

CIS211: Support for Foundations of Data Science for All

Clare Nguyen

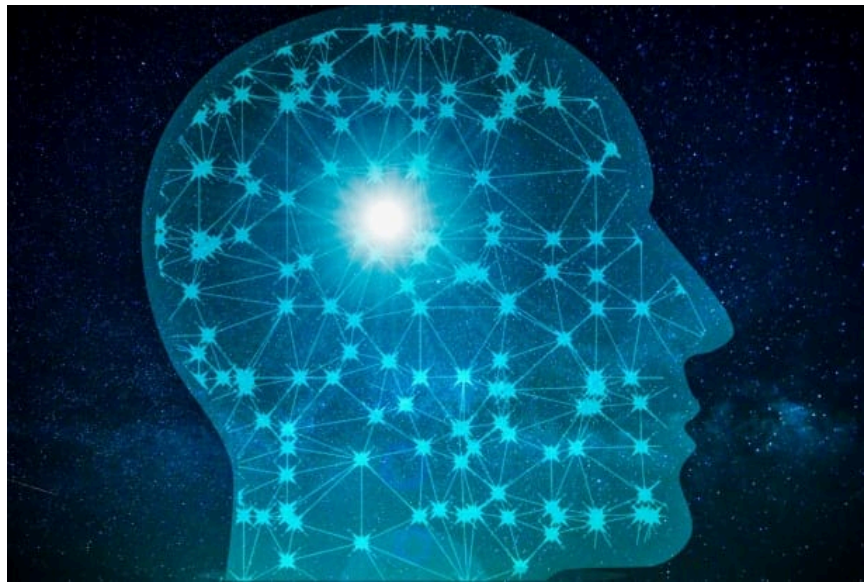
De Anza College

Workforce

CC BY 4.0



Support for Foundations of Data Science for All



Clare Nguyen

De Anza College

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

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

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



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

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

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

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

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

This text was compiled on 09/26/2026

TABLE OF CONTENTS

- Licensing**
- Program Page**
- Accessibility Statement**
- AI Use**
- About the Book**
- About the Authors**
- 1: Introduction to Data Science**
- 2: Numbers and Strings**
- 3: Data Types**
- 4: Sequences and Tables**
- 5: Plotting**
- 6: Table Manipulation**
- 7: Conditionals, Iterations, and Probabilities**
- 8: Sampling**
- 9: Hypothesis Testing**
- 10: Two Sample Tests**
- 11: Estimation**
- 12: The Mean**
- 13: Linear Regression**
- 14: Root Mean Squared Error**
- 15: Classification**
- 16: Classifier**
- Index**
- Glossary**
- Detailed Licensing**

Licensing

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

This page titled [Licensing](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by Clare Nguyen.

Program Page



De Anza College sits in the heart of Silicon Valley, where innovation and opportunity are part of daily life, and also has one of the highest costs of living in the country. For our students, that makes every dollar count, and the price of a textbook can be the difference between staying enrolled or dropping a class. Our Zero Textbook Cost (ZTC) program closes that gap by giving students full access to high-quality, faculty-created course materials at no cost from the first day of class.

De Anza is proud of its diverse student community, and we hold our ZTC materials to the same standard of excellence that defines the college. Faculty across departments write, adapt, and review these resources themselves, keeping them accurate, current, and grounded in the perspectives of the students who use them, with accessibility and inclusive design built in from the start.

This program reflects the time and care our faculty invest in their students. If you have questions, feedback, or come across any issues with a ZTC resource, contact De Anza College's ZTC Grants Coordinator, [Shagun Kaur](#).

Learn more about [De Anza ZTC Program](#) [www.deanza.edu/academic-services/oer/]

Note: The De Anza logo is a copyrighted trademark of De Anza College and may not be reproduced without permission.

Accessibility Statement

Accessibility

De Anza College is dedicated to creating learning materials that are accessible and benefit all students. Our materials follow the Web Content Accessibility Guidelines (WCAG) 2.1 Level AA. Our authors and contracted accessibility specialists checked all resources using this ASCCC OERI checklist, the accessibility tool in LibreTexts, WAVE, JAWS, and other more detailed checks for complex materials such as interactive components.

We selected LibreTexts as our preferred platform because of its [commitment to accessibility](#). In addition to the platform's inherent accessibility, we collaborated with the LibreTexts team to fix any accessibility issues we found.

Our Accessibility Review Process

Accessibility was built into the resource as it was written. This resource went through the following stages of review:

1. **Template check.** The project coordinator and instructional designers checked the first chapter for accessibility as a model to ensure later chapters meet accessibility standards.
2. **Progress checks.** The manuscript was checked at approximately 40%, 60%, and 80% completion to catch and correct accessibility issues in production.
3. **Specialist review.** Once the manuscript was complete, an accessibility expert conducted an accessibility check.
4. **Remediation.** The author and the project manager who conducted the earlier progress checks worked together to remediate the issues identified by the accessibility expert.
5. **Final check.** The resource underwent another full check by the accessibility expert before the book was deemed ready for publication.

This process combines ongoing checks during writing with a final specialist review, so issues are caught early and confirmed fixed before publication.

Report an Accessibility Problem

We are always looking for ways to make our resources more accessible. If you run into a barrier while using this resource, please let us know so we can look into it. When you [contact us](#), it helps to include:

- The name of the resource
- The location of the problem (a web address or page/chapter description)
- A description of the problem you encountered
- The device, browser, and any assistive technology you were using (e.g., Windows 11, Google Chrome, JAWS screen reader)

Contact us: [De Anza ZTC team](#)

This statement was last updated August 2026

This resource was project-managed by **Shagun Kaur**, overseeing scope, budget, and schedule; stakeholder coordination via monthly meetings; training and platform support (LibreTexts, troubleshooting, technical support); QA/accessibility check scheduling; and licensing and copyright governance.

A Note on Known Limitations

No accessibility check is perfect. We've made a genuine, sustained effort to catch and fix issues throughout the writing and review process described above, but some barriers may remain, especially in complex or interactive content. We'd rather be upfront about that than claim a standard of perfection we can't guarantee.

AI Use

AI Use in Course Development – Transparency & Standards

As part of developing this textbook and its accompanying course materials, we used artificial intelligence (AI) as a supportive educational technology to assist with idea generation, content organization, and early-stage drafting. Our use of AI was guided by principles of transparency, academic integrity, accessibility, and human oversight. At no point was AI used as a substitute for faculty expertise, scholarly judgment, or pedagogical decision-making.

The AI's role was limited to ideation and preliminary content generation. The course authors extensively reviewed, revised, verified, and edited all AI-generated material. Faculty members evaluated content for accuracy, clarity, inclusivity, accessibility, relevance, and alignment with course learning outcomes. In many cases, AI-generated text served only as a starting point and was substantially modified, expanded, reorganized, or rewritten during development.

We also recognized that AI systems can reflect biases, inaccuracies, omissions, or outdated information. For this reason, we accepted no content without human review and independent verification. We evaluated all factual claims, educational recommendations, examples, and references against current evidence, professional standards, and the authors' subject matter expertise. The human authors ultimately remain responsible for the content, not the technology used during development.

Our use of AI mirrors how many professionals increasingly use emerging technologies in education and the workplace: as tools that can enhance efficiency, creativity, and productivity while still requiring critical thinking, ethical judgment, and expert oversight. By incorporating AI transparently and responsibly, we sought not only to improve the development process but also to model thoughtful technology use for students navigating an increasingly digital and AI-assisted world.

Finally, we believe that innovation must be balanced with a commitment to equity and accessibility. Throughout the development process, all content was reviewed to ensure it met accessibility standards and supported diverse learners. AI-assisted drafting was only one component of a much larger process that included faculty collaboration, scholarly review, accessibility evaluation, and continuous revision. The result is a resource that reflects both the opportunities offered by emerging technologies and the enduring importance of human expertise in teaching and learning.

Copyright & Fair Use Considerations

We aligned with evolving IP policy frameworks, especially insights from the U.S. Copyright Office:

- The [Office's Part 3 report on AI model](#) training highlights the importance of fair use analysis and restricts reliance on generative AI when it could replicate copyrighted works—our use was limited to non-protected, openly licensed materials.
- This cautious approach is consistent with [guidance](#) that encourages transformative use, transparency, and limiting output that could inadvertently mirror protected content.

About the Book

This page titled [About the Book](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by Clare Nguyen.

About the Authors

This page titled [About the Authors](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by Clare Nguyen.

1: Introduction to Data Science

The [U.S. Census Bureau](#) defines data science as "a field of study that uses scientific methods, processes, and systems to extract knowledge and insights from data." In data science we analyze data to pull out meaningful insights, draw conclusions, make predictions, and support decision making.

Data science is part of everyday life. If you've ever seen personalized ads or browsed recommended song lists based on music that you're currently listening, you've experienced data science at work. Data science combines skills from statistics and mathematics, computer science, and domain-specific knowledge. This makes the field very versatile, with domains spanning healthcare, finance, business, education, scientific and social research, and many more.

The first important part of data science is the data itself. In this class we will discover the importance of using a large dataset with diverse data points if we want our conclusions about the data to be useful. Then, with an appropriate large and diverse dataset, we will use programming techniques to apply statistical calculation to explore the data, make predictions and inferences, and draw conclusions about the data.

A Little Tour of Data Science

We now take a look at a simple example of data science at work by exploring data from the classic book *The Secret Garden* by Frances Hodgson Burnett. We'll investigate what a large and diverse dataset means, then we explore the dataset and see whether we can make a prediction with the data. Along the way we'll see some of the code and statistical concepts that are used by data scientists.

[The book *The Secret Garden* is online in the public domain at [Project Gutenberg](#). You can use the link to read the online book if you like.]

Gather Data

The first step is to locate the data and load it into the computer memory so we can use the Python programming language to work with it.

```
import requests

# download the entire book from the Project Gutenberg website
url = 'https://www.gutenberg.org/cache/epub/113/pg113.txt'
garden = requests.get(url).text

# separate into chapters
garden_chapters = garden.split('CHAPTER ')[1:]
print(garden_chapters[0][:200]) # print the first 200 characters of the first chapter to
```

Output?

```
I.
THERE IS NO ONE LEFT
```

```
When Mary Lennox was sent to Misselthwaite Manor to live with her uncle
everybody said she was the most disagreeable-looking child ever seen.
It was true, too. She had
```

Now that we have the book chapters stored in memory at a location or variable named `garden_chapters`, we can start the data analysis. One of the most common starting points in data analysis starting point is to look at the size of the data. Below we find how many characters and how many words the book has.

```
# Find the number of characters and number of words in the book

# initialize counters at 0, since there are 0 characters and 0 words at the start
```

```
total_characters = 0
total_words = 0

# go through the chapters and count all printable characters and words
for chapter in garden_chapters:
    total_characters += len(chapter) - chapter.count(' ')
    words = chapter.split()
    total_words += len(words)

print(f'Number of characters: {total_characters:,d}')
print(f'Number of words: {total_words:,d}')
```

Output?

```
Number of characters: 379,495
Number of words: 83,346
```

In the output above we see that the character dataset and the word dataset are both large, and each dataset is likely diverse because it has a variety of characters or words, as most books do.

Analyze Data

Next we will analyze specific words in the text and see whether the data tells us something about the book.

First, we take the list of all chapters of the book and store them in a special container that's commonly used for data analysis: a DataFrame. A DataFrame is a table that consisting of rows and columns. We'll put each chapter in a row of the DataFrame, so the DataFrame has one column which contains the chapters of the book.

```
# Store the chapters into a table or a DataFrame and name it chapters

import pandas as pd

# store the chapters into a table where each chapter is one row of the table
chapters = pd.DataFrame({'chapters': garden_chapters})

# print the first 5 rows of the table to check
chapters.head()
```

Output?

chapters	
0	I.\n\nTHERE IS NO ONE LEFT\n\n\n\nWhen Mar...
1	II.\n\nMISTRESS MARY QUITE CONTRARY\n\n\n\n\n...
2	III.\n\nACROSS THE MOOR\n\n\n\n\nShe slept a...
3	IV.\n\nMARTHA\n\n\n\n\nWhen she opened her e...
4	V.\n\nTHE CRY IN THE CORRIDOR\n\n\n\n\nAt fir...

From the output above, we can see the first five rows of the DataFrame, numbered 0-4. Each row shown is the beginning part of a chapter.

The book *The Secret Garden* has several major characters and their names are: Mary, Colin, Dickon, Craven, Martha, Ben, and Medlock. Without reading the book, we want to use the dataset of the words in the book to predict which of these characters is the main character of the story.

To do this, we'll count the number of times each name appears in the book. Counting the number of occurrences of a data value in a dataset is a basic statistics concept. From the count of each name we can infer who the main character is by choosing the name with the highest count.

```
# Find the number of times the names of the characters appear

# identify the names of the characters to count
names = ['Mary', 'Colin', 'Dickon', 'Craven', 'Martha', 'Ben', 'Medlock']

# create room in memory to store the counts of each name
name_counts = pd.DataFrame(columns=['Name', 'Count'])
counts = {}

# go through each name and count how many times it appears in the chapters
for name in names:
    count = 0
    for chapter in chapters['chapters']:
        count += chapter.lower().count(name.lower())
    counts[name] = count

# store the name counts into a DataFrame for easier viewing
for name, count in counts.items():
    name_counts = pd.concat([name_counts, pd.DataFrame({'Name': [name], 'Count': [count]})])

name_counts

Output?
```

	Name	Count
0	Mary	694
1	Colin	333
2	Dickon	311
3	Craven	114
4	Martha	194
5	Ben	148
6	Medlock	134

From the output above, we see that Mary has the highest count, so we can conclude or *infer* that Mary is most likely the main character. When we infer something, we reach a conclusion about it by combining facts with reasoning, rather than by directly observing it. In this case the fact (Mary has the highest name count) and the reasoning (the most important character will appear the most times in the book) are used to infer that Mary is the main character, without actually reading the book.

Visualize Data

Even though the name count for Mary is already very convincing, often in data science we display the resulting data in an appropriate plot to clearly show the pattern in the data. This is called *data visualization*.

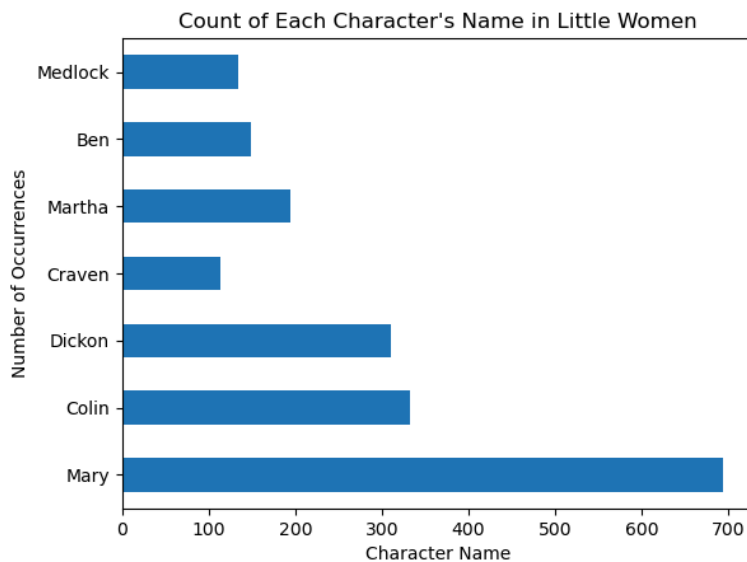
Let's plot the counts of all the names and see if data visualization really shows who the main character is.

```
# Plot the count of each name

import matplotlib.pyplot as plt

name_counts.plot(x='Name', y='Count', kind='barh', legend=False)
plt.xlabel("Character Name")
plt.ylabel("Number of Occurrences")
plt.title("Count of Each Character's Name in Little Women")
plt.show()
```

Output?



It's pretty clear that Mary's name appears the most in the book, so we can infer that Mary is the main character of the story. If you've read the book *The Secret Garden*, you can confirm whether our data science exercise reaches a correct conclusion, and if you haven't read the book, a quick online search can also confirm our conclusion.

Through this quick tour, we see a few key components of a data science application:

- A large and diverse dataset. If we only looked at one chapter of the book (a small dataset), the count of each character name in the chapter may not be the same as the count in the entire book and we might arrive at the wrong main character. This is why choosing a large and diverse dataset is important.
- Using the appropriate statistical methods and visualization methods, we can see different patterns in the data, such as how large the dataset is.
- With a good dataset and appropriate statistical methods, we can come to a correct conclusion about a characteristic of the book (such as who the main character is) without personally reading the book.

It is the extraction of meaningful patterns or insights from data that makes data science an important tool for many industries, and it also makes data science the basis for machine learning and artificial intelligence.

Summary

A data scientist uses programming and statistical techniques to analyze large and diverse datasets and infer characteristics of the data.

The ability to make inferences from data allows for data driven decisions, such as what medical treatment to use or what business policy to implement, and leads to the development of artificial intelligence models that work with and make predictions based on insights and relationships in the data.

2: Numbers and Strings

In the previous chapter we saw that data scientists apply statistical concepts on a dataset to find patterns or insights from the data. To be efficient with statistical calculations on the data, data scientists take advantage of the fast computation speed of the computer. In this chapter we learn how to use the computer to do math calculations and print the results along with text strings.

We want to write math instructions that can be understood by the computer so that it can follow the instructions and produce the result we need. Writing instructions for the computer to do work is called *programming*, and the list of instructions makes up a *program* or *code*. The language that we use to write code is the Python programming language. Python is a popular language for data science because it supports many data science tasks.

When we write a document in the English language, we need to follow the grammar rules or the syntax of the English language. Similarly, when we write instructions in the Python language, we need to follow the syntax of Python. This chapter covers the syntax of instructions that do basic calculations and printing the results.

Printing Text

To tell the computer to print a line of text or a result to screen, use the `print` function. A function is a block of Python code that does a task, and the print function's task is to print the given data to screen.

The general syntax to run or *call* any function is:

```
function_name( input to the function )
```

When we write an instruction, or a *statement*, in the above format in a Code cell and run the cell, the computer will run the block of code that belongs to the `function name`, and the block of code will use the `input to the function` to do its task.

Therefore, to print two lines of text to screen, we write:

```
print("Hello world")
print('Python is used for this output')
```

Output?

```
Hello world
Python is used for this output
```

Note that:

- The text `Hello world` is the input to the function `print`, and the text needs double quotes or single quotes around it. When the text is printed, Python automatically removes the quotes.
- Each `print` statement produces one line of output.

Note that:

- The text `Hello world` is the input to the function `print`, and the text needs double quotes or single quotes around it. When the text is printed, Python automatically removes the quotes.
- Each `print` statement produces one line of output.

Printing blank lines

If blank lines are needed in the output, there are two ways:

1. Use `"\n"` at the end of a line of text. `\n` is short for newline.

2. Use `print()` with no input

Can you predict how many lines of output will be printed from the code below?

```
print("First line of text that's followed by a blank line.\n")
print("Second line of text that's also followed by a blank line.")
print()
print("Third line of text, followed by no blank line.")
print("Fourth line of text.")
```

Output?

First line of text that's followed by a blank line.

Second line of text that's also followed by a blank line.

Third line of text, followed by no blank line.

Fourth line of text.

Arithmetic Expressions

An arithmetic expression is made of *operands* and *operators*.

An operand is a data value, and an operator is a symbol that represents the work that the computer will do.
For example:

`4 + 3`

is an expression where 4 and 3 are the operands, and `+` is the operator that tells the computer to add.

In a Code cell, we can simply type the math expression and then run the cell to tell the computer to do the work and display the result of the math expression.

`4 + 3`

Output?

7

An arithmetic expression can have the following common operators:

- `+`: add
- `-`: subtract
- `*`: multiply (note that it's an asterisk, not `x`)
- `/`: divide
- `%`: modulus, which finds the remainder of a division
- `**`: exponent
- `( )`: parentheses

An arithmetic expression can have the following common operators:

- `+`: add
- `-`: subtract
- `*`: multiply (note that it's an asterisk, not `x`)
- `/`: divide
- `%`: modulus, which finds the remainder of a division

- `**`: exponent
- `( )`: parentheses

Predict the output of each expression below and then print it to check your prediction. Predict the output of each expression below and then print it to check your prediction.

```
4 - 3
```

Output?

Note that for division, even when the result is a whole number, Python returns the result as a decimal number, as shown in the next Code cell.

```
1
```

```
4 * 3
```

Output?

```
12
```

```
4 / 3
```

Output?

```
1.3333333333333333
```

```
4 % 3
```

Output?

```
1
```

```
4 ** 3
```

Output?

```
64
```

Note that for division, even when the result is a whole number, Python returns the result as a decimal number, as shown in the next Code cell.

```
10 / 2
```

Output?

```
5.0
```

Printing Expressions

To print the result of an arithmetic expression together with some text, we need to use the print function again.

The print function can be used to print multiple data values, by placing the data values inside `()`, in the order that they should be printed, and separate them by comma.

In the following examples, the print function has two inputs. Can you predict what the print output is?

```
print(7, "is the number seven.")
```

Output?

In the example above, we see that the print function can print a number and a text string. By default, the two inputs are printed with a space in between them.

The print function can also print the result of an expression if we give the expression as an input to the print function.

```
7 is the number seven.
```

In the example above, we see that the print function can print a number and a text string. By default, the two inputs are printed with a space in between them.

The print function can also print the result of an expression if we give the expression as an input to the print function.

```
print("2 + 9 is", 2 + 9)
```

Output?

Can you predict the output of the print statement below?

How many lines of text are printed?

```
2 + 9 is 11
```

Can you predict the output of the print statement below?

How many lines of text are printed?

```
print("The result of 2 + 3 is", 2 + 3, "\nRight?")
```

Output?

```
The result of 2 + 3 is 5
Right?
```

Evaluating Expressions

Computers evaluate arithmetic expressions, or do the calculation, in the same PEMDAS order of precedence that's defined in math:

- Parentheses
- Exponents
- Multiplication and Division (left to right)
- Addition and Subtraction (left to right)

This means parentheses are evaluated first, and Addition/Subtraction are evaluated last and in the order from left to right of the expression.

In the expression: $1 + (2 ** 3 + 4) / 5 * 2$ there are 7 operators. The order that they are evaluated are:

1. Parentheses: $(2 ** 3 + 4)$
2. Exponent: $2 ** 3$ or $2^3 = 8$
3. Addition: $2 ** 3 + 4$ or $8 + 4 = 12$
4. Division: $(2 ** 3 + 4) / 5$ or $12 / 5 = 2.4$
5. Multiplication: $(2 ** 3 + 4) / 5 * 2$ or $2.4 * 2 = 4.8$
6. Addition: $(1 + (2 ** 3 + 4) / 5 * 2)$ or $1 + 4.8 = 5.8$

$(1 + (2 ** 3 + 4) / 5 * 2)$

Output?

5.8

Applying Expressions

In their data analysis work, data scientists often write multiple expressions so that the computer can run them to get results, which are then used for extracting insights from the data.

For example, given that a class has four exams, each with a possible score from 0 to 100 points. If the four exams are a major factor in determining the grade of students, then what is the most likely grade for a student who earned 84, 90, 94, 82 on the exams?

In the Code cells below:

- Write an expression to calculate the average exam score and run the cell to see the result.
- Observe the output of the cell and determine the most likely letter grade (A, B, C...) based on the average score. Then in the Code cell, write a statement to print the most likely grade that you think the student earned.

```
# Write an expression to calculate the average exam score:
```

Output?

```
# Write a print statement to print the most likely letter grade,  
# based on the average score:
```

Output?

What you did in the example above is applying expressions to the data or the exam scores, so you can get the insight (the average score) and predict the likely grade for a student.

Comments in Code

In the two Code cells above, the lines of text in English is called a *comment*. It is a way for us to write a quick note to ourselves or to the reader, to explain what the code does or explain a result of a calculation.

A comment in a Code cell starts with a `#` symbol, and ends at the end of the line.

Variables

Many times the calculations that we need to do can take several steps, where each step produces an intermediate result, and the intermediate result is then used in the next step.

An example is to calculate the speed of two cars and then print the average of their speeds. Suppose Car A takes 1.5 hours to travel 47 miles, and Car B takes 0.5 hours to travel 12 miles. What is the average speed of the two cars?

Here is the order of the 3 steps that it takes to answer the question:

- Step 1: Find the speed of Car A
- Step 2: Find the speed of Car B
- Step 3: Find the average speed of the two cars

In order to do step 3, we need to save the results of steps 1 and 2, so we can tell the computer which two values to average. The results of steps 1 and 2 are the intermediate results, they help us to get to the final result which is the average speed.

In programming it is easy to create a temporary storage place for an intermediate data value: we use an *assignment* operator.

Assignment Operator

The *assignment* operator is the `=` symbol. Here is how to use the assignment operator to store an intermediate result:

```
variable_name = data value
```

The instruction or statement above tells the computer to store data by using the `=` or assignment operator. Therefore the entire instruction is called the *assignment statement*.

Note that the `=` operator or assignment operator does not work like the `=` of math. In math, `=` means the two values on the left and right are the same value. In programming, `=` means saving data into storage.

- The right side of the `=` operator is always for data.
- The left side of the `=` operator is always the name of the data storage location or the name of the *variable*.

Recall that an operator is a symbol that causes the computer to do some work. The `=` operator tells the computer to store the value on the right into a named storage location on the left.

Below is the Python code for the 3 steps to find the average speed of the two cars:

```
# Step 1: find the speed of Car A and store the result in a variable
carA_speed = 47 / 1.5

# Step 2: find the speed of Car B and store the result in a different variable
carB_speed = 12 / 0.5

# Step 3: refer to the 2 variables to find the average speed
average_speed = (carA_speed + carB_speed) / 2

print('Car A speed:', round(carA_speed, 1), 'mph')
print('Car B speed:', round(carB_speed, 1), 'mph')
print('Average speed of the 2 cars:', round(average_speed, 1), 'mph')
```

Output?

```
Car A speed: 31.3 mph
Car B speed: 24.0 mph
Average speed of the 2 cars: 27.7 mph
```

When there are multiple instructions that work together in a logical, step-by-step order to tell the computer to do some work, like the block of code above, then we have a *computer program*.

When there are multiple instructions that work together in a logical, step-by-step order to tell the computer to do some work, like

the block of code above, then we have a *computer program*.

In the program above, note that:

- A variable name should be descriptive: it should describe the data that is stored.
The variable name `carA_speed` makes it clear which data is being stored at that location. A variable name should not be `x` or `y` as is sometimes used in math equations, because a program typically has many variables to store intermediate data, and variable names like `a`, `b`, `x`, `y` make it difficult to remember what results are stored at these locations.
- Variable names must be made of letters, numbers, and underscores, and the first character in the name cannot be a number. If the variable name begins with a number, Python will not understand the name and the program will have errors. It's similar to misspelling a word or using the wrong word in an English sentence. The sentence has an error that makes it difficult for the human reader to understand it.

```
# Run this Code cell to observe the error with a variable name.  
# Can you fix the error and run the cell to see that the code works?
```

```
first = 25  
2nd = 30  
print("The sum of the 1st and 2nd numbers is", first + second)
```

Output?

Ways to Assign Data

In the example to calculate the average car speed, we see one way to assign data:

```
carA_speed = 47 / 1.5
```

The data to be stored in `carA_speed` is the result of an expression.

There are two other common ways to assign data:

```
carA_speed = 42.5
```

The data to be stored is a *literal value*, or a value typed directly into the statement.

```
carA_speed = carB_speed
```

The data to be stored comes from a different variable.

This way is the most confusing when we first encounter it: do we set `carA_speed` to `carB_speed`? Or is it the other way around?

If we remember that it's always the data on the right being stored in the variable on the left, then it's not confusing.

The assignment statement tells the computer to store the value of `carB_speed` into the variable `carA_speed`.

Side Note

The last example `carA_speed = carB_speed`

clearly shows the difference between the math `=` operator and the programming `=` operator.

In math, `carA_speed = carB_speed`

can also be written as `carB_speed = carA_speed`

because the math `=` operator means *equal to*. Both versions say that `carA_speed` is the same as `carB_speed`, so the `=` operator is commutative, which means we can swap the variables on the left and right sides.

In programming, `carA_speed = carB_speed`

means store the speed of Car B into the variable `carA_speed`, so `carA_speed` is updated to a new value.

But `carB_speed = carA_speed`

means store the speed of Car A into the variable `carB_speed`, so `carB_speed` is updated to a new value.

The two expressions do completely different tasks.

Update a Variable

Within one program, a variable name refers to a specific data value that has been stored or assigned to it. This means:

- If we have four unique data values to save or store, then we need 4 variable names and four assignments to store the four data values.
- If we need to update a particular variable with new data, then we simply assign the new value to that variable name.

```
# Use three different variables names when we want to store three unique data values:
first_num = 10
second_num = 20
third_num = 30

print("Original 3 numbers:", first_num, second_num, third_num)

# Update first_num with a new value
first_num = 40
print("The 3 numbers after updating the first number:", first_num, second_num, third_num)

Output?
```

Note that after updating `first_num`, the value `10` that was originally in `first_num` is gone and we can't get to it anymore.

```
Original 3 numbers: 10 20 30
The 3 numbers after updating the first number: 40 20 30
```

Note that after updating `first_num`, the value `10` that was originally in `first_num` is gone and we can't get to it anymore.

```
# a. Can you predict the output of this program?
# b. How do you store and print both scores?

score = 95
score = 88
print("The 2 scores are:", score)

Output?
```

```
The 2 scores are: 88
```

Summary

In this chapter we learn how to tell the computer to perform arithmetic calculations and print the results. We also saw that storing intermediate results in variables allows us to build programs that solve problems one step at a time.

3: Data Types

We've seen that it's good to be able to store intermediate results in a program, so that we can use these results later in the program. Now we'll also see why it's important to know the type of data being stored. There are 4 main data types that we'll work with:

- Numbers
- Strings
- Sequences
- Tables

So far we've seen numbers and strings at work. In the print statement:

```
print("The 2 numbers are:", 5, "and", 6)
```

There are four inputs to the print function:

- two text strings with quotes: "The numbers are:", "and"
- two numbers: 5, 6

The reason why we want to be aware of the data type of a variable is because Python has many useful functions for each type of data. Recall that a function is a block of code that does some work. If we know the data type of a variable, then we can take advantage of these useful functions and ask Python to do the work on the data for us, so that we don't have to write code to do the work ourselves.

To check the data type of a variable, we can use the function `type`:

```
type( variable_name )

num1 = 82
num2 = 7.5

print("Type of num1 data:", type(num1))
print("Type of num2 data:", type(num2))

Output?
```

The data type `int` means that the data are *integers* or whole numbers.

The data type `float` means that the data are *floating point numbers* or decimal numbers.

```
# Can you predict the output of this code?
num1 = num2
print("Type of num1 data:", type(num1))
print("Type of num2 data:", type(num2))

Output?
```

```
Type of num1 data: <class 'float'>
Type of num2 data: <class 'float'>
```

Numbers and Strings

It's easy to differentiate between the *number* data type and *string* data type.

- A *number* is made of one or more digits 0-9, and an optional decimal point .
- A *string* is a sequence of text characters that are surrounded by double quotes or single quotes

Numbers and Their Properties

Python divides the numbers into two basic types:

- `int`: short for *integers*, includes all whole numbers such as: 12, -4, 67823, -37, 0
- `float`: short for *floating point numbers*, includes all numbers with a decimal point such as: 5.3813, -0.002841, 2683.39, 0.0

Evaluating Expressions with Numbers

- When an expression with both `int` data type and `float` data type is evaluated, the result is a `float`.

```
num1 = 6.2
num2 = 5
difference = num1 - num2
print("Type of num1:", type(num1))
print("Type of num2:", type(num2))
print("Type of the difference:", type(difference))
```

Output?

- When evaluating expressions with `float` data types, the result may be off by the last one or two digits. This is because some decimal numbers can have infinitely many digits, but data on a computer can only be stored with a finite number of bits (typically 64 bits), so some decimal numbers will have a tiny rounding error.

For most calculations in data science, the tiny rounding error is not a problem since the input data is limited to a certain number of significant digits, therefore we always round the output to the same number of significant digits, which means the last one or two digits in the output can be ignored.

```
import math

# Suppose we have an original input number which is precise to
# 1 digit after the decimal point:
num = 2.0

# Using Python's square root function, we find the square root of num.
# Note that we don't need to write code to calculate the square root,
# which involves many lines of code and a complex algorithm.
# This is because we recognize that num is a number data type, and
# Python already provides the square root function for numbers.
square_root = math.sqrt(num)

# Now we take the square root and square it, which mathematically
# should result in the exact original num value:
print("The square is", square_root ** 2)
```

Output?

The square is 2.0000000000000004

We see that the output is not exactly the same as the original `num` value. However, since `num` has a precision of one digit after the decimal point, if we also consider only one digit after the decimal point for the output, then the output exactly matches `num`.

Common Functions for Numbers

Python has many useful functions for the number data type. We'll cover some of the most common ones here and we'll discuss other functions as we continue further into data science.

Rounding a Number

To round a number to a certain number of digits after the decimal point, we use the `round` function:

```
round ( input number, number of digits after decimal point )

# Continuing with the squaring of the square root calculation above,
# Original print statement:
print("The square is", square_root ** 2)

# With rounding to 1 digit after the decimal point:
print("With rounding, the square is", round(square_root ** 2, 1))
```

Output?

```
The square is 2.0000000000000004
With rounding, the square is 2.0
```

Absolute Value of a Number

To take the absolute value, or to find the positive component of a number, we use the `abs` function.

```
abs( input number )
```

A common use of the absolute value in data science is to find the difference between two numbers.

```
num1 = 2.5
num2 = 3
# What's the difference between the 2 numbers?

# Without using the abs function, we could end up with 2 different results,
# depending on how the difference is calculated:
print("The difference is", num1 - num2)
print("The difference is", num2 - num1)

print("\nWith absolute value:")
# But if we use the absolute value of result, then we always get the
# true difference between the 2 numbers:
print("The difference is", abs(num1 - num2))
print("The difference is", abs(num2 - num1))
```

Output?

```
The difference is -0.5
The difference is 0.5
```

```
With absolute value:
The difference is 0.5
The difference is 0.5
```

Largest and Smallest of Several Numbers

When we have several numbers and want to find the largest or smallest of them, we can use the `max` and `min` functions.

```
max( two or more input numbers )
min( two or more input numbers )
```

where the input numbers are separated by commas.

```
print("The largest value of 1,5,0,-2,4 is", max(1,5,0,-2,4))
print("The smallest value of 1,5,0,-2,4 is", min(1,5,0,-2,4))
```

Output?

```
The largest value of 1,5,0,-2,4 is 5
The smallest value of 1,5,0,-2,4 is -2
```

Square Root of a Positive Number

Finding the square root of a positive number takes two steps:

1. Tell Python to look at the `math` module.

```
import math
```

A module is a grouping of similar functions. The `math` module contains many useful math functions.

2. Use the `sqrt` function of the `math` module:

```
math.sqrt( input number )
```

Note: Finding the square root of a negative number is part of a branch of math called Complex Analysis, which is not used in beginning data science.

```
import math

num = 5
print("The square root of num:", math.sqrt(num))
print("The square root of num:", round(math.sqrt(num),2))
```

Output?

```
The square root of num: 2.23606797749979
The square root of num: 2.24
```

Strings and Their Properties

Any text enclosed in double quotes or single quotes is considered a string data type in Python. The string data type is shown as `str` in Python.

```
# Number which is an int data type
num = 5
print(num, "is type", type(num))

# Number which is a string data type
num = "5"
print(num, "is type", type(num))

# Number which is a string data type
num = '5'
print(num, "is type", type(num))

# Strings typically are one or more characters, making up words or sentences
word = "Hello"
sentence = "Hello, World!"
print(word, "is type", type(word))
print(sentence, "is type", type(sentence))
```

Output?

```
5 is type <class 'int'>
5 is type <class 'str'>
5 is type <class 'str'>
Hello is type <class 'str'>
Hello, World! is type <class 'str'>
```

Common Functions for Strings

Just like with numbers, Python provides many useful functions for strings. Here we cover some of the most common ones.

Joining Two Strings

To join two strings together, one after the other, is to *concatenate* the strings. To concatenate two strings, we use the `+` operator, just like with numbers, but the result is not an addition, it's a concatenation of the strings.

```
first string + second string
```

```
string1 = "Python"  
string2 = "Language"
```

```
print("Concatenate:", string1 + string2)  
print("Concatenate with space in the middle:", string1 + " " + string2)  
print("Concatenate with a new string in the middle:", string1 + ' **--** ' + string2)
```

Output?

```
# Can you predict the output of this program?
```

```
str1 = "hello"  
str2 = "good bye"  
output_string = str1 + " and " + str2  
print(output_string)
```

Output?

```
hello and good bye
```

Comparisons

In addition to doing math calculations with numbers, many times we need to compare numbers when we analyze the data. For example, if 90% is a cutoff for an A grade, we will need to compare a student's overall percentage with 90 to see whether the student earned an A grade.

Just like with arithmetic operators, the Python language also uses the same *relational operators* or comparison operators as math when comparing two numbers or two strings.

Relational operators

There are 6 relational operators:

```
> : greater than  
< : less than  
>= : greater than or equal to  
<= : less than or equal to  
== : equal to  
!= : not equal to
```

When Python evaluates a relational expression, the result is the value `True` or `False`.

Comparing numbers

Comparing numbers works as expected in math.
Can you predict the output of the following code?

```
num1 = 5
num2 = 8

print("num1 > num2?", num1 > num2)
print("num1 < num2?", num1 < num2)
print("num1 >= num1 - 1?", num1 >= num1 - 1)
print("num1 <= num1?", num1 <= num1)
print("num1 == num2?", num1 == num2)
print("num1 != num2?", num1 != num2)
```

Output?

