode outlines a step-by-step approach to finding the maximum element in an array. It initializes a variable `max_value` with the first element of the array and then iterates through the remaining elements, updating `max_value` whenever a larger element is encountered. Finally, the algorithm outputs the maximum value found. This pseudocode provides a clear and concise representation of the logic without getting into the specific syntax of any programming language.

When creating pseudocode, one focuses on the logical flow of operations, defining steps in a manner that is independent of the syntax of a particular programming language. This abstraction allows developers to plan and communicate algorithmic solutions more effectively before committing to the intricacies of coding.

The bottom line is this: *if you can't write pseudocode, you can't write the actual code*. If you're having trouble coding up a problem, you should first try to write it in pseudocode before you try to write actual code. Once you have this pseudocode, writing actual code is (relatively) easy.

1.4: Pseudocode, part 2 is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

2: Variables, Expressions, and Statements

- [2.1: Values and Types](#)
- [2.2: Variables](#)
- [2.3: Variable names and Keywords](#)
- [2.4: Statements](#)
- [2.5: Operators and Operands](#)
- [2.6: Expressions](#)
- [2.7: Order of Operations](#)
- [2.8: String Operations](#)
- [2.9: Asking the user for input](#)
- [2.10: Comments](#)
- [2.11: Choosing Mnemonic Variable Names](#)
- [2.12: Debugging](#)
- [2.E: Variables, Expressions, and Statements \(Exercises\)](#)
- [2.G: Glossary](#)

This page titled [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](#).

2.1: 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 [2.1: Values and Types](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [2.1: Values and Types](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.2: 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 [2.2: Variables](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [2.2: Variables](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.3: 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).

Snake case is a naming convention used in computer programming and writing, particularly noticeable for its readability. In snake case, words are separated by underscores and are typically written in all lowercase letters. This style is called "snake case" because the underscores resemble a snake's undulating shape.

For example, a variable name like "user age" would be written as `user_age` in snake case. This convention is commonly used in certain programming languages and scenarios, such as variable names in Python, to create clear, readable identifiers that can be easily understood by others reading the code.

An alternative to snake case is camel case. Camel case involves writing phrases without spaces or punctuation, with each word's initial letter capitalized, except for the very first letter of the sequence. The name "camel case" comes from the visual resemblance of the capital letters to a camel's hump. In this style, a phrase like "user name" would be written as the variable `userName`. This style helps in creating concise, yet descriptive, names that enhance the readability and clarity of the code.

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 [2.3: 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](#).

- [2.3: Variable names and Keywords](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.4: 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.

This page titled [2.4: Statements](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [2.4: Statements](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.5: 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 [2.5: Operators and Operands](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [2.5: Operators and Operands](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.6: 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 2.6: Expressions is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- 2.6: Expressions by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.7: 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:

- Parentheses 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 $(\text{minute} * 100) / 60$, even if it doesn't change the result.
- Exponentiation has the next highest precedence, so $2 ** 1 + 1$ is 3, not 4, and $3 * 1 ** 3$ is 3, not 27.
- Multiplication and Division have the same precedence, which is higher than Addition 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 [2.7: Order of Operations](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **2.7: Order of Operations** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.8: 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:

The output of this program is `100150` .

This page titled [2.8: String Operations](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [2.9: String Operations](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.9: 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 keyboard¹. 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:

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:

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 [2.9: 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](#).

- [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>.

2.10: 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:

```
# compute the percentage of the hour that has elapsed
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:

```
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:

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

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

```
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 [2.10: Comments](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [2.11: Comments](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.11: 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 [2.11: 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](#).

- [2.12: Choosing Mnemonic Variable Names](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

2.12: 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 [2.12: Debugging](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [2.13: Debugging](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

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 [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](#).

- [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>.

2.G: Glossary

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 [2.G: Glossary](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

CHAPTER OVERVIEW

3: Conditional Execution

- [3.1: Boolean Expressions](#)
- [3.2: Logical Operators](#)
- [3.3: Conditional Execution](#)
- [3.4: Chained Conditionals](#)
- [3.5: Nested Conditionals](#)
- [3.6: Catching exceptions Using Try and Except](#)
- [3.7: Short-Circuit Evaluation of Logical Expressions](#)
- [3.8: Debugging](#)
- [3.E: Conditional Execution \(Exercises\)](#)
- [3.G: Conditional Execution \(Glossary\)](#)

This page titled [3: Conditional Execution](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

3.1: 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 [3.1: Boolean Expressions](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **3.1: Boolean Expressions** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

3.2: 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.

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."

```
>>> 17 and True
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 [3.2: Logical Operators](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

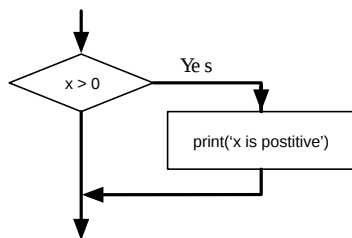
- [3.2: Logical Operators](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

3.3: 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:

```
if x > 0 :  
    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` loops¹. 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.

```
if x < 0 :  
    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:

```
>>> x = 3  
>>> if x < 10:  
...     print('Small')  
...  
Small  
>>>
```

This page titled [3.3: Conditional Execution](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

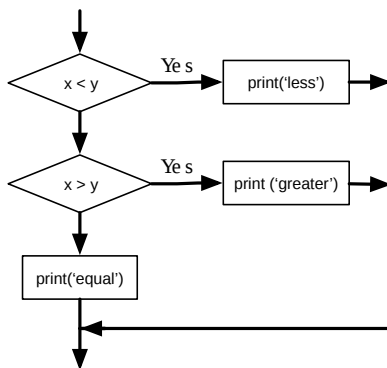
- [3.3: Conditional Execution](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

3.4: 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*:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    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.

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 [3.4: Chained Conditionals](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

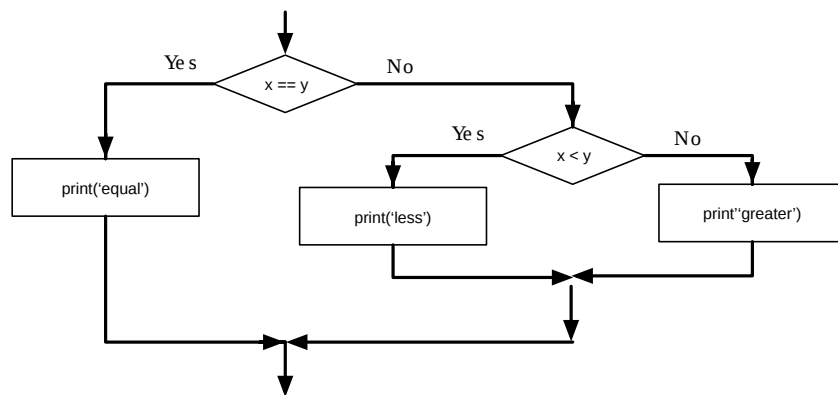
- [3.5: Chained Conditionals](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

3.5: 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 [3.5: Nested Conditionals](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [3.6: Nested Conditionals](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

3.6: 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:

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:

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 fahrenheit.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 [3.6: 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](#).

- [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>.

3.7: 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 $y \neq 0$ is *after* the (x/y) calculation so the expression fails with an error.

In the second expression, we say that $y \neq 0$ acts as a *guard* to insure that we only execute (x/y) if y is non-zero.

This page titled [3.7: 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](#).

- [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>.

3.8: 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 [3.8: Debugging](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [3.9: Debugging](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

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 [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](#).

- [3.E: Conditional Execution \(Exercises\)](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

3.G: Conditional Execution (Glossary)

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 [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](#).

CHAPTER OVERVIEW

4: Functions

- [4.1: Function Calls](#)
- [4.2: Built-in Functions](#)
- [4.3: Type Conversion Functions](#)
- [4.4: Random Numbers](#)
- [4.5: Math Functions](#)
- [4.6: Adding New Functions](#)
- [4.7: Definitions and Uses](#)
- [4.8: Flow of Execution](#)
- [4.9: Parameters and Arguments](#)
- [4.10: Fruitful functions and void functions](#)
- [4.11: Scope](#)
- [4.12: Why functions?](#)
- [4.13: Debugging](#)
- [4.E: Functions \(Exercises\)](#)
- [4.G: Functions \(Glossary\)](#)

This page titled [4: Functions](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

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 [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](#).

- [4.1: Function Calls](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

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 [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](#).

- [4.2: Built-in Functions](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

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 [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](#).

- [4.3: Type Conversion Functions](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.4: 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:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
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.

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).

```
>>> random.randint(5, 10)
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 [4.4: Random Numbers](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [4.4: Random Numbers](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.5: 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 [4.5: Math Functions](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [4.5: Math Functions](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.6: 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:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    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:

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

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.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<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:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
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 [4.6: Adding New Functions](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [4.6: Adding New Functions](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.7: Definitions and Uses

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

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 [4.7: Definitions and Uses](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [4.7: Definitions and Uses](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.8: 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 [4.8: Flow of Execution](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [4.8: Flow of Execution](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.9: 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 [4.9: Parameters and Arguments](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [4.9: Parameters and Arguments](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.10: 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.

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 [4.10: 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](#).

- [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>.

4.11: Scope

In Python, when you define a function, it can access variables that are defined outside of the function. For instance, consider a variable `x` defined outside of a function `update_x`. Inside `update_x`, you can print the value of `x` because the function has access to `x`.

```
x = 5

def update_x():
    print(x) # This will print 5, as 'x' is accessible within 'update_x'

update_x()
```

If you execute this, it will print out the value 5 because the function is able to access the global variable `x`.

However, if you try to modify `x` within `update_x`, Python will not alter the original `x` defined outside. Instead, it treats it as if you've created a new local variable `x` within the function.

```
x = 5

def update_x():
    x = 10 # Python sees this as creating a new local variable, not modifying the
    global 'x'
    print(x) # Prints 10

update_x()
print(x) # Still prints 5, as the global 'x' remains unchanged
```

You can write a function that can modify global variables from within a function. To do this, you need to declare `x` as `global` inside the function:

```
def update_x():
    global x # Declares that we intend to use and modify the global 'x'
    x = 10 # This now modifies the global 'x'
    print(x) # Prints 10

update_x()
print(x) # Also prints 10, showing that the global 'x' has been modified
```

This example illustrates Python's variable scope management and how the `global` keyword enables functions to modify variables defined outside their scope.

4.11: Scope is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

4.12: 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 [4.12: Why functions?](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **4.11: Why functions?** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.13: 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 [4.13: Debugging](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [4.12: Debugging](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

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 `compute_pay` 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 `compute_grade` 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 [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](#).

- [4.E: Functions \(Exercises\)](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

4.G: Functions (Glossary)

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 [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](#).

CHAPTER OVERVIEW

5: Iterations

- [5.1: Updating Variables](#)
- [5.2: The while Statement](#)
- [5.3: Infinite Loops](#)
- [5.4: "Infinite loops" and break](#)
- [5.5: continue, break, enumerate, zip](#)
- [5.6: Finishing iterations with continue](#)
- [5.7: Definite loops using for](#)
- [5.8: Loop patterns](#)
- [5.9: Counting and Summing Loops](#)
- [5.10: Maximum and Minimum Loops](#)
- [5.11: Debugging](#)
- [5.E: Iterations \(Exercises\)](#)
- [5.G: Iterations \(Glossary\)](#)

This page titled [5: Iterations](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

5.1: 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.

```
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`:

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

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

```
>>> x = 0
>>> x = x + 1
```

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

This page titled [5.1: Updating Variables](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [5.1: Updating Variables](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.2: 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!".

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 [5.2: The while Statement](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [5.2: The while Statement](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.3: 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 [5.3: Infinite Loops](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [5.3: Infinite Loops](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.4: "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:

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 5.4: "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](#).

- 5.4: "Infinite loops" and break by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.5: continue, break, enumerate, zip

1. Break Statement: The `break` keyword can be employed to exit a loop prematurely when a certain condition is met. For example, consider this code:

```
for ii in range(10):  
    print(ii)
```

This prints 0 through 9. `break` lets you terminate the loop early. For example, if you only want to print up to 5, you could write the code like this:

```
for ii in range(10):  
    if ii > 5: break  
    print(ii)
```

When the code exceeds 5, the break command is executed and the loop is exited. This is useful when the loop condition might be hardcoded or otherwise unchangeable, and you need an early exit strategy.

2. Continue Statement: Unlike `break`, which exits the loop, `continue` skips the remainder of the current iteration and moves on to the next one. This allows you to exit a loop iteration but without having to terminate the loop entirely.

For example, if we only want to print to 5, we could use this code:

```
for ii in range(10):  
    if ii > 5: continue  
    print(ii)
```

In this case, the continue statement does not cause the loop to end, but, once ii exceeds 5, it sends the loop back to the top before it executes the print statement. Use this when you are done with a particular iteration and want to skip any remaining code in the for loop and move on to the next iteration.

3. Enumerate Function: The `enumerate` function in Python is useful when you need both the index and the value of items in a list during iteration. Let's consider a list of fruits and say we want to print each fruit along with its index in the list.

```
# List of fruits  
fruits = ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry']  
  
# Using enumerate to get both index and value  
for index, fruit in enumerate(fruits):  
    print(index , ',' , fruit)
```

When this code is executed, it will produce the following output:

```
0, Apple  
1, Banana  
2, Cherry  
3, Date  
4, Elderberry
```

4. Zip Function: The `zip` function is a powerful tool for iterating over multiple lists in parallel. Suppose you have two lists, one containing names of students and another containing their corresponding scores, and you want to print out each student's name along with their score.

Here's how you could use `zip` to accomplish this task:

```
# List of student names  
students = ['Alice', 'Bob', 'Charlie', 'Dave']
```

```
# Corresponding scores
scores = [85, 92, 78, 90]

# Using zip to iterate over both lists
for student, score in zip(students, scores):
    print(f"Student: {student}, Score: {score}")
```

When you run this code, you'll see the following output:

```
Student: Alice, Score: 85
Student: Bob, Score: 92
Student: Charlie, Score: 78
Student: Dave, Score: 90
```

In this example, the `zip` function takes the two lists `students` and `scores`, and moves the elements into `student` and `score`, which you can use within the loop.

This approach is much cleaner than iterating through the lists separately and then indexing into them, especially when you're dealing with more than two lists. It makes the code easier to read and understand, adhering to the principles of clean and efficient coding that the speaker in your transcript emphasized.

Let me emphasize that Python's flexibility allows for multiple ways to achieve the same outcome. For example, one almost never needs to use `zip` or `enumerate`, although using those functions can make the code cleaner.

5.5: [continue](#), [break](#), [enumerate](#), [zip](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

5.6: 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).

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 [5.6: 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](#).

- [5.5: Finishing iterations with continue](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.7: 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:

In Python terms, the variable `friends` is a list¹ 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 5.7: 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.

- 5.6: Definite loops using for by Chuck Severance is licensed CC BY-NC-SA 4.0. Original source: <https://www.py4e.com/html3>.

5.8: 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 [5.8: Loop patterns](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [5.7: Loop patterns](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.9: Counting and Summing Loops

For example, to count the number of items in a list, we would write the following `for` loop:

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:

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 [5.9: 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](#).

- [5.8: Counting and Summing Loops](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.10: Maximum and Minimum Loops

To find the largest value in a list or sequence, we construct the following loop:

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:

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 [5.10: 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](#).

- [5.9: Maximum and Minimum Loops](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.11: 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 [5.11: Debugging](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [5.10: Debugging](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

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 [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](#).

- [5.E: Iterations \(Exercises\)](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

5.G: Iterations (Glossary)

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 [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](#).

CHAPTER OVERVIEW

6: Strings

- [6.1: A string is a sequence](#)
- [6.2: Getting the length of a string using len](#)
- [6.3: Traversal through a string with a loop](#)
- [6.4: String Slices](#)
- [6.5: Looping and Counting](#)
- [6.6: The in operator](#)
- [6.7: String Comparison](#)
- [6.8: String Methods](#)
- [6.9: Parsing strings](#)
- [6.10: String formatting](#)
- [6.E: Strings \(Exercises\)](#)
- [6.G: Strings \(Glossary\)](#)

This page titled [6: Strings](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

6.1: 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.

b	a	n	a	n	a
[0]	[1]	[2]	[3]	[4]	[5]

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 [6.1: 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](#).

- **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>.

6.2: 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 [6.2: 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](#).

- [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>.

6.3: 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 [6.3: 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](#).

- **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>.

6.4: 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 [6.4: String Slices](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [6.4: String Slices](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

6.5: Looping and Counting

The following program counts the number of times the letter `a` appears in a string:

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 [6.5: Looping and Counting](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [6.6: Looping and Counting](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

6.6: 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
```

You can also use the `not` with the word `in` :

```
>>> 'a' not in 'banana'
False
>>> 'x' not in 'banana'
True
```

This page titled [6.6: The in operator](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [6.7: The in operator](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

6.7: String Comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':  
    print('All right, bananas.')
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':  
    print('Your word,' + word + ', comes before banana.')
```

```
elif word > 'banana':  
    print('Your word,' + word + ', comes after banana.')
```

```
else:  
    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:

```
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 [6.7: String Comparison](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [6.8: String Comparison](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

6.8: 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 [6.8: String Methods](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [6.9: String Methods](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

6.9: 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
>>> spos = data.find(' ', atpos)
>>> print(spos)
31
>>> host = data[atpos+1:spos]
>>> 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 **6.9: Parsing strings** is shared under a [CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **6.10: Parsing strings** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](https://creativecommons.org/licenses/by-nc-sa/4.0/). Original source: <https://www.py4e.com/html3>.

6.10: String formatting

In Python programming, string formatting plays a crucial role in enhancing the readability and utility of output data. Properly formatted strings help ensure that the output is clear, consistent, and easy to interpret. Consider a scenario where you're displaying floating-point numbers representing scientific measurements. Without proper formatting, you might end up with numbers like "3.141592653589793" or "0.20000000000000001," which could be cumbersome and confusing to read. On the other hand, using string formatting allows you to control the number of decimal places, so you get cleaner, more digestible outputs like "3.14" or "0.20". The ability to precisely control output styles becomes even more important in more complex applications, such as data analysis or scientific computing, where the manner in which data is presented can significantly impact the interpretation of results. Therefore, mastering string formatting is essential for anyone looking to produce high-quality, professional-level Python code.

I will now introduce f-string formatting, a modern and convenient way to format strings. By preceding a string with the letter 'f' or 'F', you can embed expressions inside string literals, enclosed by curly braces `{}`. These expressions are evaluated at runtime, and their values are directly inserted into the string. For example, if you have variables `name` and `age`, instead of the cumbersome concatenation like `"Hello, " + name + ". You are " + str(age) + " years old."`, you can simply write `f"Hello, {name}. You are {age} years old."`

The f-string method not only makes the code cleaner but also improves readability and reduces the chance of errors. You can even control the number of decimal places in a floating-point number with ease, for example, `f"The value of pi is approximately {pi:.2f}"`, which would format the value of pi to two decimal places. This brings a level of clarity and precision to your output, streamlining both development and debugging processes.

Within the string, the {}s have the variable and how it is formatted, using the syntax `{variable:format}` .

the format contains the following format: Width.DecimalCode. For example, you could write it as `f'{dog:4.1f}'` . This create the string of the value of dog, with a width of 4 spaces, and one digit after the decimal point.

```
dog = 3.14159
f'{dog:4.1f}'
```

It outputs ' 3.1', which a length of 4 and one digit after the decimal point. If you leave out the '4', it will just format the string so that there's just one digit after the decimal point.

```
cat = 342.38780
f'{cat:.2f}'
```

Outputs '342.39'

'f' is one format code, but there are others. For example, if you want the output to be in exponential notation, you can use the 'e' code:

```
f'{cat:.2e}'
```

which yields '3.42e+02'. [Here is a list of formatting codes.](#)

Number Formatting Types

Type	Meaning
d	Decimal integer
c	Corresponding Unicode character
b	Binary format
o	Octal format
x	Hexadecimal format (lower case)
X	Hexadecimal format (upper case)
n	Same as 'd'. Except it uses current locale setting for number separator
e	Exponential notation. (lowercase e)
E	Exponential notation (uppercase E)
f	Displays fixed point number (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
%	Percentage. Multiplies by 100 and puts % at the end.

Read more about f-string formatting [here](#).

6.10: String formatting is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

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 [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](#).

- [6.E: Strings \(Exercises\)](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

6.G: Strings (Glossary)

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 [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](#).

CHAPTER OVERVIEW

7: Lists

- [7.1: A list is a sequence](#)
- [7.2: Lists are mutable](#)
- [7.3: Traversing a List](#)
- [7.4: List operations](#)
- [7.5: List Slices](#)
- [7.6: List Methods](#)
- [7.7: Deleting Elements](#)
- [7.8: Lists and Functions](#)
- [7.9: Lists and Strings](#)
- [7.10: Parsing lines](#)
- [7.11: Objects and Values](#)
- [7.12: Aliasing](#)
- [7.13: List arguments](#)
- [7.14: List comprehensions](#)
- [7.15: Debugging](#)
- [7.E: Lists \(Exercises\)](#)
- [7.G: Lists \(Glossary\)](#)

This page titled [7: Lists](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

7.1: 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 *elements* or sometimes *items*.

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 [7.1: 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](#).

- **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>.

7.2: 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 [7.2: Lists are mutable](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **8.2: Lists are mutable** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.3: 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:

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` :

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 [7.3: Traversing a List](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.3: Traversing a List](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.4: 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 [7.4: List operations](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.4: List operations](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.5: 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 [7.5: List Slices](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.5: List Slices](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.6: 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 [7.6: List Methods](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.6: List Methods](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.7: 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 [7.7: Deleting Elements](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.7: Deleting Elements](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.8: 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:

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.

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 [7.8: Lists and Functions](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **8.8: Lists and Functions** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.9: 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 [7.9: Lists and Strings](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **8.9: Lists and Strings** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.10: 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.za Sat Jan 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 [7.10: Parsing lines](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.10: Parsing lines](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.11: 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:

`a` → 'banana' `a` → 'banana'
`b` → 'banana' `b` → 'banana'

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 [7.11: Objects and Values](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **8.11: Objects and Values** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.12: 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 [7.12: Aliasing](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.12: Aliasing](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.13: 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 [7.13: List arguments](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.13: List arguments](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.14: List comprehensions

What is a List Comprehension?

A list comprehension is an elegant way to create a new list by applying operations to an existing list or iterable object (like a string). It's a core feature of Python that draws inspiration from mathematical set-builder notation.

Basic Structure

The general syntax of a list comprehension looks like this:

```
new_list = [expression for item in iterable if condition]
```

Let's break it down:

- **expression:** This defines the element that will be included in the new list. It can be a simple variable reference, a transformation of the original item, or a more complex calculation.
- **item:** A placeholder representing each element in the original iterable object.
- **iterable:** The existing list, sequence, or any object that can be iterated over.
- **if condition:** An optional filter; only elements that meet the condition will be included in the new list.

Example

Let's square the numbers in a list:

```
numbers = [1, 2, 3, 4, 5]

# Traditional way (using a for loop)
squares = []
for number in numbers:
    squares.append(number * number)

# Using a list comprehension
squares = [number * number for number in numbers]

print(squares) # Output: [1, 4, 9, 16, 25]
```

Here's an example using list comprehensions to make a list of squared numbers that are odd:

```
numbers = [1, 2, 3, 4, 5]

# Using a list comprehension
odd_squares = [number * number for number in numbers if number % 2 != 0]

print(odd_squares) # Output: [1, 9, 25]
```

This code first creates a list of numbers `[1, 2, 3, 4, 5]`. Then, it uses a list comprehension to create a new list `odd_squares`. The list comprehension iterates over the original list `numbers`, and for each number, it checks if the number is odd using the modulo operator (`%`). If the number is odd, it squares the number and adds it to the `odd_squares` list. Otherwise, it does nothing.

The final output of the code is the list `[1, 9, 25]`, which contains the squares of all the odd numbers in the original list.

Advantages of List Comprehensions

1. **Conciseness:** List comprehensions can condense several lines of traditional loop-based code into a single, readable expression.

2. **Clarity** List comprehensions often clearly convey the intent of the list transformation.
3. **Potential Efficiency:** Under certain conditions, list comprehensions may be more efficient than manually constructed loops.

When to Use List Comprehensions

List comprehensions are best suited for simple to moderately complex list transformations. For very complex scenarios, a traditional `for` loop might be more readable and maintainable.

Other things to read:

[read this](#)

[and this](#)

7.14: List comprehensions is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

7.15: 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! ~{.python}`

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.za Sat Jan 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:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    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?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
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.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From' : continue
    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.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
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 [7.15: Debugging](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- **8.14: Debugging** by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.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
Minimum: 2.0
```

This page titled [7.E: Lists \(Exercises\)](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [8.E: Lists \(Exercises\)](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

7.G: Lists (Glossary)

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 [7.G: Lists \(Glossary\)](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

CHAPTER OVERVIEW

8: Sets

[8.1: Introduction to sets](#)

[8.2: set operations](#)

[8.3: set order](#)

8: Sets is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

8.1: Introduction to sets

We already learned about lists, which is one of the most important data structures. A related data structure is the set. Unlike lists, sets ensure that there are no duplicate entries, making them useful for tasks that require uniqueness among items.

A practical example of when you'd use a set in a program is when you need to remove duplicates from a list of items. Imagine you're developing an application that collects user responses from a survey where each response includes interests or hobbies. Since users might enter duplicate interests, your task is to summarize the unique interests across all responses without repetition. You can use sets for this.

Sets in Python also support a rich set of operations, mirroring the concepts found in mathematical set theory. These include union, intersection, difference, and symmetric difference. Such operations are not just syntactically straightforward but are also optimized for performance, further underscoring the utility of sets in scenarios involving large datasets. For example, finding common elements between two sets (intersection) or combining elements from multiple sets while maintaining uniqueness (union) can be done efficiently with set operations.

Here are examples demonstrating different methods of constructing sets in Python:

- **Direct Method:** When you use curly braces `{}`, you can directly define a set with unique elements. For example, `direct_set = {1, 2, 3, 4, 5}` creates a set with elements 1 through 5. The output for this is `{1, 2, 3, 4, 5}`.
- **Using `set()` Constructor with a List:** You can also create a set from a list by passing the list as an argument to the `set()` constructor. This method automatically removes any duplicate elements. For instance, if you have a list `[1, 2, 2, 3, 4]`, you can convert this to a set using this command:

```
list_to_set = set([1, 2, 2, 3, 4])
```

which converts it into a set `{1, 2, 3, 4}`. Remember that sets do not allow duplicate items, so when you convert this list into a set, one of the 2s disappear.

If you want to create an empty set, you must use the `set()` constructor without any arguments, as `{}` will create an empty dictionary instead. An example is `empty_set = set()`, which results in `set()`, representing an empty set.

The `remove` and `add` methods in Python's set data structure are used for manipulating set contents. The `remove` method is used to delete a specific element from a set. If the specified element is present in the set, it is removed; however, if the element does not exist in the set, Python raises a `KeyError`. This underscores the importance of either checking for an element's existence before attempting to remove it or using the `discard` method instead, which does not raise an error if the element is not found. The `remove` method is particularly useful when you need to ensure that an operation only proceeds if the element to be deleted is indeed a part of the set, providing a strict control mechanism over set manipulation.

The `add` method is used to insert a new element into a set. The uniqueness of sets ensures that if the element already exists within the set, adding it again has no effect, maintaining the set's invariant of containing unique elements only. This method is straightforward and does not require checking for the element's presence beforehand, making it an efficient way to ensure that a set contains a given element.

Here are examples demonstrating the use of the `remove` and `add` methods with a Python set:

We start with a set `{1, 2, 3, 4, 5}`. It's a collection of unique integers from 1 to 5.

Using the `remove` Method: When we apply `my_set.remove(3)`, the element 3 is removed from the set, resulting in `{1, 2, 4, 5}`.

Using the `add` Method: Adding a new element to the set with `my_set.add(6)` updates the set to `{1, 2, 4, 5, 6}`.

Adding an Existing Element: Trying to add an element that's already in the set, such as 2, does not change the set because sets cannot have duplicate items in it.

8.1: Introduction to sets is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

8.2: set operations

Sets are not only powerful due to their ability to maintain unique elements but also because they offer a suite of operations that can perform complex manipulations with succinct and readable code. These operations, borrowing concepts from mathematical set theory, include unions, intersections, differences, and symmetric differences. Here are some key methods and operators you can use with sets:

Union

The union of two or more sets is a fundamental operation that combines all elements from the involved sets, ensuring that each element is unique. In Python, the union can be performed using the `union()` method or the `|` operator.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
union_set = set_a.union(set_b) # or set_a | set_b
print(union_set) # Output: {1, 2, 3, 4, 5}
```

Note that 3 appears in both sets, but only appears once in the union. The union operation is particularly useful when you need to gather all distinct items from multiple collections.