Comparing strings

Strings can also be compared. When two strings are compared:

- They are compared alphabetically.
The string **ab** comes before **ac** or is "less than" **ac**.
- If the two strings have the same starting characters, but one string has extra characters at the end, then the shorter string comes before the longer string or is "less than" the longer string.
The string **ab** comes before **abc** or is "less than" **abc**.
- Uppercase letters come before lowercase letters.
The string **ab** comes after **Ab** or is "greater than" **Ab**.

Can you predict the output of the following code?

```
num1 > num2? False
num1 < num2? True
num1 >= num1 - 1? True
num1 <= num1? True
num1 == num2? False
num1 != num2? True
```

Comparing strings

Strings can also be compared. When two strings are compared:

- They are compared alphabetically.
The string **ab** comes before **ac** or is "less than" **ac**.
- If the two strings have the same starting characters, but one string has extra characters at the end, then the shorter string comes before the longer string or is "less than" the longer string.
The string **ab** comes before **abc** or is "less than" **abc**.
- Uppercase letters come before lowercase letters.
The string **ab** comes after **Ab** or is "greater than" **Ab**.

Can you predict the output of the following code?

```
string1 = "Python"
string2 = "Programming"
string3 = "PYthon"
string4 = "pY"

print("string1 < string2?", string1 < string2)
print("string1 < string3?", string1 < string3)
print("string3 < string4?", string3 < string4)
```

Output?

```
string1 < string2? False
string1 < string3? False
```

```
string3 < string4? False
```

Summary

In this chapter we learn some useful Python functions for the number and string data types so that we can keep our coding short and still do some complex work. For example, using `math.sqrt(num)` is much shorter than writing code to [calculate the square root](#) of num.

However, data science is a field that works with large datasets, so in the next chapter we will discover two more data types that make it much easier to work with large datasets: sequences and tables. These data types can store many data values under one variable name, making it easier for us to work with large datasets.

4: Sequences and Tables

In the previous chapters we learned to work with numbers and strings, two basic data types that represent a number or a text string. In this chapter we continue by exploring a grouping of numbers or strings, which is a useful way to work with large datasets.

Suppose we need to store the names of all 15 cities in Santa Clara County, California. It would be cumbersome to create 15 `str` variables, each storing one city name. Instead we can create one sequence variable, called an `array`, that can store all 15 strings under one variable name.

Likewise, if we want to store the names of 50 students and their quiz scores, it would be cumbersome to create 50 variables for student names and 50 variables for quiz scores. Instead we can use a table, or a `DataFrame`, to store all the names and all the scores under one variable name.

Sequences or Arrays

In data science, a common way to store a sequence of data values is with an `array`. An array can store multiple data values under one variable name, making it easier to refer to the sequence of data in our code.

To create an array from a sequence of data, we use the format:

```
np.array([ list of comma-separated data values ])
```

In the code below we create an array named `cities` that can store the 15 city names in Santa Clara County.

```
import numpy as np

cities = np.array(['Campbell', 'Cupertino', 'Gilroy', 'Los Altos',
                  'Los Altos Hills', 'Los Gatos', 'Milpitas', 'Monte Sereno',
                  'Morgan Hill', 'Mountain View', 'Palo Alto', 'San Jose',
                  'Santa Clara', 'Saratoga', 'Sunnyvale'])

print("Array of Cities:")
print(cities)
```

Output?

```
Array of Cities:
['Campbell' 'Cupertino' 'Gilroy' 'Los Altos' 'Los Altos Hills' 'Los Gatos'
 'Milpitas' 'Monte Sereno' 'Morgan Hill' 'Mountain View' 'Palo Alto'
 'San Jose' 'Santa Clara' 'Saratoga' 'Sunnyvale']
```

From the code above we observe that:

- We need to import a Python module named `Numpy`, and typically `Numpy` is shortened to `np`.
- The `array` data type is defined in `Numpy`, therefore `np.array( )` is a function that tells Python to create an array, ready to store a sequence of multiple data values.
- We only need to create one variable named `cities`, and it can store 15 strings because it is an `array` data type.
- The input to the function `np.array( )` is the sequence of city names, separated by comma, and surrounded by `[ ]` to group the names together.
In Python when we use `[ ]` to group multiple values together, the grouping is called a `list`.

A Numpy array is most useful when it contains numbers, because a Numpy array comes with many math functions that can be applied to all the numbers in the array.

In the Code cell below, we create an array called `temps` and store the following temperatures: 61, 63, 67, 70, 73, 79, 81, 81, 81, 76, 67, 60

The temperatures are the 12 [monthly average high temperature](#) for the city of San Jose.

```
# Create an array named temps
SJ_temps = np.array([61, 63, 67, 70, 73, 79, 81, 81, 81, 76, 67, 60])
# Print the temps array
print(SJ_temps)
```

Output?

```
[61 63 67 70 73 79 81 81 81 76 67 60]
```

Functions for Arrays of Numbers

Just as there are multiple useful functions for numbers and strings, there are multiple useful functions for arrays of numbers.

Arithmetic

When an array contains numbers, and the array is evaluated in a math expression, the arithmetic operations *automatically* apply to all numbers in the array.

Suppose we want to decrease each temperature in the `temps` array above by two degrees, we can simply subtract 2 from the array.

```
lower_temps = SJ_temps - 2
print("Original temp:", SJ_temps)
print("Lowered temps:", lower_temps)
```

Output?

```
Original temp: [61 63 67 70 73 79 81 81 81 76 67 60]
Lowered temps: [59 61 65 68 71 77 79 79 79 74 65 58]
```

All the arithmetic operations (add, subtract, multiply, etc.) work the same way when one operand is an array and the other operand is a number.

If there are two arrays of numbers in an arithmetic expression, then the math operation is performed on corresponding numbers in each array.

```
# Create an array of average monthly high temperatures in Los Angeles
LA_temps = np.array([70, 69, 70, 74, 74, 78, 83, 84, 84, 81, 74, 68])

# Find the temperature difference between Los Angeles and San Jose
temp_diff = LA_temps - SJ_temps
print("Difference in temps:", temp_diff)
```

Output?

```
Difference in temps: [ 9  6  3  4  1 -1  2  3  3  5  7  8]
```

All arithmetic operations work the same way when the operands are both arrays.

Common Array Functions

Some useful functions when working with an array of numbers:

- `np.size( array )`: returns the count of the numbers in the array
- `np.sum( array )`: returns the total of all the numbers in the array
- `np.mean( array )`: returns the mean or the average of all the numbers in the array
- `np.round( array )`: round each number in the array to a certain number of digits after the decimal point

- `np.sort( array )`: sort the numbers in the array

```
print("Number of temps:", np.size(SJ_temps))
print("Total of temps:", np.sum(SJ_temps))
print("Average of temps:", np.mean(SJ_temps))
print("Rounded Celsius temps:", np.round(SJ_temps - 32 * 9/5, 1))
print("Sorted temps:", np.sort(SJ_temps))
```

Output?

```
Number of temps: 12
Total of temps: 859
Average of temps: 71.58333333333333
Rounded Celsius temps: [ 3.4  5.4  9.4 12.4 15.4 21.4 23.4 23.4 23.4 18.4  9.4  2.4]
Sorted temps: [60 61 63 67 67 70 73 76 79 81 81 81]
```

Creating Arrays of Numbers

There are two ways to create an array of numbers when we know what the sequence of numbers are:

1. Give the sequence of numbers as input to the function `np.array`, as seen in the examples above.
`numbers = np.array([10, 5.2, 7, -1.3])`
2. Use the `range` function as input to the function `np.array`. The `range` function will automatically create a sequence of whole numbers in the range that we specify.

There are three ways to give input to the `range` function:

- `range(end number)`: start at 0, continue to step up by 1, stop right before the `end number` is reached.
- `range(start number, end number)`: start at `start number`, continue to step up by 1, stop right before `end number` is reached.
- `range(start number, end number, step)`: start at `start number`, continue to step up or step down by `step`, stop right before `end number` is reached.

```
# create an array from 0-7, step up by 1
print("range(8):", np.array(range(8)))

# create an array from 2-7, step up by 1
print("range(2, 8):", np.array(range(2, 8)))

# create an array from 2-7, step up by 3
print("range(2, 8, 3):", np.array(range(2, 8, 3)))

# create an array from 8-3, step down by 2
print("range(8, 2, -2):", np.array(range(8, 2, -2)))
```

Output?

```
range(8): [0 1 2 3 4 5 6 7]
range(2, 8): [2 3 4 5 6 7]
range(2, 8, 3): [2 5]
range(8, 2, -2): [8 6 4]
```

Note that the sequence of numbers produced by the `range` function stops at the number closest to the `end number` but never reaches the `end number`.

Tables or DataFrames

In data science one of the most commonly used Python data storage is the `DataFrame`. A `DataFrame` is a table with one or more rows and columns, making it easy to associate multiple data sequences together.

Internally a DataFrame is made of one or more arrays, where each array is one column of the DataFrame.

Create a DataFrame with Existing Data

Suppose we have 5 students with the following student data:

- Names: Emma, Tin, Ali, Jeff, Maria
- Quiz 1 scores, corresponding with the name order above: 18, 20, 15, 17, 18
- Quiz 2 scores, corresponding with the name order above: 19, 16, 15, 16, 17

A DataFrame or a table is a simple but effective way to organize data:

- Each row of the table is for one student
- Each column of the table is for one category of data: name, quiz 1 score, quiz 2 score

With a DataFrame we can label the columns with the categories of data, making it easy to distinguish between Quiz 1 scores and Quiz 2 scores.

To use a DataFrame, we need to work with the `pandas` module, which is commonly shortened to `pd`. Then we create a DataFrame data type with all the data we have above.

```
# Work with the pandas module
import pandas as pd

# 1. Create 3 lists to store data for the 3 categories
names = ['Emma', 'Tin', 'Ali', 'Jeff', 'Maria']
quiz1 = [18, 20, 15, 17, 18]
quiz2 = [19, 16, 15, 16, 17]

# 2. Create a DataFrame with the 3 lists, labeling the columns with the category names
students = pd.DataFrame({'Name': names,
                        'Quiz 1': quiz1,
                        'Quiz 2': quiz2 })

# 3. Print the DataFrame
display(students)
```

Output?

	Name	Quiz 1	Quiz 2
0	Emma	18	19
1	Tin	20	16
2	Ali	15	15
3	Jeff	17	16
4	Maria	18	17

In the code above we observe:

- In step 1: once again the `[ ]` is used to group together multiple data values into a `list`. We name each list with a descriptive variable name.
- In step 2: we create the DataFrame with the function:

```
pd.DataFrame( )
```

- Also in step 2: we put together the 3 lists and their corresponding labels by using `{ }`. This format creates a `dictionary` in Python, allowing the computer to associate each label (such as `Name`) with its list (such as all the data in `names`). Then the

dictionary becomes the input to the `pd.DataFrame( )` function.

- In step 3: we use the function `display( )` instead of the usual `print`. The `display( )` function works with the DataFrame to print in graphical format, so that we can see the column labels and each row more clearly than using `print`.

Add a column to a DataFrame

To add a column of data to the `students` DataFrame, we use the following format:

```
DataFrame_name[ column_label ] = [ sequence of data ]
```

```
# add a new column for Quiz 3 scores
students['Quiz3'] = [18, 20, 19, 18, 19]
display(students)
```

Output?

	Name	Quiz 1	Quiz 2	Quiz3
0	Emma	18	19	18
1	Tin	20	16	20
2	Ali	15	15	19
3	Jeff	17	16	18
4	Maria	18	17	19

Create a DataFrame from a File

Typing data into a DataFrame, as in the student example above, is fine when there are few data points. But often our input data is a large dataset that's stored in a file.

The `pandas` module has several functions to read in data from a file, depending on what type of file it is.

A common file type used in data science is a CSV file, or *comma separated values* file, where each data record is in a row, and each row has multiple values that are separated by commas. We will work with the data file `cars.csv` which is from the [UCI Machine Learning Repository](#).

The data in the file are characteristics of different cars, such as number of cylinders, horsepower, weight, etc.

```
# Read data from a CSV file, where the file is stored online (at a URL)

# 1. Store the URL or location of the data file
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_

# 2. Read the CSV file into a DataFrame
cars = pd.read_csv(url)

# 3. Print the DataFrame
display(cars)
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
387	27.0	4	140.0	86.0	2790.0	15.6	82	1	ford mustang gl
388	44.0	4	97.0	52.0	2130.0	24.6	82	2	vw pickup
389	32.0	4	135.0	84.0	2295.0	11.6	82	1	dodge rampage
390	28.0	4	120.0	79.0	2625.0	18.6	82	1	ford ranger
391	31.0	4	119.0	82.0	2720.0	19.4	82	1	chevy s-10

392 rows × 9 columns

In the above example the CSV file is stored online, therefore we use the `pd.read_csv( )` function with the URL as the input.

If the CSV file is stored locally on our computer, we can also use `pd.read_csv( )` but we use the name of the file instead:

```
DataFrame_name = pd.read_csv( CSV_filename )
```

where the CSV filename ends with the extension `.csv`

Get Properties of a DataFrame

As shown in the output of the `cars` DataFrame, the display function automatically shows the size of the DataFrame: the number rows and columns.

- To get the number of rows and columns without having to print the DataFrame:

```
rows, columns = DataFrame_name.shape
```

```
rows, columns = cars.shape
print("Number of rows:", rows)
print("Number of columns:", columns)
```

Output?

- To display the first rows or last rows:

```
First rows: DataFrame name.head( num of rows )
```

```
Last rows: DataFrame name.tail( num of rows )
```

The `num of rows` input is optional. If there is no input, then `head` and `tail` show 5 rows.

Number of rows: 392

Number of columns: 9

```
display(cars.head(3))
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite

```
display(cars.tail())
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
387	27.0	4	140.0	86.0	2790.0	15.6	82	1	ford mustang gl
388	44.0	4	97.0	52.0	2130.0	24.6	82	2	vw pickup
389	32.0	4	135.0	84.0	2295.0	11.6	82	1	dodge rampage
390	28.0	4	120.0	79.0	2625.0	18.6	82	1	ford ranger
391	31.0	4	119.0	82.0	2720.0	19.4	82	1	chevy s-10

- To get a list of the column labels:
`DataFrame_name.columns.to_list()`

```
labels = cars.columns.to_list()
print("Column labels:", labels)
```

Output?

Column labels: ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'm

Access Data in a DataFrame

There are three common ways to access data in a specific column:

- `DataFrame_name.column_name`
- `DataFrame_name['column name']`
- `DataFrame_name.iloc[:, column_index ]`

For the last option we use the column index instead of the column name. Each column in a DataFrame is numbered from left to right with an *index number*, starting at 0 for the left most column, and counting up as we go to the columns on the right.

```
# look at the mpg or miles per gallon column
```

```
mpg = cars.mpg
display(mpg)
```

Output?

```
0      18.0
1      15.0
2      18.0
3      16.0
4      17.0
...
387    27.0
388    44.0
389    32.0
390    28.0
391    31.0
Name: mpg, Length: 392, dtype: float64
```

```
# Can you predict what column will be printed?
```

```
column_index3 = cars.iloc[:,3]
display(column_index3)
```

Output?

```
0      130.0
```

```
1      165.0
2      150.0
3      150.0
4      140.0
...
387     86.0
388     52.0
389     84.0
390     79.0
391     82.0
Name: horsepower, Length: 392, dtype: float64
```

```
# Can you predict what data will be printed?
```

```
last_mpg = cars['mpg'].tail(6)
display(last_mpg)
```

Output?

```
386     27.0
387     27.0
388     44.0
389     32.0
390     28.0
391     31.0
Name: mpg, dtype: float64
```

The code in the last example above shows a programming technique called *function chaining*. There are two distinct steps:

1. The output of `cars['mpg']` is the mpg column of data.
2. This column then uses `tail` to get the last 6 mpg values.

The two steps are linked or chained together in one line of code.

To get a specific data value within a column:

```
DataFrame_name.iloc[ row_index, column_index ]
```

Each row in a DataFrame is also numbered with an index number, starting at 0 for the first row and counting up by one as we go down the rows.

```
third_mpg = cars.iloc[2, 0] # note that the first row has index 0, so the third row has i
                           # and the first column has index 0
print("Third mpg:", third_mpg)
```

Output?

```
Third mpg: 18.0
```

Arithmetic in a DataFrame

Since each column of a DataFrame is an array, we can use arithmetic operations on a column and store the result as a new column in the DataFrame.

Suppose we want to add a new column that shows the weight of the cars in 1000 lb units. For example, a car that weighs 2,790 lbs will have the weight displayed as 2.790.

```
cars['weight_in_tons'] = cars.weight / 1000
display(cars.head(10))
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu	3.504
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320	3.693
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite	3.436
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst	3.433
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino	3.449
5	15.0	8	429.0	198.0	4341.0	10.0	70	1	ford galaxie 500	4.341
6	14.0	8	454.0	220.0	4354.0	9.0	70	1	chevrolet impala	4.354
7	14.0	8	440.0	215.0	4312.0	8.5	70	1	plymouth fury iii	4.312
8	14.0	8	455.0	225.0	4425.0	10.0	70	1	pontiac catalina	4.425
9	15.0	8	390.0	190.0	3850.0	8.5	70	1	amc ambassador dpl	3.850

Sort Data

The DataFrame has a convenient method to sort data in *ascending* (lowest to highest) order or *descending* (highest to lowest) order.

To sort data we must select a column that will be used to determine the sorting order. During the sort, the DataFrame will automatically move an entire row based on the sorting order of the selected column. This way all data in one row always stay together.

```
new_DataFrame = existing_DataFrame.sort_values( by=column_name )
```

The default order is *ascending* order.

To sort in *descending* order, we need to request it:

```
new_DataFrame = existing_DataFrame.sort_values( by=column_name, ascending=False )
```

```
# create a new DataFrame with the first 5 rows of the cars DataFrame
```

```
first_five = cars.head()
display(first_five)
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu	3.504

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320	3.693
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite	3.436
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst	3.433
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino	3.449

```
# display the first_five DataFrame sorted by car name
```

```
display(first_five.sort_values(by='car_name'))
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst	3.433
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320	3.693
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu	3.504
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino	3.449
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite	3.436

```
# Can you write code to do the following?
```

```
# Display the first_five DataFrame sorted by weight_in_tons, in descending order
```

Output?

Selecting Columns

Often in data analysis we need to work with only a section of a DataFrame and not the entire DataFrame. The DataFrame has several useful methods that we can use to select certain columns and rows that we want.

- Create a new DataFrame with selected columns:

```
new_DataFrame = existing_DataFrame[ list of column names ]
```

```
# Recap of cars DataFrame by showing first 5 rows
display(cars.head())
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu	3.504

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320	3.693
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite	3.436
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst	3.433
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino	3.449

```
# Create a DataFrame with only car names and mpg
```

```
car_mpg = cars[['car_name', 'mpg']]
display(car_mpg.head())
print("Type of car_mpg:", type(car_mpg))
```

Output?

	car_name	mpg
0	chevrolet chevelle malibu	18.0
1	buick skylark 320	15.0
2	plymouth satellite	18.0
3	amc rebel sst	16.0
4	ford torino	17.0

Type of car_mpg: <class 'pandas.core.frame.DataFrame'>

- Create a new array with one selected column:

```
new_array = existing_DataFrame[ column_name ].to_numpy()
```

```
weight_array = cars['weight_in_tons'].to_numpy()
print("Weight array:", weight_array)
print("Type of array:", type(weight_array))
```

Output?

```
Weight array: [3.504 3.693 3.436 3.433 3.449 4.341 4.354 4.312 4.425 3.85 3.563 3.609
3.761 3.086 2.372 2.833 2.774 2.587 2.13 1.835 2.672 2.43 2.375 2.234
2.648 4.615 4.376 4.382 4.732 2.13 2.264 2.228 2.634 3.439 3.329 3.302
3.288 4.209 4.464 4.154 4.096 4.955 4.746 5.14 2.962 2.408 3.282 3.139
2.22 2.123 2.074 2.065 1.773 1.613 1.834 1.955 2.278 2.126 2.254 2.408
2.226 4.274 4.385 4.135 4.129 3.672 4.633 4.502 4.456 4.422 2.33 3.892
4.098 4.294 4.077 2.933 2.511 2.979 2.189 2.395 2.288 2.506 2.164 2.1
4.1 3.672 3.988 4.042 3.777 4.952 4.464 4.363 4.237 4.735 4.951 3.821
3.121 3.278 2.945 3.021 2.904 1.95 4.997 4.906 4.654 4.499 2.789 2.279
2.401 2.379 2.124 2.31 2.472 2.265 4.082 4.278 1.867 2.158 2.582 2.868
3.399 2.66 2.807 3.664 3.102 2.901 3.336 1.95 2.451 1.836 2.542 3.781
3.632 3.613 4.141 4.699 4.457 4.638 4.257 2.219 1.963 2.3 1.649 2.003
2.125 2.108 2.246 2.489 2.391 2. 3.264 3.459 3.432 3.158 4.668 4.44
4.498 4.657 3.907 3.897 3.73 3.785 3.039 3.221 3.169 2.171 2.639 2.914
2.592 2.702 2.223 2.545 2.984 1.937 3.211 2.694 2.957 2.945 2.671 1.795
2.464 2.22 2.572 2.255 2.202 4.215 4.19 3.962 4.215 3.233 3.353 3.012
3.085 2.035 2.164 1.937 1.795 3.651 3.574 3.645 3.193 1.825 1.99 2.155
2.565 3.15 3.94 3.27 2.93 3.82 4.38 4.055 3.87 3.755 2.045 2.155
1.825 2.3 1.945 3.88 4.06 4.14 4.295 3.52 3.425 3.63 3.525 4.22
4.165 4.325 4.335 1.94 2.74 2.265 2.755 2.051 2.075 1.985 2.19 2.815]
```

```
2.6 2.72 1.985 1.8 1.985 2.07 1.8 3.365 3.735 3.57 3.535 3.155
2.965 2.72 3.43 3.21 3.38 3.07 3.62 3.41 3.425 3.445 3.205 4.08
2.155 2.56 2.3 2.23 2.515 2.745 2.855 2.405 2.83 3.14 2.795 3.41
1.99 2.135 3.245 2.99 2.89 3.265 3.36 3.84 3.725 3.955 3.83 4.36
4.054 3.605 3.94 1.925 1.975 1.915 2.67 3.53 3.9 3.19 3.42 2.2
2.15 2.02 2.13 2.67 2.595 2.7 2.556 2.144 1.968 2.12 2.019 2.678
2.87 3.003 3.381 2.188 2.711 2.542 2.434 2.265 2.11 2.8 2.11 2.085
2.335 2.95 3.25 1.85 2.145 1.845 2.91 2.42 2.5 2.29 2.49 2.635
2.62 2.725 2.385 1.755 1.875 1.76 2.065 1.975 2.05 1.985 2.215 2.045
2.38 2.19 2.21 2.35 2.615 2.635 3.23 3.16 2.9 2.93 3.415 3.725
3.06 3.465 2.605 2.64 2.395 2.575 2.525 2.735 2.865 1.98 2.025 1.97
2.125 2.125 2.16 2.205 2.245 1.965 1.965 1.995 2.945 3.015 2.585 2.835
2.665 2.37 2.95 2.79 2.13 2.295 2.625 2.72 ]
Type of array: <class 'numpy.ndarray'>
```

- Create a new DataFrame by dropping or removing columns:
`new_DataFrame = existing_DataFrame.drop(columns=[ column_names ])`

```
# first we check the columns of the cars DataFrame
cars.columns
```

Output?

```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
      'acceleration', 'model_year', 'origin', 'car_name', 'weight_in_tons'],
      dtype='object')
```

```
# suppose we only need the mpg, cylinders, horsepower, and car_name columns
```

```
mpg_cyl_hp= cars.drop(columns=['displacement', 'weight', 'acceleration', 'model_year', 'or
display(mpg_cyl_hp.head())
print("Type of mpg_cyl_hp:", type(mpg_cyl_hp))
```

Output?

	mpg	cylinders	horsepower	car_name
0	18.0	8	130.0	chevrolet chevelle malibu
1	15.0	8	165.0	buick skylark 320
2	18.0	8	150.0	plymouth satellite
3	16.0	8	150.0	amc rebel sst
4	17.0	8	140.0	ford torino

```
Type of mpg_cyl_hp: <class 'pandas.core.frame.DataFrame'>
```

Selecting Rows

Sometimes we only want certain rows of a DataFrame. We can use the row index to select one or more rows to create a new array or DataFrame.

- Select one row: resulting in a new array
`new_array = existing_DataFrame.iloc[ row_index ].to_numpy()`
- Selecting a range of rows: resulting in a new DataFrame
`new_DataFrame = existing_DataFrame.iloc[ np.arange( start_index, end_index ) ]`
- Selecting specific rows: resulting in a new DataFrame
`new_DataFrame = existing_DataFrame.iloc( list of index values )`

```
first_car = cars.iloc[0].to_numpy()
print(first_car)
```

```
print(type(first_car))
```

Output?

```
[np.float64(18.0) np.int64(8) np.float64(307.0) np.float64(130.0)
 np.float64(3504.0) np.float64(12.0) np.int64(70) np.int64(1)
 'chevrolet chevelle malibu' np.float64(3.504)]
<class 'numpy.ndarray'>
```

```
first_three = cars.iloc[np.arange(0,3)]
display(first_three)
print(type(first_three))
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu	3.504
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320	3.693
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite	3.436

```
<class 'pandas.core.frame.DataFrame'>
```

```
row5_row10 = cars.iloc[[4,9]]
display(row5_row10)
print(type(row5_row10))
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino	3.449
9	15.0	8	390.0	190.0	3850.0	8.5	70	1	amc ambassador dpl	3.850

```
<class 'pandas.core.frame.DataFrame'>
```

Conditional Selection

Often we want to select the rows of the DataFrame based on a certain condition, such as all cars with 4 cylinders or all cars weighing more than 3 tons.

• Select based on one condition

We can use relational operators to compare data in a specific column and select only the True output.

Recall that relational operators are <, >=, ==, etc.

```
DataFrame_name[ DataFrame_name.column_name <relational_operator> value ]
```

```
# All cars with 4 cylinders
```

```
display(cars[cars.cylinders == 4])
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
14	24.0	4	113.0	95.0	2372.0	15.0	70	3	toyota corona mark ii	2.372
18	27.0	4	97.0	88.0	2130.0	14.5	70	3	datsum pl510	2.130
19	26.0	4	97.0	46.0	1835.0	20.5	70	2	volkswagen 1131 deluxe sedan	1.835
20	25.0	4	110.0	87.0	2672.0	17.5	70	2	peugeot 504	2.672
21	24.0	4	107.0	90.0	2430.0	14.5	70	2	audi 100 ls	2.430
...	...	...	...	...	...	...	...	...	...	...
387	27.0	4	140.0	86.0	2790.0	15.6	82	1	ford mustang gl	2.790
388	44.0	4	97.0	52.0	2130.0	24.6	82	2	vw pickup	2.130
389	32.0	4	135.0	84.0	2295.0	11.6	82	1	dodge rampage	2.295
390	28.0	4	120.0	79.0	2625.0	18.6	82	1	ford ranger	2.625
391	31.0	4	119.0	82.0	2720.0	19.4	82	1	chevy s-10	2.720

199 rows x 10 columns

All cars that weigh more than 3 tons

display(cars[cars.weight_in_tons > 3])

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu	3.504
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320	3.693
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite	3.436
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst	3.433
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino	3.449
...	...	...	...	...	...	...	...	...	...	...
358	22.4	6	231.0	110.0	3415.0	15.8	81	1	buick century	3.415
359	26.6	8	350.0	105.0	3725.0	19.0	81	1	oldsmobile cutlass ls	3.725

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_ton
360	20.2	6	200.0	88.0	3060.0	17.1	81	1	ford granada gl	3.060
361	17.6	6	225.0	85.0	3465.0	16.6	81	1	chrysler lebaron salon	3.465
381	38.0	6	262.0	85.0	3015.0	17.0	82	1	oldsmobile cutlass ciera (diesel)	3.015

167 rows × 10 columns

Can you write a line of code to print all data for the car named ford ranger?

Output?

Select based on more than one condition

When we select data based on two or more conditions, each condition must be inside ().

We can require:

- All the conditions to be True:
We use the operator & between the conditions.
- Only one of the conditions needs to be True:
We use the operator | between the conditions

All cars that weigh more than 3 tons and have 4 cylinders

```
display(cars[(cars.weight_in_tons > 3) & (cars.cylinders == 4)])
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_ton
205	20.0	4	130.0	102.0	3150.0	15.7	76	2	volvo 245	3.150
207	19.0	4	120.0	88.0	3270.0	21.9	76	2	peugeot 504	3.270
297	27.2	4	141.0	71.0	3190.0	24.8	79	2	peugeot 504	3.190
313	24.3	4	151.0	90.0	3003.0	20.1	80	1	amc concord	3.003
326	30.0	4	146.0	67.0	3250.0	21.8	80	2	mercedes-benz 240d	3.250
354	28.1	4	141.0	80.0	3230.0	20.4	81	2	peugeot 505s turbo diesel	3.230

All cars that are model year 70 or 80

```
display(cars[(cars.model_year == 70) | (cars.model_year == 80)])
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu	3.504
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320	3.693
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite	3.436
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst	3.433
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino	3.449
5	15.0	8	429.0	198.0	4341.0	10.0	70	1	ford galaxie 500	4.341
6	14.0	8	454.0	220.0	4354.0	9.0	70	1	chevrolet impala	4.354
7	14.0	8	440.0	215.0	4312.0	8.5	70	1	plymouth fury iii	4.312
8	14.0	8	455.0	225.0	4425.0	10.0	70	1	pontiac catalina	4.425
9	15.0	8	390.0	190.0	3850.0	8.5	70	1	amc ambassador dpl	3.850
10	15.0	8	383.0	170.0	3563.0	10.0	70	1	dodge challenger se	3.563
11	14.0	8	340.0	160.0	3609.0	8.0	70	1	plymouth 'cuda 340	3.609
12	15.0	8	400.0	150.0	3761.0	9.5	70	1	chevrolet monte carlo	3.761
13	14.0	8	455.0	225.0	3086.0	10.0	70	1	buick estate wagon (sw)	3.086
14	24.0	4	113.0	95.0	2372.0	15.0	70	3	toyota corona mark ii	2.372
15	22.0	6	198.0	95.0	2833.0	15.5	70	1	plymouth duster	2.833
16	18.0	6	199.0	97.0	2774.0	15.5	70	1	amc hornet	2.774
17	21.0	6	200.0	85.0	2587.0	16.0	70	1	ford maverick	2.587
18	27.0	4	97.0	88.0	2130.0	14.5	70	3	datsum pl510	2.130
19	26.0	4	97.0	46.0	1835.0	20.5	70	2	volkswagen 1131 deluxe sedan	1.835
20	25.0	4	110.0	87.0	2672.0	17.5	70	2	peugeot 504	2.672
21	24.0	4	107.0	90.0	2430.0	14.5	70	2	audi 100 ls	2.430

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
22	25.0	4	104.0	95.0	2375.0	17.5	70	2	saab 99e	2.375
23	26.0	4	121.0	113.0	2234.0	12.5	70	2	bmw 2002	2.234
24	21.0	6	199.0	90.0	2648.0	15.0	70	1	amc gremlin	2.648
25	10.0	8	360.0	215.0	4615.0	14.0	70	1	ford f250	4.615
26	10.0	8	307.0	200.0	4376.0	15.0	70	1	chevy c20	4.376
27	11.0	8	318.0	210.0	4382.0	13.5	70	1	dodge d200	4.382
28	9.0	8	304.0	193.0	4732.0	18.5	70	1	hi 1200d	4.732
307	41.5	4	98.0	76.0	2144.0	14.7	80	2	vw rabbit	2.144
308	38.1	4	89.0	60.0	1968.0	18.8	80	3	toyota corolla tercel	1.968
309	32.1	4	98.0	70.0	2120.0	15.5	80	1	chevrolet chevette	2.120
310	37.2	4	86.0	65.0	2019.0	16.4	80	3	datsum 310	2.019
311	28.0	4	151.0	90.0	2678.0	16.5	80	1	chevrolet citation	2.678
312	26.4	4	140.0	88.0	2870.0	18.1	80	1	ford fairmont	2.870
313	24.3	4	151.0	90.0	3003.0	20.1	80	1	amc concord	3.003
314	19.1	6	225.0	90.0	3381.0	18.7	80	1	dodge aspen	3.381
315	34.3	4	97.0	78.0	2188.0	15.8	80	2	audi 4000	2.188
316	29.8	4	134.0	90.0	2711.0	15.5	80	3	toyota corona liftback	2.711
317	31.3	4	120.0	75.0	2542.0	17.5	80	3	mazda 626	2.542
318	37.0	4	119.0	92.0	2434.0	15.0	80	3	datsum 510 hatchback	2.434
319	32.2	4	108.0	75.0	2265.0	15.2	80	3	toyota corolla	2.265
320	46.6	4	86.0	65.0	2110.0	17.9	80	3	mazda glc	2.110
321	27.9	4	156.0	105.0	2800.0	14.4	80	1	dodge colt	2.800
322	40.8	4	85.0	65.0	2110.0	19.2	80	3	datsum 210	2.110
323	44.3	4	90.0	48.0	2085.0	21.7	80	2	vw rabbit c (diesel)	2.085
324	43.4	4	90.0	48.0	2335.0	23.7	80	2	vw dasher (diesel)	2.335
325	36.4	5	121.0	67.0	2950.0	19.9	80	2	audi 5000s (diesel)	2.950

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
326	30.0	4	146.0	67.0	3250.0	21.8	80	2	mercedes-benz 240d	3.250
327	44.6	4	91.0	67.0	1850.0	13.8	80	3	honda civic 1500 gl	1.850
328	33.8	4	97.0	67.0	2145.0	18.0	80	3	subaru dl	2.145
329	29.8	4	89.0	62.0	1845.0	15.3	80	2	volkswagen rabbit	1.845
330	32.7	6	168.0	132.0	2910.0	11.4	80	3	datsum 280-zx	2.910
331	23.7	3	70.0	100.0	2420.0	12.5	80	3	mazda rx-7 gs	2.420
332	35.0	4	122.0	88.0	2500.0	15.1	80	2	triumph tr7 coupe	2.500
333	32.4	4	107.0	72.0	2290.0	17.0	80	3	honda accord	2.290

• Select based on a partial match of a string

If the condition is based on data which is a string, we can also look for a partial match of a given text string.

```
DataFrame_name[ DataFrame_name.column_name.str.contains( "given text string", case=False ) ]
```

```
# All cars that are from mazda
```

```
display(cars[cars.car_name.str.contains("Mazda", case=False)])
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
70	19.0	3	70.0	97.0	2330.0	13.5	72	3	mazda rx2 coupe	2.330
241	21.5	3	80.0	110.0	2720.0	13.5	77	3	mazda rx-4	2.720
244	32.8	4	78.0	52.0	1985.0	19.4	78	3	mazda glc deluxe	1.985
317	31.3	4	120.0	75.0	2542.0	17.5	80	3	mazda 626	2.542
320	46.6	4	86.0	65.0	2110.0	17.9	80	3	mazda glc	2.110
331	23.7	3	70.0	100.0	2420.0	12.5	80	3	mazda rx-7 gs	2.420
345	34.1	4	91.0	68.0	1985.0	16.0	81	3	mazda glc 4	1.985
353	31.6	4	120.0	74.0	2635.0	18.3	81	3	mazda 626	2.635

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name	weight_in_tons
370	37.0	4	91.0	68.0	2025.0	18.2	82	3	mazda glc custom l	2.025
371	31.0	4	91.0	68.0	1970.0	17.6	82	3	mazda glc custom	1.970

The `case=False` input means that the match can be with either uppercase or lowercase letters. In the example above, the input string that we give to the selection is "Mazda" but all the matches are "mazda". We leave out the `case=False` if we want an exact match of uppercase or lowercase of the given text string.

Summary

In this chapter we learn two commonly used data types in data science: the array for a sequence of data, and the DataFrame for multiple data records, where each record has many data fields.

These data types, together with numbers and strings, provide the tools to analyze and perform calculations with large datasets.

5: Plotting

In the previous chapter we saw that an array and a DataFrame are convenient ways to store a dataset in an organized way that makes it easy to select columns or rows of data and analyze them. However, it's not easy to make observations about the data by looking at multiple rows and columns of many numbers. In this chapter we learn to show the data visually so that it's easy to spot the patterns and trends in the data.

This chapter covers four different types of plots that will display the data visually: scatterplots, line plots, bar charts, and histograms. We will use the same dataset about cars that we explored in the last chapters.

Gather Data

First we read the same cars dataset into a DataFrame and display the first five rows.

```
# import pandas to use the DataFrame
import pandas as pd

# 1. Store the URL of the movie data webpage
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_"

# 2. Read the CSV file into a DataFrame
cars = pd.read_csv("cars.csv")

# 3. Show the number of rows and columns
rows, columns = cars.shape
print("Number of rows:", rows)
print("Number of columns:", columns)

# 4. Print the first 10 lines of the DataFrame
print("First 10 rows:")
cars.head(10)
```

Output?

```
Number of rows: 392
Number of columns: 9
First 10 rows:
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino
5	15.0	8	429.0	198.0	4341.0	10.0	70	1	ford galaxie 500
6	14.0	8	454.0	220.0	4354.0	9.0	70	1	chevrolet impala
7	14.0	8	440.0	215.0	4312.0	8.5	70	1	plymouth fury iii
8	14.0	8	455.0	225.0	4425.0	10.0	70	1	pontiac catalina
9	15.0	8	390.0	190.0	3850.0	8.5	70	1	amc ambassador dpl

```
cars.describe()
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin
count	392.000000	392.000000	392.000000	392.000000	392.000000	392.000000	392.000000	392.000000
mean	23.445918	5.471939	194.411990	104.469388	2977.584184	15.541327	75.979592	1.576531
std	7.805007	1.705783	104.644004	38.491160	849.402560	2.758864	3.683737	0.805518
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000	1.000000
25%	17.000000	4.000000	105.000000	75.000000	2225.250000	13.775000	73.000000	1.000000
50%	22.750000	4.000000	151.000000	93.500000	2803.500000	15.500000	76.000000	1.000000
75%	29.000000	8.000000	275.750000	126.000000	3614.750000	17.025000	79.000000	2.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000	3.000000

```
import matplotlib.pyplot as plt # shorten the module name to plt
```

Output?

We now use different plots to *visualize* the data.

To plot data, we use another Python module called `matplotlib.pyplot`. This module has all the plotting functions that will plot data in specific ways.

Scatterplots

Use Case for a Scatterplot

A scatterplot shows the relationship between 2 *features* or 2 columns of the data. If one feature increases in value, does the other feature tend to increase value as well? Or does it decrease instead? Or is there simply no pattern between the two features?

Scatterplot Application

For the car dataset we want to see whether a car's `weight` is related to the `mpg` (or fuel efficiency). We suspect that the heavier the car is, the lower its fuel efficiency, since it takes more fuel to move a larger mass. But is it true?

To answer the question, we use a scatterplot to see whether there's a pattern or association between these two features. The `mpg` feature will be along the x-axis or horizontal axis, and the `weight` feature will be along the y-axis or the vertical axis. The choice of which feature goes on the x-axis vs the y-axis is arbitrary. We can also easily swap the `mpg` and `weight` on the x-axis and y-axis.

[[View](#) the color choices for plotting data in matplotlib]

```
# set the size of the plot
plt.figure(figsize=(4,4))

# create a scatterplot of mpg vs weight
plt.scatter(cars['mpg'], cars['weight'], color='green')
# x-axis data first, y-axis data second

# label (describe) the x-axis or horizontal axis
plt.xlabel('Miles per Gallon (mpg)')

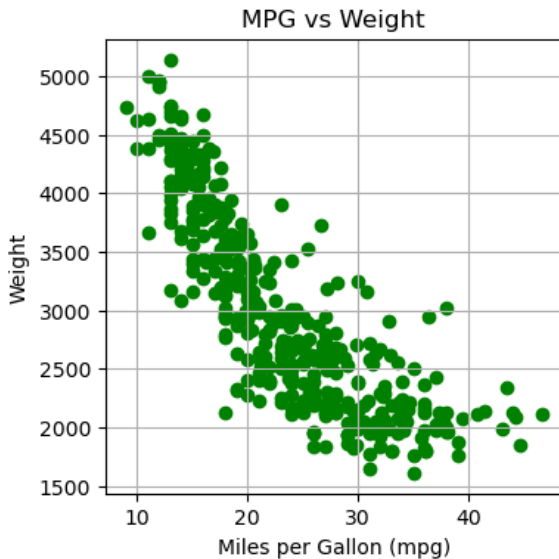
# label (describe) the y-axis or vertical axis
plt.ylabel('Weight')

# title the plot
```

```
plt.title('MPG vs Weight')

# put a grid on the plot and show the plot
plt.grid()
plt.show()
```

Output?



In the scatterplot above each green point is for one car in the dataset. See if you can locate the car with the highest **mpg** in the dataset, close to 50, which weighs a little above 2000 lbs.

The scatterplot shows a *general* pattern: the heavier the car is, the lower its fuel efficiency. This is just as we suspected, but the scatterplot makes it easy to see the relationship in the data.

When one feature generally increases or decreases with another feature, we say that there's an *association* between the two features. In this case, there is a *negative* association because as one feature increases, the other feature decreases. If both features increase together or decrease together, then it is a *positive* association.

We also note that the negative association in the plot above is a *general* trend and doesn't apply to all cars. For example, looking at the green dots above, we can find one car that weighs around 3000 lbs and is around 38 mpg, which is much higher than the 18 mpg of another car that weighs only around 2000 lbs.

Line Plots

Use Case for a Line Plot

A line plot is used to show the trend or the change in data over a range of measurements, typically a range of time. If you've ever seen a plot that shows how the price of a company stock fluctuates during a trading day, then you've seen a line plot.

Line Plot Application

For the car dataset we want to see whether the fuel efficiency of cars has increased over time. We hope that later model cars have better mileage than earlier model cars, and we'll use a line plot to see whether it's true.

For the line plot:

- The years represent our the time range that the data span. It is the *independent* variable of the line plot and is on the x-axis.

- The `mpg` changes over time. It is the *dependent* variable since it depends on the years so it is used on the y-axis.

Before we can plot the `mpg`, we need to find the average `mpg` for each year. This is because in each year there can be cars with high fuel efficiency as well as low fuel efficient cars, and we want the average fuel efficiency for that year.

Fortunately the DataFrame has a method that makes it easy to calculate the average `mpg` per year: the `groupby` method:

```
yearly_mpg = cars.groupby('model_year')['mpg'].mean().reset_index()
```

Output?

In the calculation above, we ask the DataFrame to group the cars by year or `model_year`. For each year group, we choose the `mpg` column to calculate the `mean` or the average, resulting in one average `mpg` per year. Since we end up with a small number of average `mpg` compared to the individual `mpg` of all the cars, we use the `reset_index` function to reset the index to the count of years, starting at 0.

```
display(yearly_mpg)
```

Output?

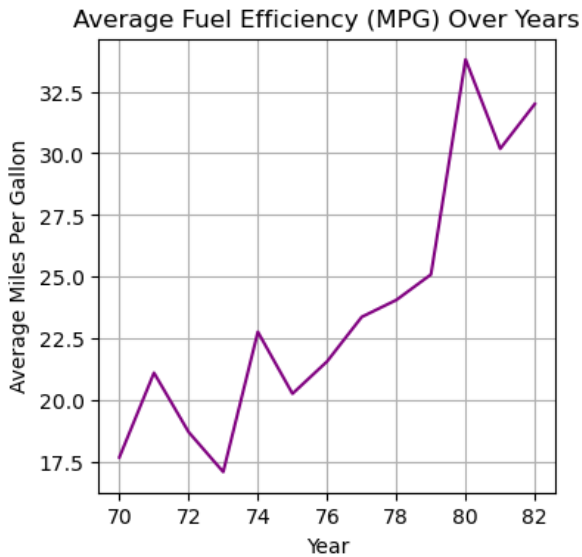
	model_year	mpg
0	70	17.689655
1	71	21.111111
2	72	18.714286
3	73	17.100000
4	74	22.769231
5	75	20.266667
6	76	21.573529
7	77	23.375000
8	78	24.061111
9	79	25.093103
10	80	33.803704
11	81	30.185714
12	82	32.000000

```
# line plot of the average mpg over the years
```

```
plt.figure(figsize=(4,4))
plt.plot(yearly_mpg['model_year'], yearly_mpg['mpg'], color='purple')
# x-axis data are the years, y-axis data are the average mpg

plt.title('Average Fuel Efficiency (MPG) Over Years')
plt.xlabel('Year')
plt.ylabel('Average Miles Per Gallon')
plt.grid()
plt.show()
```

Output?



The line plot shows that there was a big increase in the **mpg** between 1976 and 1980, and the overall trend is that fuel efficiency did indeed increase over time.

Bar Charts

Quantitative Data vs Categorical Data

Sometimes we work with data that are in a continuous range, such as the **mpg** of the cars. Data that are in a continuous range are *quantitative* data.

At other times we may work with data that are in categories instead of in a range. An example of *categorical* data would be whether a person is a full-time student, part-time student, or not a student. In the car dataset, one categorical feature is the number of cylinders.

```
print("Example of quantitative data: mpg")
print("Numbers of unique mpg values:", cars['mpg'].unique())
print()
print("Example of categorical data: cylinders")
print("Numbers of unique cylinders values:", cars['cylinders'].unique())
```

Output?

```
Example of quantitative data: mpg
Numbers of unique mpg values: [18. 15. 16. 17. 14. 24. 22. 21. 27. 26. 25. 10. 11.
 28. 19. 12. 13. 23. 30. 31. 35. 20. 29. 32. 33. 17.5 15.5
 14.5 22.5 24.5 18.5 29.5 26.5 16.5 31.5 36. 25.5 33.5 20.5 30.5 21.5
 43.1 36.1 32.8 39.4 19.9 19.4 20.2 19.2 25.1 20.6 20.8 18.6 18.1 17.7
 27.5 27.2 30.9 21.1 23.2 23.8 23.9 20.3 21.6 16.2 19.8 22.3 17.6 18.2
 16.9 31.9 34.1 35.7 27.4 25.4 34.2 34.5 31.8 37.3 28.4 28.8 26.8 41.5
 38.1 32.1 37.2 26.4 24.3 19.1 34.3 29.8 31.3 37. 32.2 46.6 27.9 40.8
 44.3 43.4 36.4 44.6 33.8 32.7 23.7 32.4 26.6 25.8 23.5 39.1 39. 35.1
 32.3 37.7 34.7 34.4 29.9 33.7 32.9 31.6 28.1 30.7 24.2 22.4 34. 38.
 44. ]
```

```
Example of categorical data: cylinders
Numbers of unique cylinders values: [8 4 6 3 5]
```

The quantitative **mpg** data are within a range from 9 to 44.
The categorical **cylinders** data are in five distinct values or categories.

Use Case for a Bar Chart

When we work with categorical data, a bar chart is used to show the count of data for each category. Each category is represented by a bar, and the length of the bar corresponds to the count of data in that category.

Bar Chart Application

For the car dataset we want to see how many cars there are for each number of cylinder category.

First we need to group the data by number of cylinders and count the number of cars in each group.

```
car_count = cars.groupby('cylinders').cylinders.count()
car_count
```

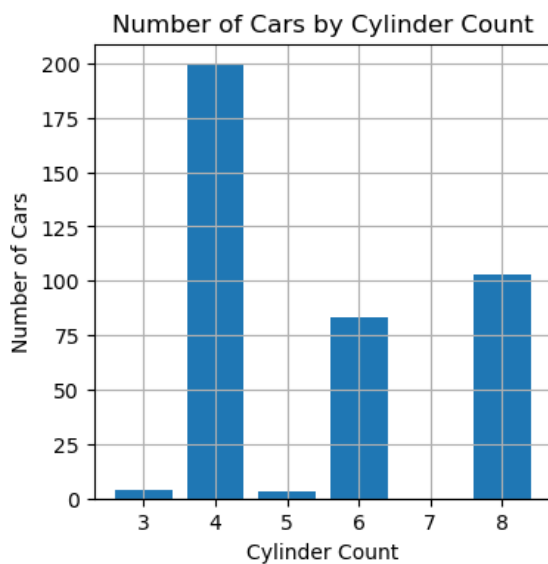
Output?

```
cylinders
3      4
4    199
5      3
6     83
8    103
Name: cylinders, dtype: int64
```

Now we use a bar chart to visually compare the count of each group.

```
plt.figure(figsize=(4,4))
plt.bar(car_count.index, car_count.values)
plt.title('Number of Cars by Cylinder Count')
plt.xlabel('Cylinder Count')
plt.ylabel('Number of Cars')
plt.grid()
plt.show()
```

Output?



From the output above, we can easily see that the dataset has the highest number of 4-cylinder cars, followed by 8-cylinder cars. There are very few 3-cylinder and 5-cylinder cars.

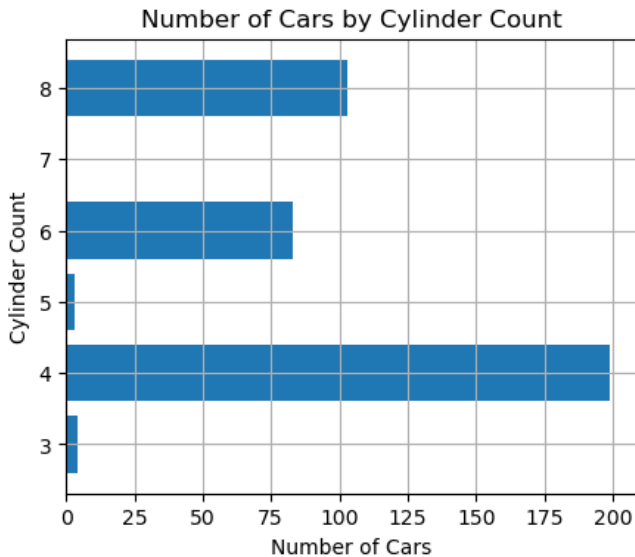
It is also possible to plot a horizontal bar chart instead of a vertical bar chart as above. We use most of the same code for plotting

except we use `barh()` for horizontal bar chart, and we swap the x-axis and y-axis labels.

```
plt.figure(figsize=(5,4))

# barh for horizontal bar chart
plt.barh(car_count.index, car_count.values)
plt.title('Number of Cars by Cylinder Count')
plt.xlabel('Number of Cars')
plt.ylabel('Cylinder Count')
plt.grid()
plt.show()
```

Output?



Generally, when the x-ticks are long strings, the bar chart is easier to read if it's a horizontal bar chart. In the first vertical bar chart, the cylinder counts along the x-axis are small numbers so they don't overlap each other in the plot, so both a vertical and horizontal bar chart can be used.

Histograms

In the previous sections we learned how to visualize the count of categorical data, an important fact about the data. From the bar chart output we can immediately see which categories have the most data points or the fewest data points, and this can help us draw conclusions on the data.

For quantitative data, which are data points that belong on a continuous range of values instead of in separate categories, it can also be helpful to know the count of data. However, finding the count of quantitative data is slightly different than finding the count of categorical data:

- For categorical data, a data point can be in one of several categories (such as full-time student, part-time student, or not a student), so it's easy to find the count of the three categories in the dataset. From the output of the bar chart, we can easily see what type of students is most common in the dataset.
- For quantitative data, most data points will have different values. For example, let's say the cost of a new car in the US is in a range from 15,000 to 125,000 dollars. If we have a dataset of 1000 new cars, most of the "out the door" price (including taxes, fees, licensing, add-ons, etc.) for these cars will be slightly different. If we used a bar chart to display the count of each "category" or each price, we will have many categories, each with a count of one or at most two or three, and this doesn't give us much information about which price ranges are the most common in the dataset.

Binning Data

To solve the problem of how to find the count of quantitative data, we use a statistical method called binning the data. When we *bin* the data, we take the entire range of data and divide it into equal intervals, each called a bin. Each bin is a small part of the entire range of data. Then we put each data point into a bin whose range includes the data point.

For the new car price example above, we can divide the range of \$15,000 to \$125,000 into bins of \$10,000 each:

- first bin: \$15,000 to \$24,999
- second bin: \$25,000 to \$34,999
- third bin: \$35,000 to \$44,999
- ...
- last bin: \$115,000 to \$125,000 (note the different end for the range here)

If a car has a price of \$27,895, then it goes in the second bin.

After all the data points have been put into bins, we can plot the bins as if they're "categories", except that the bins form a continuous range. The resulting plot is called a *histogram*.

A histogram is similar to a bar chart, except that the bars represent ranges of numerical values rather than separate categories.

Use Case for a Histogram

A histogram shows the shape of the data sequence. With a histogram we can see the minimum and maximum values, what range of values is the most common in the dataset, whether the data range is skewed to the right or left, where the average is, whether there is a gap, and whether there are *outliers*, which are extreme values compared to the rest of the data.

Histogram Application

Using the same car dataset, we display the first 10 rows of the cars DataFrame:

```
cars.head(10)
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino
5	15.0	8	429.0	198.0	4341.0	10.0	70	1	ford galaxie 500
6	14.0	8	454.0	220.0	4354.0	9.0	70	1	chevrolet impala
7	14.0	8	440.0	215.0	4312.0	8.5	70	1	plymouth fury iii
8	14.0	8	455.0	225.0	4425.0	10.0	70	1	pontiac catalina
9	15.0	8	390.0	190.0	3850.0	8.5	70	1	amc ambassador dpl

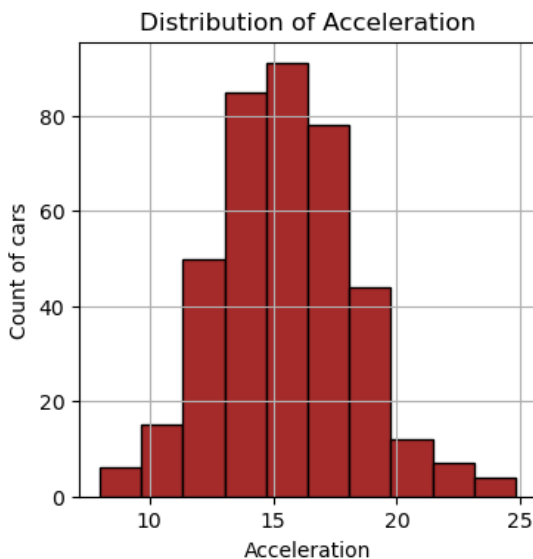
We now look at the `acceleration` column, which is the number of seconds it takes to accelerate from 0 to 60 mph. We plot the `acceleration` values in a histogram.

```
plt.figure(figsize=(4,4))
```

```
# create a histogram, the default number of bins is 10:
plt.hist(cars['acceleration'], color='brown', edgecolor='black')

# add title, labels, grid, and show the plot
plt.title('Distribution of Acceleration')
plt.xlabel('Acceleration')
plt.ylabel('Count of cars')
plt.grid()
plt.show()
```

Output?



The histogram above has 10 bars or 10 bins that cover the continuous range of **acceleration** data along the x-axis.

We see that the fastest acceleration is just under 5 seconds, and the slowest acceleration is almost 25 seconds.

Since the range of each bin is 1/10 of the entire data range, the width of each bar is the same. Therefore the height of the bar corresponds to the count of data points in a particular bin, and we see that the tallest bar or the largest count is for an acceleration of around 15-16 seconds. This tallest bin is also around the middle of the data range, so we expect the mean or average acceleration of all cars is also between 15-16 seconds.

The histogram shows us the shape of the **acceleration** data. Looking at the three tallest bars in the middle, we can say that the majority of the cars in the dataset has an acceleration of around 13 to 18 seconds, and the faster and slower accelerations slope down symmetrically on the left and right of center. The shape of this data distribution is shaped like a bell, or is a bell curve: it is symmetric, with no skew on the left or right.

We also notice that there's no gap or no empty bin, and no outliers, which are data values that are far on the left or right on the x-axis from the rest of the data.

To check the count of data points in each bin, we call the same hist() function to get the counts:

```
plt.figure(figsize=(4,4))

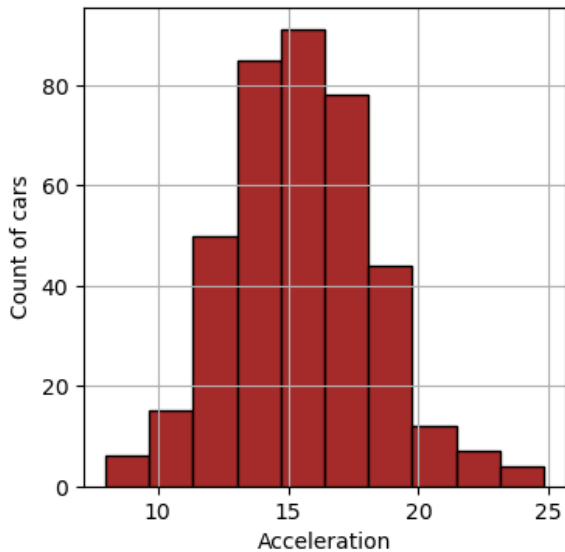
(counts, edges, patches) = plt.hist(cars['acceleration'], color='brown', edgecolor='black')
# the hist() function returns:
# the count of each bin,
# the value at the max edge of each bin,
# and the patches, the standard name from hist() for the plot figure

print("Count of each bin:", counts)
print("Max value of bin edges:", edges)
```

```
plt.xlabel('Acceleration')
plt.ylabel('Count of cars')
plt.grid()
plt.show()
```

Output?

```
Count of each bin: [ 6. 15. 50. 85. 91. 78. 44. 12.  7.  4.]
Max value of bin edges: [ 8.    9.68 11.36 13.04 14.72 16.4  18.08 19.76 21.44 23.12 24.8 ]
```



The printed counts from the output above match our expectations from viewing the plot. The middle bin has the highest count of 92 cars, and the counts of bins on both sides slope down fairly symmetrically .

We can change the bin range to smaller intervals by requesting a larger number of bins. This lets us see the histogram with more granularity:

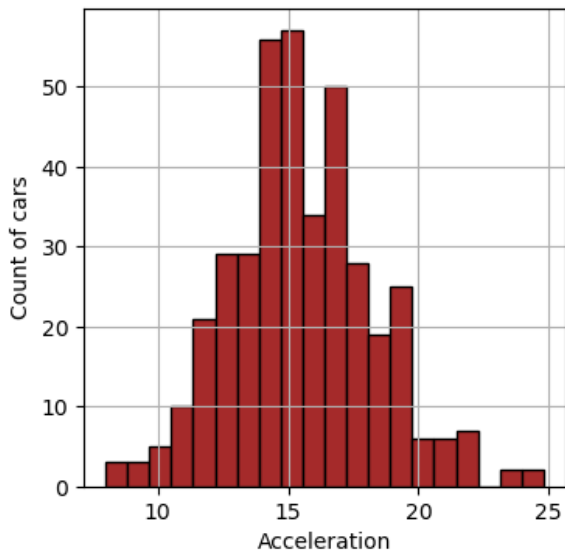
```
plt.figure(figsize=(4,4))

# asking for twice the number of bins as the default:
(counts, edges, patches) = plt.hist(cars['acceleration'], color='brown', edgecolor='black')

print("Count of each bin:", counts)
plt.xlabel('Acceleration')
plt.ylabel('Count of cars')
plt.grid()
plt.show()
```

Output?

```
Count of each bin: [ 3.  3.  5. 10. 21. 29. 29. 56. 57. 34. 50. 28. 19. 25.  6.  6.  7.  0.
 2.  2.]
```



There are now 20 bars for the 20 bins. The shape of the 20 bars are not exactly the same as with 10 bins. The bar heights are generally lower since the bar widths are half as wide. This is because there are twice as many bins so each bin has fewer data points. We also see there is a gap, the third bin from the right has no data values. But overall the *distribution* or shape of the data are similar in both histograms.

Density Scaling

Last but not least, it's important to note that in each of the two histograms above, the height of the bars corresponds to the count of data in the bins *only because the bars all have the same width*. But there are two common cases when the count of data in the bins should not directly determine the height of the bars.

Case 1

Sometimes the bars in a histogram don't have equal widths due to the nature of the data, and this can cause misleading histogram plots. As an example:

- Suppose in a histogram, the bar for bin A happens to be twice as wide as the bar for bin B, and the two bins have the same count of data.
- If we use the count to determine the height of the bars, then bin A and bin B will be the same height.
- But since the bar for bin A is twice as wide than the bar for bin B, the bar for bin A will appear twice as large as the bar for bin B, misleading the viewer into concluding that bin A has twice the count of data as bin B.

Case 2

If the count of data in the bins determines the height of the bars, then it can be difficult to compare the distribution of two datasets of different sizes. As an example:

- Suppose dataset A has twice as many data points as dataset B, and we need to compare the distribution of the two datasets.
- Since the total count of data in dataset A is larger, it means that the count in each bin will generally be larger, and the bars will be generally be higher than the bars in dataset B.
- When comparing the histograms, it would be difficult to determine whether the difference in bar heights between the 2 histograms is due to one dataset simply having more data points to go in the bins, or if it's due to the different characteristics of the datasets.

The solution for the two cases above is to:

1. Scale the count in each bin to be a ratio of the count / total data points.

This turns the count of each bin into a proportion of the dataset that falls into each bin.

It normalizes the data so we can compare the distribution of two datasets that differ greatly in size (10,000 data points vs 100 data points).

2. The *area* of the bar represents this ratio.

This means the ratio = height of the bar * width of the bar

Or: height of the bar = ratio / width of the bar.

Since the area of each bar is a ratio of count / total data points, when we add all the areas of all the bars, the total area is 1. Again, this normalizes the distribution so that different distributions can be compared together.

When the height of the bars is determined this way, it represents the *density* or concentration of data at a bin, and the histogram is called a *density histogram*.

Fortunately we don't have to do all the calculations above to plot the bar height as density instead of as counts.

The `hist()` function has a variable named `density` that determines whether the height of the bar is determined by the count in the bins or the density of the bins. The default is to use the count in the bins, but when we set `density` to `True`, then the height of the bar is determined by the density.

Below we compare the *frequency histogram* (y-axis as count) and the *density histogram* (y-axis as density).

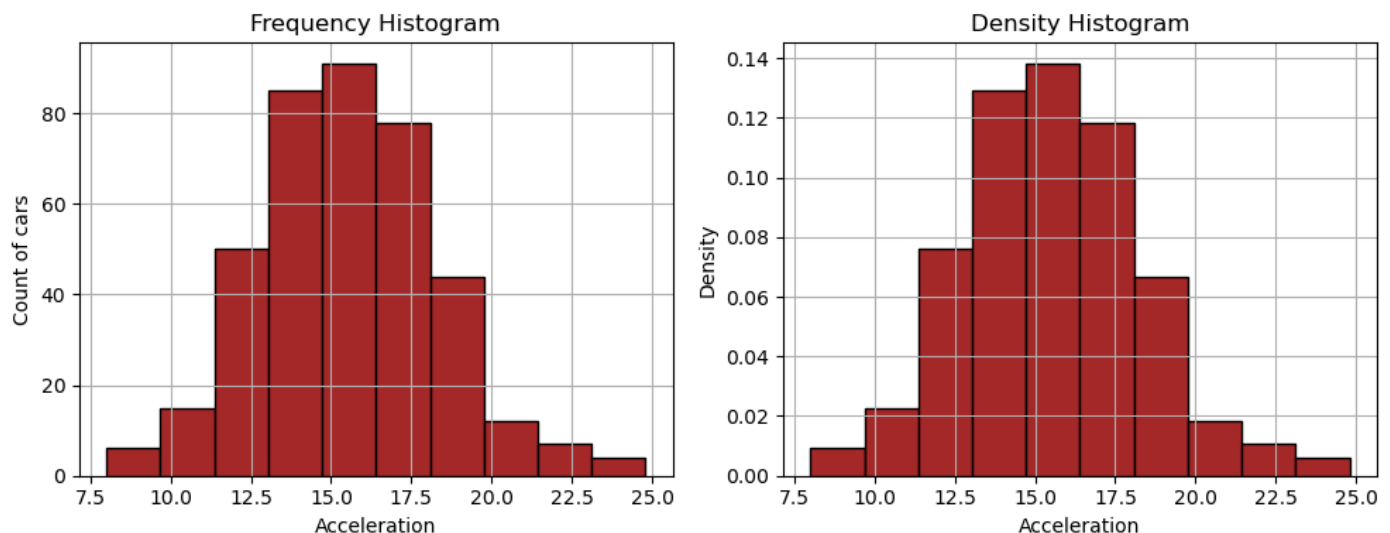
```
plt.figure(figsize=(10, 4))

# Left plot: frequency histogram
plt.subplot(1, 2, 1)
plt.hist(cars['acceleration'], color='brown', edgecolor='black')
plt.xlabel('Acceleration')
plt.ylabel('Count of cars')
plt.title('Frequency Histogram')
plt.grid()

# Right plot: density histogram
plt.subplot(1, 2, 2)
plt.hist(cars['acceleration'], color='brown', edgecolor='black', density=True)
plt.xlabel('Acceleration')
plt.ylabel('Density')
plt.title('Density Histogram')
plt.grid()

plt.tight_layout()
plt.show()
```

Output?



Note that the two distribution shapes are identical. In the frequency histogram the y-axis is the actual count of cars in a bin, while in the density histogram, the y-axis is the density or how crowded with data a bin is.

Overlaid Plots

Sometimes in data analysis it can be helpful to compare the plots of two sequences of data to visually see how the two datasets relate to each other over the same time period or under the same conditions. To facilitate this visual comparison, each of the four plotting functions we've discussed can plot more than one sequence of data in the same plot. This is to *overlay* the plots in the same figure.

To plot two or more sequences of data in the same plot, the sequences should:

- be related and have the same number of data points
- have the same scale

An example is to plot the homework scores between 2 students in a class during a semester. We work with a small dataset below:

```
scores = pd.DataFrame(data=[[20,18,18,19,20,19,18,18,19,19],
                           [18,17,16,19,18,20,19,19,19,20]],
                      columns=["HW1", "HW2", "HW3", "HW4", "HW5",
                              "HW6", "HW7", "HW8", "HW9", "HW10"],
                      index=["Student A", "Student B"])
```

scores

Output?

	HW1	HW2	HW3	HW4	HW5	HW6	HW7	HW8	HW9	HW10
Student A	20	18	18	19	20	19	18	18	19	19
Student B	18	17	16	19	18	20	19	19	19	20

We note that the sequences of scores for each student meet the conditions to be plotted together in one plot:

- The sequences of scores are related: each sequence has the scores for one student, and since the students are in the same class, the scores are for the same work.
- The sequences of scores have the same number of data points, there are 8 scores for each sequence.
- The sequences have the same scale, they are all out of 20 points.

To compare how well the 2 students did with their homework during the semester, we can use a line plot to show the trend of scores over time

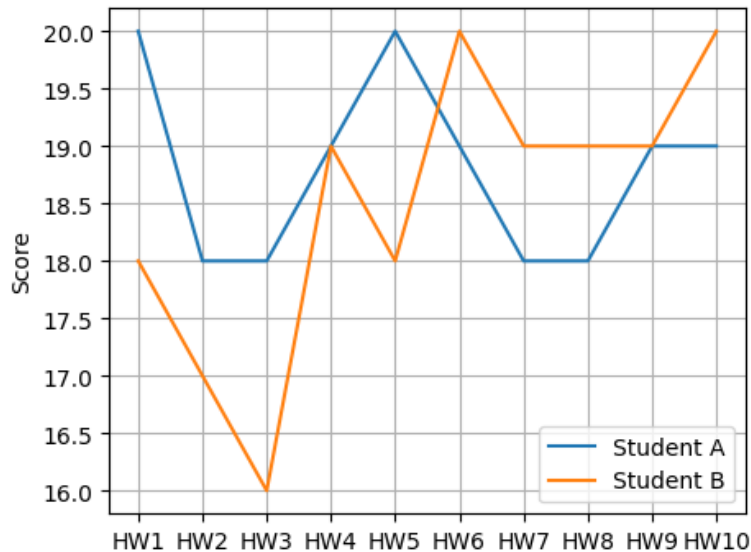
```
plt.figure(figsize=(5,4))

# plot each student score, and add a label to identify the student
plt.plot(scores.columns, scores.loc['Student A'], label='Student A')
plt.plot(scores.columns, scores.loc['Student B'], label='Student B')

plt.ylabel('Score')
plt.legend() # the legend identifies the label for each plot line

plt.grid()
plt.show()
```

Output?



The plot makes it easier to see that Student A is more consistent with the homework scores, while Student B scored lower at the beginning of the semester, but improved steadily and surpassed Student A at the end of the semester.

When overlaying two plots, matplotlib.pyplot automatically chooses different colors for the two plots. But without a legend to explain what the colors represent, it's not easy to understand the plot. In the overlaid plots above, the legend shows that Student A's data are in blue and Student B's data are in orange. This makes it easy to compare the trend in scores of the students.

In the example above we see how two line plots can be overlaid in the same plot figure to make it easier to compare them. We can also use the same steps to overlay two scatterplots (by calling `plt.scatter` twice) or two bar charts (by calling `plt.bar` twice) or two histograms (by calling `plt.hist` twice).

Summary

In this chapter we learn that data visualization or plotting helps us observe more easily trends and patterns in a dataset. By selecting the appropriate plot to display data, we can see whether two features of a dataset have an association with each other, or see the trend of a feature over time, or see the category or range that contains the most number of the data points in the dataset.

6: Table Manipulation

In this chapter we explore more advanced ways to work with data in a DataFrame.

First we learn advanced techniques to work with DataFrames: calculating descriptive statistics, separating data into groups, and shaping the DataFrame. Then we'll learn to write functions that can help us work more efficiently.

Gather Data

In this chapter we work with a dataset of food delivery time named `delivery.csv`, from [Kaggle](#). The dataset contains a company's food delivery time in different types of weather, different times of day, etc.

As usual, we read the data into a DataFrame.

```
import pandas as pd

url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_delivery.csv"
delivery = pd.read_csv(url)
print("First 5 rows:")
display(delivery.head())
print("Number of rows:", delivery.shape[0])
print("Number of columns:", delivery.shape[1])
```

Output?

First 5 rows:

	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experience
0	522	7.93	Windy	Low	Afternoon	Scooter	12	1.0
1	738	16.42	Clear	Medium	Evening	Bike	20	2.0
2	741	9.52	Foggy	Low	Night	Scooter	28	1.0
3	661	7.44	Rainy	Medium	Afternoon	Scooter	5	1.0
4	412	19.03	Clear	Low	Morning	Bike	16	5.0

```
Number of rows: 1000
Number of columns: 9
```

The dataset has 1000 delivery times (in minutes) and the conditions of the delivery: weather, traffic, time of day, vehicle.

Descriptive Statistics

Descriptive statistics, also known as *summary statistics*, describe the main characteristics of a dataset. They describe:

- The typical value of the data: *mean* or average, *median* or midpoint value
- The spread or how much variation in the data: *standard deviation* or spread of data, minimum, maximum
- The frequency and aggregate of the data: count, sum

The DataFrame has methods that make it easy to find the descriptive statistics for a feature of a dataset.

```
value = a_DataFrame[feature].statistics_method()
```

We find the descriptive statistics of the delivery time one by one:

```
# average delivery time
average_delivery_time = delivery['Delivery_Time_min'].mean()
print("Average delivery time:", round(average_delivery_time,1))

# median of delivery time
median_delivery_time = delivery['Delivery_Time_min'].median()
print("Median delivery time:", round(median_delivery_time,1))

# standard deviation of delivery time
std_delivery_time = delivery['Delivery_Time_min'].std()
print("Standard deviation of delivery time:", round(std_delivery_time,1))

# minimum delivery time
min_delivery_time = delivery['Delivery_Time_min'].min()
print("Minimum delivery time:", min_delivery_time)

# maximum delivery time
max_delivery_time = delivery['Delivery_Time_min'].max()
print("Maximum delivery time:", max_delivery_time)

# count of delivery times
count_delivery_times = delivery['Delivery_Time_min'].count()
print("Count of delivery times:", count_delivery_times)

# sum of all delivery times
sum_delivery_times = delivery['Delivery_Time_min'].sum()
print("Sum of all delivery times:", sum_delivery_times)
```

Output?

```
Average delivery time: 56.7
Median delivery time: 55.5
Standard deviation of delivery time: 22.1
Minimum delivery time: 8
Maximum delivery time: 153
Count of delivery times: 1000
Sum of all delivery times: 56732
```

We can also use the DataFrame `describe` method to find the descriptive statistics.

```
a_DataFrame.feature.describe()
```

```
delivery['Delivery_Time_min'].describe()
```

Output?

```
count    1000.000000
mean      56.732000
std       22.070915
min        8.000000
25%       41.000000
50%       55.500000
75%       71.000000
max      153.000000
Name: Delivery_Time_min, dtype: float64
```

The `describe` method gives us three additional statistics:

- The *25th percentile*: the cutoff value of the lowest 25% of the delivery times. This means 25% of the delivery times are smaller (or faster) than this value.
- The *50th percentile*: the cutoff value of 50% of the delivery times, where half the delivery times are higher than this value and half the delivery times are lower than this value.
- The *75th percentile*: the cutoff value where 75% of the delivery times are smaller than this value.

Grouping Data by a Feature

Use Case for Grouping Data

Often data scientists need to divide the dataset into groups to inspect the data in each group or to compare the groups. For example, in the delivery dataset, we can group the delivery times by weather condition to see if certain weather conditions slow down the delivery time.

To divide the data into groups based on a data feature, such as the weather, the feature must be *categorical data*.

Recall that categorical data means that the data must have distinct categories, and not be a continuous range of data. The weather is categorical data in the dataset because it can only be one of several distinct categories: windy, clear, foggy, rainy, etc.

Can you tell which features of the delivery dataset can be used with **groupby**?

Count of groups

To group the data by a feature and find the total count of each category of feature, we use the format:

```
output = a_DataFrame.groupby('feature').size()
```

The **groupby** method returns each group by **feature**, and we can use the **size** function to find the count of data in each group.

```
# group by Weather and find the count of delivery times for each weather condition
delivery.groupby('Weather').size()
```

Output?

```
Weather
Clear      470
Foggy      103
Rainy      204
Snowy       97
Windy       96
dtype: int64
```

```
# Can you write code to print the Traffic_Level and the count of each level?
```

Output?

Descriptive Statistics of Groups

To group the data by a feature and find the statistics of a certain column in each group, we use the format:

```
output = a_DataFrame.groupby('feature').a_column.statistics_method()
```

The **groupby** method returns each group by **feature**, then we select a column that we want and ask for the specific descriptive statistics.

```
delivery.groupby('Weather').Delivery_Time_min.mean()
```

Output?

```
Weather
Clear      53.082979
Foggy      59.466019
Rainy      59.794118
Snowy      67.113402
Windy      55.458333
Name: Delivery_Time_min, dtype: float64
```

We can see that the delivery time is shortest when the weather is clear, and it's the longest when the weather is snowy.

Grouping by Multiple Features

To group data by multiple features, we simply give the `groupby` method the features we want in a Python list. We do this by listing the features inside `[]`, which you may recall is the format of a list in Python.

```
# show each weather condition, and with each weather condition show all the vehicle types,
# and with each vehicle type show the average delivery time
delivery.groupby(['Weather', 'Vehicle_Type']).Delivery_Time_min.mean()
```

Output?

```
Weather  Vehicle_Type
Clear    Bike          52.154812
         Car          57.259740
         Scooter      52.435065
Foggy    Bike          60.807692
         Car          63.650000
         Scooter      54.516129
Rainy    Bike          60.602041
         Car          55.727273
         Scooter      61.403226
Snowy    Bike          65.826923
         Car          68.440000
         Scooter      68.800000
Windy    Bike          57.037736
         Car          50.150000
         Scooter      56.434783
Name: Delivery_Time_min, dtype: float64
```

With `groupby` and descriptive statistics we can get quite a bit of information from the data.