Intersection

Intersection creates a new set containing only the elements that are present in both sets. This operation is useful for finding commonalities between datasets. Similar to union, intersection can be achieved with the `intersection()` method or the `&` operator.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
intersection_set = set_a.intersection(set_b) # or set_a & set_b
print(intersection_set) # Output: {3}
```

Intersections are often used in filtering processes where items meeting a certain criteria across multiple sets are required.

Difference

The difference operation yields a set containing elements that are in one set but not in the others. It's performed using the `difference()` method or the `-` operator. This is particularly useful for identifying elements that are unique to one dataset.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
difference_set = set_a.difference(set_b) # or set_a - set_b
print(difference_set) # Output: {1, 2}
```

This operation can be used to exclude specific items from a set based on the contents of another.

Symmetric Difference

Symmetric difference produces a set with elements that are in either of the sets but not in both. It can be executed with the `symmetric_difference()` method or the `^` operator. This operation is useful for finding items that are unique to each set when comparing two datasets.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
symmetric_difference_set = set_a.symmetric_difference(set_b) # or set_a ^ set_b
print(symmetric_difference_set) # Output: {1, 2, 4, 5}
```

8.3: set order

Sets in Python are unordered. This means that the elements in a set do not have a fixed position; the order of elements can appear random and can change with every execution of your program. This unordered nature is a result of the set's primary focus, which is to ensure each element is unique and to provide efficient membership testing, not to maintain element order. Because sets do not support indexing or slicing, trying to access elements by an index will result in a `TypeError`.

However, you can still iterate over a set in Python using a `for` loop, just like you would with a list. The order in which you iterate over the elements is not guaranteed to be consistent across runs of the program, reflecting the unordered nature of sets. However, this does not impede your ability to access and perform operations on each element within the set during iteration.

Here's a basic example of iterating over a set:

```
my_set = {1, 2, 3, 4, 5}
for element in my_set:
    print(element)
```

In this example, each element in `my_set` is printed out. The output will include all the elements in `my_set`, though the order in which they appear can vary. This feature makes sets useful for cases where you need to perform operations on unique elements without caring about the order of those elements.

8.3: set order is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

9: Dictionaries

[9.1: Dictionaries](#)

[9.2: Dictionary as a Set of Counters](#)

[9.3: Dictionaries and Files](#)

[9.4: Looping and Dictionaries](#)

[9.5: Advanced Text Parsing](#)

[9.6: Debugging](#)

[9.E: Dictionaries \(Exercises\)](#)

[9.G: Dictionaries \(Glossary\)](#)

This page titled [9: Dictionaries](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

9.1: Dictionaries

A *dictionary* is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called *keys*) and a set of values. Each key maps to a value. The association of a key and a value is called a *key-value pair* or sometimes an *item*.

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function `dict` creates a new dictionary with no items. Because `dict` is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
```

The curly brackets, `{}`, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key `'one'` to the value `"uno"`. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print(eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print `eng2sp`, you might be surprised:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print(eng2sp)
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print(eng2sp['two'])
'dos'
```

The key `'two'` always maps to the value `"dos"` so the order of the items doesn't matter.

If the key isn't in the dictionary, you get an exception:

```
>>> print(eng2sp['four'])
KeyError: 'four'
```

The `len` function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The `in` operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the method `values`, which returns the values as a list, and then use the `in` operator:

```
>>> vals = list(eng2sp.values())
>>> 'uno' in vals
True
```

The `in` operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a *hash table* that has a remarkable property: the `in` operator takes about the same amount of time no matter how many items there are in a dictionary. I won't explain why hash functions are so magical, but you can read more about it at [Wikipedia.org/wiki/Hash_table](https://en.wikipedia.org/wiki/Hash_table).

Exercise 1: [wordlist2]

Write a program that reads the words in `words.txt` and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the `in` operator as a fast way to check whether a string is in the dictionary.

This page titled [9.1: Dictionaries](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [9.1: Dictionaries](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

9.2: Dictionary as a Set of Counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function `ord`), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An *implementation* is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don't have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:

We are effectively computing a *histogram*, which is a statistical term for a set of counters (or frequencies).

The `for` loop traverses the string. Each time through the loop, if the character `c` is not in the dictionary, we create a new item with key `c` and the initial value 1 (since we have seen this letter once). If `c` is already in the dictionary we increment `d[c]`.

Here's the output of the program:

```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on.

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

We can use `get` to write our histogram loop more concisely. Because the `get` method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the `if` statement.

The use of the `get` method to simplify this counting loop ends up being a very commonly used "idiom" in Python and we will use it many times in the rest of the book. So you should take a moment and compare the loop using the `if` statement and `in` operator with the loop using the `get` method. They do exactly the same thing, but one is more succinct.

This page titled [9.2: Dictionary as a Set of Counters](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [9.2: Dictionary as a Set of Counters](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

9.3: Dictionaries and Files

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let's start with a very simple file of words taken from the text of *Romeo and Juliet*.

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included.

```
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
```

We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary.

You will see that we have two `for` loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called *nested loops* because one of the loops is the *outer* loop and the other loop is the *inner* loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating "more quickly" and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count1.py
```

When we run the program, we see a raw dump of all of the counts in unsorted hash order. (the `romeo.txt` file is available at www.py4e.com/code3/romeo.txt)

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

It is a bit inconvenient to look through the dictionary to find the most common words and their counts, so we need to add some more Python code to get us the output that will be more helpful.

This page titled [9.3: Dictionaries and Files](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [9.3: Dictionaries and Files](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

9.4: Looping and Dictionaries

If you use a dictionary as the sequence in a `for` statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

Here's what the output looks like:

```
jan 100
chuck 1
annie 42
```

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have described earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:

The `for` loop iterates through the *keys* of the dictionary, so we must use the index operator to retrieve the corresponding *value* for each key. Here's what the output looks like:

```
jan 100
annie 42
```

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the `keys` method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:

Here's what the output looks like:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

First you see the list of keys in unsorted order that we get from the `keys` method. Then we see the key-value pairs in order from the `for` loop.

This page titled [9.4: Looping and Dictionaries](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [9.4: Looping and Dictionaries](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

9.5: Advanced Text Parsing

In the above example using the file `romeo.txt`, we made the file as simple as possible by removing all punctuation by hand. The actual text has lots of punctuation, as shown below.

```
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,
```

Since the Python `split` function looks for spaces and treats words as tokens separated by spaces, we would treat the words "soft!" and "soft" as *different* words and create a separate dictionary entry for each word.

Also since the file has capitalization, we would treat "who" and "Who" as different words with different counts.

We can solve both these problems by using the string methods `lower`, `punctuation`, and `translate`. The `translate` is the most subtle of the methods. Here is the documentation for `translate`:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

Replace the characters in `fromstr` with the character in the same position in `tostr` and delete all characters that are in `deletestr`. The `fromstr` and `tostr` can be empty strings and the `deletestr` parameter can be omitted.

We will not specify the `table` but we will use the `deletechars` parameter to delete all of the punctuation. We will even let Python tell us the list of characters that it considers "punctuation":

```
>>> import string  
>>> string.punctuation  
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

The parameters used by `translate` were different in Python 2.0.

We make the following modifications to our program:

```
import string

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(line.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: http://www.py4e.com/code3/count2.py
```

Part of learning the "Art of Python" or "Thinking Pythonically" is realizing that Python often has built-in capabilities for many common data analysis problems. Over time, you will see enough example code and read enough of the documentation to know where to look to see if someone has already written something that makes your job much easier.

The following is an abbreviated version of the output:

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Looking through this output is still unwieldy and we can use Python to give us exactly what we are looking for, but to do so, we need to learn about Python *tuples*. We will pick up this example once we learn about tuples.

This page titled [9.5: Advanced Text Parsing](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [9.5: Advanced Text Parsing](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

9.6: Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

Scale down the input

If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first n lines.

If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

Check summaries and types

Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

Write self-checks

Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a "sanity check" because it detects results that are "completely illogical".

Another kind of check compares the results of two different computations to see if they are consistent. This is called a "consistency check".

Pretty print the output

Formatting debugging output can make it easier to spot an error.

Again, time you spend building scaffolding can reduce the time you spend debugging.

This page titled [9.6: Debugging](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [9.6: Debugging](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

9.E: Dictionaries (Exercises)

Exercise 2: Write a program that categorizes each mail message by which day of the week the commit was done. To do this look for lines that start with "From", then look for the third word and keep a running count of each of the days of the week. At the end of the program print out the contents of your dictionary (order does not matter).

Sample Line:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Sample Execution:

```
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Exercise 3: Write a program to read through a mail log, build a histogram using a dictionary to count how many messages have come from each email address, and print the dictionary.

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Exercise 4: Add code to the above program to figure out who has the most messages in the file.

After all the data has been read and the dictionary has been created, look through the dictionary using a maximum loop (see Section [maximumloop]) to find who has the most messages and print how many messages the person has.

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 5: This program records the domain name (instead of the address) where the message was sent from instead of who the mail came from (i.e., the whole email address). At the end of the program, print out the contents of your dictionary.

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

This page titled [9.E: Dictionaries \(Exercises\)](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

- [9.E: Dictionaries \(Exercises\)](#) by [Chuck Severance](#) is licensed [CC BY-NC-SA 4.0](#). Original source: <https://www.py4e.com/html3>.

9.G: Dictionaries (Glossary)

dictionary

A mapping from a set of keys to their corresponding values.

hashtable

The algorithm used to implement Python dictionaries.

hash function

A function used by a hashtable to compute the location for a key.

histogram

A set of counters.

implementation

A way of performing a computation.

item

Another name for a key-value pair.

key

An object that appears in a dictionary as the first part of a key-value pair.

key-value pair

The representation of the mapping from a key to a value.

lookup

A dictionary operation that takes a key and finds the corresponding value.

nested loops

When there are one or more loops "inside" of another loop. The inner loop runs to completion each time the outer loop runs once.

value

An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word "value".

This page titled [9.G: Dictionaries \(Glossary\)](#) is shared under a [CC BY-NC-SA 4.0](#) license and was authored, remixed, and/or curated by [Chuck Severance](#).

CHAPTER OVERVIEW

10: Packages

- [10.1: Importing packages in python](#)
- [10.2: How to use functions in packages](#)
- [10.3: Aliasing packages](#)
- [10.4: Sub packages](#)

10: Packages is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

10.1: Importing packages in python

Python, in its core form, is a somewhat limited language that doesn't encompass every possible capability a developer might need. However, its true power shines through the extensibility provided by packages. These packages allow developers to infuse Python with additional capabilities, essentially customizing and expanding the language to meet specific requirements. For instance, while Python doesn't have built-in tools for data analysis or web development, packages like Pandas and Django fill these gaps, offering functionalities that make tasks in these domains significantly easier. This modularity and extensibility have transformed Python into one of the most versatile and widely-used programming languages today.

Before you use a package in Python, you need to import it. To begin with, you need to ensure that the package you intend to use is installed in your Python environment; this can often be done using commands such as `pip install package_name`. In ATMO 321, most of the packages we need are already installed. So you can generally ignore this step, but if you do research, you might have to do this.

Once the package is installed, you can import it into your Python script using the `import` statement followed by the package name, for instance, `import numpy`. You can also import specific functions or classes from a package using the `from ... import ...` syntax, like `from datetime import date`. Furthermore, to simplify your code and make it more readable, Python allows you to assign aliases to imported packages using the `as` keyword, for instance, `import numpy as np`. Through this mechanism, Python facilitates a more organized and efficient coding environment, granting you access to a broad spectrum of pre-defined functionalities to streamline your development process.

10.1: Importing packages in python is shared under a [not declared](#) license and was authored, remixed, and/or curated by Andy Dessler.

10.2: How to use functions in packages

When you import a package in Python, the functions within that package aren't directly accessible by their name alone. Instead, to use a function from a package, you typically need to prefix the function's name with the package's name, followed by a dot. This process is known as "namespacing," and it's essential for preventing naming conflicts between functions from different packages or between a package's functions and your own.

For example, let's say you've imported the `numpy` package, which contains a variety of mathematical functions. If you want to use the `sqrt` function (which calculates the square root of a number) from the `numpy` package, you'd write:

```
import numpy

result = numpy.sqrt(25) # This will assign 5.0 to the variable 'result'.
```

Here, the `numpy.` prefix before `sqrt` indicates that you're using the `sqrt` function from the `numpy` package. This distinction is crucial, especially in cases where multiple packages might have functions with the same name or if there's a chance of a naming collision with functions you've defined in your own code.

There are ways to bypass this prefixing if needed. For instance, if you use the `from ... import ...` syntax, you can directly access the function:

```
from numpy import sqrt

result = sqrt(25) # This uses the numpy version of sqrt without needing the 'numpy.'
prefix.
```

But be cautious with this approach, as it can lead to name clashes if you're not careful. Always ensure that you're aware of the functions and variables present in your current namespace to prevent unexpected behaviors.

10.2: How to use functions in packages is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

10.3: Aliasing packages

In Python, while importing packages, there's an option to rename or assign an alias to them for ease of use and enhanced readability. This is particularly beneficial when working with packages that have longer names or when there's a convention in the community to use a specific alias. The `as` keyword is used for this purpose. For instance, `numpy` is a popular package for numerical computations in Python, and it's a common practice among developers to import it using the alias `np`.

By writing `import numpy as np`, the entire functionality of the `numpy` package becomes accessible via the shorter prefix `np`.

```
import numpy as np  
print(np.sqrt(25))
```

This not only makes the code cleaner but also speeds up writing code, especially when the package's functions are frequently accessed. This practice of renaming has become so widespread that for certain packages like `numpy`, not aliasing the package would look odd.

10.3: Aliasing packages is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

10.4: Sub packages

Packages in Python can contain subpackages, essentially packages nested within them. These subpackages help in further organizing and categorizing functionalities, making the primary package modular and more manageable. One prime example of this hierarchical organization is seen in the `numpy` package. While `numpy` itself offers a vast array of numerical operations, it also includes subpackages like `random` to handle random number generation and related tasks.

When you want to access a function from a subpackage, you have to traverse this hierarchy using the dot notation. If, for instance, you've imported `numpy` as `np`, to use a function within the `random` subpackage, you'd employ the syntax `np.random.function_name`. For example, to generate a random float between 0 and 1, you'd use `np.random.rand()`.

It's also worth noting that it's possible to import subpackages directly, streamlining the syntax a bit. For instance, you can write `from numpy import random` and then just use `random.function_name` without the `np` prefix. Alternatively, you can even combine the use of subpackages with the aliasing technique: `import numpy.random as npr`, allowing you to access random functions using `npr.function_name`.

10.4: Sub packages is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

11: Numpy, part 1

There are several candidates in the Python language for representing the type of array structure we introduced in chapter 7. One is the plain-ol' Python **list**, which you may have used if you've taken a computer science course in Python. Turns out, **lists** are going to be too slow for us once we start dealing with a lot of data, plus there are a lot of things that it won't do for us automatically that are handy to have. Another choice is the Pandas **Series** which we'll actually introduce in chapter 11 – oddly, that one turns out to do too much, rather than too little, for our purposes here. A happy medium is the **ndarray** from the NumPy package¹. Before we do that, however, we need to learn what a “package” actually is, and how to use one.

¹Most people seem to pronounce this “NUM-pie,” although I've heard “NUM-pee” as well. Pick your poison.

[11.1: Packages](#)

[11.2: Mathematical Functions](#)

[11.3: The NumPy 'ndarray'](#)

This page titled [11: Numpy, part 1](#) 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.

11.1: Packages

Back in my day (circa 1990's) when someone wanted to write a computer program, they wrote the entire thing themselves, line by line. Everything you needed to do – from something complex like making a remote network connection to something simple like computing the average of some numbers – was up to you to build. Code sharing over the Internet just wasn't much of a thing.

Today, the reverse is true. When you write a complex data analysis program, most of the code will actually be written by others, if you do it right. This is because many, many smart people across the globe have written snippets of code to do all the common (and some not-so-common) things you'll want to do, and your job is to string them all together. Put another way: you're given most of the Legos® – and even a bunch of pre-assembled chunks made with dozens of Legos® each – and your job is to construct your masterpiece out of those building blocks.

In Python, a **package** is a repository of useful functions and methods that someone else has written. By **importing** a package into your program, you're making all those useful things available to you. Your own code can then call those functions/methods whenever you see fit. It's the modular, organized, and elegant way to do things, in addition to saving a ton of time.

The first package we'll use is called **NumPy**, which stands for “Numerical Python.” To import it, you should include this exact line of code in the *first* Code cell of your Notebook:

```
import numpy as np
```

Note that it's in all lower-case letters. Once that cell has been executed, you now have access to all the NumPy “stuff,” which is the subject of this chapter.

This page titled [11.1: Packages](#) 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.

- [8.1: Packages](#) by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

11.2: Mathematical Functions

You can see a list of mathematical functions [here](#).

11.2: Mathematical Functions is shared under a [not declared](#) license and was authored, remixed, and/or curated by Andy Dessler.

11.3: The NumPy 'ndarray'

The actual data type that the NumPy package provides is called an **ndarray**, which stands for “n-dimensional array.” If that sounds heady, it kind of is, although in this course we’re only ever going to use a *one*-dimensional array, which is super simple to understand. In fact, it looks exactly like the examples in Figure 7.1.1.

“One-dimensional” just means that there is a single index number, and the elements are all in a line.²

Creating ndarrays

There are many different ways to create an **ndarray**. We’ll learn four of them.

Way 1: `np.array([])`

The first is to use the **array()** function of the NumPy package, and give it all the values explicitly. Here’s the code to reproduce the Figure 7.1.1 examples:

```
followees = np.array(['@katyperry', '@rihanna', '@TheEllenShow'])
balances = np.array([1526.73, 98774.91, 1000000, 4963.12, 123.19])
```

It’s simple, but don’t miss the syntactical gotcha: *you must include a pair of boxies inside the bananas*. Why? Reasons.³ For now, just memorize that for this function – and this function only – we use “([...stuff...])” instead of “(...stuff...)” when we call it.

By the way, the attentive reader might object to me calling `array()` a function, instead of a method. Isn’t there a word-and-a-dot before it, and isn’t that a “method thing?” Shrewd of you to think that, but actually no, and the reason is that “**np**” isn’t the name of a variable, but the name of a package. When we say “**np.array()**” what we’re saying is: “Python, please call the **array()** function *from the np package*.” The word-and-dot syntax does double-duty.

We can call the **type()** function, as we did back on p. 17, to verify that yes indeed we have created **ndarrays**:

```
print(type(followees))
print(type(balances))
| numpy.ndarray
| numpy.ndarray
```

This is useful, but sometimes we want to know what underlying atomic data type the array is comprised of. To do that, we attach “**.dtype**” (confusingly, *without* bananas this time) to the end of the variable name. “**.dtype**” stands for “data type.” Here goes:

```
print(followees.dtype)
print(balances.dtype)
| dtype('<U13')
| dtype('float64')
```

Whoa, what does that stuff mean? It’s a bit hard on the eyes, but let me explain. The underlying data type of **followees** is (bizarrely) “U13” which in English means “strings of Unicode characters⁴, each of which is 13 characters long or less.” (If you bother to count, you’ll discover that the longest string in our **followees** array is the last one, ‘@TheEllenShow’, which is exactly 13 characters long.) The “**float64**” thing means “**floats**, each of which is represented with 64 bits⁵ in memory.

You don’t need to worry about any of those details. All you need to know is: if an array’s dtype has “<U” in it, then it’s composed of strings; and if it has the word “**int**” or “**float**” in it, it means one of those two old friends from chapter 3.

Incidentally, you'll recall from chapter 7 that an array is *homogeneous*, which means all its elements are of the same type. NumPy enforces this. If you try to combine them:

```
weird = np.array([3, 4.9, 8])
strange = np.array([18, 73.0, 'bob', 22.8])
```

you'll discover that NumPy converts them to all be of the same type:

```
print(weird)
print(weird.dtype)
print(strange)
print(strange.dtype)
| [ 3.  4.9  8. ]
| dtype('<U3')
| ['18' '73.0' 'bob' '22.8']
| dtype('float64')
```

See how the **ints** **3** and **8** from the first array were converted into the **floats** **3.** and **8.**; meanwhile, all of the numerical elements of the second array got converted to **strs**. (If you think about it, that's the only direction the conversions could go.)

Way 2: `np.zeros()`

It will often be useful to create an array, possibly a large one, with all elements equal to *zero* initially. Among other scenarios, we often need to use a bunch of counter variables to, well, count things. (Recall our incrementing technique from Section 5.1) Suppose, for example, that we had a giant array that held the numbers of likes that each Instagram photo had. When someone likes a photo, that photo's appropriate element in the array should be **incremented** (raised in value) by one. Similarly, if someone unlikes it, then its value in the array should be **decremented** by one.

An easy way to do this is NumPy's **zeros()** function:

```
photo_likes = np.zeros(400000000000)
```

(although I'll bet you don't have enough memory on your laptop to actually store an array this size! Instagram sure has a lot of pics...) When I do this on my Data Science cluster, I get this:

```
print(photo_likes)
print(photo_likes.dtype)
| array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
| float64
```

Don't miss the `"..."` in the middle of that first line! It means "there are (potentially) a lot of elements here that we're not showing, for conciseness." Also notice that `zeros()` makes an array of **floats**, not **ints**.

Way 3: `np.arange()`

Sometimes we need to create an array with regularly-spaced values, like "all the numbers from one to a million" or "all even numbers between 20 and 50." We can use NumPy's **arange()** function for this.

Normally we pass this function two arguments, like so:

```
usa_years = np.arange(1776, 2022)
print(usa_years)
print(usa_years.dtype)
| [1776 1777 1778 1779 ... 2018 2019 2020 2021]
| int64
```

If you read that code and output carefully, you should be surprised. We asked for elements in the range of 1776 to 2022, and we got...1776 through 2021. Huh?

Welcome to one of several little Python idiosyncrasies. When you use **arange()** you get an array of elements starting with the first argument, and going up through *but not including* the last number. There's a reason Python and NumPy decided to do it this way⁶, but for now it's just another random thing to memorize. If you forget, you're likely to get an "OBOE" – which stands for "off-by-one error" – a common programming error where you do almost the right thing but perform one fewer, or one more, operation than you meant to.

Anyways, other than that glitch, you can see that the function did a useful thing. We can quickly generate regularly-spaced arrays of any range of values we like. By including a third argument, we can even specify the **step size** (the interval between each pair of values):