When the weather is clear, a delivery on bike or scooter tends to be faster than with a car. The deliveries must be in a big city where bikes and scooters can move faster than cars.

But when the weather is rainy, a delivery by car tends to be faster, most likely because the food just be packed more carefully if a bike or scooter is used.

We begin to see how programming and statistical methods can help us extract insights from a dataset.

Pivoting

Sometimes it's useful to reshape the data in a table in order to see the patterns in the data more clearly. When we select certain columns (or features of the data) and display them against each other, this is called *pivoting* data.

A pivot table organizes data by categories and shows numerical values for those categories. This makes it easier to compare features than with a full dataset with many features.

Reviewing the first 12 rows of the delivery dataset:

```
delivery.head(12)
```

Output?

	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experience
0	522	7.93	Windy	Low	Afternoon	Scooter	12	1.0
1	738	16.42	Clear	Medium	Evening	Bike	20	2.0
2	741	9.52	Foggy	Low	Night	Scooter	28	1.0

	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experience
3	661	7.44	Rainy	Medium	Afternoon	Scooter	5	1.0
4	412	19.03	Clear	Low	Morning	Bike	16	5.0
5	679	19.40	Clear	Low	Evening	Scooter	8	9.0
6	627	9.52	Clear	Low	NaN	Bike	12	1.0
7	514	17.39	Clear	Medium	Evening	Scooter	5	6.0
8	860	1.78	Snowy	Low	Evening	Car	20	6.0
9	137	10.62	Foggy	Low	Evening	Scooter	29	1.0
10	812	16.86	Snowy	Medium	Afternoon	Car	13	4.0
11	77	15.54	Clear	Low	Night	Bike	29	1.0

Suppose we want to see the average delivery times for each type of vehicle under different traffic conditions.

This means we want to extract the vehicle type and the traffic level and display them against each other, which means we create a pivot table. The DataFrame method to create a pivot table is `pivot_table`. In a simple pivot, the `pivot_table` method accepts three inputs which are three features of our choice.

Our choice of data values is the `Delivery_Time_min`, and our features are `Traffic_Level` and `Vehicle_Type`.

We rearrange or pivot the features so that the `Vehicle_Type` data are along the rows, and the `Traffic_Level` data are along the columns, and the data in the new table are the mean delivery times for each condition.

- The feature that makes up the rows of data is the *index* of the pivot. The data must be categorical data.
- The feature that makes up the columns is the *columns* of the pivot. The data must be categorical data.
- The feature that fills in the new table is the *values* of the pivot.

To find the mean, we also set the `aggfunc` parameter with the `mean` method to find the average of the *values* data.

```
delivery.pivot_table(values="Delivery_Time_min",
                      index="Vehicle_Type",
                      columns="Traffic_Level",
                      aggfunc="mean")
```

Output?

	Traffic_Level	High	Low	Medium
Vehicle_Type				
Bike	63.440000	53.845361	55.949749	
Car	67.777778	55.574713	56.138462	
Scooter	65.295082	48.764706	56.071429	

The pivot table makes it easier to see that when traffic is high, a bike has the fastest average delivery time. When the traffic is low, a scooter has the fastest average delivery time. And when traffic is at a medium level, all three types of vehicle have about the same delivery time.

Joining

Sometimes the data we need are stored in different input files. Each file contains different data that belong to the same entities (such as customers, movies, purchases). In this case we could combine the data from different files into one DataFrame to analyze

In the delivery dataset, the features are all about different aspects of the delivery of the food order.

```
delivery.head()
```

Output?

	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experi
0	522	7.93	Windy	Low	Afternoon	Scooter	12	1.0
1	738	16.42	Clear	Medium	Evening	Bike	20	2.0
2	741	9.52	Foggy	Low	Night	Scooter	28	1.0
3	661	7.44	Rainy	Medium	Afternoon	Scooter	5	1.0
4	412	19.03	Clear	Low	Morning	Bike	16	5.0

Now suppose we have a `price.csv` file which also has data for the same food order, but the data is the price paid for each order.

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
price = pd.read_csv(url)
price.head()
```

Output?

	Order_ID	Price
0	522	38.41
1	738	40.57
2	741	79.60
3	661	70.80
4	412	71.31

Noting that both DataFrames contain different data, but they share the same `Order_ID`. This allows us to *join* the two tables into one table.

We join the `delivery` and `price` tables by using the format:

```
table1.merge(table2, on = common_column)
```

```
delivery_and_price = delivery.merge(price, on='Order_ID')
delivery_and_price.head()
```

Output?

	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experi
0	522	7.93	Windy	Low	Afternoon	Scooter	12	1.0
1	738	16.42	Clear	Medium	Evening	Bike	20	2.0
2	741	9.52	Foggy	Low	Night	Scooter	28	1.0
3	661	7.44	Rainy	Medium	Afternoon	Scooter	5	1.0
4	412	19.03	Clear	Low	Morning	Bike	16	5.0

Functions

Lastly for this chapter we learn a more advanced concept in programming that can make our programming more efficient and easier to maintain.

We've used functions many times in writing code for data analysis. `groupby` is a function that has code which will go through a dataset and separate the data based on the feature that it was given. Similarly, `display` is a function that will print the DataFrame that it was given.

A function is:

- a named block of code
- that is given some input and does work on that input
- and displays a result or returns a calculated value.

When we run or *call* a function by its name, the corresponding block of code runs, and if the function returns a value then we save the *return value* in a variable with an assignment `=` operator.

```
return value = function_name( zero or more input )
```

Use Case for Writing a Function

So far we've used functions that were written by other people and are stored as part of the Python language. But all the pre-written functions do a specific task, so when there is no existing Python function to do the work that we want, then it can be useful to write our own function.

Let's say there's an analysis step that requires four calculations to produce the result we want, and there is no existing function to do these calculations, so we write a short block of code to do these four calculations one after another.

Suppose a more complex analysis requires us to do the same four calculations at different times in the analysis. For each time, we can copy and paste the same four calculations and run them. But a more efficient way is to turn the short block of code with the four calculations into a function, and then we run the function.

Writing a Function

We would like to identify unusually long delivery times in the delivery dataset.

To do this, we need to do three steps:

1. Find the mean or average delivery time.
2. Find the standard deviation of the delivery time. Generally any data that's higher than three standard deviations is considered to be unusually high. (We'll cover standard deviation in more detail in a later chapter.)
3. Find the data that are above three standard deviations.

```
# find the average delivery time
average_delivery_time = delivery['Delivery_Time_min'].mean()

# find the standard deviation of delivery time
standard_deviation = delivery['Delivery_Time_min'].std()

# find the 3 standard deviations cutoff for delivery time
high_threshold = average_delivery_time + 3 * standard_deviation

# find the delivery times that are more than 3 standard deviations
too_high = delivery[delivery['Delivery_Time_min'] > high_threshold]

display(too_high['Delivery_Time_min'])
print("Average delivery time:", round(average_delivery_time,1))
```

Output?

```
29      123
127     141
379     153
452     141
784     126
Name: Delivery_Time_min, dtype: int64
```

Average delivery time: 56.7

We see that there are 5 delivery times that are much longer than the average delivery time, which is under one hour.

Suppose we now want to find the unusually high delivery times for when the weather is **Clear** or **Snowy**.

We can copy the block of code above and adjust it to select fair weather delivery times, do the calculations, and print the results. Then we copy the block of code again, adjust it again to select snowy weather delivery times, do the calculations, and print the results.

But a more convenient way would be to create a function from the block of code above.

To create a function, we need to:

- Write a function header with the function name, followed by a list of variables to store the input data that the function needs. The variables that store the input data are called the *parameters*.
The function header starts with the word **def**, short for *definition*, since we're defining the block of code that makes up the function. **def function_name(parameter1, parameter2, ...):**
- Following the header line is the *function body*, which has code that tells the function the work it needs to do. The function body is indented from the function header. We will copy the code above to be the function body and change the code so it works with the parameters.
- If the function needs to return a calculated data value, then we use the *return statement*:
return data_value

```
# convert block of code above to a function
```

```
# 1. start with the header: the name describes what the function does
#                               the parameter is the data that contains the delivery times
def find_high_delivery_times(data):
    # 2. copy code above and indent it to be inside the function
    # 3. change the code to use the parameter name
    average_delivery_time = data['Delivery_Time_min'].mean()
    standard_deviation = data['Delivery_Time_min'].std()
    high_threshold = average_delivery_time + 3 * standard_deviation
    too_high = data[data['Delivery_Time_min'] > high_threshold]
    # 4. return the delivery times that are more than 3 standard deviations
    return too_high['Delivery_Time_min']
```

Output?

After running the Code cell above so that Python knows the function definition, we can now use it with any selection of the delivery data simply by calling the function and giving it the data selection.

```
# find the unusually high delivery times for when the weather is clear

#1. filter the data to only include clear weather
clear_weather = delivery[delivery['Weather'] == 'Clear']

#2. find the unusually high delivery times by calling the function
high_delivery_times = find_high_delivery_times(clear_weather)

display(high_delivery_times)
```

Output?

```
29      123
127     141
379     153
Name: Delivery_Time_min, dtype: int64

# find the unusually high delivery times for when the weather is rainy

#1. filter the data to only include rainy weather
rainy_weather = delivery[delivery['Weather'] == 'Rainy']

#2. find the unusually high delivery times by calling the function
high_delivery_times = find_high_delivery_times(rainy_weather)

display(high_delivery_times)

Output?
```

```
452     141
Name: Delivery_Time_min, dtype: int64
```

We can see that after the function is defined, calling the function is much more convenient than copying and pasting a block of code and changing it to work with a specific dataset.

The advantage of writing *functions* are:

- Avoid repetition: we don't need to copy and paste a block of code over and over each time we want to do the same task.
- Easier to read the code: it's easier to recognize what `find_high_delivery_times` would do, compared to reading the block of code that makes up this function.
- Easier to maintain the code: if we decided to change the calculation to use two standard deviations instead of three, we would only need to make the change in the `find_high_delivery_times` function, instead of searching for the copies of the block of code to change them.

Summary

In this chapter we learn how to calculate descriptive statistics of data, separate data into groups based on features, create pivot tables to inspect multiple features, and join tables of data together. We also learn how to create functions to make our code more convenient to use. In the next chapters we'll apply them in statistical methods that help us analyze data.



7: Conditionals, Iterations, and Probabilities

In this chapter we learn the foundational concepts of statistics: randomness, probability, and simulation. Randomness and probability both work with the idea of an unknown outcome, such as when we flip a coin, we don't know if it will land heads side up or tails side up.

In order to work with unknown outcomes and observe their patterns, we first need to learn two new programming techniques that can (1) check what the outcome is and (2) repeat the experiment that produces the outcome. These techniques are called conditional statements and iterations.

Comparison Expressions

Comparison expressions, also known as *relational expressions* in programming, are used to compare two values. For example, "5 is greater than 3" is a comparison of two values.

We were introduced to comparison expressions in Chapter 3. Data Types, where we compare numbers and strings.

Python uses the same or similar symbols for comparison as with math:

- `<`: less than
- `<=`: less than or equal to
- `>`: greater than
- `>=`: greater than or equal to
- `==`: equal to
- `!=`: not equal to

The two values on the left and right of the comparison symbol (or *operator*) can be two numbers or two strings.

The result of the comparison is one of two values: `True` or `False`

```
5 > 3
```

Output?

True

```
limit = -2.5  
-2 <= limit
```

Output?

False

Comparing Sequences

When the value to be compared is an array or a column of a DataFrame, the comparison operator applies to the entire array or column. The result will be an array of `True` and `False` values.

```
import pandas as pd  
  
ice_cream = pd.DataFrame({'flavor': ['chocolate', 'strawberry', 'vanilla'],  
                           'price': [1.5, 1.25, 1]})  
ice_cream
```

Output?

	flavor	price
0	chocolate	1.50
1	strawberry	1.25
2	vanilla	1.00

We check for any price that's above \$1.25.

```
ice_cream.price > 1.25
```

Output?

```
0    True
1   False
2   False
Name: price, dtype: bool
```

Note that the result is an array of the comparison result of each price and 1.25.

If we want to know how many ice cream flavors cost more than \$1, we use the Numpy `count_nonzero` function with the resulting array.

```
import numpy as np

above_1 = ice_cream.price > 1
print("Number of ice cream flavors above $1:", np.count_nonzero(above_1))
```

Output?

```
Number of ice cream flavors above $1: 2
```

The `count_nonzero` will return the count of all `True` values. This is because internally, a Python `True` value is 1, and `False` is 0.

If we want to select the rows of the DataFrame where the cost is more than \$1, we use the comparison statement with the DataFrame to select the rows where the condition is True.

```
selected_rows = a_DataFrame[comparison]
```

The `selected_rows` will be the rows where the comparison results in True.

```
result = ice_cream[ice_cream.price > 1]
result
```

Output?

	flavor	price
0	chocolate	1.50
1	strawberry	1.25

Conditional Statements

As we see above, a comparison is a condition. `num > 0` is a comparison that describes the condition where `num` is a positive number.

A *conditional statement* is used to ask the computer to do a specific task if a certain condition is true. If the condition is not true, then the computer will not do the task.

A basic conditional statement starts with the keyword `if`:

```
if condition:
    do a task
```

This is why the conditional statement is often also called the *if statement*.

In the following Code cell:

- the `condition` is whether `num` is greater than 5
- if the condition is true, then the printing (the task) will take place
- if the condition is false, nothing is printed.

```
num = 10
if num > 5:
    print('num is greater than 5')
```

Output?

```
num is greater than 5
```

Can you change the value of `num` so that no printing will take place?

Sometimes we want to perform task 1 if the condition is true, and perform task 2 if the condition is false.

In this case, we use the longer format of the if statement:

```
if condition:
    do task 1
else:
    do task 2
```

Here we add the keyword `else`, followed by the task that needs to be done if the condition is false.

Can you predict what will be printed in the code below?

```
num = 10
if num > 5:
    print('num is greater than 5')
else:
    print('num is not greater than 5')
```

Output?

```
num is greater than 5
```

Can you change the value of `num` so the opposite will be printed?

In the above two cases, the condition can be one of two outcomes: True or False. But sometimes there can be more than two outcomes. An example would be a letter grade that a student receives, it can be one value out of 5 outcomes: A, B, C, D, or F.

In this case, we use the general format of the `if statement`, which is the longest format:

```
if condition 1:
    do task 1
elif condition 2:
    do task 2
elif condition 3:
    do task 3
...
elif condition N:
    do task N
```

```
else:  
    do task N+1
```

Here we see a new keyword `elif`, which describes one of the conditions for one of the outcomes.

To see how the `if elif else` statement works, we use the example of a letter grade for a student with a score of 78.

```
score = 78  
if score >= 90:  
    print('A')  
elif score >= 80:  
    print('B')  
elif score >= 70:  
    print('C')  
elif score >= 60:  
    print('D')  
else:  
    print('F')  
print("done")
```

Output?

```
C  
done
```

In the above example with the score of 78, the following steps will run:

- the score is compared with 90
- since the result is False, the score is compared with 80, the next condition
- since the result is False, the score is compared with 70, the next condition
- since the result is True, C is printed
- the rest of the comparisons are skipped
- the final "done" is printed

Can you predict all the steps that will be run if the score is 54?

Can you predict all the steps that will be run if the score is 95?

Iterations

By using conditional statements, we can check the outcome of an event, such as a coin toss or a letter grade from a class. Now we learn how to use *iterations* to repeat an event multiple times so that we can observe the general trend of the event. For example, the way we know that a coin is fair is if we toss the coin many times and see that about half the time, the coin lands on the heads side.

We will use the coin toss to show the importance of using iterations to see the trend of the outcome of an event.

First we need to generate the outcome, and we'll represent heads with 1 and tails with 0.

To simulate the random outcome of getting heads or tails, we use the `random` module and the `randint()` function of Python:

```
random_number = random.randint(start_num, end_num)
```

`randint` will return a random number between `start_num` and `end_num`, inclusive.

```
import random  
  
# get a random number between 0 and 1  
rand_num = random.randint(0,1)  
print(rand_num)
```

Output?

0

If we re-run the Code cell above a few times, we'll notice that sometimes the output is 1 and sometimes it's 0.

But instead of manually running the Code cell above multiple times, we can use an *iteration* technique to ask the computer to run the code repeatedly for us.

One technique is to use a *for loop*:

```
for i in range(number_of_times):  
    do a task
```

In the for loop above, a *task* will be run *number_of_times*.

The code is called a *loop* because the computer will run a *task* for the first time, then loop back to run a *task* again, then loop back to run a *task* again... until the *number_of_times* is reached.

Let's break down the components of the for loop.

- The first line of code has two keywords: *for* and *in*
- The *i* is a variable name, *i* is short for *iteration*. Since a *task* will be run *number_of_times*, each time a *task* runs is called one *iteration* of the loop.
- The *range()* function controls the *number_of_times* that the computer will loop back to run a *task*.

Now we put the simulation of the coin flip above together with the for loop to simulate tossing the coin five times:

```
for i in range(5):  
    rand_num = random.randint(0,1)  
    print(rand_num)
```

Output?

1
1
1
0
0

Since heads is 1 and tails is 0, we see that with the 5 simulated coin flips, we have a combination of heads and tails. Each time we run the code above, we get five different combinations of heads and tails.

Simulation

The simulation with five coin flips above shows us that each outcome is random and we can't predict what a specific outcome would be. But we can see a general pattern of the outcomes if we do many more coin flips, store all the outcomes, and then observe the pattern.

To store all the outcomes, we can use a Python *list* that we first discovered in Chapter 3. A *list* is a data type that can store many values in a sequence.

As we simulate each coin toss and get the outcome, we *append* or add it to the end of the list. When we've stored all data into the list, we convert the list to a Numpy array to run calculations and plot more efficiently.

```
#1. Create a list named L  
L = []  
  
#2. Loop or iterate 100 times  
for i in range(100):  
    # create a random outcome of 0 or 1
```

```
rand_num = random.randint(0,1)
# append to the list L
L.append(rand_num)

#3. convert the completed list to a numpy array
array = np.array(L)
print(array)
```

Output?

```
[1 1 0 0 1 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 0 0 1 1 0 0
 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 1 1 0 1 1 0 0 0 0 1 0
 1 0 1 0 1 0 1 0 0 0 0 1 1 0 0 0 0 1 1 0 1 1 0 0 1 1]
```

Observing the outcome from 100 coin tosses, it looks like there are about the same number of heads (1) as tails (0). This is what we initially guessed the result would be if the coin is a fair coin.

Using the `count_nonzero` function, we can count the actual number of 0s and 1s to confirm our theory:

```
number_heads = np.count_nonzero(array == 1)
number_tails = np.count_nonzero(array == 0)
print("Number of heads:", number_heads)
print("Number of tails:", number_tails)
```

Output?

```
Number of heads: 41
Number of tails: 59
```

The above is one simulation of tossing the coin 100 times. We want to repeat this simulation 200 times and observe how many heads and tails occur in each simulation.

Step 1

Create a function to simulate a coin being tossed 100 times.

We take the code to simulate the coin toss and the code to count the heads and tails and put them together in a function named `coin_toss_simulation`.

```
# coin_toss_simulation has 1 input: the number of times the coin is to be tossed
def coin_toss_simulation(number_of_times):
    # create a list to hold the random outcomes
    L = []
    # loop to simulate tossing a coin number_of_times
    for i in range(number_of_times):
        rand_num = random.randint(0,1)
        L.append(rand_num)
    # convert the completed list to a numpy array
    array = np.array(L)

    # count the number of heads and tails
    number_heads = np.count_nonzero(array == 1)
    number_tails = np.count_nonzero(array == 0)

    # return the number of heads and tails
    return number_heads, number_tails
```

Output?

Step 2

Repeat the simulation code above 200 times.

We create two more lists:

- **heads**: to store the number of heads for each simulation
- **tails**: to store the number of tails for each simulation

Then we use a for loop to run the simulation code 200 times and store the 200 pairs of heads and tails.

```
# create two empty lists to hold the number of heads and tails from each simulation
heads = []
tails = []
# loop 200 times to simulate tossing a coin 100 times
for i in range(200):
    number_heads, number_tails = coin_toss_simulation(100)
    heads.append(number_heads)
    tails.append(number_tails)
```

Output?

Since **heads** and **tails** contain the number of heads and number of tails from 200 simulations, we expect **heads** and **tails** to contain 200 values.

```
print("Number of values in heads:", len(heads))
print("Number of values in tails:", len(tails))
```

Output?

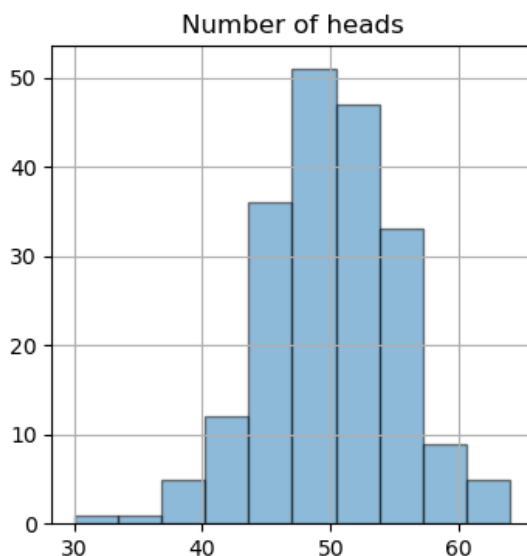
```
Number of values in heads: 200
Number of values in tails: 200
```

Instead of printing the 200 values and looking at them to find a pattern, which can be cumbersome, we take advantage of data visualization and plot them in a histogram:

```
import matplotlib.pyplot as plt

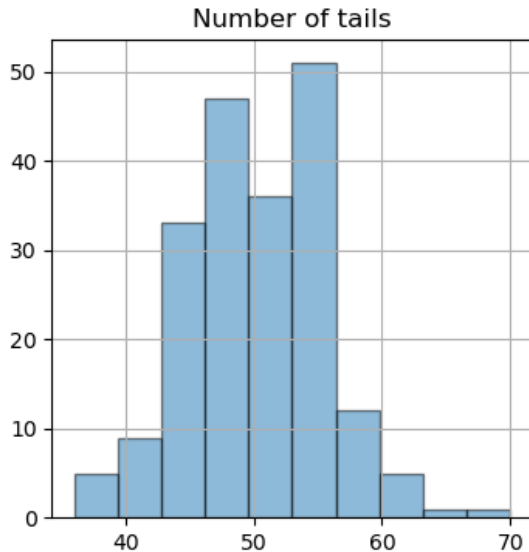
plt.figure(figsize=(4,4))
plt.hist(heads, alpha=0.5, edgecolor='black')
plt.title("Number of heads")
plt.grid()
plt.show()
```

Output?



```
plt.figure(figsize=(4,4))
plt.hist(tails, alpha=0.5, edgecolor='black')
plt.title("Number of tails")
plt.grid()
plt.show()
```

Output?



For both plots, we can see that the largest bins are around the 45-55 range. This means that out of the 200 simulations, many simulations have around 50 heads and 50 tails out of 100 coin tosses. There are fewer simulations where the ratio is 40/60, and the fewest simulations where the ratio is close to 35/65.

The two distributions above are called *empirical distributions*. They are distributions that we observe or obtain from our simulation of the coin toss. The simulation behaves exactly as we would expect for a fair coin: most simulations produce close to 50 heads and 50 tails, although the exact counts vary from one simulation to the next.

Finding Probabilities

Probability is a measure of how likely an event will happen. Probability is a value between 0 and 1, or 0% to 100%.

- When it's impossible for an event to happen, it has a probability of 0.
- When an event will happen for certain, it has a probability of 1.

An example of probability is when we toss a fair coin, the chance that a coin will land heads up is 50% or 0.5, and tails up is also 0.5.

Simulation is a powerful way to approximate the probability of the coin toss, as we've seen in the previous section. But the calculation of the probability of an event is determined through mathematics, often with the help of computers.

In this section we discuss the foundational rules for calculating probabilities by using mathematics.

Probability of an event

In math, the probability of an event is denoted as $P(\text{event})$.

In our coin toss simulation above there are 2 outcomes or 2 events:

- $P(\text{heads}) = 0.5$
- $P(\text{tails}) = 0.5$

Probability of an event not happening

If an event happens with probability $P(\text{event})$, then the probability that the event will not happen is:

$$P(\text{an event does not happen}) = 1 - P(\text{event})$$

If the chance of rain or $P(\text{rain})$ is 0.4, then the chance of it not raining is 0.6.

Probability of an outcome when multiple outcomes are equally likely

When an event has multiple, equally likely outcomes, such as rolling a six-sided die, then the chance of getting a particular side is:

$$P(\text{a particular side}) = 1 / \text{total number of sides}$$

This means when tossing a coin, which has 2 outcomes, the probability of getting tails is: 1/2.

It means when rolling a die, which has 6 sides, the probability of getting a 3 is: 1/6.

Probability of a subset of multiple outcomes

When an event has multiple, equally likely outcomes, such as rolling a die, the probability of getting a subset of N of the outcomes is:

$$P(\text{one of } N \text{ particular sides}) = N * (1 / \text{total number of sides})$$

This means when rolling a die:

- The chance of getting a 3 is: 1/6.
- The chance of getting a side that is a multiple of 2 (which is 2, 4, or 6) is: $1/6 + 1/6 + 1/6 = 3/6$.
- The chance of getting a 1 or a 6 is: $1/6 + 1/6 = 2/6$

Note that when we calculate the probability of multiple events that are *ORed* together (such as 1 or 6), we *add* together the probabilities of each event.

The combined probability of events that are *ORed* together is larger, which means it's more likely to happen.

Probability of multiple outcomes in a sequence

Suppose we have 3 cards: 1 red card, 1 green card, and 1 blue card. If we draw 2 cards out of the 3 cards without looking at them, then what is the probability of getting a red card first, then a green card second?

1. To get a red card out of the 3 cards, the probability is: 1/3
2. To get a green card out of the remaining 2 cards, the probability is: 1/2
3. The combined probability of the 2 outcomes that are in sequence is $1/3 * 1/2 = 1/6$

Note that when we calculate the probability of multiple events that are *ANDed* together (such as red and green), we *multiply* the probabilities of each event.

When events are *ANDed* together, the combined probability is smaller, which means it's less likely to happen.

Probability of at least one success

In our simulation function of 100 coin tosses above, we observe from the histogram of the number of heads that there are 35 to 65 times out of 100 that the outcome is heads.

We also know from when we simulate only one coin toss, that there's a chance that we'll have 0 heads.

How do we find the general probability of getting at least one heads with a certain number of coin tosses?

First, we hand calculate the probability for the first few coin tosses:

- For 1 coin toss
 $P(1 \text{ heads}) = 1/2$
- For 2 coin tosses
 $P(\text{at least 1 heads}) = 1 - P(2 \text{ tails}) = 1 - (1/2 * 1/2) = 3/4$
- For 3 coin tosses
 $P(\text{at least 1 heads}) = 1 - P(3 \text{ tails}) = 1 - (1/2 * 1/2 * 1/2) = 7/8$
- For 4 coin tosses
 $P(\text{at least 1 heads}) = 1 - P(4 \text{ tails}) = 1 - (1/2 * 1/2 * 1/2 * 1/2) = 15/16$

From the first 4 calculations, we can generalize the formula to N tosses:

$$P(\text{at least 1 heads}) = 1 - P(N \text{ tails}) = 1 - (1/2)^N$$

Now we can find the probability of getting at least one head for any number of coin tosses.

We also observe that as N gets larger, which means the number of coin tosses is larger, then the value $(1/2)^N$ gets smaller, which means the probability of getting at least one head approaches 1 or 100% certainty.

Summary

In this chapter we learn some new programming techniques, conditional statements and iterations. We use these techniques to work with some foundational concepts of statistics: when the outcome of an event is random, we can simulate it repeatedly to estimate the probability that the event occurs.

8: Sampling

To study or analyze a phenomenon in a large population, it is often not possible or practical to gather data from all of the population. For example, if we want to study the sleeping habits of people living in the US, it's difficult to get the sleeping data from every person living in the US. Instead, data scientists will choose a smaller number of people to represent the entire US population and obtain sleeping data from these people only. This smaller group of people is a *sample* of the US population.

In this chapter we learn the different types of *samples* and their properties.

Types of Samples

In a DataFrame, each row is the data record for one entity in the population. To sample the data and get a subset of the data, we select certain rows of the DataFrame. The way we select the rows determines the type of samples.

To illustrate the different sampling methods, we use the `cars` dataset that we've used in a previous chapter.

```
import pandas as pd

url = 'https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
cars = pd.read_csv(url)
print("Number of rows:", len(cars))
print("First 10 rows:")
cars.head(10)
```

Output?

Number of rows: 392
First 10 rows:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino
5	15.0	8	429.0	198.0	4341.0	10.0	70	1	ford galaxie 500
6	14.0	8	454.0	220.0	4354.0	9.0	70	1	chevrolet impala
7	14.0	8	440.0	215.0	4312.0	8.5	70	1	plymouth fury iii
8	14.0	8	455.0	225.0	4425.0	10.0	70	1	pontiac catalina
9	15.0	8	390.0	190.0	3850.0	8.5	70	1	amc ambassador dpl

Deterministic Sample

A *deterministic sample* is the result of identifying and selecting specific rows, without using any random chance.

One way to specify certain rows is by their row numbers or index numbers:

```
specific_rows = cars.iloc[[1, 2, 5]]
specific_rows
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
5	15.0	8	429.0	198.0	4341.0	10.0	70	1	ford galaxie 500

Another way is to specify a certain characteristic of the data:

```
specific_cylinders = cars[cars['cylinders'] == 3]
specific_cylinders
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
70	19.0	3	70.0	97.0	2330.0	13.5	72	3	mazda rx2 coupe
110	18.0	3	70.0	90.0	2124.0	13.5	73	3	mazda rx3
241	21.5	3	80.0	110.0	2720.0	13.5	77	3	mazda rx-4
331	23.7	3	70.0	100.0	2420.0	12.5	80	3	mazda rx-7 gs

Probability Sample

A *probability sample* is one where each member of the population has a known and non-zero chance of being selected to be in the sample.

The `cars` DataFrame has 398 rows, as seen in the first Code cell. If we randomly select 10 rows to create a sample, then each car has the same chance of being chosen, which is:

$P(\text{being selected}) = 10/398$

To randomly choose N rows from a DataFrame, we use the `sample()` method.

```
sample_DataFrame = a_DataFrame.sample(n = N)
```

```
# Choose 5 rows randomly from cars
random_sample = cars.sample(n = 5)
random_sample
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
175	23.0	4	115.0	95.0	2694.0	15.0	75	2	audi 100ls
129	32.0	4	71.0	65.0	1836.0	21.0	74	3	toyota corolla 1200
80	28.0	4	97.0	92.0	2288.0	17.0	72	3	datsum 510 (sw)
198	18.0	6	250.0	78.0	3574.0	21.0	76	1	ford granada ghia
301	31.8	4	85.0	65.0	2020.0	19.2	79	3	datsum 210

To choose N random values from a column of a DataFrame or an array, we use the Numpy `random.choice()` method.

```
random_sample = np.random.choice(array, N)
```

```
# Choose 4 car_name from cars
import numpy as np
random_car_names = np.random.choice(cars.car_name, 4)
print(random_car_names)
```

Output?

```
['toyota corolla' 'pontiac phoenix' 'dodge aries wagon (sw)' 'amc matador']
```

Systematic Sample

A special type of probability sample is a systematic sample, where we select every k-th value in the population.

Steps for systematic sampling `cars` to get a sample size of 10:

1. Find the sampling intervals based on the sample size. The larger the sample size, the smaller the interval.
2. Find a random starting point in the first interval.
3. Select every k-th index from the starting point.

```
# 1. Find the sample interval
interval = len(cars) // 10

#2. Find a random starting point
start = np.random.choice(interval)

#3. Select every k-th index from the starting point
systematic_sample = cars.iloc[start::interval]
systematic_sample
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
32	19.0	6	232.0	100.0	2634.0	13.0	71	1	amc gremlin
71	15.0	8	304.0	150.0	3892.0	12.5	72	1	amc matador (sw)
110	18.0	3	70.0	90.0	2124.0	13.5	73	3	maxda rx3
149	31.0	4	79.0	67.0	2000.0	16.0	74	2	fiat x1.9
188	14.5	8	351.0	152.0	4215.0	12.8	76	1	ford gran torino
227	16.0	8	400.0	180.0	4220.0	11.1	77	1	pontiac grand prix lj
266	27.2	4	119.0	97.0	2300.0	14.7	78	3	datsum 510
305	26.8	6	173.0	115.0	2700.0	12.9	79	1	oldsmobile omega brougham
344	37.7	4	89.0	62.0	2050.0	17.3	81	3	toyota tercel
383	22.0	6	232.0	112.0	2835.0	14.7	82	1	ford granada l

In step 3, `[start::interval]` means select the indices starting from the `start` starting point, all the way to the last index, and skipping every `interval` index positions.

For example: `[2::5]` means start at index 2, go to the last index, skipping every 5 indices, which means we get rows at indices: 2, 7, 12, 17, etc.

Sampling With and Without Replacement

Sampling with replacement means that each time we select an item from a population, we put it back before drawing the next one. This allows the same item to be selected more than once in the sample.

An example of sampling with replacement would be if we select a card out of a 52-card deck, look at the card face value, then put it back in the deck. The next time we select another card, we could end up with the same card we selected before.

Sampling without replacement means that each time we select an item from a population, it is removed from the population. This

means no item will appear more than once in the sample.

An example of sampling without replacement is if we select a card out of a deck of cards and put it aside. When we select another card, we cannot select the same card again.

If we roll a die three times and record the three output numbers, would that be sampling with replacement or sampling without replacement?

Empirical Distribution

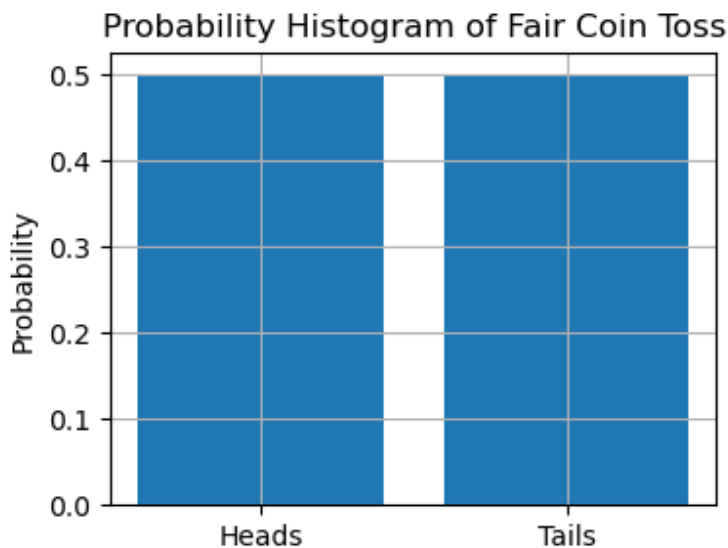
In data science, the word *empirical* means *observed*. So an empirical distribution is the histogram of observed data, such as data from a sample, since we measure and observe only the sample and not the entire population.

In the previous Chapter 7, we discussed that when a fair coin is tossed, $P(\text{heads})$ is 0.5 and $P(\text{tails})$ is 0.5. If we plotted the theoretical probability distribution, we would get two equal bars.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(4,3))
heights = [1/2, 1/2]
plt.bar(['Heads', 'Tails'], heights)
plt.ylabel('Probability')
plt.title('Probability Histogram of Fair Coin Toss')
plt.grid()
plt.show()
```

Output?



Now we use the same coin toss simulation function from Chapter 7 to get the observed numbers of heads and tails for N coin tosses:

```
import random

# coin_toss_simulation has 1 input: the number_of_times the coin is to be tossed
def coin_toss_simulation(number_of_times):
    # loop number_of_times to tossing a coin number_of_times
    L = []
    for i in range(number_of_times):
        rand_num = random.randint(0,1)
        L.append(rand_num)
```

```
outcome = np.array(L)
return outcome    # outcome is a numpy array of 0s and 1s, where 0 = heads and 1 = tails
```

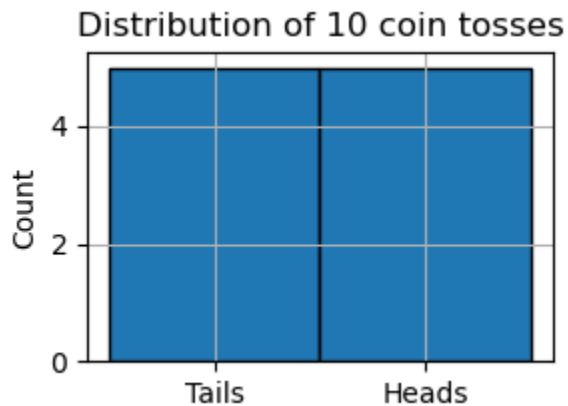
Output?

Then we call the function to simulate 10 coin tosses, and plot the distribution of the resulting number of heads and tails

```
# 10 coin tosses
outcome = coin_toss_simulation(10)

# plot the outcome
plt.figure(figsize=(3,2))
plt.hist(outcome, bins=2, edgecolor='black')
plt.xticks([0.25, 0.75], ['Tails', 'Heads'])
plt.ylabel('Count')
plt.title('Distribution of 10 coin tosses')
plt.grid()
plt.show()
```

Output?

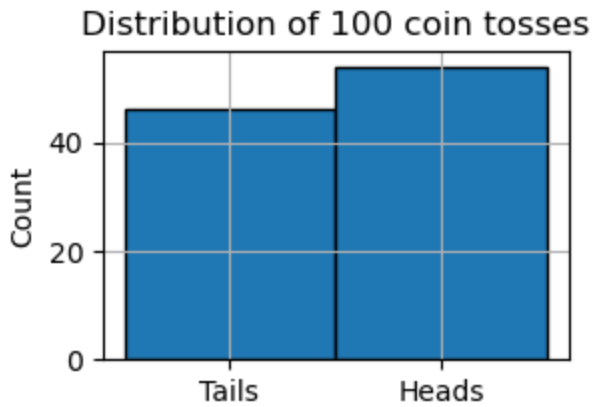


Next we repeat the same simulation for 100 coin tosses and then 1000 coin tosses.

```
# 100 coin tosses

outcome = coin_toss_simulation(100)
plt.figure(figsize=(3,2))
plt.hist(outcome, bins=2, edgecolor='black')
plt.xticks([0.25, 0.75], ['Tails', 'Heads'])
plt.ylabel('Count')
plt.title('Distribution of 100 coin tosses')
plt.grid()
plt.show()
```

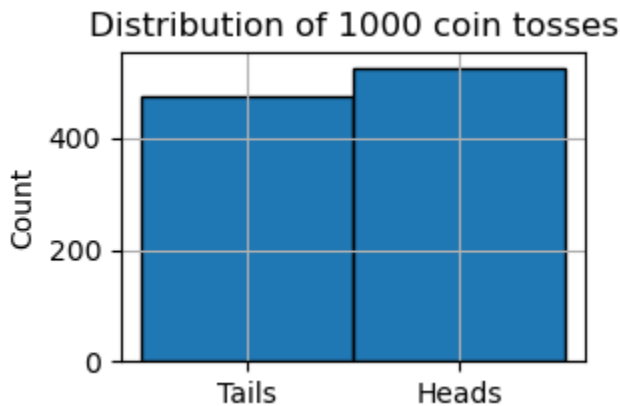
Output?



1000 coin tosses

```
outcome = coin_toss_simulation(1000)
plt.figure(figsize=(3,2))
plt.hist(outcome, bins=2, edgecolor='black')
plt.xticks([0.25, 0.75], ['Tails', 'Heads'])
plt.ylabel('Count')
plt.title('Distribution of 1000 coin tosses')
plt.grid()
plt.show()
```

Output?



If we run the Code cells for 10, 100, 1000 coin tosses several times, we can consistently see that as the number of coin tosses gets higher, the empirical histogram approaches to the calculated theoretical histogram. This is known as the *convergence of empirical distribution*, which happens because of the *Law of Large Numbers*.

The *Law of Large Numbers* states that as we repeat an experiment more times (tossing the coin more times), the average experiment result will get closer and closer to the average of the theoretical expected value.

Empirical Distribution and Sampling

The Law of Large Numbers also applies to sampling. The larger the sample size, the more closely the empirical distribution (or the distribution of the sample) resembles the population distribution.

To observe this, we continue with the delivery time dataset.

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_"
```

```
delivery = pd.read_csv(url)
print("First 5 rows")
delivery.head()
```

Output?

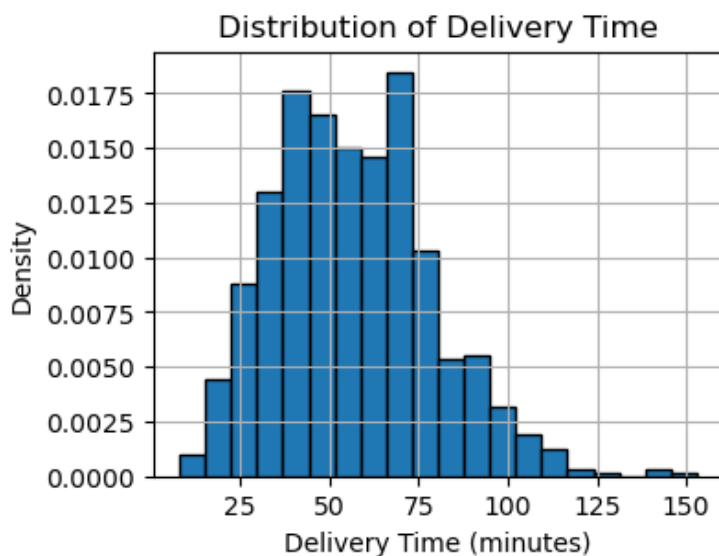
First 5 rows

	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experience
0	522	7.93	Windy	Low	Afternoon	Scooter	12	1.0
1	738	16.42	Clear	Medium	Evening	Bike	20	2.0
2	741	9.52	Foggy	Low	Night	Scooter	28	1.0
3	661	7.44	Rainy	Medium	Afternoon	Scooter	5	1.0
4	412	19.03	Clear	Low	Morning	Bike	16	5.0

First we plot the distribution of the delivery times for the entire population.

```
plt.figure(figsize=(4,3))
plt.hist(delivery.Delivery_Time_min, bins=20, edgecolor='black', density=True)
plt.title("Distribution of Delivery Time")
plt.xlabel("Delivery Time (minutes)")
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?



Note that we use a density histogram (from Chapter 5) since we will be comparing the distributions of different size datasets.

Observing the histogram, we see that the data is close to a bell curve, but has a longer right tail, where the excessively long delivery times are. We check for the number of these long delivery times.

```
(delivery.Delivery_Time_min > 130).value_counts()
```

Output?

Delivery_Time_min

```
False    997
True      3
Name: count, dtype: int64
```

Since there are only three delivery times that are excessively long, we will drop them and only consider the delivery times that are in the bulk of the data.

```
normal_delivery_times = delivery[delivery.Delivery_Time_min <= 130]
len(normal_delivery_times)
```

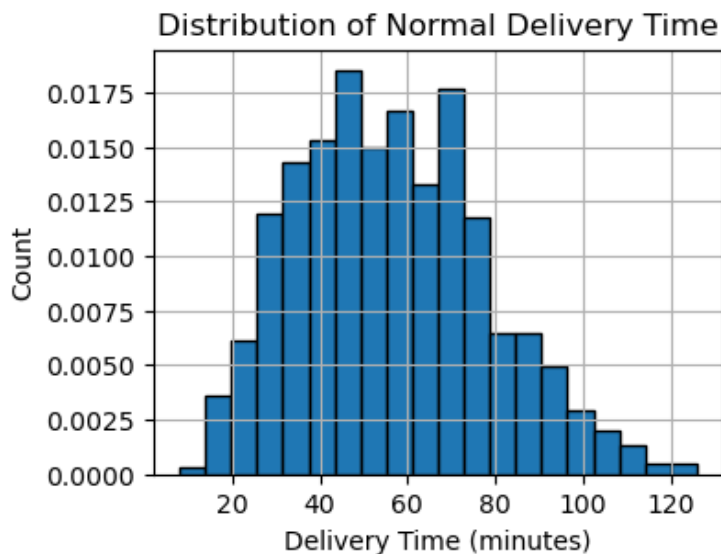
Output?

```
997
```

We plot the histogram of the normal delivery times.

```
plt.figure(figsize=(4,3))
plt.hist(normal_delivery_times.Delivery_Time_min, bins=20, edgecolor='black', density=True)
plt.title("Distribution of Normal Delivery Time")
plt.xlabel("Delivery Time (minutes)")
plt.ylabel("Count")
plt.grid()
plt.show()
```

Output?

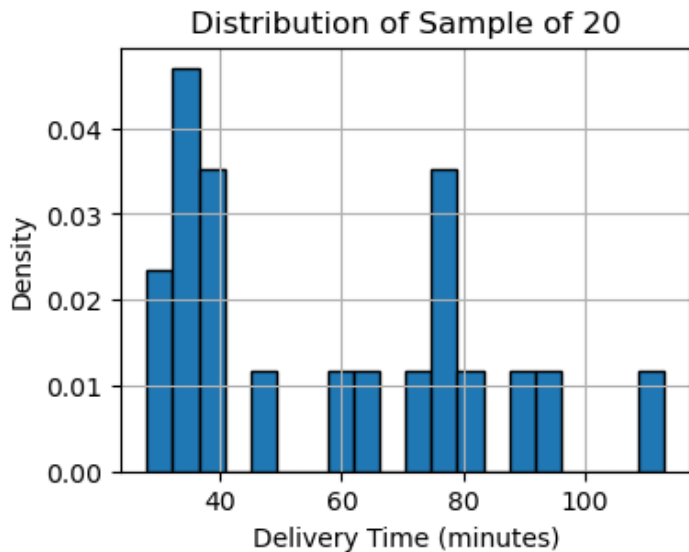


This histogram is our goal. We want the samples below to have a similar distribution.

First we randomly draw 20 delivery times out of `normal_delivery_times` and plot their distribution.

```
sample_20 = normal_delivery_times.sample(n=20)
plt.figure(figsize=(4,3))
plt.hist(sample_20.Delivery_Time_min, bins=20, edgecolor='black', density=True)
plt.title("Distribution of Sample of 20")
plt.xlabel("Delivery Time (minutes)")
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?

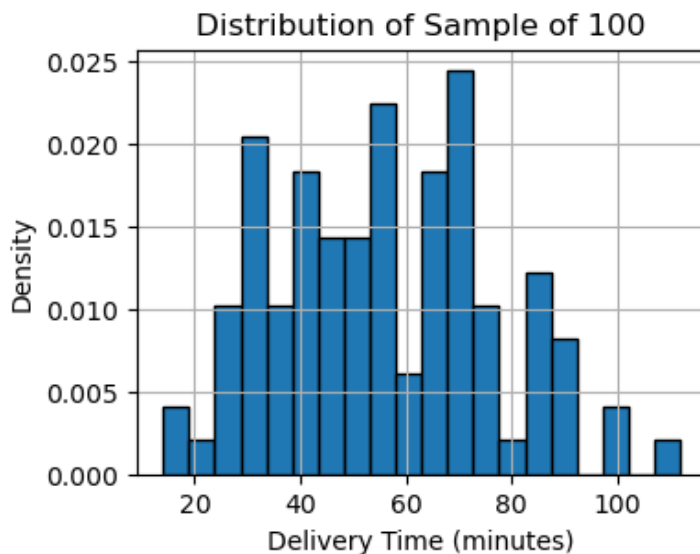


Clearly the sample is too small to have a distribution that's similar to the population distribution.

Next we draw 100 and 250 delivery times and plot their distributions.

```
sample_100 = normal_delivery_times.sample(n=100)
plt.figure(figsize=(4,3))
plt.hist(sample_100.Delivery_Time_min, bins=20, edgecolor='black', density=True)
plt.title("Distribution of Sample of 100")
plt.xlabel("Delivery Time (minutes)")
plt.ylabel("Density")
plt.grid()
plt.show()
```

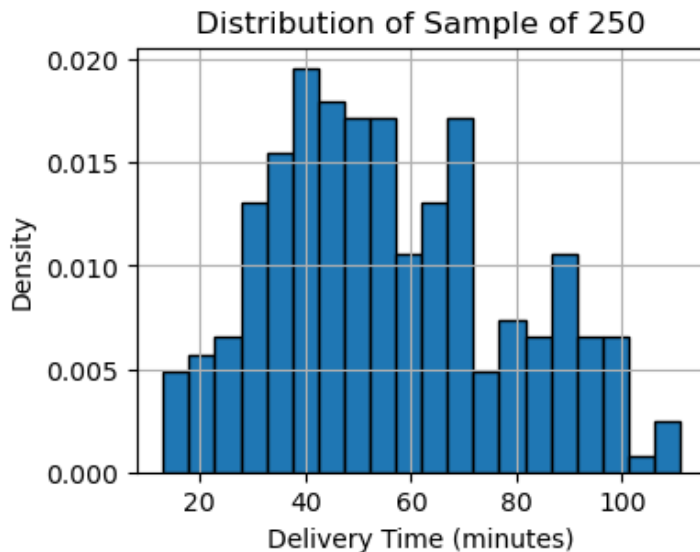
Output?



```
sample_250 = normal_delivery_times.sample(n=250)
plt.figure(figsize=(4,3))
plt.hist(sample_250.Delivery_Time_min, bins=20, edgecolor='black', density=True)
plt.title("Distribution of Sample of 250")
plt.xlabel("Delivery Time (minutes)")
```

```
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?



We can see that the larger the sample size, the more similar its distribution is to the distribution of the population.

Empirical Distribution of a Statistic

Often data scientists are interested in a numerical measurement of a population, for example, what is the mean height of 18-year-olds in California. The mean height is an unknown measurement of the 18-year-old population and is called a *parameter* of the population.

Since it is not practical and is expensive to measure the height of each 18-year-old in California, data scientists will create a sample of 18-year-olds, measure their heights, and find the mean of these heights. The resulting mean is an estimate of the mean height *parameter* and it is called a *statistic*.

Note that the *statistic* or estimate of the *parameter* is singular, it doesn't have the same meaning as the plural *statistics*, which is a field of mathematics.

The steps to find and study the *statistic* of a sample:

1. Decide what statistic to simulate.
2. Draw a random sample and calculate the statistic.
3. Repeat step 2 a number of times to get many readings of the statistic.
4. Plot the readings to observe the statistic.

Step 1. Determine the statistic to simulate

Using the delivery time dataset, we choose the median delivery time as the statistic we want to study.

Recall that the median is the midpoint of the dataset, where 50% of the data is higher and 50% of the data is lower.

We start by calculating the median for all delivery times in the dataset. This value is the value we hope to estimate with our samples.

```
print("Population median delivery time:", normal_delivery_times.Delivery_Time_min.median())
```

Output?

Population median delivery time: 55.0

Step 2. Sample and calculate the statistic

We write a function to take a sample of 250 delivery times and find the median delivery time. The function returns the median delivery time.

```
def median_sample_delivery_time():  
    sample = normal_delivery_times.sample(n=250)  
    return sample.Delivery_Time_min.median()
```

Output?

Step 3. Repeat step 2 to get many readings of the statistic

Then we write code to repeatedly call the function above 5000 times and save the returned median delivery times. Generally the number of repetition is between 1,000 and 10,000 times.

```
delivery_medians = []  
for i in range(5000):  
    delivery_medians.append(median_sample_delivery_time())  
print(len(delivery_medians))
```

Output?

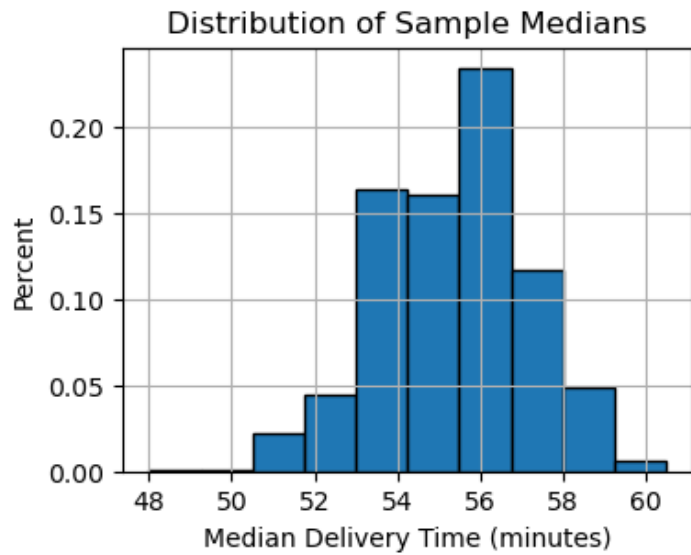
5000

Step 4. Plot the readings

We plot the `delivery_medians` distribution.

```
plt.figure(figsize=(4,3))  
plt.hist(delivery_medians, edgecolor='black', density=True)  
plt.title("Distribution of Sample Medians")  
plt.xlabel("Median Delivery Time (minutes)")  
plt.ylabel("Percent")  
plt.grid()
```

Output?



We see that the median delivery time estimates are between 50 and 60 minutes, and the majority of the readings are between 54 and 57 minutes, which match the actual median delivery times of 55 minutes for the population.

Summary

In this chapter we learn the power of sampling and simulation. Since studying an entire population is rarely practical, we rely on population sampling and simulation techniques to accurately estimate real-world statistics and distributions.

9: Hypothesis Testing

In this chapter we will apply our understanding of random samples and empirical distributions to test a *hypothesis* about a population. A hypothesis is an assumption that we make about a dataset's parameter. An example hypothesis is: people who exercise regularly report lower stress levels.

In hypothesis testing, we use a statistical process to decide whether observed data from a sample provide enough evidence to support a hypothesis about a larger population.

Steps of Hypothesis Testing

Hypothesis testing begins when we have a yes/no question to answer about a parameter of a dataset. Some examples include:

- Do people who exercise regularly have lower stress levels than those who don't exercise?
- Do students who study with background music remember the material better than those who study with no background music?
- In sports, do the home teams win more often?

The question involves two different results and is about *whether the difference is due to random chance or due to an influence*.

Hypothesis testing involves these steps:

- State the hypothesis: the null hypothesis and alternative hypothesis
- Build a model of the data for the null hypothesis
- Obtain the observed data
- Compare the observed data to the model's data and calculate the p-value
- Draw a conclusion

We will explore how hypothesis testing works with a coin toss experiment.

Suppose we have a coin and want to know if it's a fair coin, which means that it has an equal chance of landing heads or tails. We'll use the steps of hypothesis testing to see if the coin is fair.

State the Hypothesis

Since the yes/no question we try to answer has two results, such as win/loss data when a sport team plays at home and win/loss data when the team plays away from home, hypothesis testing includes two hypotheses:

1. The *null hypothesis* says that there is no significant difference between the two results. Any difference is due to random chance rather than due to the influence of a real phenomenon.
The term *null* means that any difference is due to *nothing* but chance.
2. The *alternative hypothesis* says that any difference between the two results is due to *some influence rather than chance*.

For the coin toss experiment:

1. The null hypothesis: The coin is fair, there is no influence making heads or tails more likely. $P(\text{heads})$ is 0.5.
2. The alternative hypothesis: The coin is not fair and something influences it to land on one side more often. $P(\text{heads})$ is not 0.5.

Build a Model

The *model* shows what the data would look like if the null hypothesis is true.

For the coin toss experiment, we build a model that shows the likely result if the coin is fair. But before we use the model for hypothesis testing, it is important to choose a model and assess that it is valid for our data.

Assess the Model

We want a model that truly reflects a fair coin being tossed. In Chapter 8 we see that as the sample size increases, the empirical distribution approaches the theoretical distribution. Therefore, our model will simulate a coin being tossed 500 times and records the number of heads. Then we repeat the simulation 10,000 times to get 10,000 numbers of heads. Because we repeat the simulation a large number of times, the overall distribution of our results will be similar to the theoretical distribution for a fair coin.

Use the Model

In the code below we re-use the coin toss simulation function from Chapter 8.

```
import random
import numpy as np

# coin_toss_simulation has 1 input: the number_of_times the coin is to be tossed
def coin_toss_simulation(number_of_times):
    # loop number_of_times to tossing a coin number_of_times
    L = []
    for i in range(number_of_times):
        rand_num = random.randint(0,1)
        L.append(rand_num)
    outcome = np.array(L)
    return outcome    # outcome is a numpy array of 0s and 1s, where 0 = heads and 1 = tails

# run the simulation 10,000 times, each time the coin is tossed 500 times
total_number_of_heads = []
for i in range(10000):
    # get array of 0s and 1s from tossing a coin 500 times
    results = coin_toss_simulation(500)
    # get the number of heads from the results
    number_of_heads = np.sum(results == 0)
    # save the number of heads to the total_number_of_heads list
    total_number_of_heads.append(number_of_heads)
# convert the total_number_of_heads list to a numpy array
model_heads_arr = np.array(total_number_of_heads)
```

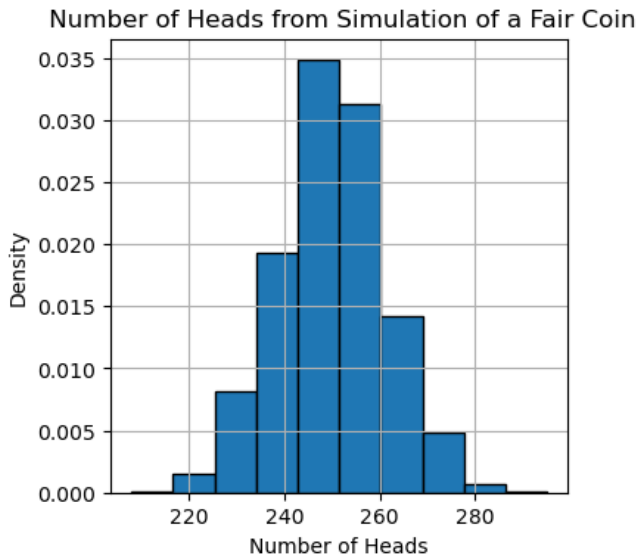
Output?

`model_heads_arr` contains 10,000 numbers, each number is the number of heads from one coin toss simulation. Next we plot the model's distribution of the number of heads.

```
import matplotlib.pyplot as plt

# plot the distribution of the number of heads from the simulation
plt.figure(figsize=(4,4))
plt.hist(model_heads_arr, edgecolor='black', density=True)
plt.title(" Number of Heads from Simulation of a Fair Coin")
plt.xlabel("Number of Heads")
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?



Because the coin is tossed many times, the distribution of the number of heads is a bell curve, as can be seen above (why it's a bell curve is discussed in Chapter 12). The bell curve shows that the majority of the simulations produce between 240 and 260 heads, which corresponds with the expected value of 250 heads out of 500 coin tosses.

Obtain the Observed Data

To truly test a real coin, we would take the coin and toss it 500 times, recording each result. But for this exercise we'll simulate the coin toss and pretend this is our actual coin toss results with the real coin.

```
# toss the coin 500 times
actual_results = coin_toss_simulation(500)
# count the number of heads and tails
observed_heads = np.sum(actual_results == 0)
observed_tails = np.sum(actual_results == 1)
print("Number of heads:", observed_heads)
print("Number of tails:", observed_tails)
```

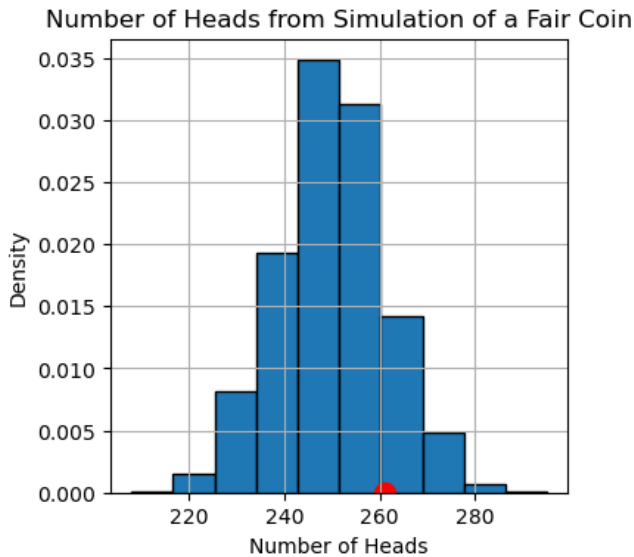
Output?

```
Number of heads: 261
Number of tails: 239
```

We bring back the model's distribution of the number of heads, and plot our observed number of heads from tossing the actual coin being tested.

```
# plotting the number of heads in the distribution of the number of heads from the simulat
plt.figure(figsize=(4,4))
plt.hist(model_heads_arr, edgecolor='black', density=True)
plt.title("Number of Heads from Simulation of a Fair Coin")
plt.xlabel("Number of Heads")
plt.ylabel("Density")
# add a red circle for the observed number of heads
plt.scatter(observed_heads, 0, color='red', s=100)
plt.grid()
plt.show()
```

Output?



In the plot we can clearly see that the observed result, the red circle, is in the range of 240-260 heads expected for a fair coin, so it looks like a fair coin.

Compare Data and Calculate p-value

We can also measure and compare the observed data and the model's data. To do this, we need to identify what the *test statistic* is. Recall from Chapter 8 that a *statistic* is a measurement of the data that summarizes an aspect of the data, such as the mean height of people in the dataset. In hypothesis testing the *test statistic* measures of how far the data is from what we expect in the null hypothesis.

The null hypothesis says that there's no influence that makes the observed data different from the model's data. This means the observed data (from tossing the tested coin) should be similar to the model's data (from tossing the fair coin). And the model's data from tossing the fair coin says that out of 500 coin tosses, we expect 250 will be heads.

Given that 250 is what we expect in the null hypothesis, the test statistic is how far the number of heads is from 250. This means we find the difference between the number of heads and 250.

Comparing the data involves 3 steps:

- find the difference between the observed number of heads and 250. This is the observed distance.
- find the difference between each of the 10,000 simulated numbers of heads and 250. This is an array of simulated distances.
- then we find what proportion of the simulated distances is larger (or farther from 250) than the observed distance.

If the proportion is large (≥ 0.05), then the observed number is well within the range of what the null hypothesis says, since the simulated result is the data of the null hypothesis.

If the percentage is small (< 0.05), then the observed number is outside the range of what the null hypothesis says.

This proportion is called the *p-value*. If the *p-value* is large, then we can reject the relative hypothesis.

And if the *p-value* is small, it is *statistically significant* and we can reject the null hypothesis.

```
observed_distance = abs(observed_heads - 250)
simulated_distances = np.abs(model_heads_arr - 250)
p_value = np.sum(simulated_distances >= observed_distance)/10000
# or:
p_value = np.mean(simulated_distances >= observed_distance)
print("p-value:", p_value)
```

Output?

p-value: 0.3487

In the calculation above:

- We use the absolute value of the difference because we want to find the distance from 250 only, we don't need to know if it's on the right or left of 250.
- To calculate the p-value, since `simulated_distances` is a numpy array, the comparison `simulated_distances >= observed_distance` results in a numpy array of 0s and 1s. Adding all the 1s (or the counts of when the condition is true) and dividing by 10,000 gives us the proportion of when the `simulated_distances` are larger than or equal to the `observed_distance`.
- Since the numpy array contains 0s and 1s, adding all the data in the array is the same as adding all the 1s. And if we add all the data in the array and divide the total by the count of data in the array, we effectively calculate the mean of the data in the array.

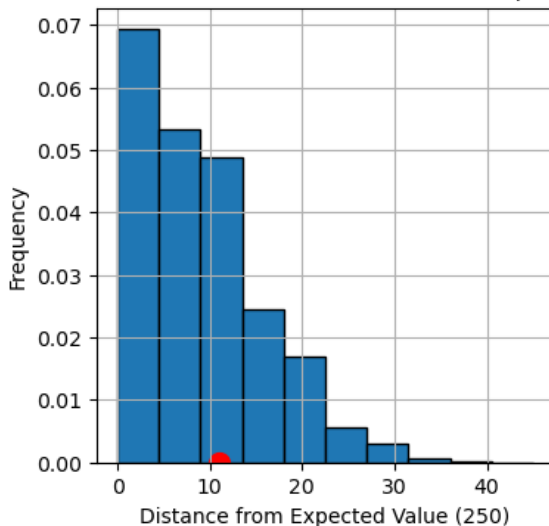
To see how the `observed_distance` is compared to the `simulated_distances` of a fair coin, we show where the `observed_distance` is in the distribution of the `simulated distances`

```
# show where the observed_distance is in a distribution of the simulated_distances
```

```
plt.figure(figsize=(4,4))
plt.hist(simulated_distances, edgecolor='black', density=True)
plt.xlabel('Distance from Expected Value (250)')
plt.ylabel('Frequency')
plt.title('Distribution of Simulated Distances from Expected Value')
# add a red circle for the observed distance
plt.scatter(observed_distance, 0, color='red', s=100)
plt.grid()
plt.show()
```

Output?

Distribution of Simulated Distances from Expected Value



We can see that the `observed_distance` is in the typical range `simulated_distances` and not in the right tail end. This is why the p-value is large: there are many `simulated_distances` that are larger than the `observed_distance`.

And with the large p-value, we can reject the *alternative hypothesis*, which states that something influences the coin to land on one side more often. Therefore there isn't enough evidence to conclude that the coin is biased.

Statistically Significant p-value

We now investigate what will happen if we have a biased coin.

We simulate the biased coin by making it likely to land heads 55% of the time, and keep everything else the same.

```
def biased_coin():
    # Generate a random float between 0 and 1
    roll = random.random()
    if roll <= 0.55: # 55% chance for heads
        return 0
    else:
        return 1

# the coin toss simulation now calls the biased_coin function instead of random.randint
def coin_toss_simulation_with_biased_coin(number_of_times):
    # loop number_of_times to tossing a coin number_of_times
    L = []
    for i in range(number_of_times):
        rand_num = biased_coin()
        L.append(rand_num)
    outcome = np.array(L)
    return outcome
```

Output?

```
# same simulation step as with the fair coin, but now with the biased coin
actual_results = coin_toss_simulation_with_biased_coin(500)
observed_heads = np.sum(actual_results == 0)
observed_tails = np.sum(actual_results == 1)
print("Number of heads:", observed_heads)
print("Number of tails:", observed_tails)

# find the p-value for the biased coin
observed_distance = abs(observed_heads - 250)
simulated_distances = np.abs(model_heads_arr - 250)
p_value = np.mean(simulated_distances >= observed_distance)
print("p-value:", p_value)
```

Output?

```
Number of heads: 272
Number of tails: 228
p-value: 0.0551
```

Since the coin has a small bias such that the $P(\text{heads})$ is 0.55 instead of 0.5, it's a good idea to run the code above a few times and observe the output. The p-value can change with each simulation of the biased coin toss, but generally it's below the 0.05 threshold for p-value.

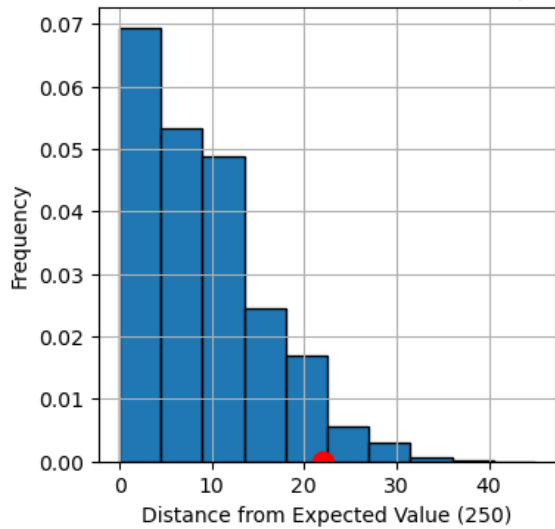
When the p-value is less than 0.05, we look at where the observed distance is compared to the distribution of the simulated distances.

```
# show where the observed_distance is in a distribution of the simulated_distances

plt.figure(figsize=(4,4))
plt.hist(simulated_distances, edgecolor='black', density=True)
plt.xlabel('Distance from Expected Value (250)')
plt.ylabel('Frequency')
plt.title('Distribution of Simulated Distances from Expected Value')
# add a red circle for the observed distance
plt.scatter(observed_distance, 0, color='red', s=100)
plt.grid()
plt.show()
```

Output?

Distribution of Simulated Distances from Expected Value



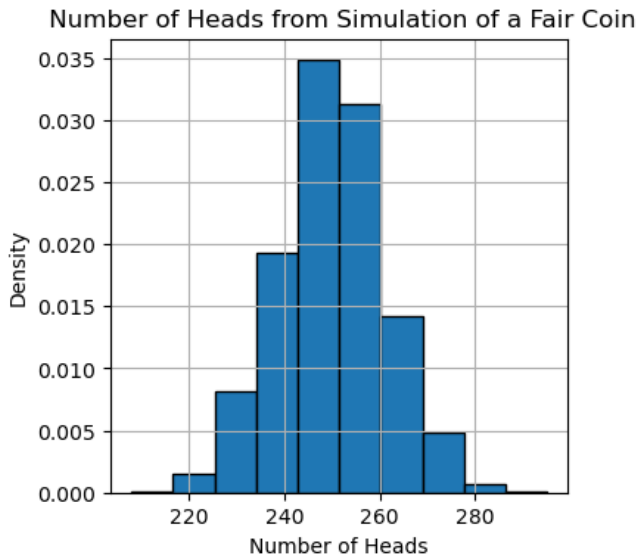
When the observed distance is at the tail end of the distribution of simulated distances, we can see that very few simulated distances are larger than the observed distance, causing the p-value to be small and statistically significant. This means we reject the null hypothesis, and the coin is biased.

Error Probability

Because the coin is tossed many times, the distribution of the number of heads is a bell curve, as discussed above and shown again here:

```
# plot the distribution of the number of heads from the simulation
plt.figure(figsize=(4,4))
plt.hist(model_heads_arr, edgecolor='black', density=True)
plt.title(" Number of Heads from Simulation of a Fair Coin")
plt.xlabel("Number of Heads")
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?



When the data has the bell curve shape or is a normal distribution, then 95% of the data is within 2 standard deviations from the mean of the data (discussed in Chapter 12). This means the remaining 5% of the data occurs much less frequently and that is why the p-value cutoff is 0.05.

For our simulated data, 2 standard deviations is:

```
print("2 standard deviations or p-value cut off:", np.std(model_heads_arr) * 2)
```

Output?

2 standard deviations or p-value cut off: 22.373348608556565

Using our p-value cutoff of 0.05, this means if a *fair* coin is tossed 500 times and the number of heads is above 250+22 or 272, or below 250-22 or 228, we will conclude *incorrectly* that the coin is not a fair coin.

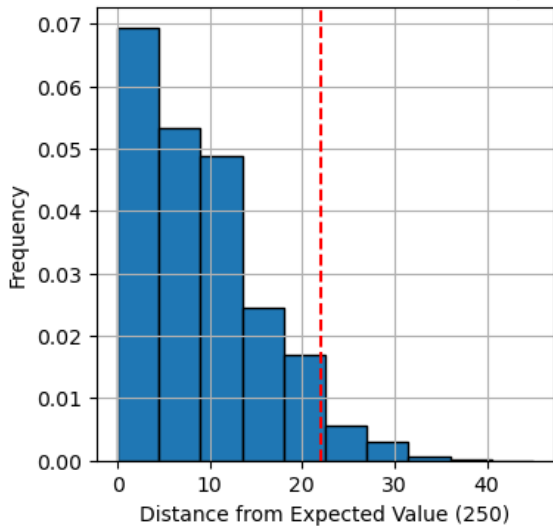
The histogram of the simulated distances from the expected value also shows that we can incorrectly conclude that it's not a fair coin.

```
# show where the p-value cut off is in a distribution of the simulated_distances

plt.figure(figsize=(4,4))
plt.hist(simulated_distances, edgecolor='black', density=True)
plt.xlabel('Distance from Expected Value (250)')
plt.ylabel('Frequency')
plt.title('Distribution of Simulated Distances from Expected Value')
# add a red dashed line at x=22
plt.axvline(x = 22, color='red', linestyle='--')
plt.grid()
plt.show()
```

Output?

Distribution of Simulated Distances from Expected Value



With the dashed red line as the p-value cut off point, we see that a fair coin still has a small chance that its distance from the expected value would be large enough that we would conclude *incorrectly* that the coin is biased.

Therefore the p-value cut off is also called the *error probability*, or the probability that we could be in error with our conclusion.

Being aware of the error probability is important. For example, it means that any conclusion made from breakthrough scientific research should not rely on a single experiment. The experiment needs to be replicated, in case the error probability leads to the wrong conclusion the first time.

Summary

In this chapter we applied sampling and simulation to build a model to do hypothesis testing. The simulated data in our model can reject or fail to reject the null hypothesis, where the hypothesis is our assumption about a parameter of the data. When we draw a conclusion about the hypothesis, we need to keep in mind that there is an error probability caused by the nature of random variation in our data.

10: Two Sample Tests

Up to now we've done statistical analysis with one data sample, such as the number of heads in a coin toss experiment. But often data scientists also need to make decisions based on comparing two data samples with each other, such as when comparing two groups of students who studied with background music and with no background music.

In this chapter we use hypothesis testing with two distinct groups of data to determine if the differences between them are just due to random chance or if the difference is meaningful.

Gather Data

The data are from the US Center for Disease Control and Prevention (CDC) and is stored at [OpenIntro](#). The CDC annually surveys the health status and habits of 350,000 people in the US, and the dataset is a random sample of 20,000 people from the survey conducted in 2000.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
health = pd.read_csv(url)
display(health.head())
print("Number of rows:", health.shape[0])
```

Output?

	genhlth	exerany	hlthplan	smoke100	height	weight	wtdesired	age	gender
0	good	0	1	0	70	175	175	77	m
1	good	0	1	1	64	125	115	33	f
2	good	1	1	1	60	105	105	49	f
3	good	1	1	0	66	132	124	42	f
4	very good	0	1	0	61	150	130	55	f

Number of rows: 20000

The rows from left to right are: general health, any exercise in the past month, health plan coverage, smoke at least 100 cigarettes in respondent's life, height, weight, desired weight, age, and gender.

We will look at two features: exercise habit or **exerany** (0 for no exercise in past month, 1 for exercise in past month), and **weight** (a quantitative variable). We want to know whether people who exercise have a lower body weight distribution than people who don't exercise.

Exploratory Data Analysis (EDA)

It might seem reasonable that people who exercise have lower body weight than those who don't exercise. However, a person's weight is influenced by their height. Taller people naturally tend to weigh more than shorter people, no matter how much they exercise.

Assessing the Model

If we only look at the weight, then when we build the model for the null hypothesis, if one of our sample groups happen to have slightly higher average height, then their weight would tend to be higher regardless of their exercise habit. We also need to take

into account the differences in height when comparing the two groups.

In this case the height is a *confounding variable*. A confounding variable is an extra factor that can bias the results when studying only weight and exercise.

To take into account both the height and weight, we will use the Body Mass Index (BMI) instead. The formula for BMI is

$$\text{BMI} = \frac{\text{weight}}{\text{height}^2} \times 703 \quad (10.1)$$

With the BMI value, the weight is scaled relative to the height.

We add another column to the `health` DataFrame to store the BMI.