```
twentieth_century_decades = np.arange(1900, 2010, 10)
prez_elections = np.arange(1788, 2024, 4)
print(twentieth_century_decades)
print(prez_elections)
| [1900 1910 1920 1930 1940 1950 1960 1970 1980 1990 2000]
| [1788 1792 1796 1800 ... 2008 2012 2016 2020]
```

Notice we had to specify 2010 and 2024 as the second argument to these function calls in order for the arrays to include 2000, and 2020, respectively. This is the same "up to but not including the end point" behavior, but extended to step sizes of greater than one.

Way 4: `np.loadtxt()`

Most of the data that we analyze will come from external files, rather than being typed in by hand in Python. For one thing, this is how it will be provided by external sources; for another, it's infeasible to actually type in anything very large.

Let me say a word about files. You probably work with them every day on your own computer, but what you might not realize is that fundamentally, they all contain the same kind of "data." You might think of a Microsoft Word document as a completely different kind of thing than a GIF image, or an MP3 song file, or a saved HTML page. But they're actually more alike than they are different. They all just contain a bunch of bits. Those bits are organized to conform to a particular kind of file specification, so that a program (like Word, Photoshop, or Spotify) can recognize and understand them. But it's still just "information." The difference between a Word doc and a GIF is like the difference between a book written in English and one written in Spanish; it's not like the difference between a bicycle and a fish.

In this course, we'll be working with **plain-text files**. This is how most of the open data sources on the Internet provide their information. A plain-text file is one that consists of readable characters, but which doesn't contain any additional formatting (like boldface, colors, margin settings, etc.). You can actually open up a plain-text file in any text editor (including Microsoft Word) and see what it contains.

In your CoCalc account, you have your own little group of files which, like those on your own computer, can be organized into **directories** (or **folders**⁷). *It is critically important that the data file you read, and the Jupyter Notebook that reads it, are in the same directory.* The #1 trouble students experience when trying to read from a text file is not having the text file itself located in

the same directory as the code that reads it. If you make this mistake, Python will simply claim to not recognize the filename you give it.

That doesn't mean your file doesn't exist! It's just not in the right place. An example of doing this *correctly* is in Figure 8.2.1. We're in a directory called "**filePractice**" (stare at the middle of the figure until you find those words) which is contained within the **home directory** that's denoted by a little house icon. Your home directory is just the starting point of your own private little CoCalc world. The slash mark between the house and the word **filePractice** indicates that **filePractice** is contained directly within, or "under," the home directory.

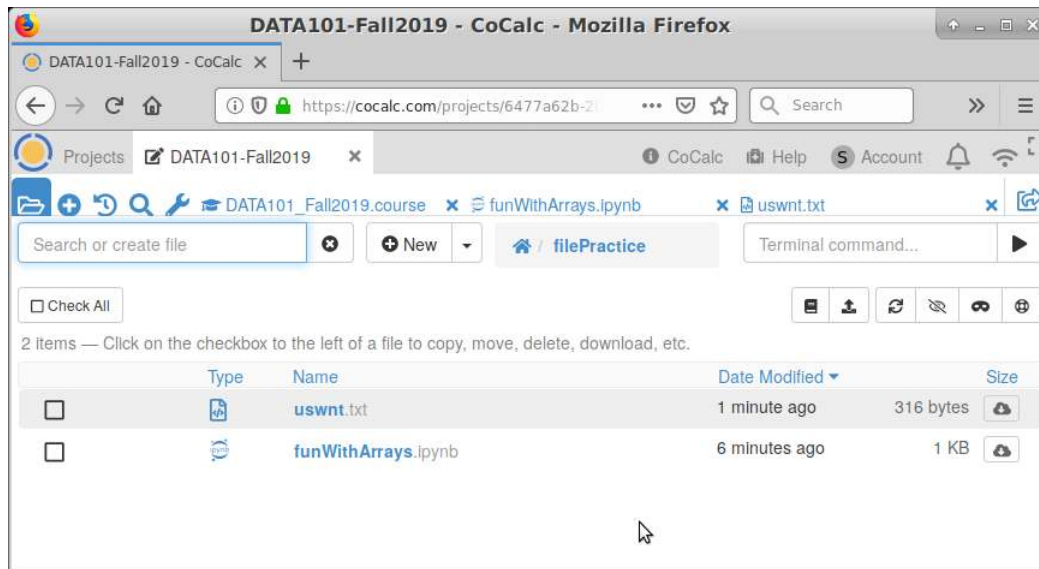


Figure 11.3.1: A directory (folder) on CoCalc, which contains two files: a plain-text file (called **uswnt.txt**) and a Jupyter Notebook which will read from it (**funWithArrays.ipynb**).

The two entries listed are a plain-text file (called **uswnt.txt**) and a Jupyter Notebook (**funWithArrays.ipynb**). You can tell that the former is a plain-text file because of the **filename extension** ".txt".⁸ If we clicked on **uswnt.txt**, we'll bring up the contents of the file, as shown in Figure 8.2.2. In this case, we have the current roster on the US Women's National Soccer team, one name per line. Perhaps the most important thing to see is that the file itself, which we will read into Python in a moment, is nothing strange or scary: you could type it yourself into Notepad or Word.⁹

This is a good time to mention that *spaces and other funny characters in filenames are considered evil*. You might think it looks better to call the notebook file "**fun with arrays.ipynb**" and the data file "**US Women's National Team roster.txt**", but I promise you it will lead to pain in the end, for a variety of fiddly reasons. It's better to use **camel case** for filenames, which is simply capitalizingEachSuccessiveWordInAPhrase.

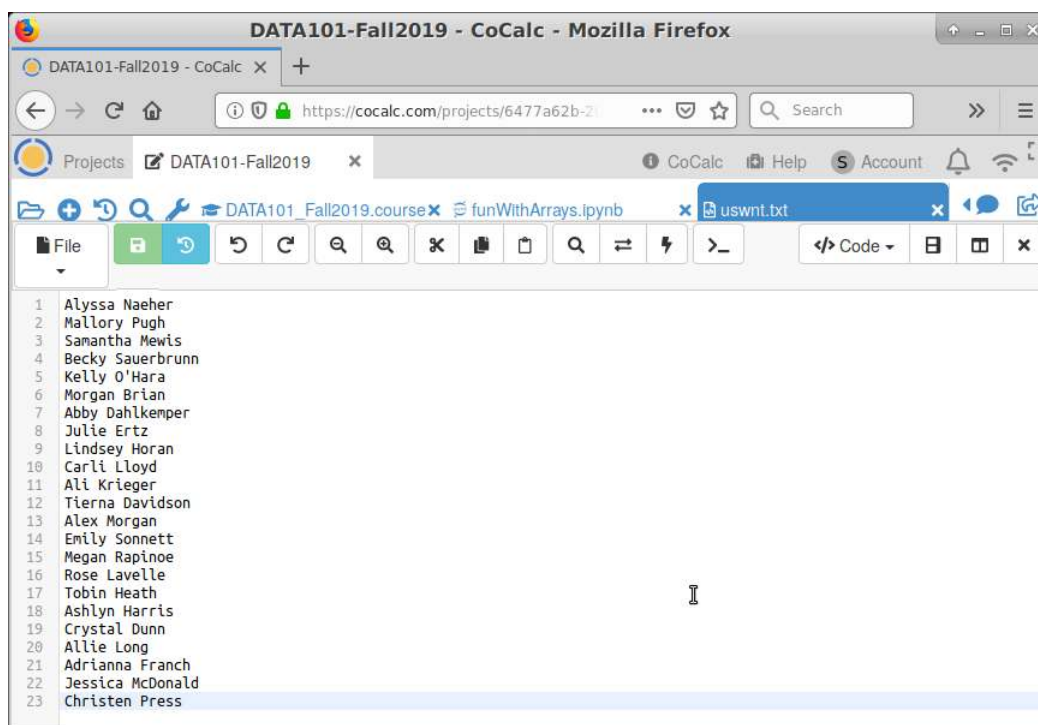


Figure 11.3.2: The contents of a plain-text file, as rendered by CoCalc.

Okay, finally back to NumPy code. If all the stars are aligned, we can write this code in a `funWithArrays.ipynb` cell to read the soccer roster into an `ndarray`:

```
roster = np.loadtxt("uswnt.txt", dtype=object, delimiter="###")
```

There's a lot of weird stuff in that line, so follow me here. The first argument is easy enough: the name of the file that contains our data. (Again, I stress that the file must be located in the same directory as the notebook!) The second argument is bizarre: we know what `dtype` means, but "`object`"? Ugh, another fiddly detail. When you read from a file into a NumPy array, you will be reading one of our three atomic types. Here are the rules:

If you want to read an array of...	...then set <code>dtype</code> to:
<code>ints</code>	<code>int</code>
<code>floats</code>	<code>float</code>
<code>strs</code>	<code>object</code>

So basically, you set `dtype` to the type of data you want in your `ndarray`...unless you want strings, in which case you put the word `object`. Sorry about that.

The last of the three arguments is even nuttier, and you actually don't need to include it at *all* if you're reading ints or floats. If you're reading `strs`, however, you need to set the `delimiter` to something *that doesn't appear in any of the `strs`*. I chose three-hashtags-in-a-row since that rarely appears in any set of text data.

Bottom line: once we've done all this, we get:

```
print(roster)

['Alyssa Naeher' 'Mallory Pugh' 'Sam Mewis' 'Becky Sauerbrunn' 'Kelly O'Hara' 'Morgan Brian' 'Abby Dahlkemper' 'Julie Ertz' 'Lindsey Horan' 'Carli Lloyd' 'Ali Krieger' 'Tierna Davidson' 'Alex Morgan' 'Emily Sonnett' 'Megan Rapinoe' 'Rose Lavelle' 'Tobin
```

Heath' 'Ashlyn Harris' 'Crystal Dunn' 'Allie Long' 'Adrianna Franch' 'Jessica McDonald'
'Christen Press']

which is pretty cool.

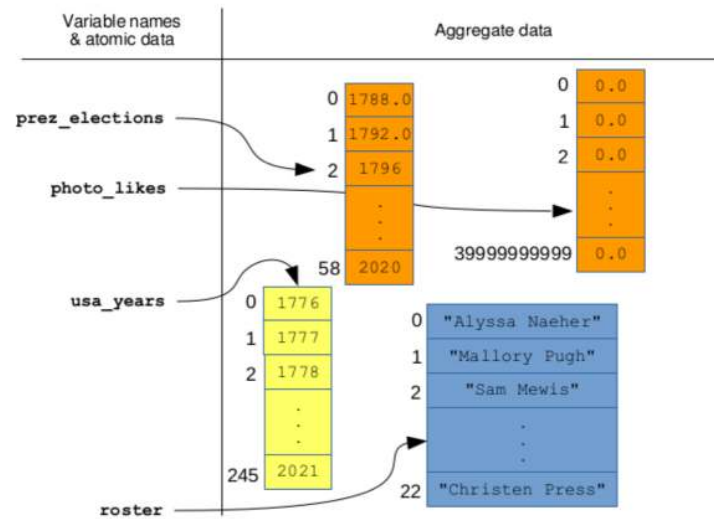


Figure 11.3.3: The memory picture of the four arrays we created in section 8.2.

This page titled [11.3: The NumPy 'ndarray'](#) 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.

- **8.2: The NumPy 'ndarray'** by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

CHAPTER OVERVIEW

12: Numpy (2 of 2)

Now that we know several options for how to *create* **ndarrays**, what can we do with them? Many and sundry things.

[12.1: Accessing Individual Elements](#)

[12.2: “Vectorized” Arithmetic Operators](#)

[12.3: Broadcasting](#)

[12.4: Copying – and Not Copying – Arrays](#)

[12.5: Sorting Arrays](#)

[12.6: More Exotic Array Modifications](#)

[12.7: Summary](#)

This page titled [12: Numpy \(2 of 2\)](#) 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.

12.1: Accessing Individual Elements

Retrieving an Element

To get the value of a specific element from an array, we use “boxie notation” with the index number:

```
print(prez_elections[0])
third_year = usa_years[2]
print("{} was the 3rd year of U.S.A.".format(third_year))
print("The highest-numbered player is {}".format( roster[len(roster)-1]))
| 788.0
| 1778 was the 3rd year of U.S.A.
| The highest-numbered player is Christen Press.
```

Remember, indices start at zero (not one) so that’s why the first line has a 0 in it.

Now examine that last line, which is kind of tricky. Whenever you have boxies, you have to first evaluate the code *inside* them to get a number. That number is then the index number Python will look up in the array. In the last line above, the code *inside* the boxies is:

...len(roster)-1...

Breaking it down, we know that **len(roster)** is 24, which means **len(roster)-1** must be 23, and so **roster[len(roster)-1]** is **Christen Press**. It’s a common pattern to get the last element of an array.¹

To test your understanding, figure out what the following code will print:

```
q = 2
r = np.array([45,17,39,99])
s = 1
print(r[q-s+1]+3)
```

The answer is at the end of the chapter.

Changing an Element

To modify an element of an array, just use the equals sign like we do for variables:

```
stooges = np.array(['Larry', 'Beavis', 'Moe'])
print(stooges)
stooges[1] = 'Curly'
print(stooges)
| ['Larry' 'Beavis' 'Moe']
| ['Larry' 'Curly' 'Moe']
```

After all, an individual element like **stooges[1]** is itself a variable pretty much like any other.

Slices

Sometimes, instead of getting just one element from an array, we want to get a whole chunk of them at a time. We're interested in the first ten elements, say, or the last twenty, or the fourteen elements starting at index 100. To do this sort of thing, we use a **slice**.

Suppose we had a list of states in alphabetical order, and wanted to snag a chunk of consecutive entries out of the middle – say, Arizona through Colorado. Consider this code:

```
states = np.array(["AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "FL", "GA", "HI"])
print(states[2:6])
| ['AZ' 'AR' 'CA' 'CO']
```

The “**2:6**” in the boxies tells Python that we want a slice with elements 2 through 5 (not through 6, as you'd expect). This is the same behavior we saw for **np.arange()** where the range goes up to, but not including, the last value. Just get used to it.

We can also omit the number before the colon, which tells Python to start at the beginning of the array:

```
print(states[:5])
| ['AL' 'AK' 'AZ' 'AR' 'CA']
```

or omit the number after the colon, which says to go until the end:

```
print(states[8:])
| ['FL' 'GA' 'HI']
```

We can even include a second colon, after which a third number specifies a step size, or stride length. Consider:

```
print(states[2:9:3])
| ['AZ' 'CO' 'FL']
```

This tells Python: “start the slice at element #2, and go up to (but not including) element #9, **by threes**.” If you count out the states by hand, you'll see that Arizona is at index 2, Colorado is at index 5, and Florida is at index 8. Hence these are the three elements included in the slice.

This slice stuff may seem esoteric, but it comes up surprisingly often.

This page titled [12.1: Accessing Individual Elements](#) 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.

- [9.2: Accessing Individual Elements](#) by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

12.2: “Vectorized” Arithmetic Operators

Recall our table of Python math operators (Figure 5.1.1). What do those things do if we use them on aggregate, instead of atomic data? The answer is: something super cool and useful.

Operating on an Array and a Single Value

Consider the following code:

```
num_likes_today = np.array([6, 61, 0, 0, 14])
num_likes_tomorrow = num_likes_today + 3
print(num_likes_tomorrow)

| [ 9 64 3 3 17 ]
```

See what happened? “Adding 3” to the array means adding 3 to each element. All in one compact line of code, we can do five – or even five billion – operations. This works for all the other Figure 5.1.1 operators as well.

For somewhat geeky reasons, this sort of thing is called a **vectorized** operation. All you need to know is that this means **fast**. And that’s “fast” in two different ways: fast to write the code (since instead of using a **loop**, which we’ll cover in 14, you just write a single statement with + and = signs), and more importantly, fast to execute. For more geeky reasons, the above code will run lightning fast even if **num_likes_today** had five hundred million elements instead of just five. As you’ll learn if you ever try it, a Python loop is much slower.²

Don’t get me wrong: there are times we’ll have to use a loop because we have no choice. But the general rule with Python is: if you can figure out how to perform a calculation without using a loop, always do it!

Operating on Two Arrays

Possibly even cooler, we can even “+” (or “-”, or “*”, or...) two entire *arrays* together. Example:

```
salaries = np.array([38000, 102000, 55750, 29500, 250000])
raises = np.array([1000, 4000, 2000, 1000, 2000])
salaries = salaries + raises
print(salaries)
```

This code produces:

```
| [ 39000 106000 57750 30500 252000 ]
```

Can you see why? “Adding” the two arrays together performed addition *element-by-element*. The result is a new array with 38000+1000 as the first element, 102000 + 4000 as the second, etc. This, too, is a lightning-fast, vectorized operation, and it too works with all the other math operators.

Just to re-emphasize one point before we go on. In the example back on p. 77, we assigned the result of the operation to a new variable, **num_likes_tomorrow**. This means that **num_likes_today** itself was *unchanged* by the code. In contrast, in the example we just did, we assigned the result of the operation back into an existing variable (**salaries**). So salaries has itself been updated as a result of that code.

This page titled [12.2: “Vectorized” Arithmetic Operators](#) 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.

- **9.3: “Vectorized” Arithmetic Operators** by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

12.3: Broadcasting

In NumPy, broadcasting refers to the practice of performing operations on arrays of different shapes and sizes in a manner that "broadcasts" the smaller array across the larger one. This concept is particularly useful for efficient memory utilization and computational speed, as it eliminates the need to explicitly replicate the smaller array multiple times to match the shape of the larger array. Imagine you have a 2D array (often called a matrix) and you want to add a 1D array (a vector) to each row of that matrix. You could use a loop to manually add the vector to each row, but that wouldn't be computationally efficient. Broadcasting automates this process. When you add the matrix and the vector together, NumPy automatically understands that you intend to add the vector to each row of the matrix. The dimensions are implicitly expanded to align the shapes of the arrays being operated upon. This intrinsic feature of NumPy enables cleaner code, faster execution, and overall, a more elegant handling of mathematical operations between arrays.

For example, let's say you have a NumPy array representing grades for a class. In this array, each row corresponds to a student and each of the three columns corresponds to an exam. You've decided to add curve points to the exams: 3 points for the first exam, 5 points for the second, and 10 points for the third. You could accomplish this task using a for loop to iterate over each column and add the corresponding curve points to all the rows in that column. While this may not be the most efficient method, it serves as an instructive example for understanding basic array manipulations.

Here's how you could code this using a for loop:

```
import numpy as np

# Grades for 6 students over 3 exams
grades = np.array([[80, 85, 88],
                  [76, 80, 82],
                  [92, 94, 96],
                  [65, 70, 75],
                  [90, 85, 88],
                  [70, 75, 80]])

# Curve points for each exam
curve_points = np.array([3, 5, 10])

# Loop through each column (exam) and add curve points
for i in range(len(curve_points)):
    grades[:, i] = grades[:, i] + curve_points[i]

print("Curved grades:")
print(grades)
```

output:

```
[[ 83 90 98]
 [ 79 85 92]
 [ 95 99 106]
 [ 68 75 85]
 [ 93 90 98]
 [ 73 80 90]]
```

In this example, the `for` loop iterates through each of the three columns of the `grades` array and adds the curve points to all the rows in that column.

In NumPy, the concept of broadcasting allows you to elegantly and efficiently apply operations across arrays of different shapes. When adding curve points to each exam in our example, you can take advantage of broadcasting to sidestep explicit looping, thereby enhancing computational efficiency. You have a 2D array of grades with 6 rows (students) and 3 columns (exams), and you want to add an array of curve points with shape `(3,)` to each row.

Here's how you could do this using broadcasting:

```
import numpy as np

# Grades for 6 students over 3 exams
grades = np.array([[80, 85, 88],
                  [76, 80, 82],
                  [92, 94, 96],
                  [65, 70, 75],
                  [90, 85, 88],
                  [70, 75, 80]])

# Curve points for each exam
curve_points = np.array([3, 5, 10])

# Add curve points to grades using broadcasting
grades = grades + curve_points

print("Curved grades:")
print(grades)
```

This generates the same output!

In this example, the `curve_points` array has shape `(3,)`, and the `grades` array has shape `(6, 3)`. When you perform the operation `grades = grades + curve_points`, NumPy takes the 3-element array and reshapes it to a 1 by 3 array. Then, it can copy the rows until it's a 6 by 3 array and can directly add the two arrays.

In other words, the dimensions of the 3-element array are automatically reconciled so that the `curve_points` array is virtually stretched, or "broadcast," over the shape of the `grades` array, allowing the addition to occur element-wise across all rows for each column.

Thus, broadcasting provides a more efficient, less verbose, and arguably more readable way to perform operations like this compared to traditional for-loop constructs.FM

Broadcasting is one of the hardest concepts in NumPy, so if you feel a bit unsure, don't worry about it. [Here's a video you can watch that will help you understand this](#). And here's [another one](#).

12.3: Broadcasting is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

12.4: Copying – and Not Copying – Arrays

Now, a surprise for the unwary. Suppose I write this code:

```
stooges = np.array(['Larry', 'Beavis', 'Moe'])
funny_people = stooges
stooges[1] = 'Curly'
print("The stooges are: {}".format(stooges))
print("The funny people are: {}".format(funny_people))
```

Take a moment and predict what you think the output will be. Then, read it and (possibly) weep:

```
|The stooges are: ['Larry' 'Curly' 'Moe'].
|The funny people are: ['Larry' 'Curly' 'Moe'].
```

Note carefully: *no Beavis*.

Now the question is why. To understand this (and virtually any other tricky programming problem) you have to return once again to the memory picture. Figure 9.4.1 shows the situation immediately before, and after, the line “**stooges[1] = 'Curly'**” executes. Crucially, *there is only one array* in memory. Both variables – **stooges** and **funny_people** – are pointing at it.

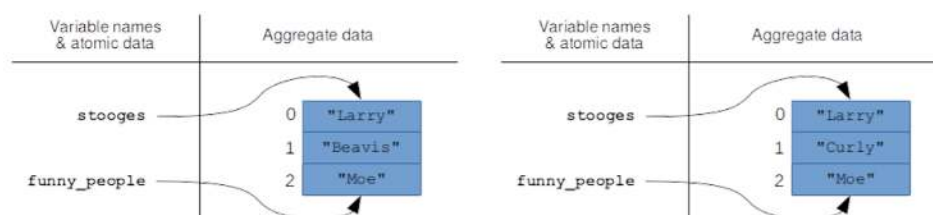


Figure 12.4.1: The code on p. 79 immediately before (left side) and after (right side) the line “**stooges[1] = 'Curly'**” is reached.

You see, if *y* contains *aggregate* (instead of atomic) data, the line “**x = y**” does not perform a copy operation. Instead, it just points the **x** variable name to the same place **y** is pointing to.

Once you grasp this, it’s easy to see why “**Beavis**” completely disappeared. There’s only one array at all, so changing **stooges** has the side effect of implicitly changing **funny_people** as well.

Actually Copying

The “point the variable to the same thing, but don’t do a copy” behavior is the default, because such copy operations are expensive (in terms of memory usage and time to execute). They’re normally not what you want anyway. Sometimes, however, you do want to produce an entire separate copy of an array, so you can modify the copy yet preserve the original. To do so, you use the **.copy()** method:

```
orig_beatles = np.array(['John', 'Paul', 'George', 'Pete'])
beatles = orig_beatles.copy()
beatles[3] = 'Ringo'
print("The Beatles were originally {}".format(orig_beatles))
print("But the ones we all know were {}".format(beatles))
```

Look carefully at that second line: it makes all the difference. Instead of making the new variable **beatles** point to the same array in memory that **orig_beatles** did, we explicitly copied the array and made **beatles** point to that new copy. The final memory picture is thus as per Figure 9.4.2, and the output is of course:

|The Beatles were originally ['John' 'Paul' 'George' 'Pete'].

|But the ones we all know were ['John' 'Paul' 'George' 'Ringo'].

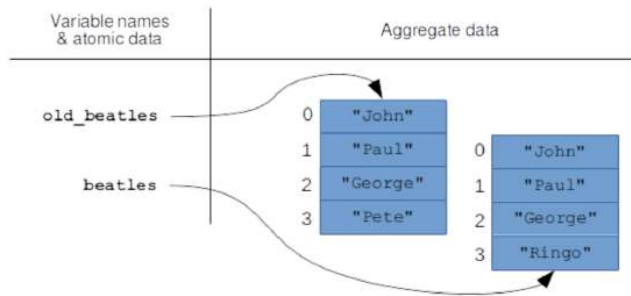


Figure 12.4.2: The memory picture after calling the `.copy()` method, instead of simply assigning to a new variable.

This page titled [12.4: Copying – and Not Copying – Arrays](#) 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.

- **9.4: Copying – and Not Copying – Arrays** by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

12.5: Sorting Arrays

A common operation in Data Science is to **sort** an array, either numerically (if the array contains **ints** or **floats**) or alphabetically (if strings). There are two ways to do this, which turn out to differ in the same way as the operations in the previous section.

One way is to call the **.sort()** method directly **on** an array. This sorts the array **in place**, which means that the actual data in memory is rearranged right then and there. As an important side effect, any *other* variable that points to the same array will *also* be sorted.

Here's an example:

```
gpas = np.array([2.86, 3.99, 3.12, 1.17])
gpas2 = gpas.copy()
gpas3 = gpas
gpas.sort()
print("gpas has: {}".format(gpas))
print("gpas2 has: {}".format(gpas2))
print("gpas3 has: {}".format(gpas3))
| gpas has: [1.17 2.86 3.12 3.99]
| gpas2 has: [2.86 3.99 3.12 1.17]
| gpas3 has: [1.17 2.86 3.12 3.99]
```

Do you see why that output was produced? It's because the memory picture after the "**gpas.sort()**" line looks like Figure 9.5.1. The **gpas** variable really is the **gpas3** variable, so when one is sorted, the other automatically is. They're both distinct from **gpas2**, though.

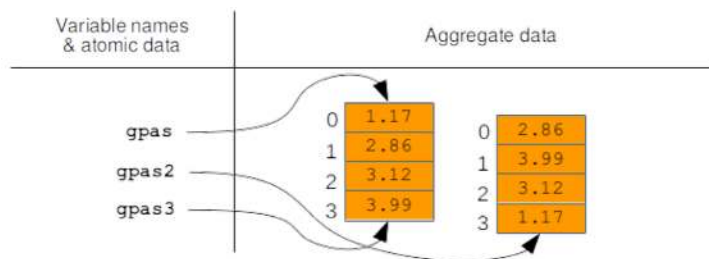


Figure 12.5.1: The state of affairs after **.sort()**ing the **gpas** array in place.

The second option is to call the **np.sort()** function and pass an array as an object. Like many Python functions, including the ones in the next section, **np.sort()** returns a *modified copy* of its argument rather than changing it in place. To illustrate:

```
nfl_teams = np.array(["Ravens", "Patriots", "Broncos", "Chargers", "Steelers"])
sorted_teams = np.sort(nfl_teams)
print(nfl_teams)
print(sorted_teams)
| ['Ravens' 'Patriots' 'Broncos' 'Chargers' 'Steelers']
| ['Broncos' 'Chargers' 'Patriots' 'Ravens' 'Steelers']
```

Observe that the `nfl_teams` variable, even though we passed it to `np.sort()`, was not *itself* sorted. The `sorted_teams` variable, on the other hand, is alphabetically sorted, because we assigned the return value from `np.sort()` to it. Again, the memory picture is shown in Figure 9.5.2.

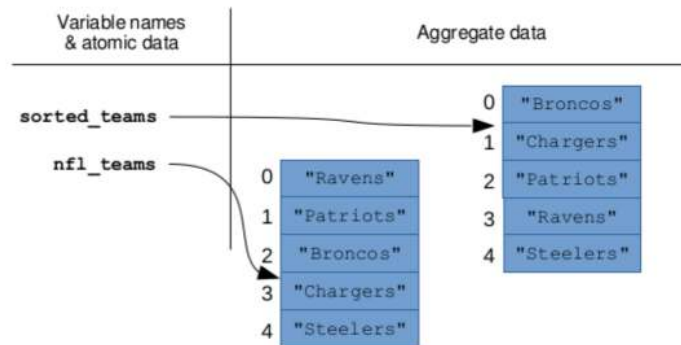


Figure 12.5.2: Calling the `np.sort()` function (as opposed to calling the `.sort()` method **on** the array) returns a sorted copy. To be clear, either one of these techniques can be used on *any* **ndarray**: whole numbers, real numbers, or text. I just chose to do real numbers in the first example and text in the second. The difference between the two is merely in what is affected: in one, the actual array in memory is modified, and in the other, a modified copy is returned.

This page titled 12.5: Sorting Arrays 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.

- 9.5: Sorting Arrays by Stephen Davies is licensed CC BY-SA 4.0. Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

12.6: More Exotic Array Modifications

There are lots of additional things you can do to an array to either modify its structure or rearrange its contents. Here's a few. **Important: all of the functions in this section return a modified copy of the array you pass to it. They do not change the array in place.**

- **np.append()** can be used to add a single element to the end of an array, or to add an entire second array of elements to it. (In the latter case, this is really **concatenation** of arrays.)
- **np.insert()** is like the first form of **np.append()**, except it inserts in the middle (or the beginning).
- **np.delete()** will remove an element of an array by *position*. In other words, you tell it which *index number* to remove, not which element.
- **np.flip()** reverses the order of elements in an array.

These functions are all summarized in Figure 9.8.1.

Remember that when you're calling a function like this – which returns a modified copy – it is perfectly acceptable to store the return value in the same variable that you passed it. This is common if you don't actually want to keep around the original:

```
ice_cream_flavors = np.flip(ice_cream_flavors)
```

In this pattern, the net effect is effectively to modify the array in place, since you're making a reversed copy, and then assigning that reversed copy to the same variable.

Anyway, here's some example code to illustrate the functions in this section:

```
clowns = np.array(["Bozo", "Krusty"])
more_clowns = np.array(["Pennywise", "Skelton"])
more_clowns = np.insert(more_clowns, 1, "Happy Slappy")
all_clowns = np.append(clowns, more_clowns)
all_clowns = np.append(all_clowns, "Ronald McDonald")
all_clowns = np.flip(all_clowns)
all_clowns = np.delete(all_clowns, 2)
print("clowns is: {}".format(clowns))
print("more_clowns is: {}".format(more_clowns))
print("all_clowns is: {}".format(all_clowns))
|clowns is: ['Bozo' 'Krusty']
|more_clowns is: ['Pennywise' 'Happy Slappy' 'Skelton']
|all_clowns is: ['Ronald' 'Skelton' 'Pennywise' 'Krusty' 'Bozo']
```

An excellent exercise to help cement your understanding of the ideas in this chapter would be to go through the above “clowns” code line by line, drawing the memory picture as you go, and then confirm that your output matches the actual output.

Yes, an excellent exercise indeed!

This page titled [12.6: More Exotic Array Modifications](#) 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.

- **9.6: More Exotic Array Modifications** by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

12.7: Summary

The table in Figure 9.8.1 gives the promised summary of the array functions, methods, and operators in this chapter.

Function	Description
<code>len(arr)</code>	Get the number of elements in the array <i>arr</i> .
<code>arr[17]</code>	Get a specific element's value from the array <i>arr</i> .
<code>arr[8] = (something)</code>	Set a specific element of the array <i>arr</i> .
<code>arr + 91</code>	Add a value to each element of <i>arr</i> , yielding a new array. (Also works with <code>-</code> , <code>*</code> , <code>/</code> , <i>etc.</i>)
<code>arr1 + arr2</code>	Add each pair of values in two arrays, yielding a new array. (Also works with <code>-</code> , <code>*</code> , <code>/</code> , <i>etc.</i>)
<code>arr1 = arr2</code>	Make <i>arr1</i> point to the same data that <i>arr2</i> points to. (<i>Not</i> a copy!)
<code>arr1 = arr2.copy()</code>	Make <i>arr1</i> point to a new, independent copy of <i>arr2</i> .
<code>arr.sort()</code>	Sort the array <i>arr</i> in place . (Numerical or alphabetical, depending on the <code>.dtype</code> .)
<code>np.sort(arr)</code>	Return a new array with the sorted elements of <i>arr</i> . (Numerical or alphabetical, depending on the <code>.dtype</code> .)
<code>np.append(arr, elem)</code>	Return a new array with <i>elem</i> tacked on to the end.
<code>np.append(arr1, arr2)</code>	Return a new array with the two arrays <i>arr1</i> and <i>arr2</i> concatenated.
<code>np.insert(arr, ind, val)</code>	Return a new array with the new value <i>val</i> inserted into position <i>ind</i> of <i>arr</i> .
<code>np.delete(arr, ind)</code>	Return a new array with the element at index <i>ind</i> removed from <i>arr</i> .
<code>np.flip(arr)</code>	Return a new array with <i>arr</i> in reverse order.

Figure 12.7.1: Handy NumPy functions, methods, and operators.

The Answer

Oh, and the answer to the puzzle on p. 75 – and also the answer to Life, the Universe, and Everything, as it turns out – is 42.

This page titled [12.7: Summary](#) is shared under a [not declared](#) 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.

- **9.8: Summary** by Stephen Davies has no license indicated. Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

CHAPTER OVERVIEW

13: Matplotlib

[13.1: Loading matplotlib](#)

[13.2: Matplotlib basics](#)

[13.3: Matplotlib basics, 2](#)

[13.4: Matplotlib tutorial](#)

13: Matplotlib is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

13.1: Loading matplotlib

Matplotlib is a highly regarded plotting package that you'll often find employed to visualize data within the scientific Python community. As one of the essential tools in data analysis, its significance cannot be overstated. That's because, at the conclusion of any thorough analysis, visualization is almost always a necessary step. It allows researchers, scientists, and analysts to translate complex numerical results into intuitive, comprehensible visuals. Graphs, charts, and other visual representations can make trends and patterns more accessible, aiding in both understanding and communication of findings. In essence, Matplotlib serves as a bridge between raw data and human insight, transforming numbers and statistics into something tangible and relatable, thereby playing a crucial role in the scientific programming.

Importing:

```
import matplotlib.pyplot as plt
```

 will allow you to access the matplotlib commands as `plt.*` — e.g., `plt.plot()` .

An alternative way to do this is:

```
from matplotlib.pyplot import *
```

This means that you can use those functions and variables directly in your code without needing to use the `plt.` prefix.

While this can make your code slightly more concise, it's generally considered bad practice, especially in larger or more complex scripts. Importing everything with `*` can lead to confusion and conflicts if different modules have functions or variables with the same names. It can make the code harder to read and maintain, as it's less clear which package each function comes from.

Using the more common `import matplotlib.pyplot as plt` keeps all the functions and variables contained within the `plt` namespace, avoiding these potential issues and making the code more transparent and maintainable.

You can also execute the command: `%pylab inline` which loads both NumPy and Matplotlib, so it's very convenient. It executes both `import matplotlib.pyplot as plt` and `from matplotlib.pyplot import *` , so you can use the `plt.` prefix or not. While this is considered deprecated, I still use it because it's very convenient.

13.1: Loading matplotlib is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

13.2: Matplotlib basics

At the core of this library is the `plot` command, which primarily takes in x and y coordinates to create a variety of plots – from line and scatter plots to bar and pie charts. It's crucial to understand that x is typically the independent variable and y, the dependent one. x and y can be either lists or numpy arrays, so the interface is quite flexible.

It pairs the elements of x and y to generate the plot points. So `plot([1, 2, 3], [4, 5, 6])` plots the point pairs [1,4], [2,5], and [3,6]. Note that you do not have to provide an x array; if you do not, it will pair the y values with sequential integers beginning with zero. So `plot([10, 12, 14])` plots the points [0,10], [1,12], and [2,14].

Matplotlib provides an extensive range of customization options to fine-tune the appearance of your plots, ensuring they are not only informative but also engaging. Central to this capability are parameters like color, marker, and line style.

Color: The color of a plot can convey a multitude of information. Matplotlib allows users to choose from a wide palette of colors. You can use standard color abbreviations like 'r' for red, 'b' for blue, or even use hexadecimal color codes for precision. It's essential to pick colors that are distinguishable, especially when multiple plots overlap, and ensure they are accessible to those with color vision deficiencies.

Markers: Markers are symbols used to represent data points. Matplotlib provides a plethora of options, from standard circles ('o') and triangles ('^') to more specialized ones like pixel markers ('.'). The choice of marker can help differentiate data series and should be chosen to best represent the nature and number of data points.

Line Style: The manner in which data points are connected or represented can be controlled using line styles. Options range from solid lines ('-') to dashed ('--'), dash-dot ('-.'), and dotted lines (':'). The choice of line style can indicate different types of data trends or emphasize certain aspects of your data.

For example, as mentioned, the combination 'r--o' would depict a series of red-colored data points marked with circles, connected by dashed lines. Such a combination not only provides a clear visual representation of data points but also the trend they follow.

These customizations can be defined using two primary methods: using a string shorthand after specifying the 'y' values or using explicit keyword arguments. Let's delve into both methods:

1. String Shorthand Method:

When calling the `plot` function, immediately after specifying the 'x' and 'y' values, you can include a string that defines the line's appearance.

- **Color:** Single letters represent colors (e.g., 'r' for red, 'g' for green, 'b' for blue).
- **Marker:** Characters represent data point symbols (e.g., 'o' for circles, '^' for triangles, 's' for squares).
- **Line Style:** Characters represent the type of line (e.g., '-' for a solid line, '--' for a dashed line, '-.' for dash-dot).

You can combine these in a string to specify multiple attributes. For example:

- `'ro-'`: Red color, circle markers, solid line.
- `'g^--'`: Green color, triangle markers, dashed line.

2. Using Keyword Arguments:

Instead of the shorthand string, you can specify each attribute using keyword arguments in the `plot` function.

- **Color:** Use the `color` keyword. Example: `color='red'`
- **Marker:** Use the `marker` keyword. Example: `marker='o'`
- **Line Style:** Use the `linestyle` keyword. Example: `linestyle='--'`

When using the `plot` function, you can combine these keywords:

```
plt.plot(x, y, color='red', marker='o', linestyle='--')
```

Both methods offer the same level of customization, so the choice comes down to personal or organizational preference. The string shorthand provides a concise way to specify common combinations of properties, while the keyword arguments offer a more verbose but perhaps clearer method, especially for those unfamiliar with the shorthand conventions.

13.3: Matplotlib basics, 2

In Matplotlib, the functions `xlim` and `ylim` are used to set the range of values that will be displayed on the x and y-axes, respectively. By default, Matplotlib automatically determines these ranges based on the data provided, aiming to showcase the entirety of the data points. However, there are scenarios where a specific focus or range is necessary, either to emphasize a particular segment of the data or to maintain consistent axis scales across multiple plots for comparison. The `xlim` function allows you to set the lower and upper bounds of the x-axis, while `ylim` does the same for the y-axis. By using these functions, you have granular control over the visible area of your plot, ensuring that your visualizations convey the intended message or highlight specific aspects of your data.

You use the commands by calling the function with the lower and upper ranges: `plt.xlim(5,10)` sets the limit of the x-axis to be 5 to 10. `ylim` is used in the same way.

The functions `xlabel` and `ylabel` are used to add descriptive labels to the x and y-axes of a plot, respectively. These labels provide context to the data being visualized, making it easier for viewers to understand what each axis represents. For instance, if you're plotting a graph of time against revenue, the x-axis might represent time (e.g., months or years), and the y-axis might represent the revenue amount (e.g., dollars or thousands of dollars). By using `xlabel` and `ylabel`, you can clearly annotate these axes with labels like "Months" for the x-axis and "Revenue (in thousands)" for the y-axis. Properly labeled axes are essential for ensuring the clarity and comprehensibility of a chart or graph, as they guide the viewer in interpreting the presented data correctly.

When working with data visualizations, there often arises the need to share or present your findings outside the immediate environment in which you created them. This is especially true when crafting reports, presentations, or even when collaborating with team members who might not have access to your exact development setup. In such instances, Matplotlib offers the function `savefig` for this.

The `savefig` command allows users to export and save their plots in a variety of common file formats suitable for diverse applications. Here's the basic syntax and some commonly used parameters:

```
plt.savefig(fname, dpi=None, format=None, bbox_inches=None, **kwargs)
```

- `fname`: This is the filename (including path). The output format can often be deduced from the file extension (e.g., `.png`, `.pdf`, `.jpeg`, etc.). If not provided, the `format` parameter will be necessary.
- `dpi`: The resolution of the saved figure in dots per inch. If not specified, it uses the default value from the `savefig.dpi` configuration parameter.
- `format`: This explicitly determines the format of the output file. Common values include `'png'`, `'pdf'`, `'jpeg'`, etc. If not provided, Matplotlib will use the extension of the `fname` to determine the format.
- `bbox_inches`: Determines which part of the figure is saved. Typical values include `'tight'` (which tries to eliminate any unnecessary padding around the figure) or `None` (which uses the default setting).
- `**kwargs`: Additional keyword arguments that are passed to the backend being used.

13.3: Matplotlib basics, 2 is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

13.4: Matplotlib tutorial

Go over these tutorials:

- [W3 schools](#)
- [Matplotlib official tutorial](#) (this is a python notebook; download it and open it in colab and you can execute the code as you read it)

If you want a really deep dive into Matplotlib, you can read [this textbook](#). It's more than you need for this class, but might be useful if you're putting together publication-quality figures.

Here is Matplotlib [cheat sheet](#).

13.4: Matplotlib tutorial is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

14: Pandas Series

[14.1: Introduction](#)

[14.2: The Pandas Series](#)

[14.3: More about Pandas series](#)

[14.4: Vectorized Arithmetic Operators](#)

[14.5: Sorting Serieses](#)

[14.6: Queries](#)

[14.7: Summary](#)

This page titled [14: Pandas Series](#) 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.

14.1: Introduction

In the world of data, imagine using NumPy and having a list of temperatures, like 70°F, 75°F, and 68°F. If all you have is this array, you don't know what each number represents. Is 70°F the temperature for Monday or for Phoenix? You'll therefore need to carry the metadata with this array, and if you're using NumPy, you would need to define another array that would have, for example, the city name associated with each temperature: "Houston", "Phoenix", "Seattle", etc. That would be inconvenient and make code harder to debug and maintain.

But with Pandas, it's different. A Pandas series is like a NumPy array, but each element of the array comes with a special tag, called an "index." The index gives you the metadata for each element in the array. For example, imagine we have an array of city temperatures; in a Pandas series, it would look like this:

Index	Temperature
New York	72°F
Los Angeles	78°F
Chicago	68°F
Houston	82°F
Miami	86°F
Seattle	64°F
Denver	70°F
Boston	66°F
San Francisco	67°F
Dallas	85°F

The index is the name of the city and the data is the temperature. This also makes accessing elements of the array much simpler. If these temperatures were in a NumPy array and you wanted to know the temperature of Chicago, you need to figure out which element number Chicago is and then access it using `array[index]` syntax; in this case, Chicago's temperature is `array[2]`.

But with Pandas, because the index is stored with the data, you can just ask for the temperature of Chicago: `array['Chicago']`.

14.1: Introduction is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

14.2: The Pandas Series

The standard way to import Pandas is to use this line:

```
import pandas as pd
```

This code should go at the top of your first notebook cell, right under your `import numpy as np` line. The two go hand in hand.

Let's begin talking about Pandas series. A **Series** is conceptually a set of key-value pairs. The keys are normally all of the same type (e.g., all integers or all strings), and so are the values, although the keys might be of a different type than the values (e.g., the keys could be strings and the values could be floats).

The Pandas package calls the keys “the index,” which is an overlap with the term we used for ordinary arrays.

Creating a Series

Here are a few common ways of creating a Pandas Series object in memory.

Way 1: create an empty Series

Perhaps this first one sounds dumb, but we will indeed have occasion to start off with an empty **Series** and then add key/value pairs to it from there. The code is simple:

```
my_new_series = pd.Series()
```

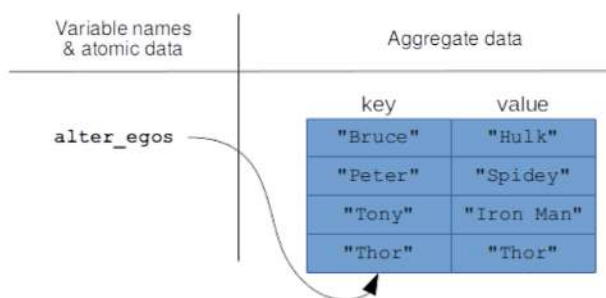
Voilà.

Way 2: `pd.Series([], index=[])`

As with NumPy **ndarrays**, we can explicitly list the values we want in a new **Series**. We also have to list the **index** values (the keys). The syntax for doing so is:

```
alter_egos = pd.Series(['Hulk', 'Spidey', 'Iron Man', 'Thor'], index=
['Bruce', 'Peter', 'Tony', 'Thor'])
```

This creates the **Series** shown in Figure 11.1.2.



Way 3: “wrapping” an array

Associative arrays, and the Pandas **Series** we’ve been using to implement them, are inherently *one-dimensional* data structures. This is just like the NumPy arrays we used before. Pandas **Series** also provide a bunch of features for manipulating, querying, computing, and even graphing aspects of their content. It’s a lot of rich stuff on top of plain-old NumPy.

For this reason, it’s common to want to create a **Series** that just “wraps” (or encloses) an underlying NumPy **ndarray**, and provides all that rich stuff.

The way to do this is simple:

```
my_numpy_array = np.array(['Ghost', 'Pumpkin', 'Vampire', 'Witch'])
my_pandas_enhanced_thang = pd.Series(my_numpy_array)
```

You can then treat **my_pandas_enhanced_thang** as an ordinary aggregate variable which has the more sophisticated operations of next chapter automatically glommed on to it. The keys (index values) of this thang will simply be the integers 0 through 3.

Way 4: `pd.read_csv()`

Finally, there's reading data from a text file. Data typically resides in sources and files external to our programming environment, and we want to do everything we can to play ball with this open universe.

One common data format is called **CSV**, which stands for **comma-separated** values. Files in this format are normally named with a **".csv"** extension. As the name suggests, the lines in such a file consist of values separated by commas. For example, suppose there's a file called **disney_rides.csv** whose contents looked like this:

Pirates of the Carribean,	25
Small World,	20
Peter Pan,	29

These are the current expected wait time (in minutes) for each of these Disney World rides at some point of the day.

To read this into Python, we use the **pd.read_csv()** function. It's a bit awkward since it has several mandatory arguments if you want to deal with **Series**. Here's how it works:

```
wait_times = pd.read_csv('disney_rides.csv', index_col=0, squeeze=True, header=None)
```

Most of that junk is just to memorize for now, not to fully understand. If you're curious, **index_col=0** tells Pandas that the first (0th) column – namely, the ride names – should be treated as the **index** for the **Series**. The **header=None** means “there is no separate header row at the top of the file, Pandas, so don't try to treat it like one.” If our **.csv** file *did* have a summary row at the top, containing labels for the two columns, then we'd skip the **header=None** part. Finally, “**squeeze=True**” tells Pandas, “since this is so skinny anyway – just two columns – let's have **pd.read_csv()** return us a **Series**, rather than a more complex **DataFrame** object (which is the subject of a later chapter).”

This page titled [14.2: The Pandas Series](#) 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.

- **11: Associative Arrays in Python (1 of 3)** by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.
- **11.1: The Pandas Series** by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

14.3: More about Pandas series

In NumPy, you access individual elements using its integer index, with zero-based indexing: `array[0]` for the first element, `array[1]` for the second, and so on. You can also use negative integers to index elements from the end of the array; for example, `array[-1]` will give you the last element.

if you have Pandas series named `animal_weights` with the index the animal types and the data the weight in pounds, it might look like this:

```
Elephant 13000
Horse     1200
Kangaroo  200
Penguin   25
Python    150
```

We use the `iloc` method for integer-location based indexing like NumPy. For example, `animal_weights.iloc[0]` will return the first row:

```
Elephant 13000
```

and `animal_weights.iloc[-1]` will return the last row:

```
Python 150
```

In NumPy, you can extract a subset of the array using the syntax `array[start:stop:step]`. For instance, `array[2:5]` would extract elements at indices 2, 3, and 4. In a similar way, you can extract subsets of a Pandas series with `animal_weights.iloc[2:5]`, which returns

```
Horse     1200
Kangaroo   200
Penguin    25
```

If this were all there were, then Pandas would present little benefit over NumPy. But Pandas also lets you access rows based on the index labels, not their integer location. For example, we can access the weight of a Kangaroo with the syntax: `animal_weights['Kangaroo']` or, even simpler, `animal_weights.Kangaroo`. This makes accessing data much simpler than NumPy because you don't have to figure out which row the data you want is — you just use the index label to find it.

You can also select several rows using slicing similar to NumPy. For example, you can extract the middle three rows using the syntax: `animal_weights.loc['Horse':'Penguin']`. This would return the Pandas series:

```
Horse     1200
Kangaroo   200
Penguin    25
```

Three things to note: 1) when you slice into a Pandas series, you get the entire row: both the indices and values. 2) Remember that label-based indexing uses the `.loc` method, while integer indexing uses `.iloc`. 3) Unlike integer indexing, where the extracted subset does NOT include the last number, when you use label-based indexing, you get row corresponding to the last index (in this case, 'Penguin').

14.3: More about Pandas series is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

14.4: Vectorized Arithmetic Operators

As with NumPy **ndarrays**, you can apply arithmetic operators like **+** and ***** to entire Serieses at a time, which is not only easy code to write but also runs blazing fast. But the Pandas Series is even smarter than that.

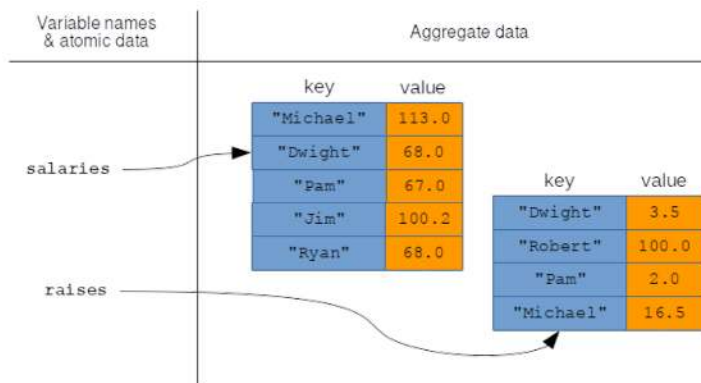


Figure 14.4.1: Two **Serieses** in memory

Consider the memory picture in Figure 12.1. Here we have two **Serieses**, one pointed to by a **salaries** variable and the other by **raises**, which are of different sizes and which have overlapping, but not identical, sets of keys. What do you suppose Pandas would do if we executed this code?

```
new_salaries = salaries + raises
```

The answer, happily, is the smartest possible thing it could do. Pandas gets neither confused nor stifled by the fact that the keys are in different orders in the two **Serieses**, and instead it does what you surely want: add corresponding elements, with matching keys, and produce a new Series with all of those sums.

The actual result in this case is in Figure 12.2.2, and the output is here:

```
new_salaries = salaries + raises
print(new_salaries)
```

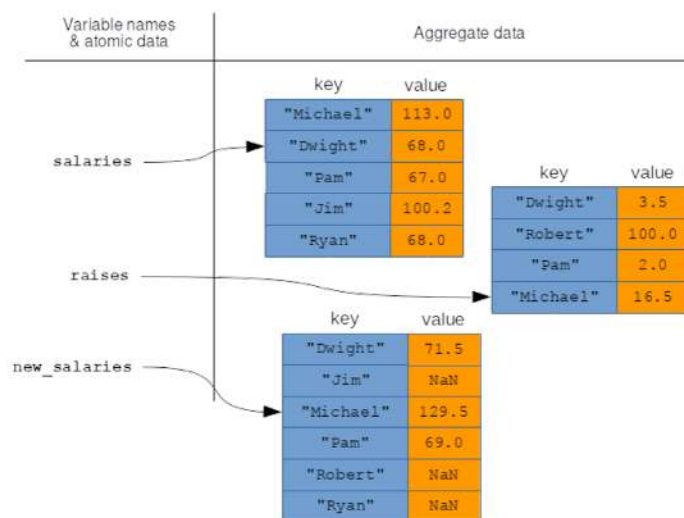


Figure 14.4.2: The result of **+**ing two **Serieses** that don't have all the same keys.

```
| Dwight 71.5
| Jim NaN
| Michael 129.5
| Pam 69.0
| Robert NaN
| Ryan NaN
| dtype: float64
```

Convince yourself that **Dwight**'s \$68,000 salary got added to his \$3,500 raise, that **Michael**'s \$113,000 salary was added to his \$16,500 raise, *etc.*

Don't get freaked out by those **NaN** entries just yet. The special value "**NaN**" stands for "**not a number**," and basically means that Pandas has to throw up its hands in that case. And with good cause. Jim has a current salary of \$100,200 in the first Series, but has no value at all in the second one (no raise for Jim this year? Haven't decided what his raise will be yet? Something else?) So Pandas does the safe thing, shrugs, and says "dunno." We say that the **Jim** entry in the **new_salaries Series** is a **missing value**. The same is true for **Robert** and **Ryan**, each of whom was present in only one of the two operands.

Now I know what you're thinking: "can't Pandas just assume the salary and/or raise is 0 if there's a missing one?" The answer is that yes it can, but it won't do so unless you give the go-ahead. Pandas is being cautious here, and doesn't want to introduce errors into your data stream by false assumptions. (Maybe in your company, for instance, there's a default entry-level salary that every employee receives who's unspecified in the **salary Series**. Or maybe the yearly raise is always assumed to be a flat 2.5% cost-of-living raise unless explicitly specified.)

If we do want Pandas to assume a certain default value, we have to change tactics a bit and go with the **add()** function (or **sub()**, **mul()**, or **div()**):

```
new_salaries = pd.Series.add(salaries, raises, fill_value=0)
print(new_salaries)
| Dwight 71.5
| Jim 100.2
| Michael 129.5
| Pam 69.0
| Robert 100.0
| Ryan 68.0
| dtype: float64
```

The **fill_value** argument is the important one here: it specifies what default value to use if one of the addends is missing a key from the other. Now the result is as in Figure 12.2.3. You can, of course, choose a **fill_value** other than zero, if you wish.

As with NumPy arrays, we can add (or subtract, or multiply, ...) a single atomic value to a series as well:

```
cost_of_living_increase = salaries * .025
print(cost_of_living_increase)
```

Figure 14.4.3: Using **add()** instead, and passing a **fill_value**.

```
| Micheal 2.825
```

```
|Dwight 1.700
|Pam 1.675
|Jim 2.505
|Ryan 1.700
|dtype: float64
```

```
salaries = salaries + cost_of_living_increase
print(salaries)
|Micheal 115.825
|Dwight 69.700
|Pam 68.675
|Jim 102.705
|Ryan 69.700
|dtype: float64
```

It can sometimes be useful to do string concatenation as well, for instance if we had employee first names and last names in two **Serieses** with their employee ID as the index:

```
firsts = pd.Series(['Hannibal', 'Clarice', 'Multiple', 'Buffalo'], index=[666, 1993, 47, 988])
lasts = pd.Series(['Starling', 'Crawford', 'Lecter', 'Bill', 'Miggs'], index=[1993, 1650, 666, 988, 47])
print(firsts + " " + lasts)
|47 Multiple Miggs
|666 Hannibal Lecter
|988 Buffalo Bill
|1650 NaN
|1993 Clarice Starling
|dtype: object
```

This page titled [14.4: Vectorized Arithmetic Operators](#) 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.

- [12.2: Vectorized Arithmetic Operators](#) by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

14.5: Sorting Serieses

Sorting is slightly more complex than for arrays, since there are two things we might want to sort by: the Series' index, or the values themselves. Correspondingly, there are two methods: `.sort_index()` and `.sort_values()`:

```
print(anti_vamps.sort_index())
```

```
| Buffy 120  
| Rubert 150  
| Willow 200  
| Xander 72  
| dtype: int64
```

```
print(anti_vamps.sort_values())
```

```
| Xander 72  
| Buffy 120  
| Rubert 150  
| Willow 200  
| dtype: int64
```

Like NumPy's `np.sort()` function (but unlike its `.sort()` method; refer back to Section 9.5), neither of these methods actually sort the **Series** in place; instead, they return sorted copies. However, they can be made to work in place, by including “`inplace=True`” as an argument:

```
heroes_dumb_to_smart = anti_vamps.sort_values()
```

```
print(heroes_dumb_to_smart)
```

```
| Xander 72  
| Buffy 120  
| Rubert 150  
| Willow 200  
| dtype: int64
```

```
print(anti_vamps)
```

```
| Buffy 120  
| Xander 72  
| Willow 200  
| Rubert 150  
| dtype: int64
```

```
anti_vamps.sort_values(inplace=True)
print(anti_vamps)
|Xander 72
|Buffy 120
|Rubert 150
|Willow 200
|dtype: int64
```

Another useful feature of both `.sort_X` methods is the ability to *reverse* sort. By adding “**ascending=False**” as an argument (with or without also including the “**inplace=True**” argument; they are combinable with a comma) you produce the reverse order:

```
heroes_smart_to_dumb = anti_vamps.sort_values(ascending=False)
print(heroes_smart_to_dumb)
|Willow 200
|Rubert 150
|Buffy 120
|Xander 72
|dtype: int64
```

```
anti_vamps.sort_index(inplace=True, ascending=False)
print(anti_vamps)
|Xander 72
|Willow 200
|Rubert 150
|Buffy 120
|dtype: int64
```

This page titled [14.5: Sorting Serieses](#) 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.

- [12.4: Sorting Serieses](#) by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

14.6: Queries

One of the most powerful things we'll do with a data set is to **query** it. This means that instead of specifying (say) a particular key, or something like “the minimum” or “the maximum,” we provide our own custom criteria and ask Pandas to give us all values that **match**. This kind of operation is also sometimes called **filtering**, because we're taking a long list of items and sifting out only the ones we want.

The syntax is interesting: you still use the boxies (like you do when giving a specific key) but inside the boxies you put a **condition** that will be used to select elements. It's best seen with an example. Re-using the **understanding** variable from above, we can query it and ask for all the elements greater than 5:

```
more_than_five = understanding[understanding > 5]
print(more_than_five)
| 4 15
| 2 13
| 9 7
| dtype: int64
```

The new thing here is the “**understanding > 5**” thing inside the boxies. The result of this query is itself a **Series**, but one in which everything that doesn't match the condition is filtered out. Thus we only have three elements instead of five. Notice the keys didn't change, and they also had nothing to do with the query: our query was about *values*.

We could change this, if we were interested in putting a restriction on *keys* instead, using the **.index** syntax:

```
index_more_than_five = understanding[understanding.index > 5]
print(index_more_than_five)
| 10 4
| 12 3
| 9 7
| dtype: int64
```

See how tacking on “**.index**” in the query made all the difference.

Query Operators

Now I have a surprise for you. It makes perfect sense to use the character “>” (called “greater-than,” “right-angle-bracket,” or simply “wakka”) to mean “greater than.” And the character “<” makes sense as “less than.” Unfortunately, the others don't make quite as much sense. See the top table in Figure 13.1.1.

“Greater/less than or equal to” isn't hard to remember, and it's a good thing Python doesn't require symbols like “≤” or “≥” since those are hard to find on your keyboard. You just type both symbols back-to-back, with no space. More problematic are the last two entries in the top table. The “!=” operator (pronounced “bangequals”) is used as a stand-in for “≠” which also isn't keyboard friendly. And that one doesn't have a good mnemonic; you just have to memorize it.

By far the most error-prone of this set is the “==” (**double-equals**) operator, which simply means “equals.” **Yes, you do have to use double-equals instead of single-equals in your queries, and yes it matters.** As additional incentive, let me inform you that if you use single-equals when you needed to use double-equals, *it will seem to work at first*, but you will silently get the wrong answer.

Memorize this fact! Failing to use double-equals is quite possibly the single most common programming error for beginners.

Simple query operators:

Symbol	Meaning
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	<i>not</i> equal to
==	equal to

Compound query operators:

Symbol	Meaning
&	and
	or
~	not

Figure 14.6.1: Query operators: simple and compound

Here are some more examples to test your understanding. Make sure you understand why each output is what it is.

```
understanding = pd.Series([15,4,13,3,7], index=[4,10,2,12,9])
```

```
print(understanding[understanding <= 7])
```

```
| 10 4
```

```
| 12 3
```

```
| 9 7
```

```
| dtype: int64
```

```
print(understanding[understanding != 13])
```

```
| 4 15
```

```
| 10 4
```

```
| 12 3
```

```
| 9 7
```

```
| dtype: int64
```

```
print(understanding[understanding == 3])
```

```
| 12 3
```

```
| dtype: int64
```

```
print(understanding[understanding.index >= 9])
```

```
| 10 4
```

```
| 12 3
```

```
| 9 7
| dtype: int64
| dtype: int64
```

Compound Queries

Often, your query will involve more than one criterion. This is called a **compound condition**. It's not as common with **Serieses** as it will be with **DataFrames** in a couple chapters, but there are still uses for it here.

Suppose I want all the key/value pairs of **understanding** where the value is *between 5 and 14*. This is really two conditions masquerading as one: we want all pairs where (1) the value is greater than 5, **and** also (2) the value is less than 14. I put the word “**and**” in boldface in the previous sentence because that's the operator called for here. We only want elements in our results where *both* things are true, and therefore, we “and together the two conditions.” (“And” is being used as a verb here.)

The way to achieve this is as follows. The syntax is nutty, so pay close attention:

```
x = understanding[(understanding > 5) & (understanding < 14)]
print(x)
| 2 13
| 9 7
| dtype: int64
```

First, notice that we put each of our two conditions *in bananas* here. This is *not* optional, as it turns out: you'll get a non-obvious error message if you omit them. Second, see how we combined the two with the “&” operator from the bottom half of Figure 13.1.1. The result, then, was only the elements that satisfied *both* conditions.

It can be tricky to figure out whether you want an **and** or an **or**. Unfortunately they don't always correspond to their colloquial English usage. Let's see what happens if we switch the “&” symbol to a “|” (pronounced “pipe”):

```
y = understanding[(understanding > 5) | (understanding < 14)]
print(y)
| 4 15
| 10 4
| 2 13
| 12 3
| 9 7
| dtype: int64
```

You can see that we got everything back. That's because **or** means “only give me the elements where *either one* of the conditions, or both, are true.” In this case, this is guaranteed to match everything, because if you think about it, *every* number is either greater than five, or less than fourteen, or both. (Think deeply.)

Even though in this example it didn't do anything exciting, an “**or**” does sometimes return a useful result. Consider this example:

```
z = understanding[(understanding.index > 10) | (understanding > 5)]
print(z)
| 4 15
```

```
| 2 13
| 12 3
| dtype: int64
```

Here we're asking for all key/value pairs in which *either* the key is greater than ten, or the value is greater than ten, or both. This reeled in exactly three fish as shown above. If we changed this “|” to an “&”, we'd have caught *no* fish. (Take a moment to convince yourself of that.)

The last entry in Figure 13.1.1 is the “~” sign, which is pronounced “tilde,” “twiddle,” or “squiggle.” It corresponds to the English word **not**, although in an unusual place in the sentence. Here's an example:

```
a = understanding[~(understanding.index > 10) | (understanding > 10)]
print(a)
| 4 15
| 10 4
| 2 13
| 9 7
| dtype: int64
```

Search for and stare at the squiggle in that line of code. In English, what we said was “give me elements where either the key is *not* greater than ten, or the value is greater than ten, or both.” The four matching elements are shown above.

Changing the “or” back to an “and” here gives us this output instead:

```
b = understanding[~(understanding.index > 10) & (understanding > 10)]
print(b)
| 4 15
| 2 13
| dtype: int64
```

These are the only two rows where *both* conditions are true (and remember that the first one is “not-ted.”)

It can be tricky to get compound queries right. As with most things, it just takes some practice.

Queries on strings

So far our examples have involved only numbers. Pandas also lets us perform queries on text data, specifying constraints on such things as the length of strings, letters in certain positions, and case (upper/lower).

Let's return to the Marvel-themed series from section 11.1:

```
alter_egos = pd.Series(['Hulk', 'Spidey', 'Iron Man', 'Thor'], index=
['Bruce', 'Peter', 'Tony', 'Thor'])
```

By appending “.str” to the end of the variable name, we can get access to most of the string-based methods we'd like to use. For instance, find all the values with exactly four letters:

```
four_letter_names = alter_egos[alter_egos.str.len() == 4]
```

```
print(four_letter_names)
```

```
| Bruce Hulk
```

```
| Thor Thor
```

```
| dtype: object
```

or all the values that contain a space:

```
spaced_out = alter_egos[alter_egos.str.contains(' ')]
```

```
print(spaced_out)
```

```
| Tony Iron Man
```

```
| dtype: object
```

or all the keys whose first character is a **T**:

```
to_a_tee = alter_egos[alter_egos.index.str.startswith('T')]
```

```
print(to_a_tee)
```

```
| Tony Iron Man
```

```
| Thor Thor
```

```
| dtype: object
```

or all entries where either the value is greater than five letters long or the key is the same as the value:

```
huh = alter_egos[(alter_egos.str.len() > 5) | (alter_egos.index == alter_egos)]
```

```
print(huh)
```

```
| Peter Spidey
```

```
| Thor Thor
```

```
| dtype: object
```

The possibilities are endless. Some of the more common functions are summarized in Figure 13.1.2.

Function	Description
<code>ser.str.len()</code>	Set a condition on the length of a string.
<code>ser.str.startswith(str)</code>	Request only strings that begin with certain letter(s).
<code>ser.str.endswith(str)</code>	Request only strings that end with certain letter(s).
<code>ser.str.contains(str)</code>	Request only strings that contain certain letter(s) somewhere in them.
<code>ser.str.isupper()</code>	Request only strings that are in all upper-case.
<code>ser.str.islower()</code>	Request only strings that are in all lower-case.

Figure 14.6.2: Common query methods for string data.

Last word

A couple things before we move on. You've noticed that in all the above examples, it was necessary to type the **Series** variable name several times:

```
understanding[(understanding < 12) | (understanding > 18)]  
alter_egos[(alter_egos.str.isupper()) & (alter_egos.str.len() < 10)]
```

There's really no way around that, sorry; you just have to get used to it. A very common beginner error is to try and write this:

```
understanding[understanding < 12 | > 18]
```

This seems to make perfect sense, especially since it mimics the natural English sentence: "give me all values where **understanding** is *less than 12 or greater than 18*." Unfortunately, it doesn't work like that in Python. The rule is: each side of an and or an or must be a complete sentence. The phrase "**understanding** is greater than 18" counts as a complete sentence, but "is greater than 18" does not.

Also, whenever I see a line of code that specifies a key to a **Series** (or array), I mentally pronounce the opening boxie ("[" as the word "of". So when I read:

```
print(x[5])
```

I say to myself "print **x of** five."

However, whenever I see a *query*, I mentally pronounce the boxie as the word "where". So when I read:

```
print(x[x > 12])
```

I say to myself "print **x where** x is greater than 12." I've found this helpful in making sense of the meaning of queries, since they're complicated enough as it is!

This page titled [14.6: Queries](#) 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.

- **13.1: Queries** by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

14.7: Summary

All the functions from this chapter are summarized in Figure 12.6.1.

Function	Description
<code>len(<i>ser</i>)</code>	Get the number of key/value pairs in the Series <i>ser</i> .
<code><i>ser</i>['Five Guys']</code>	Get the value of a specific key from the Series <i>ser</i> .
<code><i>ser</i>.iloc[73]</code>	Treating the key/values pairs in the Series <i>ser</i> as ordered, get a specific numbered (from 0) value.
<code><i>ser</i>.index[73]</code>	Treating the key/values pairs in the Series <i>ser</i> as ordered, get a specific numbered (from 0) key.
<code><i>ser</i>['Firehouse'] = ...</code>	Set the value for a key of the Series <i>ser</i> .
<code><i>ser</i>['New Rest'] = ...</code>	Add an additional key/value pair to the Series <i>ser</i> . (Same syntax as the previous.)
<code><i>ser</i> + 13</code>	Add a quantity to each value of <i>ser</i> , yielding a new Series . (Also works with <code>-</code> , <code>*</code> , <code>/</code> , <i>etc.</i>)
<code><i>ser1</i> + <i>ser2</i></code>	Add pairs of values that have matching keys in two Serieses , yielding a new Series . Use <code>NaN</code> for the value of any key that doesn't appear in both <i>ser1</i> and <i>ser2</i> . (Also works with <code>-</code> , <code>*</code> , <code>/</code> , <i>etc.</i>)
<code>pd.Series.add(<i>ser1</i>, <i>ser2</i>, fill_value=<i>x</i>)</code>	Add pairs of values that have matching keys in two Serieses , yielding a new Series . Use <i>x</i> for any missing values. (Also works with <code>sub()</code> , <code>mul()</code> , <code>div()</code> , <i>etc.</i>)
<code><i>ser1</i> = <i>ser2</i></code>	Make <i>ser1</i> point to the same data that <i>ser2</i> points to. (<i>Not</i> a copy!)
<code><i>ser1</i> = <i>ser2</i>.copy()</code>	Make <i>ser1</i> point to a new, independent copy of <i>ser2</i> .
<code><i>ser</i>.sort_index()</code>	Return a copy of the Series <i>ser</i> which is sorted by the keys. Can also pass <code>"inplace=True"</code> to change <i>ser</i> itself, and/or pass <code>"ascending=False"</code> to get reverse order.
<code><i>ser</i>.sort_values()</code>	Same as above, except that sorting is done with respect to values, not keys.
<code><i>ser1</i>.append(<i>ser2</i>)</code>	Return a new Series with <i>ser1</i> 's and <i>ser2</i> 's key/value pairs smooshed together. (Bad things may happen if <i>ser1</i> and <i>ser2</i> share some of the same keys.)

Figure 14.7.1: Handy functions, methods, and operators for Pandas **Serieses**.

This page titled [14.7: Summary](#) 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.

- [12.6: Summary](#) by [Stephen Davies](#) is licensed [CC BY-SA 4.0](#). Original source: <https://eng.libretexts.org/api/deki/files/58148/crystal1.pdf>.

CHAPTER OVERVIEW

15: Pandas Dataframes

- 15.1: Dataframe basics
- 15.2: Accessing elements
- 15.3: Groupby()
- 15.4: Math operations

15: Pandas Dataframes is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

15.1: Dataframe basics

In the realm of data analysis, the pandas DataFrame is a cornerstone structure, a tabular arrangement akin to a spreadsheet or SQL table. A DataFrame essentially consists of multiple pandas Series that share a common index. For illustration, consider a gradebook where the index represents the names of the students, and the columns denote various assignments. Each column is, in essence, a pandas Series, compiled together under a single DataFrame.

	Quiz1	Quiz2	Quiz3	Homework1	Homework2	Exam1	Exam2
Alice	96	74	83	90	93	91	92
Bob	88	99	96	99	77	95	83
Charlie	85	96	98	79	92	75	94
David	80	73	83	97	92	83	97
Emily	83	99	72	83	79	88	87
Frank	76	80	73	87	88	83	84
Grace	85	70	95	79	82	81	75
Helen	75	97	71	86	88	75	92
Isaac	89	82	99	77	93	75	78
Jill	89	72	94	76	80	78	79

The index is the column of names and the other columns are grades for an assignment.

Creating a DataFrame can be done in numerous ways. For example, I generated the DataFrame above with this code:

```
names = ['Alice', 'Bob', 'Charlie', 'David', 'Emily', 'Frank', 'Grace', 'Helen', 'Isaac', 'Jill']

# Define the column names
columns = ['Quiz1', 'Quiz2', 'Quiz3', 'Homework1', 'Homework2', 'Exam1', 'Exam2']

# Generate random integers between 70 and 100 for the values
values = np.random.randint(70, 101, size=(len(names), len(columns)))

# Create the DataFrame
df = pd.DataFrame(values, index=names, columns=columns)
```

Alternatively, let's say you have two Series, each representing scores from different exams, "Exam1" and "Exam2." The index for each Series contains the names of the students.

Here's how you could create these Series:

```
1 # Create two pandas Series with names as the index
2 exam1 = pd.Series([85, 90, 78, 92], index=['Alice', 'Bob', 'Charlie', 'David'])
3 exam2 = pd.Series([88, 84, 76, 95], index=['Alice', 'Bob', 'Charlie', 'David'])
```

```
# Concatenate Series into a DataFrame
df = pd.concat([exam1, exam2], axis=1, keys=['Exam1', 'Exam2'])
```

df is now:

	Exam1	Exam2
Alice	85	88
Bob	90	84
Charlie	78	76
David	92	95

Let's talk about the "axis=1" keyword in the `concat()` command. In the pandas library, the "axis" determines the direction in which an operation is performed. This is a fundamental concept that not only applies to concatenation but also to other operations like summation, mean calculation, and data manipulation. The axis parameter can essentially take on one of two values: `0` or `1`.

With `axis=1`, the columns are placed side by side, as in the example above.

If axis=0, then the Series would be concatenated end-to-end, into one long column:

	Score
Student	
Alice	85
Bob	90
Charlie	78
David	92
Alice	88
Bob	84
Charlie	76
David	95

15.1: Dataframe basics is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

15.2: Accessing elements

Accessing elements within a DataFrame is a crucial aspect of data manipulation and analysis, and the pandas library offers a robust set of tools for this purpose. The `.iloc` and `.loc` attributes serve as the primary mechanisms for data retrieval. They each have their own set of capabilities, dictated by the type of index you're working with — integer-based for `.iloc` and label-based for `.loc`.

Let's consider a simple DataFrame that has names as row indices and exam scores as columns:

```
data = {'Exam1': [85, 90, 78], 'Exam2': [88, 84, 76]}
index = ['Alice', 'Bob', 'Charlie']
df = pd.DataFrame(data, index=index)
```

This produces:

```
In [13]: df
Out[13]:
```

	Exam1	Exam2
Alice	85	88
Bob	90	84
Charlie	78	76

Using `.iloc`, which relies on integer-based positioning, you could access Alice's Exam1 score (which is 85) with:

```
alice_exam1_score = df.iloc[0, 0]
```

Here, the first `0` refers to the row (Alice), and the second `0` refers to the column (Exam1). The value returned is 85.

Conversely, `.loc` operates with label-based indices. To access the same score using `.loc`, you would use:

```
alice_exam1_score = df.loc['Alice', 'Exam1']
```

This produces the same thing, 85.

The `.iloc` and `.loc` attributes are incredibly flexible. For instance, if you only specify a row, either by its integer index with `.iloc` or its label with `.loc`, pandas will return all columns for that row:

```
alice_all_scores = df.loc['Alice']
```

```
alice_all_scores = df.iloc[0]
```

Interestingly, the output in this case would be a Series object, where the original DataFrame's column names serve as the Series' index:

```
Exam1    85
Exam2    88
Name: Alice, dtype: int64
```

Both `.iloc` and `.loc` allow slicing and more complex queries, facilitating the retrieval of multiple rows and columns at once. For example, to get the Exam1 scores for both Alice and Bob, you could use:

```
alice_bob_exam1 = df.loc[['Alice', 'Bob'], 'Exam1']
alice_bob_exam1 = df.iloc[0:2, 0]
```

Accessing columns in a DataFrame is a routine operation, and pandas offers two primary ways to accomplish this: using square brackets `[]` or the dot notation `.`. The square brackets are incredibly versatile, allowing for both label-based and, occasionally, integer-based indexing, making them compatible with a wide array of DataFrame configurations. For instance, if your DataFrame `df` has a column named 'Exam1', you could access this column by executing `df['Exam1']`. This method works regardless of whether the column name includes spaces or other special characters. Moreover, square brackets facilitate the selection of multiple columns. You can pass a list of column names like `df` to retrieve multiple columns, which will be returned as a DataFrame.

In contrast, the dot notation provides a quicker but somewhat restricted way to access DataFrame columns. Utilizing dot notation involves appending the column name to the DataFrame object as an attribute: `df.Exam1`. While this is certainly more concise and

often more readable, there are limitations to be aware of. Firstly, this approach is feasible only when the column name is a valid Python identifier, meaning it doesn't contain spaces, dots, or other special characters. Secondly, dot notation is not applicable for variable column names. For example, if the column name is stored in a variable, you would have to use the square bracket notation. Lastly, dot notation doesn't allow for the same complex querying or selection of multiple columns in the way that square brackets do.

15.2: Accessing elements is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

15.3: Groupby()

The `.groupby()` method in Pandas is a cornerstone for data aggregation and summarization tasks. Think of it as the Swiss Army knife for slicing, dicing, and summarizing datasets. It allows you to segment your data into meaningful groups, apply a function to each group independently, and then combine the results, all in a few lines of code. This is a fundamental operation in data analytics, enabling you to extract valuable insights from raw data efficiently.

Let's generate a DataFrame with some random data about 8th graders:

```
# Generate random names as the index
random_names = ['Alice', 'Bob', 'Charlie', 'David', 'Emily', 'Frank', 'Grace',
                'Helen', 'Isaac', 'Jill']

# Generate random gender based on the names
gender = ['Girl', 'Boy', 'Boy', 'Boy', 'Girl', 'Boy', 'Girl', 'Girl', 'Boy', 'Girl']

# Generate random weights and heights
weights = np.random.randint(120, 151, size=10)
heights = np.random.randint(40, 61, size=10)

# Generate random eye colors
eye_colors = np.random.choice(['Blue', 'Brown'], size=10)

# Create the DataFrame
df_demo = pd.DataFrame({
    'Gender': gender,
    'Weight': weights,
    'Height': heights,
    'Eye Color': eye_colors
}, index=random_names)
```

It generates this DataFrame:

	Gender	Weight	Height	Eye Color
Alice	Girl	122	60	Blue
Bob	Boy	124	58	Brown
Charlie	Boy	138	44	Brown
David	Boy	126	46	Brown
Emily	Girl	143	55	Brown
Frank	Boy	148	54	Blue
Grace	Girl	121	40	Brown
Helen	Girl	142	44	Brown
Isaac	Boy	124	57	Blue
Jill	Girl	149	49	Blue

The "Gender" column specifies whether the individual is a boy or a girl, the "Weight" column contains random weights between 120 and 150 pounds, and the "Height" column features random heights between 40 and 60 inches, and "Eye Color" is randomly selected to be "Blue" or "Brown".

Now, let's explore how to use `.groupby()` to obtain some insights from this DataFrame. We'll start by grouping the data by gender and calculating the average weight and height for each group.