```
# find the BMI from the weight and height
BMI = health.weight / (health.height ** 2) * 703

# add to the health DataFrame
health['bmi'] = BMI
display(health.head(10))
```

Output?

	genhlth	exerany	hlthplan	smoke100	height	weight	wt Desire	age	gender	bmi
0	good	0	1	0	70	175	175	77	m	25.107143
1	good	0	1	1	64	125	115	33	f	21.453857
2	good	1	1	1	60	105	105	49	f	20.504167
3	good	1	1	0	66	132	124	42	f	21.303030
4	very good	0	1	0	61	150	130	55	f	28.339156
5	very good	1	1	0	64	114	114	55	f	19.565918
6	very good	1	1	0	71	194	185	31	m	27.054553
7	very good	0	1	0	67	170	160	45	m	26.622856
8	good	0	1	1	65	150	130	27	f	24.958580
9	good	1	1	0	70	180	170	44	m	25.824490

We see that on row index 0 someone weighs 175 lbs, which is heavier than the person on row index 8, whose weight is 150. But because of their height difference, they have very similar BMI. Now we can compare standardized body mass (BMI) against exercise habit.

Prepare Data

Next we concentrate on the two features, exercise habit and BMI, by creating a new DataFrame `exercise` that only has the `exerany` and `bmi` columns.

```
exercise_bmi = health[['exerany', 'bmi']]
display(exercise_bmi.head())
print("Number of rows:", exercise_bmi.shape[0])
```

Output?

	exerany	bmi
0	0	25.107143
1	0	21.453857

	exerany	bmi
2	1	20.504167
3	1	21.303030
4	0	28.339156

Number of rows: 20000

We find the count and proportion of exercise and non-exercise groups.

```
# Find the exercise and no exercise groups
exercise = exercise_bmi[exercise_bmi["exerany"] == 1]
no_exercise = exercise_bmi[exercise_bmi["exerany"] == 0]

print("No exercise (0): count =", len(exercise), "    proportion =", round(len(exercise)/len
print("Exercise (1): count =", len(no_exercise), "    proportion = ", round(len(no_exercise)
```

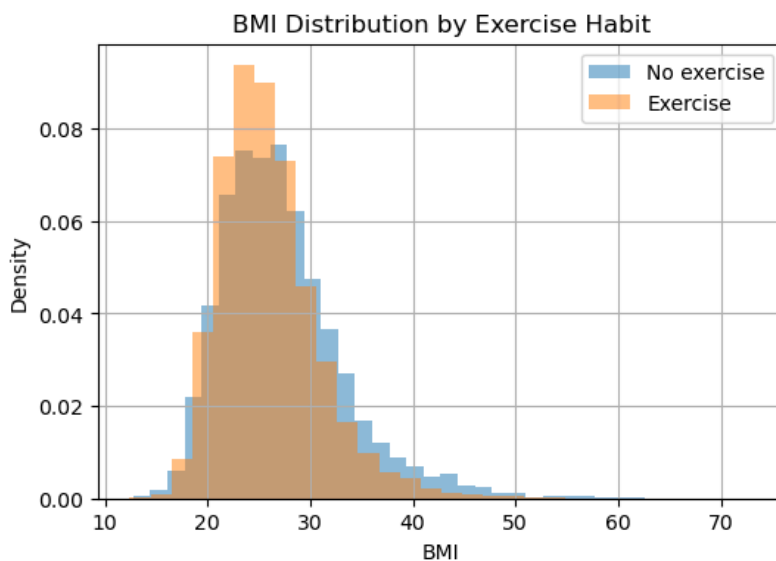
Output?

```
No exercise (0): count = 14914    proportion = 0.75
Exercise (1): count = 5086    proportion = 0.25
```

Then we plot the distribution of the BMI of the two groups.

```
plt.figure(figsize=(6, 4))
plt.hist(no_exercise["bmi"], bins=30, density=True, alpha=0.5, label="No exercise")
plt.hist(exercise["bmi"], bins=30, density=True, alpha=0.5, label="Exercise")
plt.legend()
plt.title("BMI Distribution by Exercise Habit")
plt.xlabel("BMI")
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?



It looks like the BMI distributions overlap quite a bit for the exercise and non-exercise groups. We also see that there seems to be a right tail in the distribution, which means that there are some people with BMI values that are much higher than those of most people.

The Test Statistic

To compare the exercise and no exercise group's BMI, we find the difference in the mean BMI of the two groups.

```
# Calculate the mean BMI for each group
mean_bmi_exercise = exercise["bmi"].mean()
mean_bmi_no_exercise = no_exercise["bmi"].mean()
observed_bmi_difference = mean_bmi_exercise - mean_bmi_no_exercise

print("Mean BMI for exercise group:", round(mean_bmi_exercise, 2))
print("Mean BMI for no exercise group:", round(mean_bmi_no_exercise, 2))
print("Difference (exercise - no exercise):", round(observed_bmi_difference, 2))
```

Output?

```
Mean BMI for exercise group: 25.98
Mean BMI for no exercise group: 27.27
Difference (exercise - no exercise): -1.29
```

There is a small difference: the mean BMI of the exercise group is lower than the mean BMI of the no exercise group by 1.29.

Is the 1.29 difference due to random chance or is there evidence of a difference between the two groups?
We will use hypothesis testing to find the answer.

Hypothesis Testing

We'll use the same steps as in Chapter 9, but this time we're working with two data samples so our model for the null hypothesis will be built differently.

State the Hypothesis

The *null hypothesis*: There is no difference between the mean BMI of the exercise and no exercise groups. Any difference is due to random chance.

The *alternative hypothesis*: There is a difference between the mean BMI of the exercise and no exercise groups.

Build the Model

Recall from Chapter 9 that we want to build a model for the null hypothesis, which means the model should show us what the test statistic looks like if there is no difference in the mean BMI of the exercise and no exercise groups.

If there is no difference, then whether a person is labeled as "exercise" or "no exercise" should not be associated with their BMI. We can model this by randomly shuffling the **exerany** (the exercise habit) column among the people in the dataset. This is called *random permutation*.

Random Permutation Example

To see how random permutation works, we copy the first seven rows of the **exercise_bmi** DataFrame into an example DataFrame so we can randomly shuffle the **exerany** column.

```
example = pd.DataFrame(exercise_bmi.head(7))
display(example)
```

Output?

exerany		bmi
0	0	25.107143

exerany		bmi
1	0	21.453857
2	1	20.504167
3	1	21.303030
4	0	28.339156
5	1	19.565918
6	1	27.054553

Then we shuffle the `exerany` column by using the same `sample` function discussed in Chapter 8:

```
sample_DataFrame = a_DataFrame.sample(n = N)
```

But this time we use `replace=False` for sampling with no replacement, and we use `frac=1` so 100% of the data are selected.

```
shuffled_exerany = example["exerany"].sample(frac=1, replace=False)
display(shuffled_exerany)
```

Output?

```
5    1
2    1
4    0
0    0
6    1
3    1
1    0
Name: exerany, dtype: int64
```

Note that the index values show us that the rows of `exerany` had been shuffled randomly.

We combine the `shuffled_exerany` column with the original `example` DataFrame.

```
# add the shuffled column to example,
# and reset the index so it matches the index of the DataFrame
example["shuffled_exerany"] = shuffled_exerany.reset_index(drop=True)
display(example)
```

Output?

	exerany	bmi	shuffled_exerany
0	0	25.107143	1
1	0	21.453857	1
2	1	20.504167	0
3	1	21.303030	0
4	0	28.339156	1
5	1	19.565918	1
6	1	27.054553	0

As the `example` DataFrame shows, the BMI of each person is the same, but after shuffling, the person has been randomly assigned to the exercise or no exercise group. Note also that the shuffle guarantees that we will end up with the same number of people in each group. The original `exerany` feature has four people in the exercise group and three people in the no exercise group, and the `shuffled_exerany` feature has the same counts.

Build the Null Distribution

After shuffling the `exerany` feature of the dataset, we calculate the mean BMI of the two *shuffled* groups and find the difference. This difference is due to random chance, as assumed by the null hypothesis.

Then we repeat the two steps: 1) shuffle and 2) find the mean BMI difference between the two groups, so we can have a large number of differences. These differences will show us the range of values we would expect from random chance.

To get ready to repeat the two steps of shuffling and finding the difference, we write a function that does the two steps.

```
def one_permutation(dataframe) :
    # create a copy of the original dataframe to add a new shuffled_treatment column
    dataframe_permutation = dataframe.copy()

    # create a shuffled_exerany column
    shuffled_exerany = dataframe_permutation["exerany"].sample(frac=1, replace=False)
    dataframe_permutation["shuffled_exerany"] = shuffled_exerany.reset_index(drop=True)

    # separate the data into two shuffled groups
    exercise = dataframe_permutation[dataframe_permutation["shuffled_exerany"] == 1]
    no_exercise = dataframe_permutation[dataframe_permutation["shuffled_exerany"] == 0]

    # calculate the mean BMI of each group
    exercise_mean_bmi = exercise["bmi"].mean()
    no_exercise_mean_bmi = no_exercise["bmi"].mean()

    # return the difference between the mean BMI values
    return exercise_mean_bmi - no_exercise_mean_bmi
```

Output?

We repeat the permutation 1000 times to get enough simulated differences that can occur by random chance.

```
# create a list to store the 1000 differences
differences = []

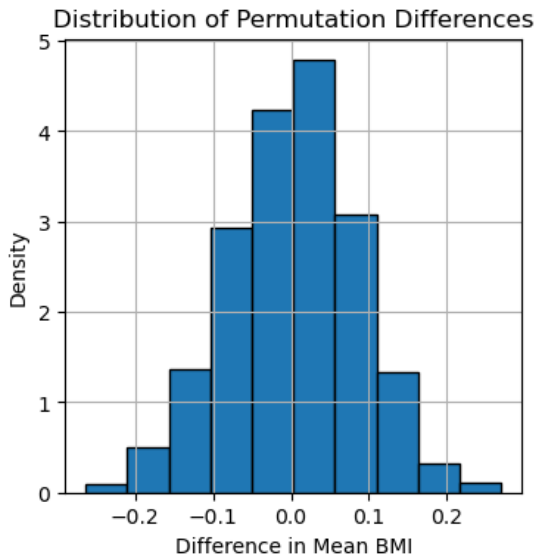
for i in range(1000):
    one_difference = one_permutation(exercise_bmi)
    differences.append(one_difference)
mean_bmi_difference_array = np.array(differences)
```

Output?

We plot the distribution of the differences.

```
plt.figure(figsize=(4, 4))
plt.hist(mean_bmi_difference_array, density=True, edgecolor='black')
plt.title("Distribution of Permutation Differences")
plt.xlabel("Difference in Mean BMI")
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?



As expected, if the null hypothesis is true and the difference in mean BMI of the two groups is due to random chance, then the difference tends to be close to 0. This can be seen in the distribution above, with most of the differences between -0.15 and 0.15.

Compare Observed Data and the Model's Data

When we calculated the test statistic with the observed data in an earlier step, we found that Exercise group mean BMI - No exercise group mean BMI = -1.29

Given the null distribution above, -1.29 is very far outside the range of what the 1000 permutations produced by random chance. Therefore, it's unlikely that the difference in the observed data is due to random chance.

p-value Calculation

We can calculate the p-value to see if it agrees with our conclusion above. We use the same calculation of p-value as in Chapter 8.

```
observed_distance = abs(observed_bmi_difference)
simulated_distance = np.abs(mean_bmi_difference_array)
p_value = np.mean(simulated_distance >= observed_distance)
print(p_value)
```

Output?

0.0

The calculated p-value of 0.0 is expected. With the 1000 permutations that we did, none of them had a distance larger than 1.29. But this doesn't mean that the p-value is exactly 0.0. It appears 0.0 for 1000 simulations, so we can say that the p-value is smaller than 1/1000 or smaller than 0.001.

With a p-value value that's less than 0.001, the p-value is statistically significant. This agrees with our conclusion from observing the null distribution, therefore we reject the null hypothesis and conclude that there is evidence of a difference in mean BMI between the exercise and no exercise groups.

Observational Data

It is important to note that our hypothesis testing above provides evidence that there is an *association* between BMI value and

exercise habit. There seems to be a difference in BMI values among people who exercise and those who don't, but we cannot say that exercise causes the difference in BMI. This is because people were not randomly assigned to the exercise or no exercise group.

If people are randomly assigned to a group, the people who have healthy diets can be assigned to either group, and older and younger people can be assigned to either group. Random assignment tends to balance these other characteristics between the groups, making exercise habit one of the primary difference between the groups. This makes it possible to infer that exercise may cause a difference in mean BMI.

On the other hand, when people are surveyed for their health condition, as in our dataset, then the two groups can be different in many ways. For example, people who exercise may also have healthy diets or are younger. These factors, the confounding variables, may also influence BMI so it's not possible to say that exercise itself causes the difference in BMI.

The dataset for the exercise habit and mean BMI is *observational data*. They are data gathered by measuring or recording behaviors, events, or phenomena as they naturally occur, without manipulating any characteristics of the data. Observational data can only show evidence of association between two factors, they cannot be used to determine cause and effect.

Causality and A/B Testing

We now look at an example of *experimental data*, which is the opposite of *observational data*. In experimental data people are randomly assigned to two groups, and this makes it possible to infer a cause and effect relationship as discussed above.

The [Criteo Uplift Prediction](#) dataset contains data from online advertising experiments conducted by Criteo, a digital advertising company. Users were randomly assigned to either an advertising treatment group or a control group. Criteo could show advertisements to users in the treatment group but withheld its advertising from users in the control group. Then the company recorded whether the user visited the advertiser's website.

This type of randomized experiment is commonly called an A/B test. A/B testing is a type of randomized experiment used to compare two alternatives. Participants are randomly assigned to one of two groups:

- Group A (control): receives the existing or control condition.
- Group B (treatment): receives the new condition being tested.

We then compare an outcome from the two groups to determine whether the treatment made a difference. Because participants are randomly assigned, A/B testing can provide evidence that the treatment caused a difference in the outcome.

A/B testing is widely used in business, technology, and marketing to determine whether a change causes an improvement in a desired outcome.

Gather Data

The original dataset contains millions of records and many features. For this example, we use a random sample of 20,000 users and focus on two features:

- treatment: whether the user was randomly assigned to the control group (0) or advertising treatment group (1)
- visit: whether the user did not visit (0) or visited (1) the advertiser's website

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
ad = pd.read_csv(url)
print("Last 10 rows")
display(ad.tail(10))
print("Number of rows:", ad.shape[0])
```

Output?

Last 10 rows

	treatment	visit
19990	1	0
19991	1	0
19992	1	1
19993	1	0
19994	1	0
19995	1	0
19996	1	0
19997	1	0
19998	1	0
19999	1	0

Number of rows: 20000

In the last 10 rows, the users were all in the treatment group, and only one of them visited the website.

The Test Statistic

We first see how many users from each group visited or not visited the website.

```
ad.value_counts()
```

Output?

```
treatment  visit
1          0    16136
0          0    2916
1          1     828
0          1     120
Name: count, dtype: int64
```

We see that for the treatment group, 16,136 people did not visit the website and 828 visited. For the control group, 2916 people did not visit the website and 120 people visited.

For each group we find the rate of visit, which is the proportion

$$\frac{\text{number of visits}}{\text{total count of visits and no visits}}$$

We observe that since the data is 1 or 0 only, if we find the sum of the visits (all the 1 values + all the 0 values), the result equals the sum of all the 1 values, which is also the number of visits. This means the fraction above can be written as

$$\frac{\text{sum of visits}}{\text{total count of visits and no visits}} = \text{the mean of visits}$$

The test statistic is the difference in the proportion of website visits for each group. Does the treatment group have a higher proportion of website visits than that of the control group?

```
# select a group's visit column and find the mean
control_visit_proportion = round(ad[ad["treatment"] == 0]["visit"].mean(), 4)
treatment_visit_proportion = round(ad[ad["treatment"] == 1]["visit"].mean(), 4)
observed_visit_difference = round(treatment_visit_proportion - control_visit_proportion, 4)
print("Proportion of control group that visited the website", control_visit_proportion)
print("Proportion of treatment group that visited the website", treatment_visit_proportion)
print("Observed difference:", observed_visit_difference)
```

Output?

```
Proportion of control group that visited the website 0.0395
Proportion of treatment group that visited the website 0.0488
Observed difference: 0.0093
```

We see that the observed difference between the two groups is only 0.0093 or 0.93%. This looks like a small percentage, but we don't know yet if it's due to random chance. We will use hypothesis testing to investigate.

Hypothesis Testing

The question we ask about the treatment and control groups:

Is the difference in visit proportions likely due to random chance, or is there evidence that advertising increases the visit proportion?

State the Hypothesis

The *null hypothesis*: There is no difference in the proportion of website visits between the treatment group and control group. Any difference is due to random chance.

The *alternative hypothesis*: There is a difference in the proportion of the website visits between the treatment and control groups.

Build the Model

Just as in the previous example with the mean BMI difference, we will randomly shuffle the `treatment` column while leaving the `visit` column unchanged for one permutation, then we find the difference in visit proportion between the two groups. Then we repeat the permutation 1000 times to get the distribution of differences.

```
def one_permutation(dataframe) :
    # create a copy of the original dataframe to add a new shuffled_treatment column
    dataframe_permutation = dataframe.copy()

    # create a shuffled_treatment column
    shuffled_treatment = dataframe_permutation["treatment"].sample(frac=1, replace=False)
    dataframe_permutation["shuffled_treatment"] = shuffled_treatment.reset_index(drop=True)

    # separate the data into two shuffled groups
    treatment = dataframe_permutation[dataframe_permutation["shuffled_treatment"] == 1]
    control = dataframe_permutation[dataframe_permutation["shuffled_treatment"] == 0]

    # calculate the mean BMI of each group
    treatment_proportion = treatment["visit"].mean()
    control_proportion = control["visit"].mean()

    # return the difference between the mean BMI values
    return treatment_proportion - control_proportion
```

Output?

We repeat the permutation to get 1000 differences and plot them.

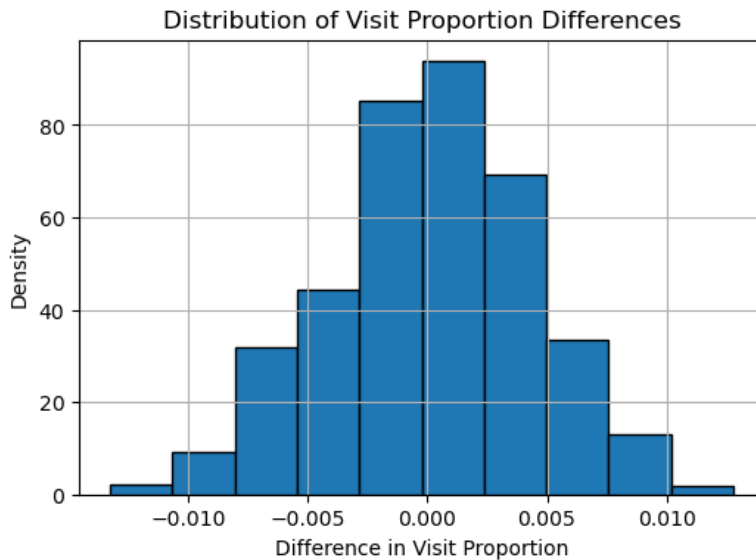
```
# create a list to store the 1000 differences
differences = []

for i in range(1000):
    one_difference = one_permutation(ad)
    differences.append(one_difference)
visit_difference_array = np.array(differences)
```

Output?

```
plt.figure(figsize=(6, 4))
plt.hist(visit_difference_array, density=True, edgecolor='black')
plt.title("Distribution of Visit Proportion Differences")
plt.xlabel("Difference in Visit Proportion")
plt.ylabel("Density")
plt.grid()
plt.show()
```

Output?



Compare the Observed Data and the Model's Data

If the null hypothesis is true, then the difference in visit proportion between the two groups would be centered around 0. The distribution shows that most of the differences are in the range of -0.005 and 0.005. Our observed difference is about 0.009, which is near the right tail of the distribution. This means the observed difference is likely rare.

We calculate the p-value to confirm that it is rare.

```
observed_distance = abs(observed_visit_difference)
simulated_distance = np.abs(visit_difference_array)
p_value = np.mean(simulated_distance >= observed_distance)
print(p_value)
```

Output?

0.022

Conclusion

The p-value of 0.036 is smaller than the significance level of 0.05. Therefore it is statistically significant and we reject the null hypothesis. We conclude that the data provide evidence that the treatment group has a higher website visit rate than the control group, and because users were randomly assigned to the treatment and control groups, there is evidence that the advertising treatment caused an increase in website visits.

However, the distribution shows that there are differences that are as large as or even larger than the observed difference of 0.009. This means that they can occasionally occur due to random chance. Our p-value of 0.036 means that, if the null hypothesis was true, differences at least as extreme as the observed difference occurred in about 3.6% of our simulations.

Therefore, rejecting the null hypothesis does not guarantee that it is false. There is still a possibility that we observed an unusual

result due to random chance. Rejecting a null hypothesis that is actually true is called a Type I error.

Total Variation Distance

In the previous example we worked with a dataset with two categories of results: visit or no visit. With two categories, the statistic that we want to measure is simple: it's the difference in visit proportion between the two groups. We don't have to separately consider the no-visit proportion because it is determined by the visit proportion.

But suppose there are three categories of results: website visit, no visit, and purchase. Now each group has three proportions, and finding the difference in the proportions of only one category is not sufficient. This is where the *Total Variation Distance* or TVD comes in. TVD is a way to measure the difference between two distributions while taking into account the proportion of *all* categories.

To calculate the TVD:

- for each category, find the absolute value of the difference in proportion
- add the absolute values together
- divide the sum by 2

To see how TVD is calculated, we create an example of the three categories of result: visit, no visit, and purchase. We also have their proportions for both treatment and control groups, and the difference between the proportions:

```
example = pd.DataFrame(
    {
        "control": [0.8, 0.14, 0.06],
        "treatment": [0.7, 0.2, 0.1],
        "difference": [-0.10, 0.06, 0.04]
    },
    index=["no visit", "visit", "purchase"],
)
display(example)
```

Output?

	control	treatment	difference
no visit	0.80	0.7	-0.10
visit	0.14	0.2	0.06
purchase	0.06	0.1	0.04

We note that for the control and treatment groups, each proportion adds up to 1.0.

We also note that the decrease in "no visit" (-0.10) is redistributed to "visit" and "purchase" which together increase by the same amount (0.06 and 0.04). Therefore 0.10 or 10% of the proportion has been redistributed among the categories. We could think of this as 10% of the people changing categories to turn the control distribution into the treatment distribution.

If we simply add the difference of the two groups, we would get: $-0.10 + 0.06 + 0.04 = 0.0$

The differences cancel out because the proportions in both groups have to add up to 1.0.

Instead we add the absolute value of the difference so the decreases and increases do not cancel each other, but this results in a value that's doubled what it should be: $0.10 + 0.06 + 0.04 = 0.20$

This counts the redistribution twice: once when the 0.10 leaves no visit, and again when the same 0.10 appears as 0.06 in visit and 0.04 in purchase.

Therefore we divide by 2 to get the actual difference between the two groups.

```
# calculate the TVD

absolute_difference = np.abs(example["treatment"] - example["control"])
total = np.sum(absolute_difference)
```

```
tvd = total / 2  
print("TVD:", round(tvd, 2))
```

Output?

TVD: 0.1

The TVD gives us one number that summarizes the differences across all the categories.

A TVD of 0 means the distributions are identical. The farther the TVD is from 0, the more different the two distributions are.

Summary

In this chapter we apply our sampling and simulation skills to two-sample hypothesis testing, which is used to compare two groups of data. We also see the importance of random assignment to groups, which allows us to determine whether a treatment may have caused a difference in the outcome.

11: Estimation

In the previous two chapters we used hypothesis testing to draw conclusions about a parameter of a dataset. In this chapter we learn how to use the dataset to estimate a parameter for the entire population.

An example of *estimation* is when we survey a sample of voters on their potential voting choice for a candidate, and then we use the sample's data to estimate the percentage of voters nationwide that will vote for that candidate. During the estimation we need to consider that the voters in the sample are randomly chosen. Different random samples can produce slightly different results simply due to chance. We therefore need to account for this sampling variability when making an estimate.

We start by getting an understanding of percentiles: what it is and what does it mean when a value is in the 75th percentile, for example. Then we continue with the concept of bootstrap through resampling to make an estimation.

Percentile

A dataset that contains numerical data can be sorted in increasing or decreasing order. After sorting, each number in the dataset has a particular position or a rank, and a *percentile* is the value at a particular rank. A percentile, as the name implies, is a value that is between 0 and 100.

A simple definition of a percentile:

The 80th percentile is the smallest value in the dataset that is greater than or equal to 80% of the data.

If we have a sequence of n numbers, to find the number that's at the p th percentile:

- Sort the numbers in increasing order.
- $k = (p / 100) * n$
- If k is an integer, take the k th element of the sorted numbers.
- If k is not an integer, round it up to the next integer, and take that element of the sorted numbers.

Example:

- A sorted sequence is 5, 8, 20, 24, 30
- The 80th percentile is:
 $k = (80 / 100) * 5 = 4$, the 4th element is 24
- The 35th percentile is:
 $k = (35/100) * 5 = 1.75$, rounding 1.75 to 2, the 2nd element is 8

Now that we have a general idea of percentile, we apply it to the larger dataset, which is the **delivery** dataset for a food delivery company that we used in an earlier chapter.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
delivery = pd.read_csv(url)
print("First 5 rows:")
delivery.head()
```

Output?

First 5 rows:

	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experience
0	522	7.93	Windy	Low	Afternoon	Scooter	12	1.0
1	738	16.42	Clear	Medium	Evening	Bike	20	2.0
2	741	9.52	Foggy	Low	Night	Scooter	28	1.0
3	661	7.44	Rainy	Medium	Afternoon	Scooter	5	1.0
4	412	19.03	Clear	Low	Morning	Bike	16	5.0

Since we're interested in the delivery distance only, we create a variable for the distance and plot the distance distribution.

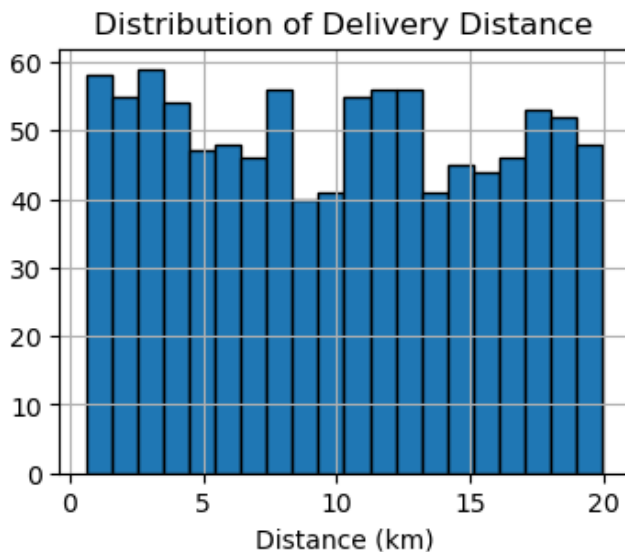
```
distance = delivery['Distance_km']
distance.head()
```

Output?

```
0    7.93
1   16.42
2    9.52
3    7.44
4   19.03
Name: Distance_km, dtype: float64
```

```
plt.figure(figsize=(4,3))
plt.hist(distance, bins=20, edgecolor='black')
plt.xlabel('Distance (km)')
plt.title('Distribution of Delivery Distance')
plt.grid()
plt.show()
```

Output?



We see that the distances range from 1 km to 20 km, and there's no clear mode, which means no delivery distance that occurs much more frequently than the others.

To find the 85th percentile, we can use the same calculation steps as above, or we can use the `percentile` function of numpy:

```
p = np.percentile(data_sequence, percentile)

print("85th percentile:", np.percentile(distance, 85))
```

Output?

85th percentile: 17.1375

The delivery distance of 17.14km is at or above 85% of the delivery distances.

Quartile

The range of percentiles, which is between 0 and 100, can be divided into four sections where each section is called a *quartile*.

- The *first quartile* is the 25th percentile
- The *second quartile* is the 50th percentile. The 50th percentile is also called the *median*, the midpoint of the sorted data.
- The *third quartile* is the 75th percentile.

In a distribution, values above the 25th percentile and below the 75th percentile is in the *middle 50%* interval.

For the delivery distances, we calculate:

```
print("First quartile:", np.percentile(distance, 25))
print("Second quartile:", np.percentile(distance, 50))
print("Third quartile:", np.percentile(distance, 75))
print("Middle 50% is between:", np.percentile(distance, 25), "and", np.percentile(distance
```

Output?

```
First quartile: 5.105
Second quartile: 10.19
Third quartile: 15.0175
Middle 50% is between: 5.105 and 15.0175
```

Gather Data

We use data from the [US Bureau of Labor and Statistics \(BLS\)](#) which contains the 2024 unemployment rate for each of the 50 states. The data are in the file `state_unemployment.csv`.

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
us = pd.read_csv(url)
display(us.head(10))
print("Number of rows:", us.shape[0])
```

Output?

	state	unemployment_rate
0	Alabama	3.1
1	Alaska	4.6
2	Arizona	3.6
3	Arkansas	3.5
4	California	5.3
5	Colorado	4.3
6	Connecticut	3.2
7	Delaware	3.7
8	Florida	3.4
9	Georgia	3.5

Number of rows: 50

```
# print all the states to check  
us.state
```

Output?

```
0      Alabama  
1      Alaska  
2      Arizona  
3      Arkansas  
4      California  
5      Colorado  
6      Connecticut  
7      Delaware  
8      Florida  
9      Georgia  
10     Hawaii  
11     Idaho  
12     Illinois  
13     Indiana  
14     Iowa  
15     Kansas  
16     Kentucky  
17     Louisiana  
18     Maine  
19     Maryland  
20     Massachusetts  
21     Michigan  
22     Minnesota  
23     Mississippi  
24     Missouri  
25     Montana  
26     Nebraska  
27     Nevada  
28     New Hampshire  
29     New Jersey  
30     New Mexico  
31     New York  
32     North Carolina  
33     North Dakota  
34     Ohio  
35     Oklahoma  
36     Oregon  
37     Pennsylvania  
38     Rhode Island  
39     South Carolina  
40     South Dakota  
41     Tennessee  
42     Texas  
43     Utah  
44     Vermont  
45     Virginia  
46     Washington  
47     West Virginia  
48     Wisconsin  
49     Wyoming  
Name: state, dtype: object
```

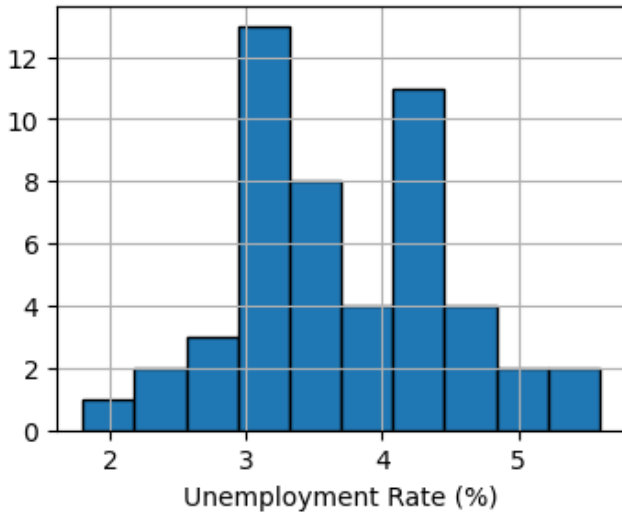
We call the DataFrame `us` because it contains the unemployment rates of all 50 US states. Since our data includes all 50 states, we have data for the entire *population*.

We look at the distribution of unemployment rates and find the mean unemployment rate of the population.

```
plt.figure(figsize=(4, 3))
plt.hist(us['unemployment_rate'], edgecolor='black')
plt.xlabel('Unemployment Rate (%)')
plt.title('Distribution of Unemployment Rates by State')
plt.grid()
plt.show()
```

Output?

Distribution of Unemployment Rates by State



```
mean_us_unemployment = us["unemployment_rate"].mean()
print("Mean unemployment:", round(mean_us_unemployment,2))
```

Output?

Mean unemployment: 3.68

Our goal is to see if a sample of 10 states can be used to estimate the mean unemployment rate of the entire population.

Take a Sample

Suppose we don't have the data for all 50 states, and instead we only have the unemployment data for 10 states that had been randomly chosen out of the 50 states.

Here is our sample of 10 states and the mean unemployment rate of the sample.

```
# take a random sample of 10, and reset the index of these states
# so we don't use their old index in the 50 states

original_sample = us.sample(n=10).reset_index(drop=True)
original_sample
```

Output?

	state	unemployment_rate
0	Michigan	4.7
1	Washington	4.5
2	Massachusetts	4.0

	state	unemployment_rate
3	Rhode Island	4.3
4	Alabama	3.1
5	Oregon	4.2
6	Virginia	2.9
7	Montana	3.0
8	Kansas	3.6
9	Idaho	3.7

```
original_mean = original_sample["unemployment_rate"].mean()
print("Mean unemployment rate for sample:", round(original_mean, 2))
```

Output?

Mean unemployment rate for sample: 3.8

We see that the mean unemployment rate of the sample is different from the mean unemployment rate of the population (3.68%). This shows that a statistic calculated from a random sample may not exactly match the population statistic.

How confident can we be that our sample gives us a reasonable estimate of the population?

Bootstrap the Sample

We've discovered that the Law of Large Numbers says that as the number of observations increases, the average of those observations tends to get closer to the population average. But we only have one sample of 10 states, how do we get the large number of observations to use to estimate the population average?

The answer is to *bootstrap* the sample. The term *bootstrap* comes from the saying that someone *pull themselves up from their bootstrap*, which is when the person uses what they have to reach their goal, without external help.

We will resample our original sample of 10 states to get many new samples to do our estimation. The key point in resampling is that we *sample with replacement* to create new samples that are different from the original.

Sample with Replacement

If we resample the original sample without replacement, we'll end up with the same sample

```
# The original sample
display(original_sample)
```

Output?

	state	unemployment_rate
0	Michigan	4.7
1	Washington	4.5
2	Massachusetts	4.0
3	Rhode Island	4.3
4	Alabama	3.1
5	Oregon	4.2

	state	unemployment_rate
6	Virginia	2.9
7	Montana	3.0
8	Kansas	3.6
9	Idaho	3.7

```
# resample without replacement
```

```
new_sample = original_sample.sample(n=10, replace=False)
new_sample
```

Output?

	state	unemployment_rate
1	Washington	4.5
7	Montana	3.0
2	Massachusetts	4.0
3	Rhode Island	4.3
5	Oregon	4.2
4	Alabama	3.1
0	Michigan	4.7
6	Virginia	2.9
8	Kansas	3.6
9	Idaho	3.7

We see that the **resample** data are the same 10 states, just in a different order. This is the reshuffling that we used for a permutation in hypothesis testing in the previous chapter.

Now we sample with replacement. As we recall from Chapter 8, this means when a state is randomly selected and copied to the new resample, it remains available to be selected again. As a result, a state can appear more than once in the sample. This also means that some of the 10 original states may not appear in the resample at all.

```
# resample with replacement
```

```
new_sample = original_sample.sample(n=10, replace=True)
new_sample
```

Output?

	state	unemployment_rate
7	Montana	3.0
2	Massachusetts	4.0
0	Michigan	4.7
6	Virginia	2.9
4	Alabama	3.1
3	Rhode Island	4.3
4	Alabama	3.1

	state	unemployment_rate
4	Alabama	3.1
9	Idaho	3.7
7	Montana	3.0

If we run the Code cell above a few times, we observe that the resamples are slightly different each time.

The Bootstrap

The bootstrap consists of two steps:

- Build one bootstrap sample by resampling the original sample, and finding the statistic of this bootstrap sample.
- Repeat the process to create many bootstrap samples and calculate the statistic to each one. These statistics form the bootstrap distribution.

The bootstrap distribution can then be used to estimate the population statistic.

One Bootstrap Sample

We write a function named `resample` that will resample the original sample and calculate and return the mean unemployment rate of the new sample.

```
def resample():
    new_sample = original_sample.sample(n=10, replace=True)
    return np.mean(new_sample["unemployment_rate"])

# show the mean unemployment rate for the new sample
print("Mean unemployment rate of bootstrap sample:", round(resample(), 2))
```

Output?

Mean unemployment rate of bootstrap sample: 3.68

When we read in data above, we calculated the `mean_us_unemployment` or actual population mean of 3.68%.

We see that this population mean is pretty close to the bootstrap sample mean shown in the output above. However, the bootstrap sample mean will vary each time we create a new bootstrap sample, because the resampling creates a different new sample each time. If we run the Code cell above several times, we can see that the bootstrap sample mean changes with every resampling.

Therefore we want to repeat the process to create many bootstrap samples and calculate the mean of each one. The variation in these means will help us with estimating the population mean.

Repeat with Many Samples

We run the `resample` function 1000 times. Each time we create a bootstrap sample and find its mean, then we find and store the bootstrap mean in an array. Last we plot the array, which is the bootstrap distribution.

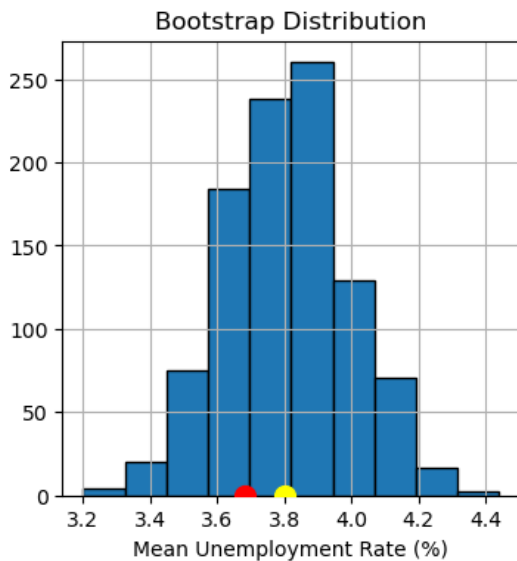
```
# create a list to store the bootstrap means
means = []
# run resample 1000 times and store the mean unemployment rate of each sample
for i in range(1000):
    means.append(resample())
# convert the list to an array
mean_rate_array = np.array(means)
```

Output?

```
# show the distribution of the means

plt.figure(figsize=(4,4))
plt.hist(mean_rate_array, edgecolor='black')
plt.xlabel('Mean Unemployment Rate (%)')
plt.title('Bootstrap Distribution')
# add a red circle for the population mean
plt.scatter(mean_us_unemployment, 0, color='red', s=100, zorder=3)
# add a yellow circle for the bootstrap mean
plt.scatter(original_mean, 0, color='yellow', s=100, zorder=3)
plt.grid()
plt.show()
```

Output?



We observe that the bootstrap distribution is centered around the yellow circle, which represents the original sample mean, as expected. And we also notice that the red circle, the population mean, falls within the main range of the bootstrap distribution.

Next we discuss this main range in more detail.

Confidence Interval

The bootstrap distribution shows a range of possible means from our resampling. Most of the bootstrap means are clustered near the center, while fewer are at the two ends of the distribution.

Rather than using the entire distribution to estimate the population mean, we can focus on the values in the middle of the distribution. We want a range of values that is wide enough to give us confidence that our estimate will be within the range, but not so wide that the estimation is not useful. A commonly used range is the middle 95% of the bootstrap distribution, called a *95% confidence interval*. 95% is common because it includes most of the bootstrap distribution while still leaving out the more extreme values at both ends.

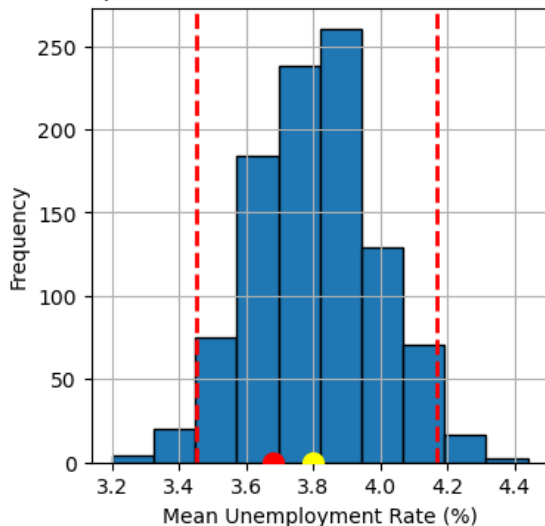
To find the middle 95%, we leave out 2.5% at each end of the bootstrap distribution. Therefore, the 95% confidence interval extends from the 2.5th percentile to the 97.5th percentile. In the histogram of the bootstrap distribution below, the 95% confidence interval is between the two dashed lines.

```
plt.figure(figsize=(4,4))
# plot the bootstrap means
plt.hist(mean_rate_array, edgecolor='black')
# find the 2.5th and 97.5th percentiles
lower_percentile = np.percentile(mean_rate_array, 2.5)
```

```
upper_percentile = np.percentile(mean_rate_array, 97.5)
# plot the 2.5th and 97.5th percentile lines
plt.axvline(lower_percentile, color='red', linestyle='--', linewidth=2)
plt.axvline(upper_percentile, color='red', linestyle='--', linewidth=2)
# plot the population mean
plt.scatter(mean_us_unemployment, 0, color='red', s=100, zorder=3)
# plot the sample mean
plt.scatter(original_mean, 0, color='yellow', s=100, zorder=3)
# show labels
plt.xlabel('Mean Unemployment Rate (%)')
plt.ylabel('Frequency')
plt.title('Bootstrap Distribution with 95% Confidence Interval')
plt.grid()
plt.show()
```

Output?

Bootstrap Distribution with 95% Confidence Interval



The plot shows the 95% confidence interval between the red lines. The red circle is the population mean, and it is inside the 95% confidence interval.

```
print("The 95% confidence interval is between", round(lower_percentile, 2), "and", round(upper_percentile, 2))
print("The red circle or population mean is", round(mean_us_unemployment, 2))
print("The yellow circle or original sample mean is", round(original_mean, 2))
```

Output?

The 95% confidence interval is between 3.45 and 4.17
The red circle or population mean is 3.68
The yellow circle or original sample mean is 3.8

Our conclusion is:

```
print("The estimate mean unemployment rate for the 50 US states is", round(original_mean, 2))
print("The actual mean unemployment rate is estimated to be between", round(lower_percentile, 2), "and", round(upper_percentile, 2))
```

Output?

The estimate mean unemployment rate for the 50 US states is 3.8
The actual mean unemployment rate is estimated to be between 3.45 and 4.17

Studies of the bootstrap method have shown that, if we repeatedly took random samples from the population and constructed a 95% confidence interval for each sample, then about 95 out of every 100 confidence intervals would contain the actual population

mean.

Estimation Example

We now use the bootstrap to work with a sample where we don't know the actual population parameter. We'll use the sample to estimate the population parameter and construct a 95% confidence interval.

Gather Data

We will use the dataset from the CDC survey of people's health habits and conditions.

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
health = pd.read_csv(url)
display(health.head())
print("Number of rows:", health.shape[0])
```

Output?

	genhlth	exerany	hlthplan	smoke100	height	weight	wt desire	age	gender
0	good	0	1	0	70	175	175	77	m
1	good	0	1	1	64	125	115	33	f
2	good	1	1	1	60	105	105	49	f
3	good	1	1	0	66	132	124	42	f
4	very good	0	1	0	61	150	130	55	f

Number of rows: 20000

We see that the sample is 20000 people. We want to estimate the age of the population, so we first find the range of ages in the sample.

```
print("Youngest age:", health.age.min())
print("Oldest age:", health.age.max())
```

Output?

Youngest age: 18
Oldest age: 99

The Parameter

The age range shows that we have a sample of adults. We would like to estimate the mean age of all adults in the population.

The Bootstrap

First we find the mean of the original sample.

```
original_mean = health.age.mean()
print("Original sample mean:", round(original_mean, 2))
```

Output?

Original sample mean: 45.07

As in the previous example, we write a function to resample. Then we repeat the resample 1000 times to create the bootstrap distribution of sample means.

```
def resample():
    new_sample = health.age.sample(n=len(health), replace=True)
    return np.mean(new_sample)

means = []
for i in range(1000):
    means.append(resample())
mean_array = np.array(means)
```

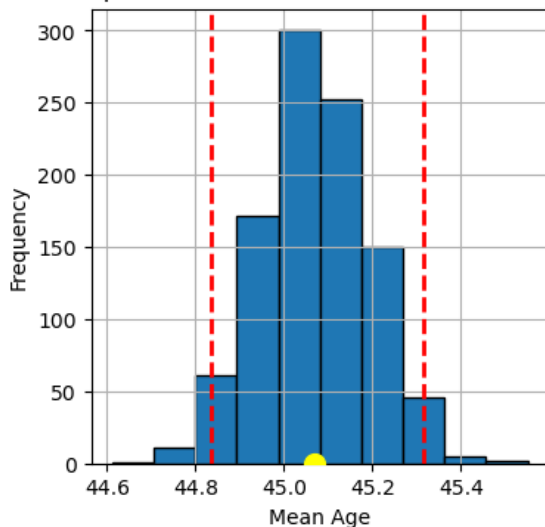
Output?

We plot the bootstrap distribution, the 95% confidence interval, and our estimate.

```
plt.figure(figsize=(4,4))
# plot the bootstrap means
plt.hist(mean_array, edgecolor='black')
# find the 2.5th and 97.5th percentiles
lower_percentile = np.percentile(mean_array, 2.5)
upper_percentile = np.percentile(mean_array, 97.5)
# plot the 2.5th and 97.5th percentile lines
plt.axvline(lower_percentile, color='red', linestyle='--', linewidth=2)
plt.axvline(upper_percentile, color='red', linestyle='--', linewidth=2)
# plot the sample mean
plt.scatter(original_mean, 0, color='yellow', s=100, zorder=3)
# show labels
plt.xlabel('Mean Age')
plt.ylabel('Frequency')
plt.title('Bootstrap Distribution with 95% Confidence Interval')
plt.grid()
plt.show()
```

Output?

Bootstrap Distribution with 95% Confidence Interval



```
print("The 95% confidence interval is between", round(lower_percentile, 2), "and", round(upper_percentile, 2))
print("The yellow circle or original sample mean is", round(original_mean, 2))
```

Output?

The 95% confidence interval is between 44.84 and 45.31
 The yellow circle or original sample mean is 45.07

Our conclusion:

```
print("The estimate mean age for the population is", round(original_mean, 2))
print("The actual mean age is estimated to be between", round(lower_percentile, 2), "and",
      Output?
```

```
The estimate mean age for the population is 45.07
The actual mean age is estimated to be between 44.84 and 45.31
```

Relationship between Confidence Interval and Hypothesis Testing

In the previous chapter we learned that hypothesis testing determines whether there is enough evidence to reject a hypothesis about a population parameter. A confidence interval provides another way to make the same decision.

For the estimation of the mean age example, suppose the null hypothesis is: The population mean age is 40 years, and the alternative hypothesis is: The population mean age is not 40 years.

Our 95% confidence interval for the population mean age is 44.83 to 45.31 years. Because the null hypothesis mean age of 40 years is outside this interval, we reject the null hypothesis. This is the same conclusion we would reach from a hypothesis test with a p-value less than 0.05. In general, if the null hypothesis value lies outside a 95% confidence interval, the corresponding hypothesis test will have a p-value less than 0.05.

Guidelines for Using the Bootstrap

We have seen that the bootstrap is a simple way to estimate a population parameter and construct a confidence interval. To get reliable results, there are a few points to keep in mind:

1. The sample should be representative of the population. Since the bootstrap repeatedly resamples from the original sample, any bias in the original sample will also appear in the bootstrap samples.
2. The sample should be reasonably large. A larger sample will likely produce more reliable bootstrap estimates because it represents the population better. Note that in the estimation of the mean unemployment of the 50 states, the 95% confidence interval is larger than the 95% confidence interval of the estimation of the mean age. This is because the sample size of the second estimation is large.
3. Use many bootstrap resamples. Repeating the resampling 1000 times is sufficient for simple applications. Typically 10,000 resamples are used in general applications.
4. The bootstrap works best for common statistics such as the mean, median, and proportion. The parameter that we want to estimate must not be at the edge of the population distribution such as the max, min, or very high or very low percentiles.

Summary

In this chapter, we learn how to estimate an unknown population parameter with a sample of the population. We repeatedly resample from the original sample to get the bootstrap distribution, then we construct a 95% confidence interval to estimate a range for the population parameter. We also learn when the bootstrap works well and how a 95% confidence interval is connected to hypothesis testing.

12: The Mean

In the previous chapter we saw how a sample can be used to estimate a population parameter. In this chapter we will learn how to describe data using measurements of center and spread, and how concepts such as the normal distribution and the Central Limit Theorem help us understand why estimates from different samples can differ.

Properties of the Mean

Calculation of the Mean

The *mean* is also commonly called the average. It is found by summing all the data values in the dataset and dividing by the total number of data values.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

array = np.array([12, 3, 23, 4, 15])
print(sum(array)/len(array))
print(np.mean(array))
```

Output?

```
11.4
11.4
```

When Data is 1 and 0, or True and False

When the data sequence contains only 1's and 0's, then:

- The sum is the number of 1's. Since adding 0 does not change the total, each 1 increases the sum by one. The final sum is simply the count of the 1's.
- The mean is the proportion of the 1's. The mean is the sum divided by the total number of values. Since the sum is the number of 1's, the mean equals the fraction or proportion of values that are 1.

```
array = np.array([1,0,1,1,1,0])
print("Count of 1's:", sum(array))
print("Proportion of 1's:", sum(array)/len(array))
print("The mean:", np.mean(array))
```

Output?

```
Count of 1's: 4
Proportion of 1's: 0.6666666666666666
The mean: 0.6666666666666666
```

For computers, True means 1 and False means 0. Therefore, when the data sequence contains True and False, we can calculate the count of True values and proportion of True values the same way.

```
array = np.array([True, False, True, True, True, False])
print("Count of True:", sum(array))
print("Proportion of True:", sum(array)/len(array))
print("The mean:", np.mean(array))
```

Output?

```
Count of True: 4
```

Proportion of True: 0.6666666666666666
The mean: 0.6666666666666666

The Mean and the Distribution

The mean can be calculated by multiplying each unique data value by its proportion in the dataset, and then adding the results.

In an array with four values: 1, 2, 2, 6, the unique data values are 1, 2, and 6. Their proportions in the dataset are:

- 1 appears once, so its proportion is 1/4.
- 2 appears twice, so its proportion is 2/4.
- 6 appears once, so its proportion is 1/4.

```
array1 = np.array([1,2,2,6])
print("The mean:", 1*1/4 + 2*2/4 + 6*1/4)
print("The mean:", np.mean(array1))
```

Output?

The mean: 2.75
The mean: 2.75

In the following `array2`, the unique data values are also 1, 2, 6, and each value's proportion is the same as in `array1`:

```
array2 = np.array([1,1,2,2,2,2,6,6])
print("The mean:", 1*2/8 + 2*4/8 + 6*2/8)
print("The mean:", np.mean(array2))
```

Output?

The mean: 2.75
The mean: 2.75

The above examples show that *if two arrays have the same distribution* (same proportion for each unique data value), *then they have the same mean*.

We recall from Chapter 5 that a *density histogram* scales the bars so that they represent proportions instead of counts. This makes it easy to compare the distributions of datasets even when they contain different numbers of data values.

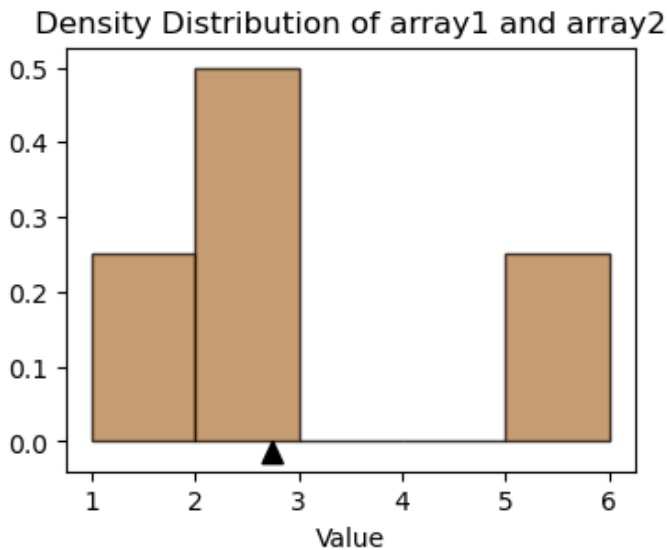
We plot the density distribution of `array1` and `array2`:

```
plt.figure(figsize=(4,3))
plt.hist(array2, bins=np.arange(1,7), density=True, edgecolor='black', alpha=0.5)
plt.hist(array1, bins=np.arange(1,7), density=True, edgecolor='black', alpha=0.5)
plt.title("Density Distribution of array1 and array2")
plt.xlabel("Value")
```

```
# You don't need to write the following code.
# The code draws a triangle at the mean.
triangle_x = [2.75 - 0.1, 2.75 + 0.1, 2.75]
triangle_y = [-0.03, -0.03, 0]
plt.fill(triangle_x, triangle_y, color='black', zorder=3)
plt.ylim(bottom=-0.04)
```

```
plt.show()
```

Output?



It looks like only one distribution was plotted, but actually both `array1` and `array2` distributions were plotted. Since they're the same distribution, the two histograms overlap each other perfectly. You can turn one of the `plt.hist()` lines of code into a comment so it doesn't run, and you'll see that the plot above is made of a blue histogram and an orange histogram that overlap each other.

For the histogram above, imagine that the x-axis is a straight board or a plank, and the three bars rest on top of the plank. As shown, the plank is balanced on a fulcrum, which is the black triangle.

- If the fulcrum is in the middle of the plank, which is at 3.5, the plank will tilt left because there's more "weight" on the left side.
- If the fulcrum is at 1.5, then the plank will tilt right.
- If the fulcrum is at 2.75 as shown, which is where the mean is, then the plank will be balanced.

The mean is the balance point of the histogram.

The Mean and the Median

Recall that the median is the midpoint of the sorted data values. *If a distribution is symmetrical, then the mean and the median are the same.*

Below we create a symmetrical distribution, where the left and right side are balanced.

```
array = np.array([1,2,2,3,3,3,4,4,5])
print("The mean:", np.mean(array))
print("The median:", np.median(array))
```

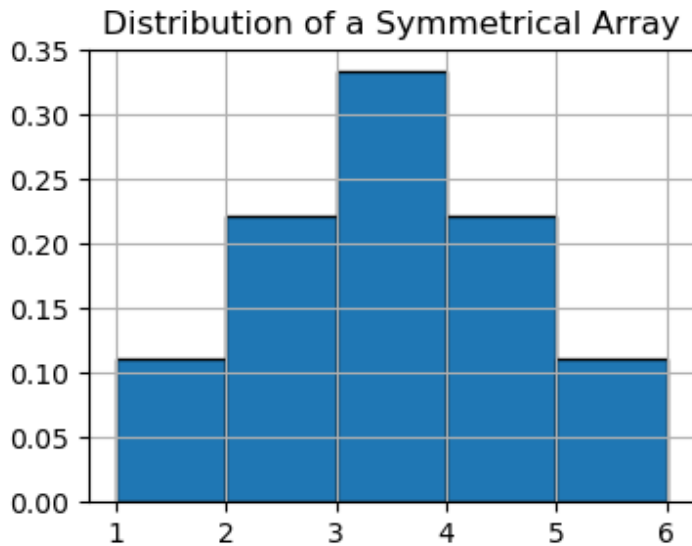
Output?

```
The mean: 3.0
The median: 3.0
```

```
plt.figure(figsize=(4,3))
plt.hist(array, bins=np.arange(1,7), density=True, edgecolor='black')
plt.title("Distribution of a Symmetrical Array")
plt.grid()
plt.plot()
```

Output?

[]



When the distribution is not symmetric, or is skewed, so that one side extends farther than the other side (also called having a tail), then the mean is pulled away from the median towards the tail.

Below we create an array with a right hand tail.

```
array = np.array([1,2,2,3,3,3,4,4,4,4,5,5,6,6,7,8,12,15,16,20])
print("The mean:", np.mean(array))
print("The median:", np.median(array))
```

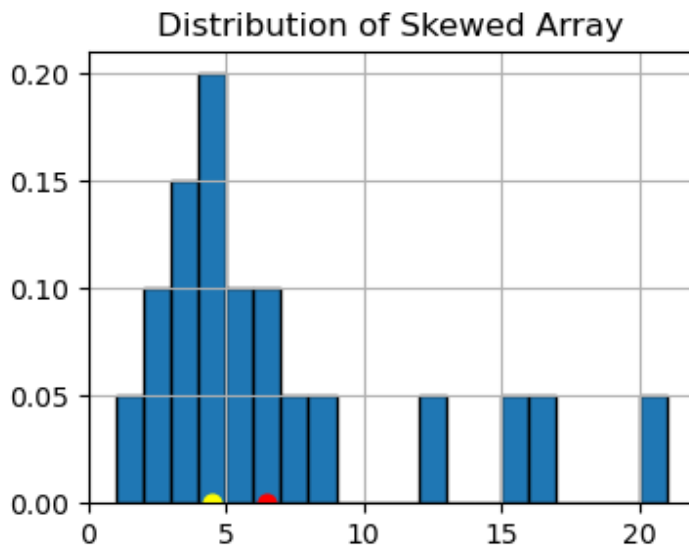
Output?

The mean: 6.5
The median: 4.5

```
plt.figure(figsize=(4,3))
plt.hist(array, bins=np.arange(1,22), density=True, edgecolor='black')
plt.title("Distribution of Skewed Array")
plt.scatter(np.mean(array), 0, color='red', s=40, zorder=2)
plt.scatter(np.median(array), 0, color='yellow', s=40, zorder=2)
plt.grid()
plt.plot()
```

Output?

[]



The histogram is skewed right and has a right hand tail, therefore the red mean is to the right of the yellow median.

The mean is affected by the tail. As values in the right tail become larger, they pull the mean to the right, making the mean larger. In contrast, the median is determined by the middle value, so it is not as affected by a few extreme values in the tail.

For skewed distributions with a long tail, we often use the median instead of the mean to describe the center of the data. In the distribution above, most of the data values are between 2 and 6, so the median of 4.5 is a better description of the typical value than the mean of 6.5, which is pulled to the right by the long tail.

Measures of Spread

We saw earlier that the mean is the balance point of the histogram, and that data spread out from the mean on both the left and right sides of the mean. In this section we learn to measure the spread of the data.

We begin by finding the deviation of each data value from the mean. A deviation is simply the distance between a data value and the mean. The following steps show how to calculate the deviations.

The Deviation

We find the difference between the mean and each data point. This difference is called the *deviation* from the mean. The deviation is positive if the data value is greater than the mean, and negative if the data is less than the mean.

```
# create an array of 4 numbers and find the mean
```

```
array = np.array([12, 5, 9, 3])
# find the mean
the_mean = np.mean(array)
print("The mean:", the_mean)
```

Output?

The mean: 7.25

```
# find the deviations
```

```
# subtracts the mean from each value in array,
# creating an array of differences
deviations = array - np.mean(array)
```

```
print("The deviations:", deviations)
```

Output?

The deviations: [4.75 -2.25 1.75 -4.25]

Note that if we add all the deviations, the result is always 0. This makes sense because the mean is the balance point for the histogram. The sum of distances on the left of the mean should be equal to the sum of distances on the right of the mean.

```
print(sum(deviations))
```

Output?

0.0

The Variance

Our goal is to measure the spread of the data, or how far on average the data values are from the mean. To calculate the average, we need to add all the distances of data from the mean and divide by the number of data values. However, we see from above that adding all the deviations results in 0, so this approach isn't useful to measure the spread.

Instead, we square the deviations and then find the average of the squared deviations. We square the deviations to remove the negative signs so the sum would not be 0.

The *variance* is the mean of the squared deviations.

```
# square each deviation and find the mean of the squares
variance = np.mean(deviations**2)
```

```
print("The variance:", variance)
```

Output?

The variance: 12.1875

We also note that squaring the deviations gives more weight to the data that are far from the mean. For example, a dataset of deviations [2, -2, 2, -2] will have a variance of $(4 + 4 + 4 + 4)/4 = 4$. And a dataset of deviations [-2, -2, -2, 6] will have a variance of $(4 + 4 + 4 + 36)/4 = 12$. One deviation increase from 2 to 6 causes a much larger increase in variance.

As a result, a dataset with some values far from the center has a larger variance than a dataset where the values are all clustered close to the mean.

The Standard Deviation

Since we need to square the deviations in order to find the variance, we now take the square root to "undo" the squaring. The result is the *standard deviation*.

```
sd = np.sqrt(variance)
print("The standard deviation:", round(sd, 2))
```

Output?

The standard deviation: 3.49

Lucky for us, numpy also has the `std()` function to calculate the standard deviation of an array, so we don't have to go through all three steps above when we want to find the standard deviation.

```
print("The standard deviation:", round(np.std(array), 2))
```

Output?

The standard deviation: 3.49

The standard deviation of 3.49 means that the data values are typically about 3.49 units away from the mean. The larger the standard deviation, the more spread out the data are.

In the example above the data values are from 3 to 12. Since the standard deviation is 3.49, we can say that the data are fairly spread out.

Typically a standard deviation is useful when comparing the spread of two datasets. A dataset with a standard deviation of 1.2 is less spread out than a dataset with a standard deviation of 3.9.

Standard Deviation Example

We use the CDC survey of people's health habits and status. We will look at the standard deviation, or SD, of the **age** column

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
health = pd.read_csv(url)
health.head()
```

Output?

	genhlth	exerany	hlthplan	smoke100	height	weight	wt desire	age	gender
0	good	0	1	0	70	175	175	77	m
1	good	0	1	1	64	125	115	33	f
2	good	1	1	1	60	105	105	49	f
3	good	1	1	0	66	132	124	42	f
4	very good	0	1	0	61	150	130	55	f

We now find the mean and standard deviation (SD) of the **age**.

```
mean_age = np.mean(health["age"])
sd_age = np.std(health["age"])
print("Mean Age:", round(mean_age,2))
print("Standard deviation:", round(sd_age,2))
```

Output?

Mean Age: 45.07
Standard deviation: 17.19

Then we look at the oldest age and compare it against the SD.

```
# find the oldest age
oldest = health["age"].max()
print("Oldest age:", oldest)
print("Number of standard deviations:", round((oldest - mean_age)/sd_age,2))
```

Output?

Oldest age: 99
Number of standard deviations: 3.14

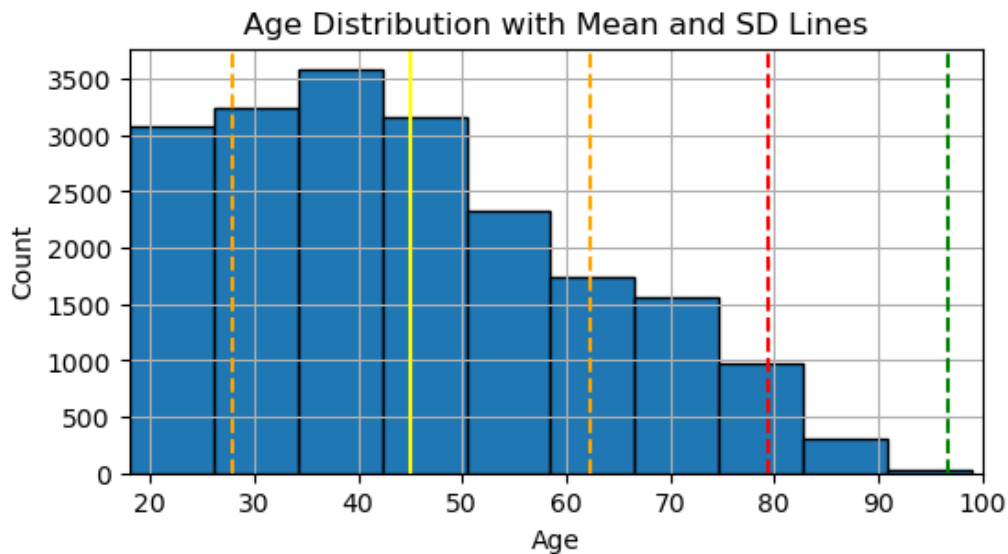
The oldest age of 99 is 3 standard deviations from the mean age.

We look at the age distribution and the 1 SD, +2 SD, and +3 SD lines.

```
plt.figure(figsize=(6,3))
plt.hist(health["age"], edgecolor='black')
```

```
plt.axvline(mean_age, color='yellow', linestyle='--')
plt.axvline(mean_age + sd_age, color='orange', linestyle='--')
plt.axvline(mean_age - sd_age, color='orange', linestyle='--')
plt.axvline(mean_age + 2*sd_age, color='red', linestyle='--')
plt.axvline(mean_age + 3*sd_age, color='green', linestyle='--')
plt.xlim(18, 100)
plt.title("Age Distribution with Mean and SD Lines")
plt.xlabel("Age")
plt.ylabel("Count")
plt.grid()
plt.show()
```

Output?



We make the following observations from the histogram:

- The age range is from 18 to 99.
- The mean, or balance point of the distribution, is the yellow line.
- The ± 1 SD lines are the orange lines.
- The $+2$ SD line is red, the $+3$ SD line is green. We don't show the negative SD lines since they're outside of the range of data.
- The oldest age of 99 is just outside of the $+3$ SD line.
- Most ages appear to be within 2 standard deviations of the mean.

Chebyshev's Theorem

From the histogram, we observe that most of the ages are within 2 standard deviations of the mean. This is not a coincidence. Chebyshev's Theorem tells us that, for any dataset, most of the data values lie within a few standard deviations of the mean, regardless of the shape of the distribution.

More specifically, Chebyshev's Theorem states that:

- At least 75% of the data lie within 2 standard deviations of the mean.
- At least 89% of the data lie within 3 standard deviations of the mean.

Since the oldest age of 99 is just beyond 3 standard deviations from the mean, it is an unusually high age compared with the rest of the dataset.

Standard Units or Z-Scores

So far, we have described how far a data value is from the mean by saying it is 1 standard deviation, 2 standard deviations, or 3 standard deviations away. But there's a simpler way to describe the number of standard deviations as a single number. This number is the *standard unit* or *z-score*.

A z-score tells us how many standard deviations a data value is above or below the mean. Positive z-scores represent values above the mean, while negative z-scores represent values below the mean.

Examples of z-score values:

- A value at the mean has a z-score of 0.
- A value one standard deviation above the mean has a z-score of +1, or is 1 standard unit above the mean.
- A value two standard deviations below the mean has a z-score of -2, or is 2 standard units below the mean.
- The oldest age in our dataset is just over 3 standard deviations above the mean, so its z-score is just over +3 or 3 standard units above the mean.

To find the number of standard units or z-score for a dataset:

```
def standard_units(array):
    return (array - np.mean(array))/np.std(array)

# in the function above, the mean is subtracted from each data value,
# then the differences are divided by the SD to get the number of SDs,
# and an array of number of SDs (z-score) is returned
```

Output?

Using the ages in the survey dataset above, we find the standard units for age:

```
age_s_units = standard_units(health["age"])
age_s_units.head()
```

Output?

```
0    1.857333
1   -0.701958
2    0.228693
3   -0.178467
4    0.577687
Name: age, dtype: float64
```

Then we add the `age_s_units` as a new column in the `health` DataFrame.

```
health["age_su"] = age_s_units
health.head()
```

Output?

	genhlth	exerany	hlthplan	smoke100	height	weight	wtdesired	age	gender	age_su
0	good	0	1	0	70	175	175	77	m	1.857333
1	good	0	1	1	64	125	115	33	f	-0.701958
2	good	1	1	1	60	105	105	49	f	0.228693
3	good	1	1	0	66	132	124	42	f	-0.178467
4	very good	0	1	0	61	150	130	55	f	0.577687

Next we sort in descending order the DataFrame by the `age_su` column so we can see the longest delay times.

```
print("10 oldest ages:")
health.sort_values("age_su", ascending=False).head(10)
```

Output?

10 oldest ages:

	genhlth	exerany	hlthplan	smoke100	height	weight	wt desire	age	gender	age_su
899	very good	0	1	0	70	150	150	99	f	3.136979
6709	excellent	1	1	1	62	120	115	99	f	3.136979
10349	good	1	1	1	60	130	130	97	f	3.020647
17050	good	1	1	1	64	128	128	96	f	2.962481
11911	very good	0	1	0	62	110	110	95	f	2.904316
16083	good	1	1	0	62	127	127	95	f	2.904316
7085	very good	1	1	0	66	125	130	94	m	2.846150
19502	fair	0	1	0	66	140	140	94	m	2.846150
2527	very good	1	1	0	61	118	118	94	f	2.846150
1443	poor	1	1	0	63	102	102	93	f	2.787984

We see that there are probably quite a few people whose age is more than 2 standard units.

Since Chebyshev's Theorem says that at least 75% of the data should be 2 standard units or less, we want to check if Chebyshev's bound holds true for this dataset.

We find the proportion of people whose age is 2 standard units or less, then we find the proportion of people whose age is 3 standard units or less.

```
proportion_2_su = len(health[np.abs(health["age_su"]) <= 2]) / len(health)
print("Proportion of ages at 2 standard units or less:", round(proportion_2_su, 4))

proportion_3_su = len(health[np.abs(health["age_su"]) <= 3]) / len(health)
print("Proportion of ages at 3 standard units or less:", round(proportion_3_su, 4))
```

Output?

Proportion of ages at 2 standard units or less: 0.9693
Proportion of ages at 3 standard units or less: 0.9999

Chebyshev's Theorem does hold for the age data. In fact the theorem gives a conservative lower bound.

- 97% of the ages are within 2 standard units, compared to Chebyshev's boundary of at least 75%.
- Almost 100% of the ages are within 3 standard units, compared to Chebyshev's boundary of at least 89%.
Looking at the first 10 rows of the sorted DataFrame, we see that only three out of 20,000 people (ages 99, 99, and 97) are above 3 standard units above the mean.

Because the age distribution is concentrated near the mean, a much larger percentage of the data falls within 2 and 3 standard units than Chebyshev's Theorem guarantees. In the next section, we will see that for data that are approximately normally distributed, these percentages can be predicted even more precisely.

The Normal Distribution and Standard Units

The normal distribution is also known as the Gaussian distribution or the bell curve. It has a symmetric bell-shaped curve, with a peak at the mean value and tapering off equally on both left and right sides. The normal distribution has a special relationship with the standard deviation.

We first create a normal distribution and fill in the areas under the normal curve that are within 1 standard deviation and within 2 standard deviations.

```
# You don't need to study this code.
# The code is to demo the standard deviations of the normal distribution.

from scipy.stats import norm

mean = 0
sd = 1

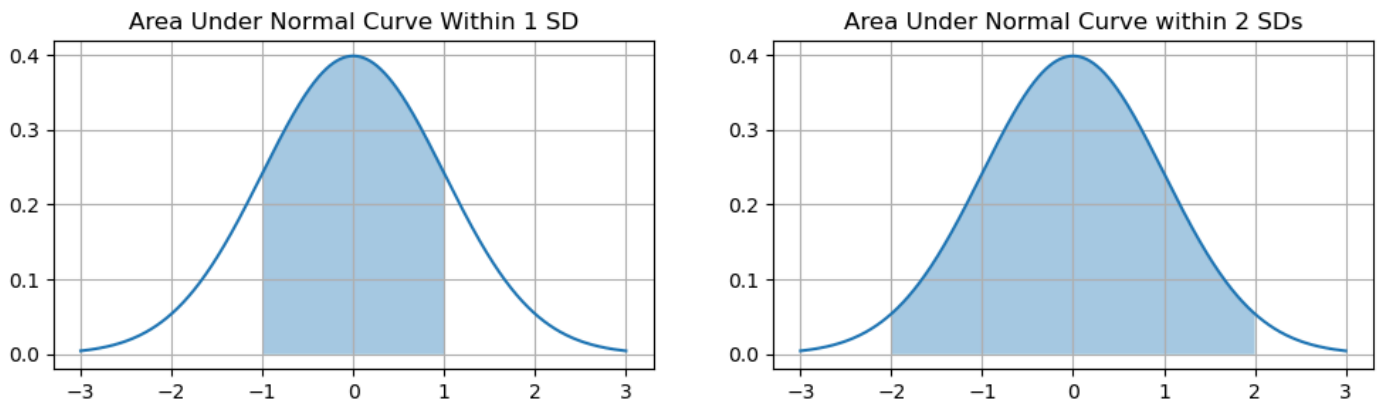
# create 1000 x values centered around 0
x = np.linspace(mean - 3*sd, mean + 3*sd, 1000)
# create y values that form the normal distribution
y = norm.pdf(x, mean, sd)

# create 2 side-by-side plots
fig, axes = plt.subplots(1, 2, figsize=(12, 3))

# plot normal distribution with filled area within 1 SD
axes[0].plot(x, y)
x_fill_1 = np.linspace(mean - sd, mean + sd, 1000)
y_fill_1 = norm.pdf(x_fill_1, mean, sd)
axes[0].fill_between(x_fill_1, y_fill_1, alpha=0.4)
axes[0].set_title('Area Under Normal Curve Within 1 SD')
axes[0].grid(True)

# plot normal distribution with filled area within 2 SD
axes[1].plot(x, y)
x_fill_2 = np.linspace(mean - 2*sd, mean + 2*sd, 1000)
y_fill_2 = norm.pdf(x_fill_2, mean, sd)
axes[1].fill_between(x_fill_2, y_fill_2, alpha=0.4)
axes[1].set_title('Area Under Normal Curve within 2 SDs')
axes[1].grid(True)
```

Output?



We can see that the area under the curve that's within 2 SDs covers a substantial part of the entire area under the curve.

To calculate the area under the curve, we use the `cdf` function of the `norm` module. The `cdf` or *Cumulative Distribution Function* returns the proportion of

$$\frac{\text{area under the curve up to an x value}}{\text{total area under the curve}}$$

To call the `cdf` function we use the format:

```
area_up_to_x = norm.cdf(x)
```

Using this function, we find the proportion of the area under the curve within 1 SD and within 2 SDs compared to the total area under the curve.