```
grouped_by_gender = df_demo.groupby('Gender').mean()
```

The line of code `grouped_by_gender = df_demo.groupby('Gender').mean()` performs two key operations on the `df_demo` DataFrame. First, it groups the DataFrame by the "Gender" column, segregating the data into subsets based on the unique gender labels ('Boy' and 'Girl'). This is the "splitting" phase in the `groupby` operation. Second, it calculates the mean for each numerical column within these gender-based groups. In our example, it computes the average "Weight" and "Height" for both boys and girls. The result is stored in a new DataFrame called `grouped_by_gender`, where the index consists of the unique gender labels and the columns contain the corresponding average weights and heights. This one-liner essentially condenses the original DataFrame into a summary table, providing a quick snapshot of how the numerical attributes differ by gender:

	Weight	Height
Gender		
Boy	138.2	52.8
Girl	143.6	49.8

It's worth noting that the "Eye Color" column is automatically excluded from the mean calculation since it's a non-numerical column. This is one of the conveniences of using `.groupby()`; it can smartly handle different data types while performing calculations.

We can, of course, group by eye color:

```
# Group the df_demo DataFrame by 'Eye Color' and calculate the mean for numerical columns
grouped_by_eye_color = df_demo.groupby('Eye Color').mean()
```

This produces the DataFrame:

	Weight	Height
Eye Color		
Blue	138.250000	48.0
Brown	142.666667	53.5

This example illustrates the power of `.groupby()` for quickly summarizing data based on specific categories. With just a single line of code, we've gained a valuable understanding of how weight and height differ between gender and eye color in this dataset. This kind of data summarization is crucial for a myriad of applications, from scientific research to business analytics.

15.3: `Groupby()` is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

15.4: Math operations

Pandas equips you with an extensive array of methods to perform calculations on DataFrames. These built-in methods simplify the process of data analysis, allowing you to focus on interpreting results rather than writing complex algorithms from scratch.

In this example, let's use the `.mean()` method. In Chapter 14.1, we introduced a DataFrame with grades in it named `df`:

	Quiz1	Quiz2	Quiz3	Homework1	Homework2	Exam1	Exam2
Alice	96	74	83	90	93	91	92
Bob	88	99	96	99	77	95	83
Charlie	85	96	98	79	92	75	94
David	80	73	83	97	92	83	97
Emily	83	99	72	83	79	88	87
Frank	76	80	73	87	88	83	84
Grace	85	70	95	79	82	81	75
Helen	75	97	71	86	88	75	92
Isaac	89	82	99	77	93	75	78
Jill	89	72	94	76	80	78	79

We can apply the `.mean()` function two primary ways, depending on what kind of average you're interested in—either by student or by assignment. The `axis` keyword in the `.mean()` function specifies this behavior.

Setting `axis=0` (the default) calculates the average of the column, in this case for each assignment:

	Quiz1	Quiz2	Quiz3	Homework1	Homework2	Exam1	Exam2
Alice	96	74	83	90	93	91	92
Bob	88	99	96	99	77	95	83
Charlie	85	96	98	79	92	75	94
David	80	73	83	97	92	83	97
Emily	83	99	72	83	79	88	87
Frank	76	80	73	87	88	83	84
Grace	85	70	95	79	82	81	75
Helen	75	97	71	86	88	75	92
Isaac	89	82	99	77	93	75	78
Jill	89	72	94	76	80	78	79



Avg. down columns

It produces:

	Average Score
Assignment	
Quiz1	84.6
Quiz2	84.2
Quiz3	86.4
Homework1	85.3
Homework2	86.4
Exam1	82.4
Exam2	86.1

Conversely, setting `axis=1` calculates the average for each student, providing a snapshot of individual performance across all assignments.

	Quiz1	Quiz2	Quiz3	Homework1	Homework2	Exam1	Exam2
Alice	96	74	83	90	93	91	92
Bob	88	99	96	99	77	95	83
Charlie	85	96	98	79	92	75	94
David	80	73	83	97	92	83	97
Emily	83	99	72	83	79	88	87
Frank	76	80	73	87	88	83	84
Grace	85	70	95	79	82	81	75
Helen	75	97	71	86	88	75	92
Isaac	89	82	99	77	93	75	78
Jill	89	72	94	76	80	78	79

→ Avg. across rows

It produces:

	Average Score
Student	
Alice	88.428571
Bob	91.000000
Charlie	88.428571
David	86.428571
Emily	84.428571
Frank	81.571429
Grace	81.000000
Helen	83.428571
Isaac	84.714286
Jill	81.142857

15.4: Math operations is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

16: Date and time variables

In climate and atmospheric sciences, using time variables in data analysis is essential. Weather and climate are not static; they change over time. So, if you're trying to study something like global temperature rise or patterns of rainfall, you need to take into account how these elements change over days, months, or even years.

But time is a hard variable. The reason lies in the complex and irregular structure of time itself. Unlike other variables, e.g., latitude, that may be more straightforward to quantify, time comes with layers of complication. For example, not all years have the same number of days; we switch between 365 and 366 days depending on whether it's a leap year or not. Months vary in length from 28 to 31 days, adding another layer of complexity to data collected on a monthly basis.

Beyond this, each day consists of 24 hours, each hour has 60 minutes, and each minute has 60 seconds. This hierarchical and non-uniform structure of time introduces a level of complexity that researchers have to consider when designing their studies. Say you're looking at hourly data for air temperature. If February's data looks different than March's, is that because of some atmospheric condition or simply because February often has fewer days?

In this chapter, we will introduce you to analysis of time in Python, with an emphasis on how it is used in Pandas.

[16.1: np.datetime64](#)

[16.2: Date and time in Pandas](#)

[16.3: Extracting components of date and time](#)

[16.4: Setting the index to be date and time](#)

[16.5: Resampling](#)

[16.6: Other resources](#)

16: Date and time variables is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

16.1: np.datetime64

In Python, time is normally stored in a specialized data type, `np.datetime64`, provided by the NumPy library. Unlike conventional data types like integers (`int`), floating-point numbers (`float`), or strings (`str`), `np.datetime64` is designed specifically to handle dates and times. It offers capabilities that make it easy to capture various time scales, from nanoseconds all the way to centuries. It also integrates seamlessly with NumPy's array operations, making it more convenient for tasks like time-based indexing, resampling, and aggregation — and, as we'll see later this chapter, it also integrates with Pandas. This is particularly useful in climate and atmospheric sciences, where time series data often require sophisticated manipulation.

Creating a date using NumPy's `np.datetime64` data type is relatively straightforward. You can create a `datetime64` object by invoking the data type and passing in a string that represents the date you're interested in.

Here's a simple example:

```
import numpy as np
my_date = np.datetime64('2023-10-26')
```

In this example, `my_date` is now a `np.datetime64` object representing October 26, 2023 (you can test this with `type(my_date)`). You can also specify the time down to the nanosecond if that level of detail is needed. For example:

```
my_datetime = np.datetime64('2023-10-26T12:34:56.789')
```

Here, `my_datetime` represents the date and time down to the millisecond for October 26, 2023, at 12:34:56.789.

Because `np.datetime64` is integrated with the NumPy library, it is generally compatible with NumPy's array operations. This makes it useful for performing complex operations on time series data, which is often a requirement in climate and atmospheric sciences.

16.1: `np.datetime64` is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

16.2: Date and time in Pandas

Because much atmospheric and climate data are time series, you will frequently use date and time variables in data analysis.

First, you can get a Pandas series of regularly spaced date and times using the `pd.date_range()` function. The function's flexibility is one of its strong suits. You can specify a start and end date, and even set a frequency for the range, such as 'D' for daily or 'M' for monthly intervals. For example, to create a date range from January 1, 2023, to January 5, 2023, with daily intervals, you would write:

```
import pandas as pd
date_range = pd.date_range(start='2023-01-01', end='2023-01-05', freq='D')
```

This would generate a series with dates from January 1 to January 5 (note that the series includes the end date). It's actually a data structure called a `DatetimeIndex`, but you don't have to worry too much about that. Internally, Pandas uses `np.datetime64` to represent times and dates within these data structures.

Another valuable function in Pandas for working with dates is `pd.to_datetime()`. This function converts arguments to datetime format, allowing you to transform a wide array of date representations into a standardized datetime object. It can accept a single date string, a list, or even an entire Pandas Series. This is particularly useful when you're dealing with raw data that may present dates in inconsistent formats. For example:

```
single_date = pd.to_datetime('2023-01-01')
date_list = pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03'])
```

In the first example, `single_date` will hold a single datetime object for January 1, 2023. In the second example, `date_list` will contain a `DatetimeIndex` with the three specified dates.

Both `pd.date_range()` and `pd.to_datetime()` are essential tools for working with time series data.

16.2: Date and time in Pandas is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

16.3: Extracting components of date and time

If you have a Pandas DataFrame with a column of ``np.datetime64`` type, you can easily extract various parts of the date, like the year, month, day, and hour.

To do this, first ensure that the column is in datetime format and, if not, use ``pd.to_datetime()`` to convert it. Once that's done, you can use the ``dt`` accessor followed by the attribute you're interested in.

For example, if your DataFrame is named ``df`` and the datetime column is named ``timestamp``, you can access the year by using ``df['timestamp'].dt.year``, the month with ``df['timestamp'].dt.month``, the day with ``df['timestamp'].dt.day``, and the hour with ``df['timestamp'].dt.hour``. These commands will return a new Pandas Series containing just the year, month, day, or hour, which you can then use for further analysis or operations.

Here is an example. First, let's define a DataFrame called ``df``:

```
import pandas as pd
import numpy as np

# Create a sample DataFrame with np.datetime64 data
df = pd.DataFrame({
    'timestamp': pd.date_range(start='2021-01-01', end='2021-01-05', freq='12H'),
    'value': np.random.random(9)
})
```

In this case, `freq = '12H'` means that the frequency is every 12 hours. The DataFrame looks like:

RESULT		timestamp	value
0	2021-01-01	00:00:00	0.040253
1	2021-01-01	12:00:00	0.603451
2	2021-01-02	00:00:00	0.176442
3	2021-01-02	12:00:00	0.879241
4	2021-01-03	00:00:00	0.786113
5	2021-01-03	12:00:00	0.332471
6	2021-01-04	00:00:00	0.663636
7	2021-01-04	12:00:00	0.680227
8	2021-01-05	00:00:00	0.392615

Let's add some new columns that contain the year, month, day, and hour of the timestamp:

```
# Extract year, month, day, and hour
df['year'] = df['timestamp'].dt.year
df['month'] = df['timestamp'].dt.month
df['day'] = df['timestamp'].dt.day
df['hour'] = df['timestamp'].dt.hour
```

Now the DataFrame looks like:

RESULT						
	timestamp	value	year	month	day	hour
0	2021-01-01 00:00:00	0.040253	2021	1	1	0
1	2021-01-01 12:00:00	0.603451	2021	1	1	12
2	2021-01-02 00:00:00	0.176442	2021	1	2	0
3	2021-01-02 12:00:00	0.879241	2021	1	2	12
4	2021-01-03 00:00:00	0.786113	2021	1	3	0
5	2021-01-03 12:00:00	0.332471	2021	1	3	12
6	2021-01-04 00:00:00	0.663636	2021	1	4	0
7	2021-01-04 12:00:00	0.680227	2021	1	4	12
8	2021-01-05 00:00:00	0.392615	2021	1	5	0

You can also use this syntax to extract rows whose dates or times satisfy some constraint. For example, if you want to extract the rows for which hour = 12, you could use this:

```
filtered_df = df[df['timestamp'].dt.hour == 12]
```

16.3: Extracting components of date and time is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

16.4: Setting the index to be date and time

You will often want to make a the date and time into the index. The `.set_index()` method in Pandas serves a critical role in organizing your DataFrame by setting one of your columns as the index. By setting time as the index, you facilitate a wide range of operations that become both more intuitive and efficient. For instance, you can easily filter data to specific time periods, resample at different frequencies, or carry out rolling calculations like moving averages. It essentially allows the DataFrame to become "aware" that it is dealing with time-ordered data, paving the way for specialized time series functionalities.

Here's a simple example to demonstrate the concept:

```
import pandas as pd
```

```
# Create a DataFrame with a date column
df = pd.DataFrame({
    'Date': pd.date_range(start='2023-01-01', end='2023-01-05', freq='D'),
    'Temperature': [30, 32, 31, 29, 28]
})

# Set the date column as the index
df = df.set_index('Date')
```

Now, the DataFrame `df` has 'Date' as its index:

Temperature	
Date	
2023-01-01	30
2023-01-02	32
2023-01-03	31
2023-01-04	29
2023-01-05	28

This makes time-based operations straightforward. For example, extracting all records for January becomes a simple slice operation: `df.loc['2023-01-01':'2023-01-31']`.

By setting the time variable as the DataFrame's index, you're effectively streamlining subsequent analyses and manipulations that rely on this chronological ordering. This is why time often serves as an ideal index, particularly in fields that require meticulous analysis of time-based phenomena.

16.4: Setting the index to be date and time is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

16.5: Resampling

Resampling is a vital operation for changing the frequency of your observations. For example, you may have monthly data and want annual averages. Resampling can do this. Or you have daily data and you want hourly data. Resampling can do this, too.

Pandas makes this process straightforward through its built-in `.resample()` method. This method allows you to downsample or upsample your data easily. Downsampling involves reducing the frequency of the data points, say from minute-level to hourly data, and typically requires an aggregation method like mean, sum, or max to represent the new, less frequent data points. Upsampling, on the other hand, increases the frequency and uses interpolation to fill in the gaps.

For example, if you have hourly temperature data and you want the daily average temperature, you could use:

```
daily_avg_temp = df['Temperature'].resample('D').mean()
```

In this case, the `.resample('D')` function changes the frequency to daily, and the `.mean()` function calculates the average temperature for each day. If you want to know the daily maximum temperature, you could change the `.mean()` to `.max()`.

If you have daily data but need to fill in hourly values, you could use interpolation:

```
hourly_temp = df['Temperature'].resample('H').interpolate()
```

Here, the `.resample('H')` function changes the frequency to hourly, and the `.interpolate()` function fills in the hourly temperature values based on the available daily data.

Note that, in order to use this, the index must be a `DateTimeIndex` (e.g., a list of times).

16.5: Resampling is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

16.6: Other resources

Here are some additional resources for data/time.

[Pydata.org](https://pydata.org)

realpython.com

earthdatascience.org

16.6: Other resources is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

17: Xarray

17.1: Introduction

17.2: DataArrays

17.3: Selecting values

17.4: Datasets

17.5: Reading netCDF files

17.6: Groupby and Resample

17.7: Additional resources

17: Xarray is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

17.1: Introduction

Pandas is a great tool for handling one-dimensional data, like a time series from one meteorological station. However, atmospheric sciences often requires analysis of multidimensional data. Consider, for example, that you might want to analyze 4-dimensional model fields (latitude, longitude, pressure, and time), from several ensemble members of different models. This is ia a six-dimensional data structure (latitude, longitude, pressure, time, model, ensemble number). This is where Xarray comes int: it is expert at handling multidimensional data with precision and makes analysis of the data easy and intuitive.

You normally import Xarray with this command:

```
import xarray as xr
```

[17.1: Introduction](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

17.2: DataArrays

Xarray's DataArray is a counterpart to Pandas' Series. Much like a Series, a DataArray holds a single data set along with labeled dimensions, coordinates, and metadata. However, while a Series is limited to one-dimensional data, a DataArray can hold two or more dimensions, which is crucial for representing more complex datasets.

Consider a simple example where you're working with temperature data from various cities. An Xarray DataArray could hold this data like this:

```
import xarray as xr
import numpy as np, pandas as pd

data = np.array([[22, 20, 18], [18, 16, 14], [30, 32, 34]])
times = pd.date_range("2023-01-01", periods=3, freq='6H')
cities = ['New York', 'London', 'Mumbai']

temperature_data_array = xr.DataArray(data, coords=[cities, times], dims=['city',
'time'])
```

`temperature\_data\_array` is a two-dimensional DataArray with cities and times as coordinates.

Converting a Pandas Series into an Xarray DataArray is straightforward due to the compatibility between these two libraries. Each Series has an index, and during conversion, this index becomes a coordinate in the DataArray. Here's how you can perform this conversion:

Let's begin with a Pandas Series example:

```
# Create a Pandas Series with an index
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
```

To convert this Series into a DataArray, you can simply use the `xr.DataArray` constructor:

```
# Convert the Pandas Series to an Xarray DataArray
data_array = xr.DataArray(series)
```

After this conversion, the `data\_array` object is an Xarray DataArray. The labeled index 'a', 'b', 'c' from the Pandas Series becomes the coordinate for the DataArray, and the integer values become the data.

You can check the contents of the `data\_array` to confirm the structure:

```
print(data_array)
```

This command will show you the values, the coordinate, and the associated dimension.

Furthermore, if your Series includes a `DatetimeIndex` or any other type of specialized index, Xarray will handle this appropriately, maintaining the nature of the index as a coordinate in the DataArray. This feature is particularly useful for time series data, which is common in atmospheric and environmental sciences.

17.2: DataArrays is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

17.3: Selecting values

If you have the following DataArray:

```
import xarray as xr
import numpy as np, pandas as pd

data = np.array([[22, 20, 18], [18, 16, 14], [30, 32, 34]])
times = pd.date_range("2023-01-01", periods=3, freq='D')
cities = ['New York', 'London', 'Mumbai']

temperature_data_array = xr.DataArray(data, coords=[cities, times], dims=['city',
'time'])
```

You can access the data in an Xarray using labels, much like you would in a Pandas Series, using `.loc[]` and `.iloc[]`.

`.loc[]` uses labels, so you could access elements this way:

`temperature_data_array.loc['New York', '2023-01-02']` would return a value of 20. One issue here is that you have to know the order of the labels; if you reverse the order of labels, e.g., `temperature_data_array.loc['2023-01-02', 'New York']`, this will generate an error.

The `.iloc[]` command works exactly the same way, using the integer position to access the values. For example, we can access the same point (New York on 2023-01-02) with this command:

```
temperature_data_array.iloc[0,1]
```

A more common and convenient way of selecting parts of a DataArray is with the `.sel()` method:

```
temperature_data_array.sel(city='New York', time='2023-01-02')
```

This yields the same data point. There is also an `.isel()` method, which uses the index number to access the array:

```
temperature_data_array.isel(city=0, time=1)
```

This selects the same point, New York (element zero) and 2023-01-02 (element 1). This would retrieve the same data point as all the previous examples.

Note that `.sel()` and `.isel()` can be considered analogous to `.loc[]` and `.iloc[]`.

What if you want to get a range of points? You can use the `slice()` function with `.sel()` and `.isel()`. As an example, let's define a DataArray `ds`:

```
# Define the dimensions
latitudes = np.linspace(-90, 90, 181) # 181 points from -90 to 90 degrees
longitudes = np.linspace(-180, 180, 361) # 361 points from -180 to 180 degrees
times = pd.date_range('2023-01-01', periods=10, freq='D') # 10 days of data

# Create a 3D array of random data
data = np.random.rand(len(times), len(latitudes), len(longitudes))

# Create the DataArray
ds = xr.DataArray(data, coords=[('time', times), ('lat', latitudes), ('lon',
longitudes)])
```

Now let's pretend we want to select data for latitudes from -45 to 45, and for the first five days of January 2023.

```
selected_data = ds.sel(lat=slice(-45, 45), time=slice('2023-01-01', '2023-01-05'))
```

This will yield a subset of the original DataArray `ds` that is confined to the specified latitude and time slices. Since we don't include any slicing for longitudes, the resulting DataArray includes all longitudes.

To slice from the beginning or to the end of the array, use `None` as the endpoint. For example, if we want all latitudes from the beginning of the latitudes up to 45, we would use the command `slice(None, 45)` . If you want latitudes starting at 45 and going to the end of the latitudes, use the command `slice(45, None)` .

You can also use this with `.isel()` . For example, if we want elements 3 through 8 of latitudes and 2 through 5 of longitudes, we could use this syntax:

```
selected_data = ds.isel(lat=slice(3,9), lon=slice(2,6))
```

Note that our `.isel()` slice indices have an end number that is one larger than what we want: if we want elements 3-8, we need to `slice(3, 9)` . This only applies to `.isel()` ; for `.sel()` , the end value of the slice is included in the extracted data.

[17.3: Selecting values](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

17.4: Datasets

In pandas, a DataFrame can be thought of as a collection of Series objects that share a common index, allowing for the alignment of data across rows. This structure is exceedingly efficient for handling one-dimensional sequences of values, typically time series or other forms of sequentially indexed data, which are common in various domains including finance, economics, and simple observational studies.

Similarly, Xarray Datasets can be thought of as a collection of DataArrays with common coordinates, similar to the way a Pandas DataFrame is a collection of Series. Xarray's Dataset enables the alignment and joint operation of multiple DataArrays across their shared coordinates, providing a robust framework for managing the intricacies of multidimensional datasets, like those generated from climate models or satellite observations.

Here is one example of how you would create a Dataset:

```
import numpy as np
import pandas as pd
import xarray as xr

# Define the coordinates
latitudes = np.linspace(-90, 90, 181) # 181 points from South to North Pole
longitudes = np.linspace(-180, 180, 361) # 361 points for full longitude range
pressure_levels = np.array([1000, 850, 700, 500, 300, 200, 100]) # in hPa
times = pd.date_range('2005-01-01', '2005-12-01', freq='MS') # Monthly intervals for 2005

# Generate random data
temperature_data = np.random.rand(len(times), len(pressure_levels), len(latitudes),
len(longitudes))
precipitation_data = np.random.rand(len(times), len(pressure_levels), len(latitudes),
len(longitudes))

# Create the xarray Dataset
ds = xr.Dataset({
    'temperature': ([ 'time', 'pressure', 'latitude', 'longitude'], temperature_data),
    'precipitation': ([ 'time', 'pressure', 'latitude', 'longitude'],
precipitation_data)
}, coords={
    'time': times,
    'pressure': pressure_levels,
    'latitude': latitudes,
    'longitude': longitudes
})
```

In this example, the `xr.Dataset` constructor is used to create a new Dataset. The first argument is a dictionary that maps variable names to dimension names and data arrays. The `coords` argument is a dictionary that assigns coordinate labels to the dimensions. The `times` array generated with `pd.date_range` provides monthly intervals throughout the year 2005, while `latitudes` and `longitudes` are evenly spaced points on the globe, and `pressure_levels` represent typical atmospheric pressure levels in hPa.

Accessing the variables within an Xarray dataset is straightforward, similar to how one might interact with a dictionary in Python. Each variable in the dataset is a Xarray DataArray, which can be retrieved using the variable name as the key. For example, to access the 'temperature' variable from the dataset `ds` created in the previous example, you simply use `ds['temperature']` or `ds.temperature`. This retrieves the DataArray corresponding to temperature, complete with its associated dimensions, coordinates, and attributes. This structure allows for intuitive querying and manipulation of the data. You can similarly access the 'precipitation' DataArray using `ds['precipitation']` or `ds.precipitation`. These variables can be handled independently or in conjunction with

other dataset variables, depending on the analysis required. It's this simplicity in variable access that makes Xarray a potent tool for handling high-dimensional datasets typical in fields like atmospheric sciences.

When performing calculations or operations on an Xarray dataset, the operation propagates across all the included variables, applying the computation en masse. This attribute of Xarray is particularly advantageous for dataset-wide adjustments or analyses. Consider a scenario where we want to add one to every data point in the dataset. We could do this with the command `ds = ds + 1`, which would add one to each variable in `ds`.

Similarly, when performing coordinate-based selections, the operation is universally applied. If you were to select a subset of data along a specific coordinate, say a particular range of latitudes, executing something like `ds.sel(latitude=slice(10, 20))` would concurrently extract the slice for all variables within `ds` that correspond to those latitudes. It would return a new dataset with the same variables but with the requested range of latitudes. Remember that this is a view, not a copy, so you may get irregular results if you try to modify the contents of the view.

17.4: Datasets is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

17.5: Reading netCDF files

A NetCDF (Network Common Data Form) file is a format designed to store array-oriented scientific data, particularly useful for storing large sets of multidimensional gridded fields, such as temperatures, humidity, wind speed across different points in time and space. It is a staple in the atmospheric and oceanic sciences due to its capability to handle complex data structures and its support for various data types.

To read a NetCDF file in Xarray, one would utilize the `open_dataset()` function provided by the library. This function seamlessly handles the task of loading the data from a NetCDF file into an Xarray dataset, where variables and coordinates are automatically aligned with the corresponding dimensions. The syntax `ds = xr.open_dataset('path_to_file.nc')` would open the NetCDF file and read the data into an Xarray dataset `ds`, with the structure of the dataset reflecting the multidimensional nature of the original data, preserving its attributes and metadata, thus making it readily accessible for further analysis.

17.5: Reading netCDF files is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

17.6: Groupby and Resample

Xarray's **groupby** functionality is similar to that of pandas and it allows one to aggregate over one or multiple dimensions based on a coordinate. In atmospheric sciences, this feature is invaluable, as it allows for the segmentation of complex, high-dimensional datasets into meaningful groups based on specified criteria, such as time periods or spatial regions, and then apply operations to each group independently.

For instance, suppose we have a dataset of monthly global temperature readings, spanning several years. We may wish to compute the monthly average (i.e., the average of all Januaries, the average of all Februaries, ...). In Xarray, we could accomplish this using the `.groupby()` method along with the coordinate we want to group by, which in this case could be `'time.month'`. Once grouped, we would then apply an aggregation method `.mean()`.

Here's a short example demonstrating this:

```
import xarray as xr
import numpy as np
import pandas as pd

# Set the seed for reproducibility
np.random.seed(0)

# Create a range of monthly periods across multiple years (2005-2007, for example)
time = pd.date_range('2005-01-01', '2007-12-31', freq='M')

# Create a DataArray with random temperature data for each month
temperature = xr.DataArray(np.random.rand(len(time), 2, 2),
                           dims=('time', 'lat', 'lon'),
                           coords={'time': time,
                                    'lat': [10, 20],
                                    'lon': [50, 60]})

# Create a DataArray with random precipitation data for each month
precipitation = xr.DataArray(np.random.rand(len(time), 2, 2),
                              dims=('time', 'lat', 'lon'),
                              coords={'time': time,
                                       'lat': [10, 20],
                                       'lon': [50, 60]})

# Combine into a Dataset
ds = xr.Dataset({'temperature': temperature, 'precipitation': precipitation})

# Group by 'time.month' and calculate the mean for each month across all years
monthly_mean = ds.groupby('time.month').mean('time')

print(monthly_mean)
```

Here is the output:

```
<xarray.Dataset>
Dimensions:      (lat: 2, lon: 2, month: 12)
Coordinates:
  * lat          (lat) int64 10 20
  * lon          (lon) int64 50 60
  * month        (month) int64 1 2 3 4 5 6 7 8 9 10 11 12
Data variables:
  temperature    (month, lat, lon) float64 0.4836 0.3663 ... 0.3119 0.4013
  precipitation   (month, lat, lon) float64 0.6187 0.5886 ... 0.5526 0.425
```

Note that what `.groupby()` has done is take the months from the years and average them, so that the coordinate `month = 1, 2, ...` is the average of Januarys, average of Februarys, ..., from three years of the dataset. It has also retained the `'lat'` and `'lon'` coordinates.

By changing the function after the `.groupby()` method, we can get other statistics of the group. For example, we could get the maximum of each month with the command: `monthly_max = ds.groupby('time.month').max('time')`.

A related process is **resampling**, a powerful tool that enables one to restructure a dataset from one time resolution to another, aggregating the data according to a given rule. It can be particularly useful when looking for long-term trends or smoothing out short-term fluctuations. This process involves defining a new time frequency and then specifying an aggregation method, such as taking the mean, sum, or maximum over each period in the resampled dataset.

If we use the same example Dataset from above, here's how we can take the dataset with monthly data and resample it to get annual averages:

```
ds.resample(time='A').mean()
```

It returns:

```
<xarray.Dataset>
Dimensions:          (lat: 2, lon: 2, time: 3)
Coordinates:
  * lat              (lat) int64 10 20
  * lon              (lon) int64 50 60
  * time              (time) datetime64[ns] 2005-12-31 2006-12-31 2007-12-31
Data variables:
  temperature        (time, lat, lon) float64 0.5198 0.6375 ... 0.6292 0.5636
  precipitation       (time, lat, lon) float64 0.6056 0.5719 ... 0.5868 0.5283
```

In the above code, ``ds.resample(time='A')`` changes the dataset's temporal resolution from monthly ('M') to annual ('A'). Following resampling, `.mean()` computes the average of the data points for each year, which could reveal year-over-year changes and trends that are less discernible in monthly variations.

Note that Xarray resample command is `.resample(time='A')`. This is different from Pandas, which would just require `.resample('A')` because Xarray needs to know which coordinate to resample over.

17.6: Groupby and Resample is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

17.7: Additional resources

[Xarray tutorial](#) from Ryan Abernathy's Research Computing in Earth Science class

[Xarray chapter](#) from Ryan Abernathy's Earth and Environmental Data Science book

17.7: Additional resources is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

18: Regression

[18.1: Introduction](#)

[18.2: scipy.stats.linregress\(\)](#)

[18.3: sklearn](#)

[18.4: multiple linear regression](#)

[18.5: More resources](#)

[18: Regression](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

18.1: Introduction

Regression analysis is a cornerstone of data science, providing a powerful way to model and analyze relationships between variables and forecast outcomes. It operates on the principle of finding the best-fitting line or curve that represents the trend of data points within a dataset. This modeling allows data scientists to not only understand which variables are significant predictors but also to quantify the strength of the impact these predictors have on the response variable. It is an indispensable tool in predictive analytics, enabling the construction of models that can forecast trends, determine the strength of predictors, and even drive decision-making processes by providing concrete, actionable insights.

The linear least-squares fitting method is pivotal in data analysis, offering a robust approach for determining the linear equation that most closely approximates the relationship between two variables. This technique minimizes the sum of the squares of the vertical deviations from each data point to the line, hence the name 'least squares.' By focusing on the smallest aggregate distance, it yields the line of best fit, which serves as a predictive model and a tool for inference. The elegance of the least-squares fit lies in its simplicity and efficiency, which allows for straightforward interpretation and computation. It provides a foundation for understanding more complex relationships by starting with a linear approximation. In practice, the linear least-squares method not only aids in prediction but also in evaluating the strength of associations, helping to discern underlying patterns in data across scientific, industrial, and research disciplines. Its central role is further cemented by its inclusion in fundamental statistical learning techniques, where it acts as a stepping stone to more advanced methods.

18.1: Introduction is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

18.2: scipy.stats.linregress()

`scipy.stats.linregress` is a function from the SciPy library that performs a linear least-squares regression for two sets of measurements.

Here's a step-by-step explanation of how to use `scipy.stats.linregress` for a linear fit:

1. Import the function: You start by importing `linregress` from `scipy.stats`.
2. Prepare your data: You should have two arrays of data, `x` and `y`, where `x` is the independent variable and `y` is the dependent variable you're trying to predict.
3. Execute the regression: Call `linregress` with your `x` and `y` data arrays. The function will return a `LinregressResult` object that contains the results of the regression.
4. Extract results: From the result object, you can extract the slope, intercept, r -value (coefficient of determination), p -value (hypothesis test for non-correlation), and standard error of the estimated gradient.

Here's an example using `scipy.stats.linregress`:

Let's say we have some data points that represent the relationship between hours studied (x) and exam scores (y). We want to perform a linear regression to understand this relationship better.

```
import numpy as np
from scipy.stats import linregress
import matplotlib.pyplot as plt

# Sample data
x = np.array([1, 2, 3, 4, 5]) # Hours studied
y = np.array([2, 3, 5, 7, 11]) # Exam score

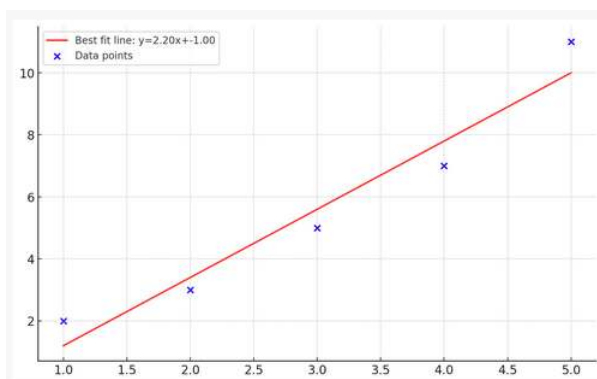
# Perform linear regression
fit1 = linregress(x, y)

# Use the slope and intercept to construct the best fit line
y_pred = fit1.intercept + fit1.slope * x

# Plotting the data points
plt.scatter(x, y, color='blue', label='Data points')

# Plotting the best fit line
plt.plot(x, y_pred, color='red', label=f'Best fit line: y={slope:.2f}x+{intercept:.2f}')

# Adding the legend
plt.legend()
```



18.2: `scipy.stats.linregress()` is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

18.3: sklearn

Scikit-learn, often abbreviated as sklearn, is a versatile and widely-used open-source machine learning library for Python. It provides a range of supervised and unsupervised learning algorithms through a consistent interface. Essential tools for data mining and data analysis are also part of this library, making it a go-to choice for many data scientists and researchers. Scikit-learn is built upon the SciPy (Scientific Python) ecosystem and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. Its simplicity, accessibility, and broad range of algorithms make it very useful for educational purposes, yet it's powerful enough to be used in large-scale industrial applications. Scikit-learn includes a variety of algorithms for classification, regression, clustering, dimensionality reduction, model selection, and preprocessing. Its robustness, ease of use, and good documentation have made it one of the most popular machine learning libraries in the world.

While sklearn can do a lot of sophisticated analyses, it can also do the simplest, a linear fit. To do so, follow these steps:

1. Import the necessary classes: Use the `LinearRegression` class from the `sklearn.linear_model` module.
2. Prepare your data:
 - Ensure your independent variable(s) (features) are in a two-dimensional array (or matrix). If you have a single feature, you'll need to reshape your array to be two-dimensional.
 - The dependent variable (target) should be in a one-dimensional array.
3. Create an instance of the `LinearRegression` class:
 - Instantiate the `LinearRegression` class to create the model.
4. Fit the model:
 - Use the `fit` method on the model instance with your independent and dependent variables as arguments. This step is where the model learns the coefficients from your data.
5. Check the coefficients:
 - After fitting the model, you can check the `coef_` and `intercept_` attributes of your `LinearRegression` object to see the slope and intercept of the fitted line.
6. Make predictions:
 - With the model fitted, you can now predict the dependent variable for new independent variable(s) using the `predict` method.

Here's an example of how to do a linear fit with scikit-learn for a dataset with a single feature:

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Sample data
X = np.array([[1], [2], [3], [4], [5]]) # Independent variable (feature), needs to be
2D for sklearn
y = np.array([2, 3, 5, 7, 11])          # Dependent variable (target), 1D is fine

# Create an instance of the model
model = LinearRegression()

# Fit the model
model.fit(X, y)

# After fitting the model, we can check the slope (coef_) and intercept (intercept_)
slope = model.coef_[0]
intercept = model.intercept_

# Make predictions (optional)
X_new = np.array([[6], [7]]) # New data for prediction
y_pred = model.predict(X_new)
```

```
# slope, intercept, and predictions are now available
print(f"Slope: {slope}, Intercept: {intercept}, Predictions: {y_pred}")
```

This code will output the slope and intercept of the best fit line, as well as predictions for the `X\_new` data points. If you have multiple features, `X` would be a matrix with multiple columns, one for each feature. The `y` array remains one-dimensional.

18.3: [sklearn](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

18.4: multiple linear regression

Multiple linear regression is a statistical method used to understand the relationship between one dependent variable and two or more independent variables. By creating an equation, this method predicts the value of the dependent variable based on the values of the independents. In practice, it helps to answer questions like how changes in factors like price and promotion affect sales, or how different health indicators can predict a patient's blood pressure. It's a step beyond simple linear regression, which only considers one independent variable, allowing for more complex analyses and more accurate predictions when several factors are at play simultaneously.

In the provided multiple linear regression example using scikit-learn, we have a dataset with two independent variables: hours studied and hours slept, and one dependent variable, which is the exam score.

```
from sklearn.linear_model import LinearRegression
import numpy as np

# Sample data with two independent variables (features) and one dependent variable (target)
# Let's assume X1 represents hours studied and X2 represents hours slept
# Y represents the exam score
X = np.array([[1, 8], [2, 7], [3, 6], [4, 5], [5, 4]]) # Features: Hours Studied, Hours Slept
y = np.array([5, 12, 14, 22, 32]) # Target: Exam Score

# Create an instance of the model
model = LinearRegression()

# Fit the model
model.fit(X, y)

# Coefficients for each independent variable
coefficients = model.coef_

# Intercept of the model
intercept = model.intercept_

# Let's predict the exam score for someone who studied for 6 hours and slept for 3 hours
new_data = np.array([[6, 3]])
predicted_score = model.predict(new_data)
coefficients, intercept, predicted_score
```

After fitting the model with the sample data, we obtain the following output:

```
(array([ 3.2, -3.2]), 26.599999999999998, array([36.2]))
```

- The coefficients for the independent variables are approximately 3.2 for hours studied and -3.2 for hours slept. This suggests that for each additional hour studied, the exam score is expected to increase by 3.2 points, and for each additional hour slept, the score is expected to decrease by 3.2 points, according to the model.

- The intercept of the model is approximately 26.6, which is the expected exam score if both independent variables were zero (which isn't practically possible, but it is a feature of the linear model).

Using this model, we predict that someone who studied for 6 hours and slept for 3 hours would have an exam score of approximately 36.2 points. This prediction is based on the linear relationship learned from the provided sample data.

18.4: multiple linear regression is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

18.5: More resources

[Here's a good linear regression exercise](#)

18.5: More resources is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

19: Cartopy

[19.1: Abernathy cartopy chapter](#)

[19: Cartopy](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

19.1: Abernathy cartopy chapter

It is [here](#).

19.1: Abernathy cartopy chapter is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

Back Matter

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

[Detailed Licensing](#)

Index

A

arguments

[4.9: Parameters and Arguments](#)

C

Catching exceptions

[3.6: Catching exceptions Using Try and Except](#)

D

debugging (python)

[5.11: Debugging](#)

dot notation

[4.5: Math Functions](#)

E

except (python)

[3.6: Catching exceptions Using Try and Except](#)

H

here

[19.1: Abernathy cartopy chapter](#)

I

Infinite loops

[5.4: "Infinite loops" and break](#)

iterations

[5: Iterations](#)

P

parameters

[4.9: Parameters and Arguments](#)

python (language)

- 2: Variables, Expressions, and Statements
 - 2.1: Values and Types
 - 2.2: Variables
 - 2.3: Variable names and Keywords
 - 2.4: Statements
 - 2.5: Operators and Operands
 - 2.6: Expressions
 - 2.7: Order of Operations
 - 2.8: String Operations
 - 2.9: Asking the user for input
 - 2.10: Comments
 - 2.11: Choosing Mnemonic Variable Names
 - 2.12: Debugging
- 2.E: Variables, Expressions, and Statements (Exercises)
- 2.G: Glossary
- 3: Conditional Execution
 - 3.1: Boolean Expressions
 - 3.2: Logical Operators
 - 3.3: Conditional Execution
 - 3.4: Chained Conditionals
 - 3.5: Nested Conditionals
 - 3.6: Catching exceptions Using Try and Except
 - 3.7: Short-Circuit Evaluation of Logical Expressions
 - 3.8: Debugging
- 3.E: Conditional Execution (Exercises)
- 3.G: Conditional Execution (Glossary)
- 4: Functions
 - 4.1: Function Calls
 - 4.2: Built-in Functions
 - 4.3: Type Conversion Functions
 - 4.4: Random Numbers
 - 4.5: Math Functions
 - 4.6: Adding New Functions
 - 4.7: Definitions and Uses
 - 4.8: Flow of Execution
 - 4.9: Parameters and Arguments
 - 4.10: Fruitful functions and void functions
 - 4.12: Why functions?
 - 4.13: Debugging
- 4.E: Functions (Exercises)
- 4.G: Functions (Glossary)
- 5: Iterations
 - 5.1: Updating Variables
 - 5.2: The while Statement
 - 5.3: Infinite Loops
 - 5.4: "Infinite loops" and break
 - 5.6: Finishing iterations with continue
 - 5.7: Definite loops using for
 - 5.8: Loop patterns
 - 5.9: Counting and Summing Loops
 - 5.10: Maximum and Minimum Loops
 - 5.11: Debugging
- 5.E: Iterations (Exercises)
- 5.G: Iterations (Glossary)
- 6: Strings
 - 6.1: A string is a sequence
 - 6.2: Getting the length of a string using len
 - 6.3: Traversal through a string with a loop
 - 6.4: String Slices
 - 6.5: Looping and Counting
 - 6.6: The in operator
 - 6.7: String Comparison
 - 6.8: String Methods
 - 6.9: Parsing strings
- 6.E: Strings (Exercises)
- 6.G: Strings (Glossary)
- 7: Lists
 - 7.1: A list is a sequence
 - 7.2: Lists are mutable
 - 7.3: Traversing a List
 - 7.4: List operations
 - 7.5: List Slices
 - 7.6: List Methods
 - 7.7: Deleting Elements
 - 7.8: Lists and Functions
 - 7.9: Lists and Strings
 - 7.10: Parsing lines
 - 7.11: Objects and Values

- 7.12: Aliasing
- 7.13: List arguments
- 7.15: Debugging
- 7.E: Lists (Exercises)
- 7.G: Lists (Glossary)
- 9: Dictionaries
 - 9.1: Dictionaries
 - 9.2: Dictionary as a Set of Counters
 - 9.3: Dictionaries and Files
 - 9.4: Looping and Dictionaries
 - 9.5: Advanced Text Parsing
 - 9.6: Debugging
- 9.E: Dictionaries (Exercises)
- 9.G: Dictionaries (Glossary)

T

try (python)

- 3.6: Catching exceptions Using Try and Except

V

Variables (python)

- 2.2: Variables

void functions

- 4.10: Fruitful functions and void functions

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: [ATMO 321: Python for Atmospheric Sciences](#)

Webpages: 174

Applicable Restrictions: Noncommercial

All licenses found:

- [CC BY-NC-SA 4.0](#): 52.9% (92 pages)
- [Undeclared](#): 38.5% (67 pages)
- [CC BY-SA 4.0](#): 8.6% (15 pages)

By Page

- [ATMO 321: Python for Atmospheric Sciences](#) - *Undeclared*
 - [Front Matter](#) - *Undeclared*
 - [TitlePage](#) - *Undeclared*
 - [InfoPage](#) - *Undeclared*
 - [Table of Contents](#) - *Undeclared*
 - [Licensing](#) - *Undeclared*
 - [1: Introduction](#) - *Undeclared*
 - [1.1: Hello world!](#) - *Undeclared*
 - [1.2: What is programming?](#) - *Undeclared*
 - [1.3: Pseudocode](#) - *Undeclared*
 - [1.4: Pseudocode, part 2](#) - *Undeclared*
 - [2: Variables, Expressions, and Statements](#) - [CC BY-NC-SA 4.0](#)
 - [2.1: Values and Types](#) - [CC BY-NC-SA 4.0](#)
 - [2.2: Variables](#) - [CC BY-NC-SA 4.0](#)
 - [2.3: Variable names and Keywords](#) - [CC BY-NC-SA 4.0](#)
 - [2.4: Statements](#) - [CC BY-NC-SA 4.0](#)
 - [2.5: Operators and Operands](#) - [CC BY-NC-SA 4.0](#)
 - [2.6: Expressions](#) - [CC BY-NC-SA 4.0](#)
 - [2.7: Order of Operations](#) - [CC BY-NC-SA 4.0](#)
 - [2.8: String Operations](#) - [CC BY-NC-SA 4.0](#)
 - [2.9: Asking the user for input](#) - [CC BY-NC-SA 4.0](#)
 - [2.10: Comments](#) - [CC BY-NC-SA 4.0](#)
 - [2.11: Choosing Mnemonic Variable Names](#) - [CC BY-NC-SA 4.0](#)
 - [2.12: Debugging](#) - [CC BY-NC-SA 4.0](#)
 - [2.E: Variables, Expressions, and Statements \(Exercises\)](#) - [CC BY-NC-SA 4.0](#)
 - [2.G: Glossary](#) - [CC BY-NC-SA 4.0](#)
 - [3: Conditional Execution](#) - [CC BY-NC-SA 4.0](#)
 - [3.1: Boolean Expressions](#) - [CC BY-NC-SA 4.0](#)
 - [3.2: Logical Operators](#) - [CC BY-NC-SA 4.0](#)
 - [3.3: Conditional Execution](#) - [CC BY-NC-SA 4.0](#)
 - [3.4: Chained Conditionals](#) - [CC BY-NC-SA 4.0](#)
 - [3.5: Nested Conditionals](#) - [CC BY-NC-SA 4.0](#)
 - [3.6: Catching exceptions Using Try and Except](#) - [CC BY-NC-SA 4.0](#)
 - [3.7: Short-Circuit Evaluation of Logical Expressions](#) - [CC BY-NC-SA 4.0](#)
 - [3.8: Debugging](#) - [CC BY-NC-SA 4.0](#)
 - [3.E: Conditional Execution \(Exercises\)](#) - [CC BY-NC-SA 4.0](#)
 - [3.G: Conditional Execution \(Glossary\)](#) - [CC BY-NC-SA 4.0](#)
 - [4: Functions](#) - [CC BY-NC-SA 4.0](#)
 - [4.1: Function Calls](#) - [CC BY-NC-SA 4.0](#)
 - [4.2: Built-in Functions](#) - [CC BY-NC-SA 4.0](#)
 - [4.3: Type Conversion Functions](#) - [CC BY-NC-SA 4.0](#)
 - [4.4: Random Numbers](#) - [CC BY-NC-SA 4.0](#)
 - [4.5: Math Functions](#) - [CC BY-NC-SA 4.0](#)
 - [4.6: Adding New Functions](#) - [CC BY-NC-SA 4.0](#)
 - [4.7: Definitions and Uses](#) - [CC BY-NC-SA 4.0](#)
 - [4.8: Flow of Execution](#) - [CC BY-NC-SA 4.0](#)
 - [4.9: Parameters and Arguments](#) - [CC BY-NC-SA 4.0](#)
 - [4.10: Fruitful functions and void functions](#) - [CC BY-NC-SA 4.0](#)
 - [4.11: Scope](#) - *Undeclared*
 - [4.12: Why functions?](#) - [CC BY-NC-SA 4.0](#)
 - [4.13: Debugging](#) - [CC BY-NC-SA 4.0](#)
 - [4.E: Functions \(Exercises\)](#) - [CC BY-NC-SA 4.0](#)
 - [4.G: Functions \(Glossary\)](#) - [CC BY-NC-SA 4.0](#)
 - [5: Iterations](#) - [CC BY-NC-SA 4.0](#)
 - [5.1: Updating Variables](#) - [CC BY-NC-SA 4.0](#)
 - [5.2: The while Statement](#) - [CC BY-NC-SA 4.0](#)
 - [5.3: Infinite Loops](#) - [CC BY-NC-SA 4.0](#)
 - [5.4: "Infinite loops" and break](#) - [CC BY-NC-SA 4.0](#)
 - [5.5: continue, break, enumerate, zip](#) - *Undeclared*

- 5.6: Finishing iterations with continue - CC BY-NC-SA 4.0
- 5.7: Definite loops using for - CC BY-NC-SA 4.0
- 5.8: Loop patterns - CC BY-NC-SA 4.0
- 5.9: Counting and Summing Loops - CC BY-NC-SA 4.0
- 5.10: Maximum and Minimum Loops - CC BY-NC-SA 4.0
- 5.11: Debugging - CC BY-NC-SA 4.0
- 5.E: Iterations (Exercises) - CC BY-NC-SA 4.0
- 5.G: Iterations (Glossary) - CC BY-NC-SA 4.0
- 6: Strings - CC BY-NC-SA 4.0
 - 6.1: A string is a sequence - CC BY-NC-SA 4.0
 - 6.2: Getting the length of a string using len - CC BY-NC-SA 4.0
 - 6.3: Traversal through a string with a loop - CC BY-NC-SA 4.0
 - 6.4: String Slices - CC BY-NC-SA 4.0
 - 6.5: Looping and Counting - CC BY-NC-SA 4.0
 - 6.6: The in operator - CC BY-NC-SA 4.0
 - 6.7: String Comparison - CC BY-NC-SA 4.0
 - 6.8: String Methods - CC BY-NC-SA 4.0
 - 6.9: Parsing strings - CC BY-NC-SA 4.0
 - 6.10: String formatting - *Undeclared*
 - 6.E: Strings (Exercises) - CC BY-NC-SA 4.0
 - 6.G: Strings (Glossary) - CC BY-NC-SA 4.0
- 7: Lists - CC BY-NC-SA 4.0
 - 7.1: A list is a sequence - CC BY-NC-SA 4.0
 - 7.2: Lists are mutable - CC BY-NC-SA 4.0
 - 7.3: Traversing a List - CC BY-NC-SA 4.0
 - 7.4: List operations - CC BY-NC-SA 4.0
 - 7.5: List Slices - CC BY-NC-SA 4.0
 - 7.6: List Methods - CC BY-NC-SA 4.0
 - 7.7: Deleting Elements - CC BY-NC-SA 4.0
 - 7.8: Lists and Functions - CC BY-NC-SA 4.0
 - 7.9: Lists and Strings - CC BY-NC-SA 4.0
 - 7.10: Parsing lines - CC BY-NC-SA 4.0
 - 7.11: Objects and Values - CC BY-NC-SA 4.0
 - 7.12: Aliasing - CC BY-NC-SA 4.0
 - 7.13: List arguments - CC BY-NC-SA 4.0
 - 7.14: List comprehensions - *Undeclared*
 - 7.15: Debugging - CC BY-NC-SA 4.0
 - 7.E: Lists (Exercises) - CC BY-NC-SA 4.0
 - 7.G: Lists (Glossary) - CC BY-NC-SA 4.0
- 8: Sets - *Undeclared*
 - 8.1: Introduction to sets - *Undeclared*
 - 8.2: set operations - *Undeclared*
 - 8.3: set order - *Undeclared*
- 9: Dictionaries - CC BY-NC-SA 4.0
 - 9.1: Dictionaries - CC BY-NC-SA 4.0
 - 9.2: Dictionary as a Set of Counters - CC BY-NC-SA 4.0
 - 9.3: Dictionaries and Files - CC BY-NC-SA 4.0
 - 9.4: Looping and Dictionaries - CC BY-NC-SA 4.0
 - 9.5: Advanced Text Parsing - CC BY-NC-SA 4.0
 - 9.6: Debugging - CC BY-NC-SA 4.0
 - 9.E: Dictionaries (Exercises) - CC BY-NC-SA 4.0
 - 9.G: Dictionaries (Glossary) - CC BY-NC-SA 4.0
- 10: Packages - *Undeclared*
 - 10.1: Importing packages in python - *Undeclared*
 - 10.2: How to use functions in packages - *Undeclared*
 - 10.3: Aliasing packages - *Undeclared*
 - 10.4: Sub packages - *Undeclared*
- 11: Numpy, part 1 - CC BY-SA 4.0
 - 11.1: Packages - CC BY-SA 4.0
 - 11.2: Mathematical Functions - *Undeclared*
 - 11.3: The NumPy 'ndarray' - CC BY-SA 4.0
- 12: Numpy (2 of 2) - CC BY-SA 4.0
 - 12.1: Accessing Individual Elements - CC BY-SA 4.0
 - 12.2: “Vectorized” Arithmetic Operators - CC BY-SA 4.0
 - 12.3: Broadcasting - *Undeclared*
 - 12.4: Copying – and Not Copying – Arrays - CC BY-SA 4.0
 - 12.5: Sorting Arrays - CC BY-SA 4.0
 - 12.6: More Exotic Array Modifications - CC BY-SA 4.0
 - 12.7: Summary - *Undeclared*
- 13: Matplotlib - *Undeclared*
 - 13.1: Loading matplotlib - *Undeclared*
 - 13.2: Matplotlib basics - *Undeclared*
 - 13.3: Matplotlib basics, 2 - *Undeclared*
 - 13.4: Matplotlib tutorial - *Undeclared*
- 14: Pandas Series - CC BY-SA 4.0
 - 14.1: Introduction - *Undeclared*
 - 14.2: The Pandas Series - CC BY-SA 4.0
 - 14.3: More about Pandas series - *Undeclared*
 - 14.4: Vectorized Arithmetic Operators - CC BY-SA 4.0
 - 14.5: Sorting Serieses - CC BY-SA 4.0
 - 14.6: Queries - CC BY-SA 4.0
 - 14.7: Summary - CC BY-SA 4.0
- 15: Pandas Dataframes - *Undeclared*
 - 15.1: Dataframe basics - *Undeclared*
 - 15.2: Accessing elements - *Undeclared*
 - 15.3: Groupby() - *Undeclared*
 - 15.4: Math operations - *Undeclared*

- 16: Date and time variables - *Undeclared*
 - 16.1: np.datetime64 - *Undeclared*
 - 16.2: Date and time in Pandas - *Undeclared*
 - 16.3: Extracting components of date and time - *Undeclared*
 - 16.4: Setting the index to be date and time - *Undeclared*
 - 16.5: Resampling - *Undeclared*
 - 16.6: Other resources - *Undeclared*
- 17: Xarray - *Undeclared*
 - 17.1: Introduction - *Undeclared*
 - 17.2: DataArrays - *Undeclared*
 - 17.3: Selecting values - *Undeclared*
 - 17.4: Datasets - *Undeclared*
 - 17.5: Reading netCDF files - *Undeclared*
 - 17.6: Groupby and Resample - *Undeclared*
 - 17.7: Additional resources - *Undeclared*
- 18: Regression - *Undeclared*
 - 18.1: Introduction - *Undeclared*
 - 18.2: scipy.stats.linregress() - *Undeclared*
 - 18.3: sklearn - *Undeclared*
 - 18.4: multiple linear regression - *Undeclared*
 - 18.5: More resources - *Undeclared*
- 19: Cartopy - *Undeclared*
 - 19.1: Abernathy cartopy chapter - *Undeclared*
- Back Matter - *Undeclared*
 - Index - *Undeclared*
 - Glossary - *Undeclared*
 - Detailed Licensing - *Undeclared*
 - Detailed Licensing - *Undeclared*

Detailed Licensing

Overview

Title: [ATMO 321: Python for Atmospheric Sciences](#)

Webpages: 174

Applicable Restrictions: Noncommercial

All licenses found:

- [CC BY-NC-SA 4.0](#): 52.9% (92 pages)
- [Undeclared](#): 38.5% (67 pages)
- [CC BY-SA 4.0](#): 8.6% (15 pages)

By Page

- [ATMO 321: Python for Atmospheric Sciences](#) - *Undeclared*
 - [Front Matter](#) - *Undeclared*
 - [TitlePage](#) - *Undeclared*
 - [InfoPage](#) - *Undeclared*
 - [Table of Contents](#) - *Undeclared*
 - [Licensing](#) - *Undeclared*
 - [1: Introduction](#) - *Undeclared*
 - [1.1: Hello world!](#) - *Undeclared*
 - [1.2: What is programming?](#) - *Undeclared*
 - [1.3: Pseudocode](#) - *Undeclared*
 - [1.4: Pseudocode, part 2](#) - *Undeclared*
 - [2: Variables, Expressions, and Statements](#) - *CC BY-NC-SA 4.0*
 - [2.1: Values and Types](#) - *CC BY-NC-SA 4.0*
 - [2.2: Variables](#) - *CC BY-NC-SA 4.0*
 - [2.3: Variable names and Keywords](#) - *CC BY-NC-SA 4.0*
 - [2.4: Statements](#) - *CC BY-NC-SA 4.0*
 - [2.5: Operators and Operands](#) - *CC BY-NC-SA 4.0*
 - [2.6: Expressions](#) - *CC BY-NC-SA 4.0*
 - [2.7: Order of Operations](#) - *CC BY-NC-SA 4.0*
 - [2.8: String Operations](#) - *CC BY-NC-SA 4.0*
 - [2.9: Asking the user for input](#) - *CC BY-NC-SA 4.0*
 - [2.10: Comments](#) - *CC BY-NC-SA 4.0*
 - [2.11: Choosing Mnemonic Variable Names](#) - *CC BY-NC-SA 4.0*
 - [2.12: Debugging](#) - *CC BY-NC-SA 4.0*
 - [2.E: Variables, Expressions, and Statements \(Exercises\)](#) - *CC BY-NC-SA 4.0*
 - [2.G: Glossary](#) - *CC BY-NC-SA 4.0*
 - [3: Conditional Execution](#) - *CC BY-NC-SA 4.0*
 - [3.1: Boolean Expressions](#) - *CC BY-NC-SA 4.0*
 - [3.2: Logical Operators](#) - *CC BY-NC-SA 4.0*
 - [3.3: Conditional Execution](#) - *CC BY-NC-SA 4.0*
 - [3.4: Chained Conditionals](#) - *CC BY-NC-SA 4.0*
 - [3.5: Nested Conditionals](#) - *CC BY-NC-SA 4.0*
 - [3.6: Catching exceptions Using Try and Except](#) - *CC BY-NC-SA 4.0*
 - [3.7: Short-Circuit Evaluation of Logical Expressions](#) - *CC BY-NC-SA 4.0*
 - [3.8: Debugging](#) - *CC BY-NC-SA 4.0*
 - [3.E: Conditional Execution \(Exercises\)](#) - *CC BY-NC-SA 4.0*
 - [3.G: Conditional Execution \(Glossary\)](#) - *CC BY-NC-SA 4.0*
 - [4: Functions](#) - *CC BY-NC-SA 4.0*
 - [4.1: Function Calls](#) - *CC BY-NC-SA 4.0*
 - [4.2: Built-in Functions](#) - *CC BY-NC-SA 4.0*
 - [4.3: Type Conversion Functions](#) - *CC BY-NC-SA 4.0*
 - [4.4: Random Numbers](#) - *CC BY-NC-SA 4.0*
 - [4.5: Math Functions](#) - *CC BY-NC-SA 4.0*
 - [4.6: Adding New Functions](#) - *CC BY-NC-SA 4.0*
 - [4.7: Definitions and Uses](#) - *CC BY-NC-SA 4.0*
 - [4.8: Flow of Execution](#) - *CC BY-NC-SA 4.0*
 - [4.9: Parameters and Arguments](#) - *CC BY-NC-SA 4.0*
 - [4.10: Fruitful functions and void functions](#) - *CC BY-NC-SA 4.0*
 - [4.11: Scope](#) - *Undeclared*
 - [4.12: Why functions?](#) - *CC BY-NC-SA 4.0*
 - [4.13: Debugging](#) - *CC BY-NC-SA 4.0*
 - [4.E: Functions \(Exercises\)](#) - *CC BY-NC-SA 4.0*
 - [4.G: Functions \(Glossary\)](#) - *CC BY-NC-SA 4.0*
 - [5: Iterations](#) - *CC BY-NC-SA 4.0*
 - [5.1: Updating Variables](#) - *CC BY-NC-SA 4.0*
 - [5.2: The while Statement](#) - *CC BY-NC-SA 4.0*
 - [5.3: Infinite Loops](#) - *CC BY-NC-SA 4.0*
 - [5.4: "Infinite loops" and break](#) - *CC BY-NC-SA 4.0*
 - [5.5: continue, break, enumerate, zip](#) - *Undeclared*

- 5.6: Finishing iterations with continue - CC BY-NC-SA 4.0
- 5.7: Definite loops using for - CC BY-NC-SA 4.0
- 5.8: Loop patterns - CC BY-NC-SA 4.0
- 5.9: Counting and Summing Loops - CC BY-NC-SA 4.0
- 5.10: Maximum and Minimum Loops - CC BY-NC-SA 4.0
- 5.11: Debugging - CC BY-NC-SA 4.0
- 5.E: Iterations (Exercises) - CC BY-NC-SA 4.0
- 5.G: Iterations (Glossary) - CC BY-NC-SA 4.0
- 6: Strings - CC BY-NC-SA 4.0
 - 6.1: A string is a sequence - CC BY-NC-SA 4.0
 - 6.2: Getting the length of a string using len - CC BY-NC-SA 4.0
 - 6.3: Traversal through a string with a loop - CC BY-NC-SA 4.0
 - 6.4: String Slices - CC BY-NC-SA 4.0
 - 6.5: Looping and Counting - CC BY-NC-SA 4.0
 - 6.6: The in operator - CC BY-NC-SA 4.0
 - 6.7: String Comparison - CC BY-NC-SA 4.0
 - 6.8: String Methods - CC BY-NC-SA 4.0
 - 6.9: Parsing strings - CC BY-NC-SA 4.0
 - 6.10: String formatting - *Undeclared*
 - 6.E: Strings (Exercises) - CC BY-NC-SA 4.0
 - 6.G: Strings (Glossary) - CC BY-NC-SA 4.0
- 7: Lists - CC BY-NC-SA 4.0
 - 7.1: A list is a sequence - CC BY-NC-SA 4.0
 - 7.2: Lists are mutable - CC BY-NC-SA 4.0
 - 7.3: Traversing a List - CC BY-NC-SA 4.0
 - 7.4: List operations - CC BY-NC-SA 4.0
 - 7.5: List Slices - CC BY-NC-SA 4.0
 - 7.6: List Methods - CC BY-NC-SA 4.0
 - 7.7: Deleting Elements - CC BY-NC-SA 4.0
 - 7.8: Lists and Functions - CC BY-NC-SA 4.0
 - 7.9: Lists and Strings - CC BY-NC-SA 4.0
 - 7.10: Parsing lines - CC BY-NC-SA 4.0
 - 7.11: Objects and Values - CC BY-NC-SA 4.0
 - 7.12: Aliasing - CC BY-NC-SA 4.0
 - 7.13: List arguments - CC BY-NC-SA 4.0
 - 7.14: List comprehensions - *Undeclared*
 - 7.15: Debugging - CC BY-NC-SA 4.0
 - 7.E: Lists (Exercises) - CC BY-NC-SA 4.0
 - 7.G: Lists (Glossary) - CC BY-NC-SA 4.0
- 8: Sets - *Undeclared*
 - 8.1: Introduction to sets - *Undeclared*
 - 8.2: set operations - *Undeclared*
 - 8.3: set order - *Undeclared*
- 9: Dictionaries - CC BY-NC-SA 4.0
 - 9.1: Dictionaries - CC BY-NC-SA 4.0
 - 9.2: Dictionary as a Set of Counters - CC BY-NC-SA 4.0
 - 9.3: Dictionaries and Files - CC BY-NC-SA 4.0
 - 9.4: Looping and Dictionaries - CC BY-NC-SA 4.0
 - 9.5: Advanced Text Parsing - CC BY-NC-SA 4.0
 - 9.6: Debugging - CC BY-NC-SA 4.0
 - 9.E: Dictionaries (Exercises) - CC BY-NC-SA 4.0
 - 9.G: Dictionaries (Glossary) - CC BY-NC-SA 4.0
- 10: Packages - *Undeclared*
 - 10.1: Importing packages in python - *Undeclared*
 - 10.2: How to use functions in packages - *Undeclared*
 - 10.3: Aliasing packages - *Undeclared*
 - 10.4: Sub packages - *Undeclared*
- 11: Numpy, part 1 - CC BY-SA 4.0
 - 11.1: Packages - CC BY-SA 4.0
 - 11.2: Mathematical Functions - *Undeclared*
 - 11.3: The NumPy 'ndarray' - CC BY-SA 4.0
- 12: Numpy (2 of 2) - CC BY-SA 4.0
 - 12.1: Accessing Individual Elements - CC BY-SA 4.0
 - 12.2: “Vectorized” Arithmetic Operators - CC BY-SA 4.0
 - 12.3: Broadcasting - *Undeclared*
 - 12.4: Copying – and Not Copying – Arrays - CC BY-SA 4.0
 - 12.5: Sorting Arrays - CC BY-SA 4.0
 - 12.6: More Exotic Array Modifications - CC BY-SA 4.0
 - 12.7: Summary - *Undeclared*
- 13: Matplotlib - *Undeclared*
 - 13.1: Loading matplotlib - *Undeclared*
 - 13.2: Matplotlib basics - *Undeclared*
 - 13.3: Matplotlib basics, 2 - *Undeclared*
 - 13.4: Matplotlib tutorial - *Undeclared*
- 14: Pandas Series - CC BY-SA 4.0
 - 14.1: Introduction - *Undeclared*
 - 14.2: The Pandas Series - CC BY-SA 4.0
 - 14.3: More about Pandas series - *Undeclared*
 - 14.4: Vectorized Arithmetic Operators - CC BY-SA 4.0
 - 14.5: Sorting Serieses - CC BY-SA 4.0
 - 14.6: Queries - CC BY-SA 4.0
 - 14.7: Summary - CC BY-SA 4.0
- 15: Pandas Dataframes - *Undeclared*
 - 15.1: Dataframe basics - *Undeclared*
 - 15.2: Accessing elements - *Undeclared*
 - 15.3: Groupby() - *Undeclared*
 - 15.4: Math operations - *Undeclared*

- 16: Date and time variables - *Undeclared*
 - 16.1: np.datetime64 - *Undeclared*
 - 16.2: Date and time in Pandas - *Undeclared*
 - 16.3: Extracting components of date and time - *Undeclared*
 - 16.4: Setting the index to be date and time - *Undeclared*
 - 16.5: Resampling - *Undeclared*
 - 16.6: Other resources - *Undeclared*
- 17: Xarray - *Undeclared*
 - 17.1: Introduction - *Undeclared*
 - 17.2: DataArrays - *Undeclared*
 - 17.3: Selecting values - *Undeclared*
 - 17.4: Datasets - *Undeclared*
 - 17.5: Reading netCDF files - *Undeclared*
 - 17.6: Groupby and Resample - *Undeclared*
 - 17.7: Additional resources - *Undeclared*
- 18: Regression - *Undeclared*
 - 18.1: Introduction - *Undeclared*
 - 18.2: scipy.stats.linregress() - *Undeclared*
 - 18.3: sklearn - *Undeclared*
 - 18.4: multiple linear regression - *Undeclared*
 - 18.5: More resources - *Undeclared*
- 19: Cartopy - *Undeclared*
 - 19.1: Abernathy cartopy chapter - *Undeclared*
- Back Matter - *Undeclared*
 - Index - *Undeclared*
 - Glossary - *Undeclared*
 - Detailed Licensing - *Undeclared*
 - Detailed Licensing - *Undeclared*

Index

A

arguments

[4.9: Parameters and Arguments](#)

C

Catching exceptions

[3.6: Catching exceptions Using Try and Except](#)

D

debugging (python)

[5.11: Debugging](#)

dot notation

[4.5: Math Functions](#)

E

except (python)

[3.6: Catching exceptions Using Try and Except](#)

H

here

[19.1: Abernathy cartopy chapter](#)

I

Infinite loops

[5.4: "Infinite loops" and break](#)

iterations

[5: Iterations](#)

P

parameters

[4.9: Parameters and Arguments](#)

python (language)

- 2: Variables, Expressions, and Statements
 - 2.1: Values and Types
 - 2.2: Variables
 - 2.3: Variable names and Keywords
 - 2.4: Statements
 - 2.5: Operators and Operands
 - 2.6: Expressions
 - 2.7: Order of Operations
 - 2.8: String Operations
 - 2.9: Asking the user for input
 - 2.10: Comments
 - 2.11: Choosing Mnemonic Variable Names
 - 2.12: Debugging
- 2.E: Variables, Expressions, and Statements (Exercises)
- 2.G: Glossary
- 3: Conditional Execution
 - 3.1: Boolean Expressions
 - 3.2: Logical Operators
 - 3.3: Conditional Execution
 - 3.4: Chained Conditionals
 - 3.5: Nested Conditionals
 - 3.6: Catching exceptions Using Try and Except
 - 3.7: Short-Circuit Evaluation of Logical Expressions
 - 3.8: Debugging
- 3.E: Conditional Execution (Exercises)
- 3.G: Conditional Execution (Glossary)
- 4: Functions
 - 4.1: Function Calls
 - 4.2: Built-in Functions
 - 4.3: Type Conversion Functions
 - 4.4: Random Numbers
 - 4.5: Math Functions
 - 4.6: Adding New Functions
 - 4.7: Definitions and Uses
 - 4.8: Flow of Execution
 - 4.9: Parameters and Arguments
 - 4.10: Fruitful functions and void functions
 - 4.12: Why functions?
 - 4.13: Debugging
- 4.E: Functions (Exercises)
- 4.G: Functions (Glossary)
- 5: Iterations
 - 5.1: Updating Variables
 - 5.2: The while Statement
 - 5.3: Infinite Loops
 - 5.4: "Infinite loops" and break
 - 5.6: Finishing iterations with continue
 - 5.7: Definite loops using for
 - 5.8: Loop patterns
 - 5.9: Counting and Summing Loops
 - 5.10: Maximum and Minimum Loops
 - 5.11: Debugging
- 5.E: Iterations (Exercises)
- 5.G: Iterations (Glossary)
- 6: Strings
 - 6.1: A string is a sequence
 - 6.2: Getting the length of a string using len
 - 6.3: Traversal through a string with a loop
 - 6.4: String Slices
 - 6.5: Looping and Counting
 - 6.6: The in operator
 - 6.7: String Comparison
 - 6.8: String Methods
 - 6.9: Parsing strings
- 6.E: Strings (Exercises)
- 6.G: Strings (Glossary)
- 7: Lists
 - 7.1: A list is a sequence
 - 7.2: Lists are mutable
 - 7.3: Traversing a List
 - 7.4: List operations
 - 7.5: List Slices
 - 7.6: List Methods
 - 7.7: Deleting Elements
 - 7.8: Lists and Functions
 - 7.9: Lists and Strings
 - 7.10: Parsing lines
 - 7.11: Objects and Values

- 7.12: Aliasing
- 7.13: List arguments
- 7.15: Debugging
- 7.E: Lists (Exercises)
- 7.G: Lists (Glossary)
- 9: Dictionaries
 - 9.1: Dictionaries
 - 9.2: Dictionary as a Set of Counters
 - 9.3: Dictionaries and Files
 - 9.4: Looping and Dictionaries
 - 9.5: Advanced Text Parsing
 - 9.6: Debugging
- 9.E: Dictionaries (Exercises)
- 9.G: Dictionaries (Glossary)

T

try (python)

- 3.6: Catching exceptions Using Try and Except

V

Variables (python)

- 2.2: Variables

void functions

- 4.10: Fruitful functions and void functions

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: [Texas A&M](#)

Webpages: 184

Applicable Restrictions: Noncommercial

All licenses found:

- [CC BY-NC-SA 4.0](#): 50% (92 pages)
- [Undeclared](#): 41.8% (77 pages)
- [CC BY-SA 4.0](#): 8.2% (15 pages)

By Page

- [Texas A&M - Undeclared](#)
 - [Front Matter - Undeclared](#)
 - [TitlePage - Undeclared](#)
 - [InfoPage - Undeclared](#)
 - [Table of Contents - Undeclared](#)
 - [Licensing - Undeclared](#)
 - [ATMO 321: Python for Atmospheric Sciences - Undeclared](#)
 - [Front Matter - Undeclared](#)
 - [TitlePage - Undeclared](#)
 - [InfoPage - Undeclared](#)
 - [Table of Contents - Undeclared](#)
 - [Licensing - Undeclared](#)
 - [1: Introduction - Undeclared](#)
 - [1.1: Hello world! - Undeclared](#)
 - [1.2: What is programming? - Undeclared](#)
 - [1.3: Pseudocode - Undeclared](#)
 - [1.4: Pseudocode, part 2 - Undeclared](#)
 - [2: Variables, Expressions, and Statements - CC BY-NC-SA 4.0](#)
 - [2.1: Values and Types - CC BY-NC-SA 4.0](#)
 - [2.2: Variables - CC BY-NC-SA 4.0](#)
 - [2.3: Variable names and Keywords - CC BY-NC-SA 4.0](#)
 - [2.4: Statements - CC BY-NC-SA 4.0](#)
 - [2.5: Operators and Operands - CC BY-NC-SA 4.0](#)
 - [2.6: Expressions - CC BY-NC-SA 4.0](#)
 - [2.7: Order of Operations - CC BY-NC-SA 4.0](#)
 - [2.8: String Operations - CC BY-NC-SA 4.0](#)
 - [2.9: Asking the user for input - CC BY-NC-SA 4.0](#)
 - [2.10: Comments - CC BY-NC-SA 4.0](#)
 - [2.11: Choosing Mnemonic Variable Names - CC BY-NC-SA 4.0](#)
 - [2.12: Debugging - CC BY-NC-SA 4.0](#)
 - [2.E: Variables, Expressions, and Statements \(Exercises\) - CC BY-NC-SA 4.0](#)
 - [2.G: Glossary - CC BY-NC-SA 4.0](#)
 - [3: Conditional Execution - CC BY-NC-SA 4.0](#)
 - [3.1: Boolean Expressions - CC BY-NC-SA 4.0](#)
 - [3.2: Logical Operators - CC BY-NC-SA 4.0](#)
 - [3.3: Conditional Execution - CC BY-NC-SA 4.0](#)
 - [3.4: Chained Conditionals - CC BY-NC-SA 4.0](#)
 - [3.5: Nested Conditionals - CC BY-NC-SA 4.0](#)
 - [3.6: Catching exceptions Using Try and Except - CC BY-NC-SA 4.0](#)
 - [3.7: Short-Circuit Evaluation of Logical Expressions - CC BY-NC-SA 4.0](#)
 - [3.8: Debugging - CC BY-NC-SA 4.0](#)
 - [3.E: Conditional Execution \(Exercises\) - CC BY-NC-SA 4.0](#)
 - [3.G: Conditional Execution \(Glossary\) - CC BY-NC-SA 4.0](#)
 - [4: Functions - CC BY-NC-SA 4.0](#)
 - [4.1: Function Calls - CC BY-NC-SA 4.0](#)
 - [4.2: Built-in Functions - CC BY-NC-SA 4.0](#)
 - [4.3: Type Conversion Functions - CC BY-NC-SA 4.0](#)
 - [4.4: Random Numbers - CC BY-NC-SA 4.0](#)
 - [4.5: Math Functions - CC BY-NC-SA 4.0](#)
 - [4.6: Adding New Functions - CC BY-NC-SA 4.0](#)
 - [4.7: Definitions and Uses - CC BY-NC-SA 4.0](#)
 - [4.8: Flow of Execution - CC BY-NC-SA 4.0](#)
 - [4.9: Parameters and Arguments - CC BY-NC-SA 4.0](#)
 - [4.10: Fruitful functions and void functions - CC BY-NC-SA 4.0](#)
 - [4.11: Scope - Undeclared](#)

- 4.12: Why functions? - *CC BY-NC-SA 4.0*
- 4.13: Debugging - *CC BY-NC-SA 4.0*
- 4.E: Functions (Exercises) - *CC BY-NC-SA 4.0*
- 4.G: Functions (Glossary) - *CC BY-NC-SA 4.0*
- 5: Iterations - *CC BY-NC-SA 4.0*
 - 5.1: Updating Variables - *CC BY-NC-SA 4.0*
 - 5.2: The while Statement - *CC BY-NC-SA 4.0*
 - 5.3: Infinite Loops - *CC BY-NC-SA 4.0*
 - 5.4: "Infinite loops" and break - *CC BY-NC-SA 4.0*
 - 5.5: continue, break, enumerate, zip - *Undeclared*
 - 5.6: Finishing iterations with continue - *CC BY-NC-SA 4.0*
 - 5.7: Definite loops using for - *CC BY-NC-SA 4.0*
 - 5.8: Loop patterns - *CC BY-NC-SA 4.0*
 - 5.9: Counting and Summing Loops - *CC BY-NC-SA 4.0*
 - 5.10: Maximum and Minimum Loops - *CC BY-NC-SA 4.0*
 - 5.11: Debugging - *CC BY-NC-SA 4.0*
 - 5.E: Iterations (Exercises) - *CC BY-NC-SA 4.0*
 - 5.G: Iterations (Glossary) - *CC BY-NC-SA 4.0*
- 6: Strings - *CC BY-NC-SA 4.0*
 - 6.1: A string is a sequence - *CC BY-NC-SA 4.0*
 - 6.2: Getting the length of a string using len - *CC BY-NC-SA 4.0*
 - 6.3: Traversal through a string with a loop - *CC BY-NC-SA 4.0*
 - 6.4: String Slices - *CC BY-NC-SA 4.0*
 - 6.5: Looping and Counting - *CC BY-NC-SA 4.0*
 - 6.6: The in operator - *CC BY-NC-SA 4.0*
 - 6.7: String Comparison - *CC BY-NC-SA 4.0*
 - 6.8: String Methods - *CC BY-NC-SA 4.0*
 - 6.9: Parsing strings - *CC BY-NC-SA 4.0*
 - 6.10: String formatting - *Undeclared*
 - 6.E: Strings (Exercises) - *CC BY-NC-SA 4.0*
 - 6.G: Strings (Glossary) - *CC BY-NC-SA 4.0*
- 7: Lists - *CC BY-NC-SA 4.0*
 - 7.1: A list is a sequence - *CC BY-NC-SA 4.0*
 - 7.2: Lists are mutable - *CC BY-NC-SA 4.0*
 - 7.3: Traversing a List - *CC BY-NC-SA 4.0*
 - 7.4: List operations - *CC BY-NC-SA 4.0*
 - 7.5: List Slices - *CC BY-NC-SA 4.0*
 - 7.6: List Methods - *CC BY-NC-SA 4.0*
 - 7.7: Deleting Elements - *CC BY-NC-SA 4.0*
 - 7.8: Lists and Functions - *CC BY-NC-SA 4.0*
 - 7.9: Lists and Strings - *CC BY-NC-SA 4.0*
 - 7.10: Parsing lines - *CC BY-NC-SA 4.0*
 - 7.11: Objects and Values - *CC BY-NC-SA 4.0*
 - 7.12: Aliasing - *CC BY-NC-SA 4.0*
 - 7.13: List arguments - *CC BY-NC-SA 4.0*
 - 7.14: List comprehensions - *Undeclared*
 - 7.15: Debugging - *CC BY-NC-SA 4.0*
 - 7.E: Lists (Exercises) - *CC BY-NC-SA 4.0*
 - 7.G: Lists (Glossary) - *CC BY-NC-SA 4.0*
- 8: Sets - *Undeclared*
 - 8.1: Introduction to sets - *Undeclared*
 - 8.2: set operations - *Undeclared*
 - 8.3: set order - *Undeclared*
- 9: Dictionaries - *CC BY-NC-SA 4.0*
 - 9.1: Dictionaries - *CC BY-NC-SA 4.0*
 - 9.2: Dictionary as a Set of Counters - *CC BY-NC-SA 4.0*
 - 9.3: Dictionaries and Files - *CC BY-NC-SA 4.0*
 - 9.4: Looping and Dictionaries - *CC BY-NC-SA 4.0*
 - 9.5: Advanced Text Parsing - *CC BY-NC-SA 4.0*
 - 9.6: Debugging - *CC BY-NC-SA 4.0*
 - 9.E: Dictionaries (Exercises) - *CC BY-NC-SA 4.0*
 - 9.G: Dictionaries (Glossary) - *CC BY-NC-SA 4.0*
- 10: Packages - *Undeclared*
 - 10.1: Importing packages in python - *Undeclared*
 - 10.2: How to use functions in packages - *Undeclared*
 - 10.3: Aliasing packages - *Undeclared*
 - 10.4: Sub packages - *Undeclared*
- 11: Numpy, part 1 - *CC BY-SA 4.0*
 - 11.1: Packages - *CC BY-SA 4.0*
 - 11.2: Mathematical Functions - *Undeclared*
 - 11.3: The NumPy 'ndarray' - *CC BY-SA 4.0*
- 12: Numpy (2 of 2) - *CC BY-SA 4.0*
 - 12.1: Accessing Individual Elements - *CC BY-SA 4.0*
 - 12.2: "Vectorized" Arithmetic Operators - *CC BY-SA 4.0*
 - 12.3: Broadcasting - *Undeclared*
 - 12.4: Copying – and Not Copying – Arrays - *CC BY-SA 4.0*
 - 12.5: Sorting Arrays - *CC BY-SA 4.0*
 - 12.6: More Exotic Array Modifications - *CC BY-SA 4.0*
 - 12.7: Summary - *Undeclared*
- 13: Matplotlib - *Undeclared*
 - 13.1: Loading matplotlib - *Undeclared*
 - 13.2: Matplotlib basics - *Undeclared*
 - 13.3: Matplotlib basics, 2 - *Undeclared*
 - 13.4: Matplotlib tutorial - *Undeclared*
- 14: Pandas Series - *CC BY-SA 4.0*

- 14.1: Introduction - *Undeclared*
- 14.2: The Pandas Series - *CC BY-SA 4.0*
- 14.3: More about Pandas series - *Undeclared*
- 14.4: Vectorized Arithmetic Operators - *CC BY-SA 4.0*
- 14.5: Sorting Serieses - *CC BY-SA 4.0*
- 14.6: Queries - *CC BY-SA 4.0*
- 14.7: Summary - *CC BY-SA 4.0*
- 15: Pandas Dataframes - *Undeclared*
 - 15.1: Dataframe basics - *Undeclared*
 - 15.2: Accessing elements - *Undeclared*
 - 15.3: Groupby() - *Undeclared*
 - 15.4: Math operations - *Undeclared*
- 16: Date and time variables - *Undeclared*
 - 16.1: np.datetime64 - *Undeclared*
 - 16.2: Date and time in Pandas - *Undeclared*
 - 16.3: Extracting components of date and time - *Undeclared*
 - 16.4: Setting the index to be date and time - *Undeclared*
 - 16.5: Resampling - *Undeclared*
 - 16.6: Other resources - *Undeclared*
- 17: Xarray - *Undeclared*
 - 17.1: Introduction - *Undeclared*
 - 17.2: DataArrays - *Undeclared*
 - 17.3: Selecting values - *Undeclared*
 - 17.4: Datasets - *Undeclared*
 - 17.5: Reading netCDF files - *Undeclared*
 - 17.6: Groupby and Resample - *Undeclared*
 - 17.7: Additional resources - *Undeclared*
- 18: Regression - *Undeclared*
 - 18.1: Introduction - *Undeclared*
 - 18.2: scipy.stats.linregress() - *Undeclared*
 - 18.3: sklearn - *Undeclared*
 - 18.4: multiple linear regression - *Undeclared*
 - 18.5: More resources - *Undeclared*
- 19: Cartopy - *Undeclared*
 - 19.1: Abernathy cartopy chapter - *Undeclared*
- Back Matter - *Undeclared*
 - Index - *Undeclared*
 - Glossary - *Undeclared*
 - Detailed Licensing - *Undeclared*
 - Detailed Licensing - *Undeclared*
- Back Matter - *Undeclared*
 - Index - *Undeclared*
 - Glossary - *Undeclared*
 - Detailed Licensing - *Undeclared*