```
# Area within 1 SD is between -1 and 1
print("Proportion of area within 1 SD:", round(norm.cdf(1) - norm.cdf(-1), 3))

# Area within 2 SD is between -2 and 2
print("Proportion of area within 2 SDs:", round(norm.cdf(2) - norm.cdf(-2), 3))

# Area within 3 SD is between -3 and 3
print("Proportion of area within 3 SDs:", round(norm.cdf(3) - norm.cdf(-3), 3))

Output?
```

```
Proportion of area within 1 SD: 0.683
Proportion of area within 2 SDs: 0.954
Proportion of area within 3 SDs: 0.997
```

We note that for a normal distribution:

- About 95% of the area under the curve is within 2 standard deviations of the mean, and about 99.7% is within 3 standard deviations of the mean.
- These percentages are much higher than Chebyshev's lower bounds of 75% and 89%. This is because Chebyshev's Theorem applies to any distribution, regardless of its shape, while the percentages above apply only to the normal distribution.

Because the normal distribution has a specific bell shape, we can make much stronger statements about the proportion of data within 2 or 3 standard deviations than we can for an arbitrary distribution.

The Central Limit Theorem

The *central limit theorem* says that if we take a large number of random samples from a population and find the mean of each sample, then the distribution of the means will approach a normal distribution in shape, regardless of the original population's distribution.

Just like with Chebyshev's Theorem, we will see how the central limit theorem works by investigating the food delivery dataset.

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_delivery"
delivery = pd.read_csv(url)
delivery.head()

Output?
```

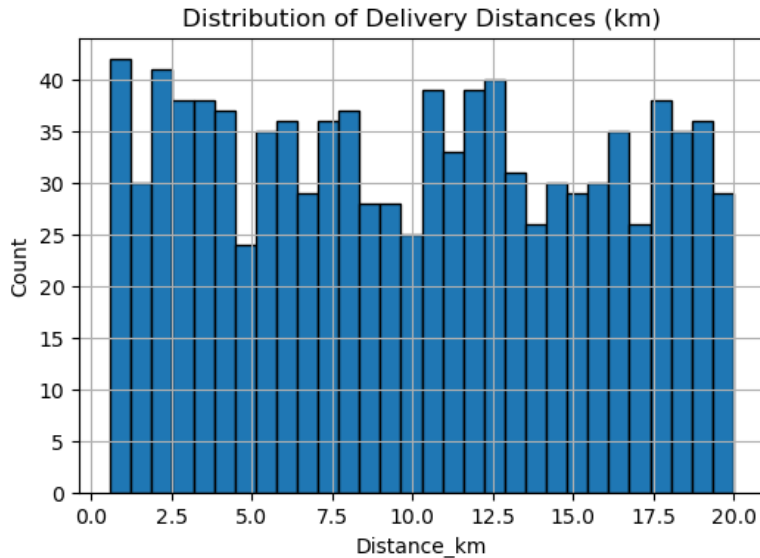
	Order_ID	Distance_km	Weather	Traffic_Level	Time_of_Day	Vehicle_Type	Preparation_Time_min	Courier_Experience
0	522	7.93	Windy	Low	Afternoon	Scooter	12	1.0
1	738	16.42	Clear	Medium	Evening	Bike	20	2.0
2	741	9.52	Foggy	Low	Night	Scooter	28	1.0
3	661	7.44	Rainy	Medium	Afternoon	Scooter	5	1.0
4	412	19.03	Clear	Low	Morning	Bike	16	5.0

We look at the delivery distance `Distance_km` distribution.

```
plt.figure(figsize=(6,4))
plt.hist(delivery["Distance_km"], bins=30, edgecolor='black')
plt.title("Distribution of Delivery Distances (km)")
plt.xlabel("Distance_km")
plt.ylabel("Count")
plt.grid()
```

```
plt.show()
```

Output?



The distribution of delivery distances is not a bell curve or normal distribution. The distances are spread fairly evenly from about 1 to 20 kilometers.

We will repeatedly take random samples from this dataset and calculate the mean distance of each sample. According to the Central Limit Theorem, the distribution of these sample means will become approximately a normal distribution.

We find the mean and standard deviation of the delivery distance.

```
delivery_mean = np.mean(delivery["Distance_km"])
delivery_standard_dev = np.std(delivery["Distance_km"])
print("The mean:", round(delivery_mean,2))
print("The SD:", round(delivery_standard_dev,2))
```

Output?

```
The mean: 10.06
The SD: 5.69
```

Next we take a large number of samples of the delivery times and plot the mean of the samples.

We write a function to select a sample of `num_of_data` values with replacement and find the mean of the sample.

```
def get_sample_mean(num_of_data):
    return np.mean(delivery["Distance_km"].sample(num_of_data, replace=True ))
```

Output?

We call the `get_sample_mean` 10,000 times to get 10,000 means of delivery time

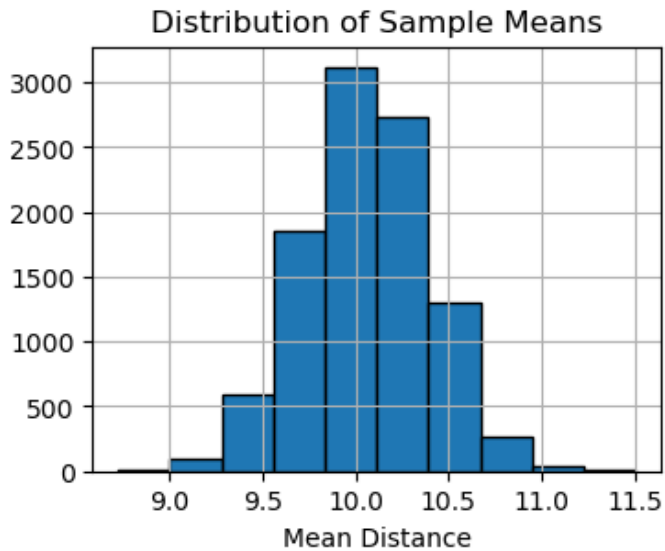
```
means = []
for i in range(10000):
    # get the mean of a sample of 300 delivery times
    # and append to the means list
    means.append(get_sample_mean(300))
# convert to a numpy array
mean_array = np.array(means)
```

Output?

We plot the distribution of the sample means.

```
plt.figure(figsize=(4,3))
plt.hist(mean_array, edgecolor='black')
plt.xlabel('Mean Distance')
plt.title('Distribution of Sample Means')
plt.grid()
plt.show()
```

Output?



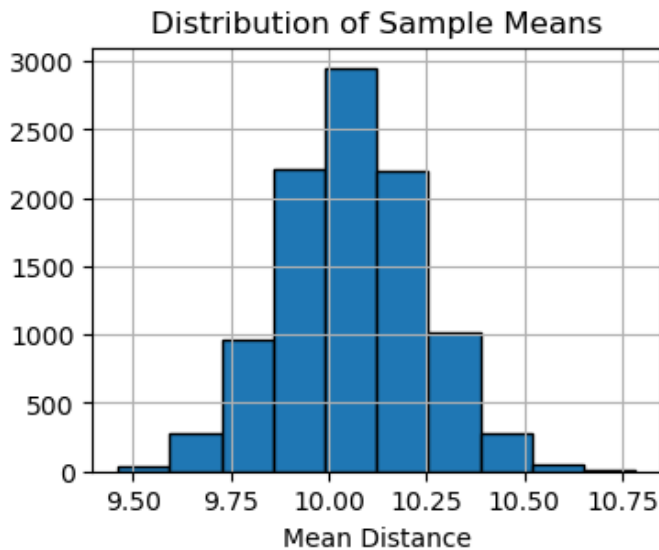
We see that the distribution of the sample means is approximately bell-shaped, as predicted by the central limit theorem. We also see that the distribution is centered around the mean, which we calculated above to be 10.06.

We now repeat the same sampling steps as above, but our sample size is 1,000 instead of 300.

```
means = []
for i in range(10000):
    means.append(get_sample_mean(1000))
mean_array = np.array(means)

plt.figure(figsize=(4,3))
plt.hist(mean_array, edgecolor='black')
plt.xlabel('Mean Distance')
plt.title('Distribution of Sample Means')
plt.grid()
plt.show()
```

Output?



We notice that once again the distribution of sample means is a normal distribution, just as predicted by the central limit theorem, and that the distribution is centered around the mean of 10.06.

But we also notice that for samples of 1000 values, the distribution is narrower than the distribution for samples of 300 values. This makes sense because as the sample size increases, each individual data value (including the unusually high value or low value) has less influence on the sample mean. As a result, the sample mean varies less from sample to sample, and the means from repeated samples are closer together around the mean. This results in a narrower sampling distribution.

The Standard Deviation of the Sampling Distribution

The distribution of the sample means is also called the *sampling distribution*. The spread of the sampling distribution tells us how much the sample mean will vary from sample to sample.

The standard deviation of the sampling distribution is:

$$\text{SD of sampling distribution} = \frac{\text{population SD}}{\sqrt{\text{sample size}}}$$

The standard deviation of the sampling distribution is often called the standard error of the mean, or simply the standard error.

```
print("SD of sampling distribution for sample of 300:", round(np.std(delivery["Distance_km"],
print("SD of sampling distribution for sample 1000:", round(np.std(delivery["Distance_km"],
```

Output?

```
SD of sampling distribution for sample of 300: 0.33
SD of sampling distribution for sample 1000: 0.18
```

The SD of the larger sample (1000) is smaller than the SD of the smaller sample (300). A smaller SD means the sampling distribution is narrower so the sample mean is more likely to be close to the population mean.

As the equation above shows, the SD of the sampling distribution decreases in reverse proportion with $\sqrt{\text{sample size}}$. To reduce the SD by a factor of 10, the sample size must increase by a factor of 100.

Summary

In this chapter we learned how to describe the center and spread of a dataset using the mean, median, variance, and standard deviation. We also learned how z-scores or standard units, Chebyshev's Theorem, the normal distribution, and the Central Limit

Theorem help us understand the distribution of data and why larger samples produce more reliable estimates of the population mean.

13: Linear Regression

In the previous chapter, we learned how to describe a single variable using measurements such as the mean and standard deviation, and how variation affects the accuracy of estimates. In this chapter, we turn our attention to the relationship between two variables. We will learn how to measure the strength of that relationship with correlation and describe it with a regression line.

Scatterplots

In a dataset with multiple features, or columns describing the data, often we want to know whether two features are related. For example, with the car dataset that we first worked with in Chapter 4, we want to know whether the weight of the car and the fuel efficiency (MPG) are related. Does a heavier car tend to have lower MPG?

A scatterplot is used to help us visually determine if two features are related. Scatterplots were first introduced in Chapter 5, and in this chapter we use them to investigate whether there is an association between two features.

For the car dataset, a scatterplot displays each car as a point, with one feature plotted on the horizontal axis and the other on the vertical axis. When all the cars are plotted together, the points can form a pattern that reveals the direction, form, and strength of the relationship between the two features.

We start with the car dataset. The two features we investigate are `mpg` and `weight`.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
cars = pd.read_csv(url)
display(cars.head())
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

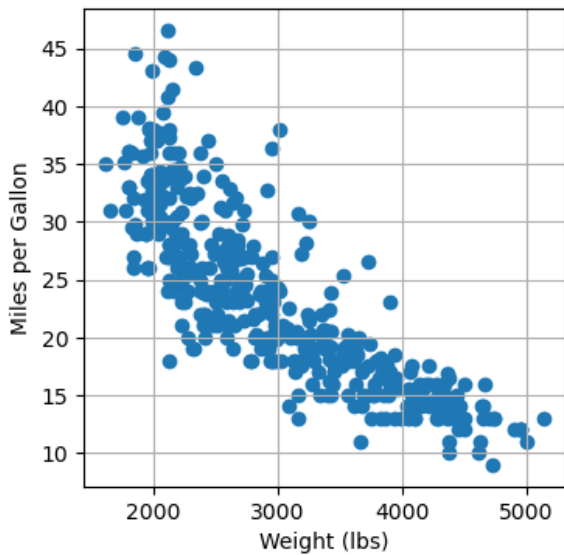
The scatterplot is created by the function

```
plt.scatter(x_axis_variable, y_axis_variable)
```

It doesn't matter which variable is placed on the horizontal axis and which is placed on the vertical axis. Swapping the axes will rotate the scatterplot, but the same relationship between the two variables is still visible.

```
plt.figure(figsize=(4,4))
plt.scatter(cars["weight"], cars["mpg"])
plt.xlabel("Weight (lbs)")
plt.ylabel("Miles per Gallon")
plt.grid()
plt.show()
```

Output?



In the scatterplot above, we can see that as the weight increases, the mpg tends to decrease. Although the points do not fall exactly on a straight line, they form a downward trend.

Correlation

The points in the scatterplot above form a pattern with a downward trend: as the weight increases, the mpg tends to decrease. This type of relationship between two features or two variables is called *correlation*. Correlation describes how closely the two variables are related.

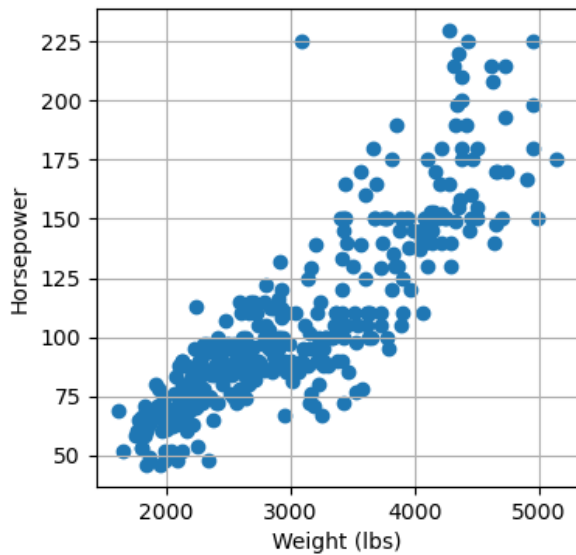
When one variable tends to decrease when the other variable increases, they have a negative correlation. When two variables tend to increase together, they have a positive correlation. If there is no clear relationship between the variables, they have little or no correlation.

The `weight` and `mpg` variables have a *negative correlation*.

We look at the `weight` and `horsepower` of the cars.

```
plt.figure(figsize=(4,4))
plt.scatter(cars["weight"], cars["horsepower"])
plt.xlabel("Weight (lbs)")
plt.ylabel("Horsepower")
plt.grid()
plt.show()
```

Output?

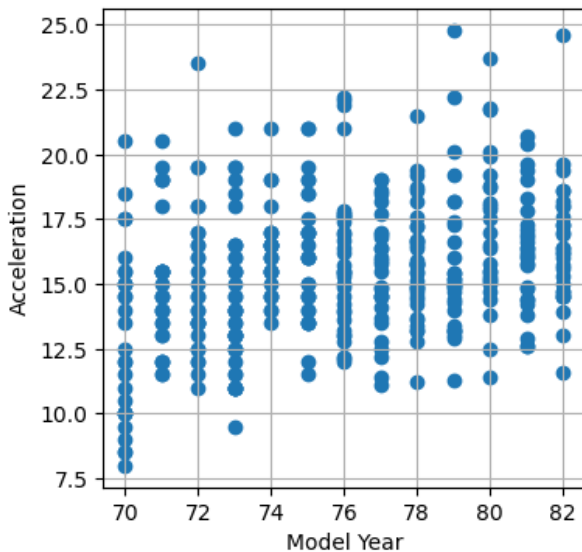


We see that as the weight increases, the horsepower also tends to increase. They have a *positive correlation*.

Lastly we look at `model_year` and `acceleration`.

```
plt.figure(figsize=(4,4))
plt.scatter(cars["model_year"], cars["acceleration"])
plt.xlabel("Model Year")
plt.ylabel("Acceleration")
plt.grid()
plt.show()
```

Output?



The scatterplot above looks different than the first two scatterplots.

Each vertical column of points represents one model year. Since the cars were made only in whole years (1970, 1971, 1972, ...), there are no cars with model years between them, so the points appear in vertical columns instead of being spread evenly across the horizontal axis.

Although there is a very slight upward trend, the points have a wide spread in each year. This means knowing the model year tells us very little about a car's acceleration. We say they have *little or no correlation*.

Scatterplots in Standard Units

In Chapter 12 we learned about *standard units* or the **z-score**. A value in standard units tells us how many standard deviations it is above or below the mean of the dataset.

We wrote the `standard_units` function to convert every value in an `array` to standard units:

```
def standard_units(array):
    return (array - np.mean(array))/np.std(array)
```

Output?

We will now apply this function to both variables in a scatterplot. Although the numbers on the axes will change, the overall pattern of the scatterplot will remain the same.

We work with the `weight` and `horsepower` of the cars.

We convert the `weight` and `horsepower` data into standard units and create a new DataFrame to store them.

```
su = pd.DataFrame({'weight (z)': standard_units(cars['weight']),
                  'horsepower (z)': standard_units(cars['horsepower'])})
display(su)
```

Output?

	weight (z)	horsepower (z)
0	0.620540	0.664133
1	0.843334	1.574594
2	0.540382	1.184397
3	0.536845	1.184397
4	0.555706	0.924265
...	...	...
387	-0.221125	-0.480448
388	-0.999134	-1.364896
389	-0.804632	-0.532474
390	-0.415627	-0.662540
391	-0.303641	-0.584501

392 rows × 2 columns

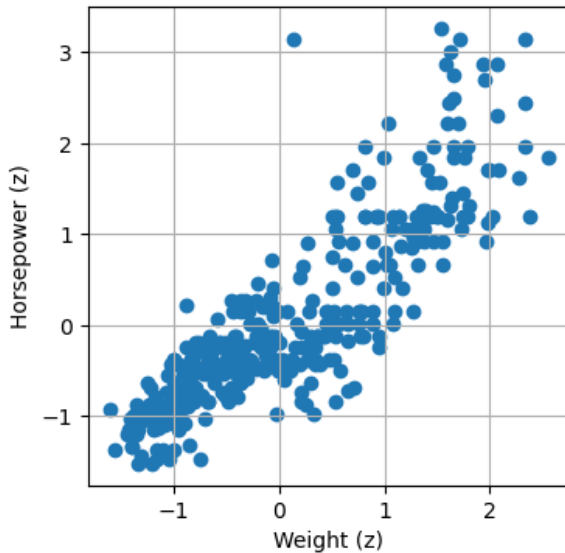
Note that the data for both `weight (z)` and `horsepower (z)` are now centered around 0. The positive z values come from actual weight or horsepower values that are greater than the mean, and the negative z values come from actual weight or horsepower values that are less than the mean. The numbers have changed, but the relative position of the data values remains the same.

We now display the `weight (z)` vs. `horsepower (z)` in a scatterplot to show that the plot in standard units still has the same shape as the plot with actual units in pounds and horsepower shown above.

```
plt.figure(figsize=(4,4))
plt.scatter(su["weight (z)", su["horsepower (z)"]])
```

```
plt.xlabel("Weight (z)")
plt.ylabel("Horsepower (z)")
plt.grid()
plt.show()
```

Output?



Although the axes are now measured in standard units instead of in actual units of pounds and horsepower, the scatterplot has the same overall shape as before. This shows that converting data to standard units does not change the relationship between the two variables.

The Correlation Coefficient

A scatterplot visually shows us the relationship between two variables. To measure the strength of that relationship, we calculate the *correlation coefficient*.

The correlation coefficient is denoted by r and has the following characteristics:

- r is a value between 1 and -1.
- When $r = 1$, all the points lie on a straight line with a positive correlation.
- When $r = 0$, there is no linear correlation.
- When $r = -1$, all the points lie on a straight line with a negative correlation.

We observe the shape of the scatterplot and its relationship to the r value below.

```
# You don't need to study the following code.
# It's used to demo the different r values and its direct
# relationship with the clustering of the data.

plt.figure(figsize=(6,6))

# r = 0.95
plt.subplot(2, 2, 1)
x1 = np.linspace(0, 10, 100)
y1 = x1 * 0.95 + np.random.randn(100)
plt.scatter(x1, y1)
plt.title('subplot 1: r = 0.95')
plt.grid()

# r = 0.5
```

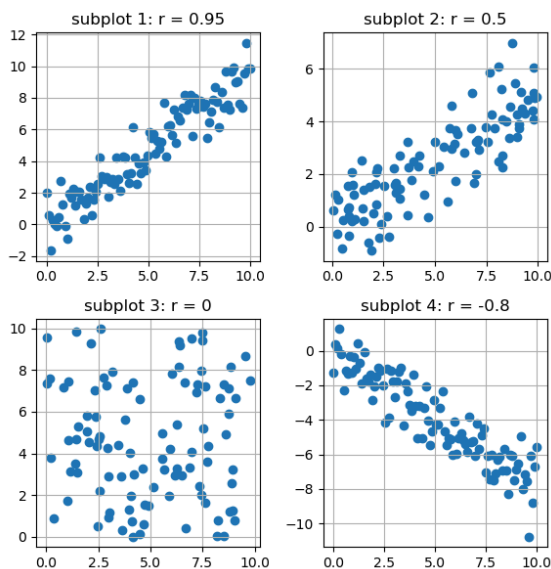
```
plt.subplot(2, 2, 2)
x2 = np.random.rand(100) * 10
y2 = x2 * 0.5 + np.random.randn(100)
plt.scatter(x2, y2)
plt.title('subplot 2: r = 0.5')
plt.grid()

# r = 0
plt.subplot(2, 2, 3)
x3 = np.random.rand(100) * 10
y3 = np.random.rand(100) * 10
plt.scatter(x3, y3)
plt.title('subplot 3: r = 0')
plt.grid()

# r = -0.8
plt.subplot(2, 2, 4)
x4 = np.linspace(0, 10, 100)
y4 = -x4 * 0.8 + np.random.randn(100)
plt.scatter(x4, y4)
plt.title('subplot 4: r = -0.8')
plt.grid()

plt.tight_layout()
plt.show()
```

Output?



The correlation coefficient r is the average of the products of the two variables, after both variables have been converted to standard units, or have been standardized.

Multiplying the two standardized variables captures two aspects of their relationship:

- **Direction.** If both variables are above their means or both are below their means, their product is positive. If one variable is above its mean and the other is below its mean, their product is negative. When we average these products, a positive average indicates a positive correlation, while a negative average indicates a negative correlation.
- **Strength.** A large product means that both variables are far from their means at the same time. A small product means that one or both variables are close to their means. Therefore, for the correlation coefficient r to have a large magnitude, many data points must move far from their means together in a consistent direction, which means that they have a strong linear correlation.

We define a function `correlation` that calculates the `r` value for 2 sequences of data:

```
def correlation(array1, array2):
    return np.mean(standard_units(array1) * standard_units(array2))
```

Output?

We test the `correlation` function, with the `x1` and `y1` data from subplot 1 above, and with the `x3` and `y3` data from subplot 3 above.

```
print("r for subplot 1:", round(correlation(x1, y1), 2))
print("r for subplot 3:", round(correlation(x3, y3), 2))
```

Output?

```
r for subplot 1: 0.94
r for subplot 3: -0.02
```

The calculated `r` results correspond with the scatterplots' shapes.

Generally, the closer the value of `r` is to 1 or -1, the stronger the linear relationship between the variables. The closer `r` is to 0, the weaker the linear relationship.

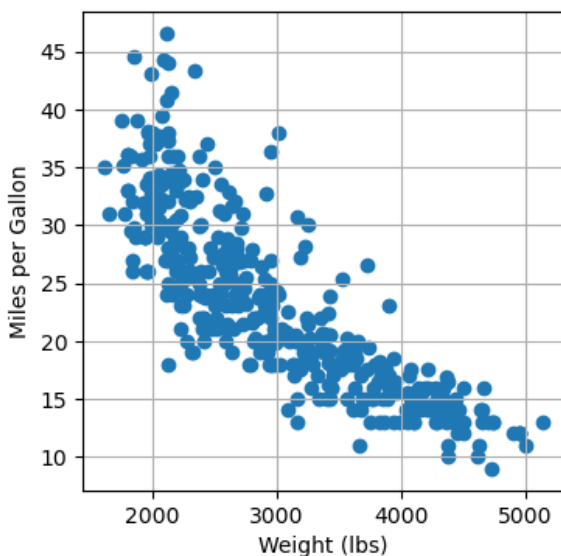
There is no universal cutoff for deciding what is considered "strong" or "weak." The scatterplot should always be examined together with the value of `r`.

The Regression Line

We now look more closely at the `weight` and `mpg` of the cars. Here's the scatterplot again:

```
plt.figure(figsize=(4,4))
plt.scatter(cars["weight"], cars["mpg"])
plt.xlabel("Weight (lbs)")
plt.ylabel("Miles per Gallon")
plt.grid()
plt.show()
```

Output?



In the scatterplot we can see that heavier cars generally have lower mpg or lower fuel efficiency.

The points do not fall exactly on a straight line, but they form a clear downward pattern. We can imagine drawing a line that passes through the middle of the points. The line doesn't have to touch all the data points, but it gets as close as possible on average to all the data points.

If we follow such a line, we can predict a car's mpg based on its weight. For example, if the car is 4000 lbs, we can estimate that the car's mpg will be about 15. And if the car is 3000 lbs, we can estimate that the mpg will be about 22.

The line we used above to make these predictions is called the *regression line*, or the *line of best fit*. It summarizes the overall linear relationship between the two variables. For each value of **weight** on the horizontal axis, the regression line gives a predicted value of **mpg** on the vertical axis.

Calculate the Regression Line

In the previous example, we estimated the regression line visually. However, different people might draw slightly different lines. Linear regression gives us a mathematical method for calculating the line that best fits the data.

In this case **weight** is the predictor variable because it is used to make the prediction, and **mpg** is the response variable because it is the value being predicted. The predictor variable is placed on the x-axis, and the response variable is placed on the y-axis.

- **weight** = predictor, on x-axis
- **mpg** = response, on y-axis

Use Standard Units

Just as the correlation coefficient is calculated using standard units, the regression line is also calculated using standard units. We will later convert the regression line back to the original units.

We convert the **weight** and **mpg** into standard units.

```
x = standard_units(cars["weight"])
y = standard_units(cars["mpg"])
```

Output?

When the two data sequences **x** and **y** have standard units, then the regression line is described as:

$$\hat{y} = r \times x$$

where **r** is the correlation coefficient, and $\hat{y}$ (called y hat) is the predicted value of **y** on the regression line for a given value of **x**.

The equation says that for every value **x**, we can find the corresponding $\hat{y}$ on the regression line by multiplying **x** by **r**.

If **x** and **y** had a perfect positive linear relationship, then **r** = 1 and the regression line is simply: $\hat{y} = x$

However, two variables are usually not perfectly correlated, so the **r** value adjusts the slope of the line to reflect the direction and strength of the relationship.

For the weight and mpg, the **r** value calculated with our **correlation** function from earlier:

```
r = correlation(x, y)
print("r:", r)
```

Output?

r: -0.8322442148315758

We note that **r** is negative as expected from the downward trend in the scatterplot. It's also reasonably close to -1, which agrees with the strong linear correlation that we observe from the scatterplot.

Now that we've found the **r** value, given any **x** value, we can multiply it by **r** to find the $\hat{y}$ value of the regression line, which is our

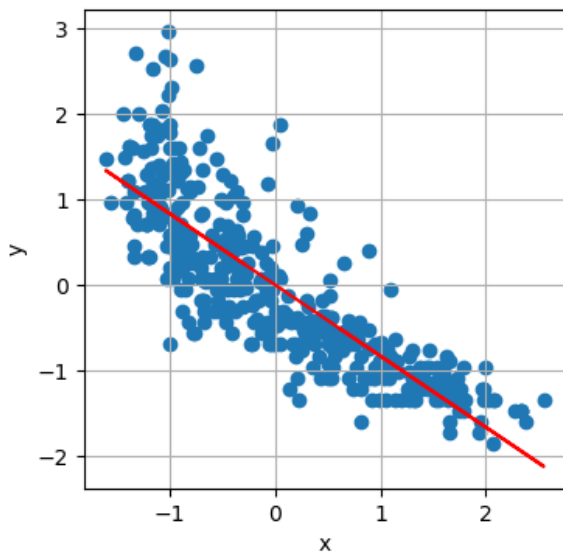
predicted $\hat{y}$ value.

We display the same scatterplot and add a calculated regression line in red.

```
plt.figure(figsize=(4,4))

plt.scatter(x, y)
regression_line = r * x
plt.plot(x, regression_line, color='red')
plt.xlabel("x")
plt.ylabel("y")
plt.grid()
plt.show()
```

Output?



For the red regression line above, which is describe as:

$$\hat{y} = r \times x$$

- the *slope* of the line is r . The sign of r determines the direction of the line. Since r is negative, the line slopes downward: as x increases, the predicted value $\hat{y}$ decreases. The closer r is to -1 , the steeper the downward slope.
- the *intercept* of the line is at $(0, 0)$. The intercept is where the line crosses the y -axis. Because both x and y are measured in standard units, their means are 0 . Therefore, when $x=0$ (or the mean of x), the predicted value is also $\hat{y}=0$ (or the mean of y).

Convert to Original Units

The regression line:

$$\hat{y} = r \times x$$

predicts values only in standard units. To make predictions with the original data, we must convert the regression line back to the original units.

Recall the definition of the standard units:

$$\text{standard units} = \frac{\text{data} - \text{mean}}{\text{standard deviation}}$$

Substitute the definition of `standard_units` for $\hat{y}$ and x in the equation for the regression line:

$$\frac{\hat{y} - \text{mean}_{\hat{y}}}{SD_y} = r \times \frac{x - \text{mean}_x}{SD_x}$$

Solve for $\hat{y}$:

$$\hat{y} = r \times \frac{SD_y}{SD_x} \times (x - \text{mean}_x) + \text{mean}_{\hat{y}}$$

Distribute $(x - \text{mean}_x)$

$$\hat{y} = (r \times \frac{SD_y}{SD_x} \times x) - (r \times \frac{SD_y}{SD_x} \times \text{mean}_x) + \text{mean}_{\hat{y}}$$

Rearrange into the standard form of a line: $y = mx + b$

$$\hat{y} = (r \times \frac{SD_y}{SD_x}) \times x + (\text{mean}_{\hat{y}} - (r \times \frac{SD_y}{SD_x} \times \text{mean}_x))$$

In the equation of the line: $y = mx + b$, m is the slope and b is the intercept.

Therefore the slope of the regression line is:

$$\text{slope} = r \times \frac{SD_y}{SD_x}$$

The intercept is:

$$\text{intercept} = \text{mean}_{\hat{y}} - (r \times \frac{SD_y}{SD_x} \times \text{mean}_x)$$

or:

$$\text{intercept} = \text{mean}_{\hat{y}} - \text{slope} \times \text{mean}_x$$

We can now write a function to find the slope and the intercept:

```
def slope(x, y):
    return correlation(x, y) * np.std(y) / np.std(x)
```

Output?

```
def intercept(x, y):
    return np.mean(y) - slope(x, y) * np.mean(x)
```

Output?

We now apply the functions to the cars `weight` and `mpg` data. First we review the dataset.

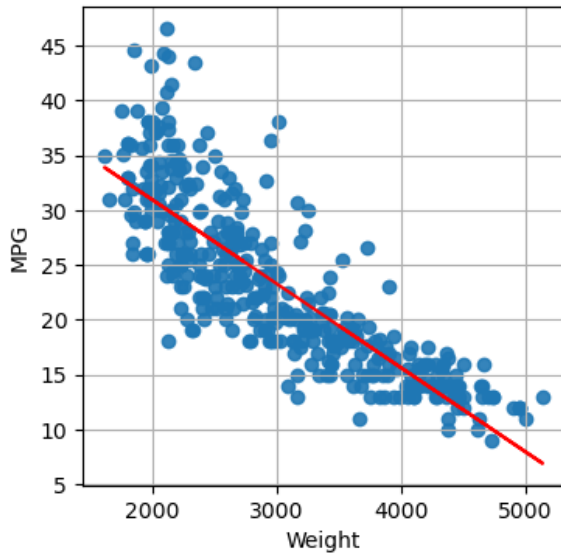
```
weight_mpg_slope = slope(cars["weight"], cars["mpg"])
weight_mpg_intercept = intercept(cars["weight"], cars["mpg"])
```

Output?

Displaying the scatterplot for `weight` vs `mpg`, along with the regression line.

```
plt.figure(figsize=(4,4))
plt.scatter(cars["weight"], cars["mpg"], alpha=0.9)
# regression line y = slope * x + intercept
plt.plot(cars["weight"], weight_mpg_slope * cars["weight"] + weight_mpg_intercept, color='
plt.xlabel("Weight")
plt.ylabel("MPG")
plt.grid()
plt.show()
```

Output?



With the linear regression line, we now have a consistent way to predict the MPG or $\hat{y}$, given a car's weight, which is x .

```
weight = 4000
predicted_mpg = weight_mpg_slope * weight + weight_mpg_intercept
print("Predicted MPG for a car weighing", weight, "lbs:", round(predicted_mpg, 1))
```

Output?

Predicted MPG for a car weighing 4000 lbs: 15.6

Meaning of the Slope

With the `weight` vs `mpg` data, the slope is:

```
print(round(weight_mpg_slope, 2))
```

Output?

-0.01

The slope of the regression line is -0.01 . This means that, on average, for every additional pound of weight, the predicted MPG decreases by about 0.01 MPG. Equivalently, a car that is 100 pounds heavier is predicted to get about 1 MPG less.

In general, the slope of a regression line represents the average change in the predicted value of y for each one-unit increase in x . A positive slope means the predicted value of y increases as x increases, while a negative slope means the predicted value of y decreases as x increases.

Summary

In this chapter, we learned how to investigate the relationship between two variables using scatterplots, the correlation coefficient, and the regression line. We saw how scatterplots reveal patterns in the data, how the correlation coefficient measures the direction and strength of a linear relationship, and how the regression line can be used to predict one variable from another.

14: Root Mean Squared Error

In the previous chapter, we learned how a regression line models the relationship between two variables and how the correlation coefficient describes the strength of that relationship. In this chapter, we will evaluate how well a regression line predicts new values. We will measure the prediction errors and find ways to measure how close the predicted values are to the actual data.

Review: Find the Regression Line

We revisit the cars dataset to work with `weight` and `mpg` and find the regression line.

We read in the data and show the scatterplot of `weight` vs `mpg`.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

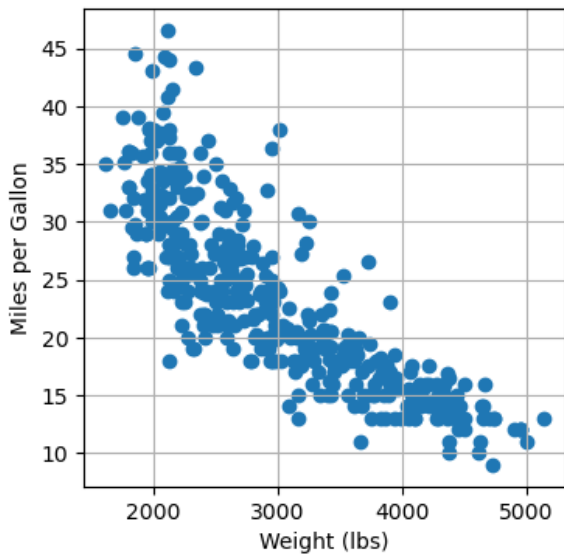
url = 'https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
cars = pd.read_csv(url)
cars.head()
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

```
plt.figure(figsize=(4,4))
plt.scatter(cars["weight"], cars["mpg"])
plt.xlabel("Weight (lbs)")
plt.ylabel("Miles per Gallon")
plt.grid()
plt.show()
```

Output?



Next we bring in the `standard_units` and `correlation` functions from the previous chapter.

```
def standard_units(array):
    return (array - np.mean(array))/np.std(array)

# correlation function uses standard_units function to calculate the correlation coefficient
def correlation(array1, array2):
    return np.mean(standard_units(array1) * standard_units(array2))
```

Output?

Then we bring in the `slope` and `intercept` functions from the previous chapter.

We use these functions to find the slope and intercept of the regression line.

```
# slope function uses correlation function to calculate the slope of the regression line
def slope(x, y):
    return correlation(x, y) * np.std(y) / np.std(x)

# intercept function uses slope function to calculate the y-intercept of the regression line
def intercept(x, y):
    return np.mean(y) - slope(x, y) * np.mean(x)

# find the slope and intercept of the regression line for weight and mpg
reg_line_slope = slope(cars['weight'], cars['mpg'])
print("slope:", round(reg_line_slope, 2))
reg_line_intercept = intercept(cars['weight'], cars['mpg'])
print("intercept:", round(reg_line_intercept, 2))
```

Output?

```
slope: -0.01
intercept: 46.22
```

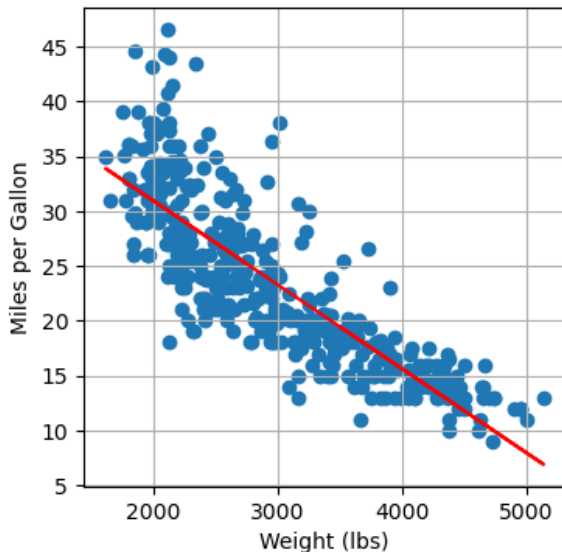
We recall what the slope of the regression line means:

For every 1-pound increase in weight, the predicted MPG decreases by about 0.01, or, for every 100-pound increase in weight, the predicted MPG decreases by about 1 mile per gallon.

We plot the prediction line in the scatterplot by using the equation for a line: $\hat{y} = \text{slope} \times x + \text{intercept}$, where $\hat{y}$ represents the predicted values.

```
plt.figure(figsize=(4,4))
plt.scatter(cars['weight'], cars['mpg'])
plt.plot(cars['weight'], reg_line_slope * cars['weight'] + reg_line_intercept, color='red')
plt.xlabel('Weight (lbs)')
plt.ylabel('Miles per Gallon')
plt.grid()
plt.show()
```

Output?



We recall that the regression line is the *best fit* line, so it doesn't necessarily pass through all the data points.

In the plot above, the blue points are the actual y values or the actual MPG. The red regression line represents predicted values $\hat{y}$ or the predicted MPG for each weight. We see that the red line passes through some of the blue points. At these points the predicted MPG matches the actual MPG exactly. But we also see many blue points that lie above or below the red regression line, which means the predicted values are different from the actual MPG values. The farther a blue point is from the regression line, the larger the prediction error. We want to measure how accurate the predictions are.

The Residuals

We measure the prediction error by comparing the actual MPG with the predicted MPG. This difference is called the *residual*.

The residual is the vertical distance between a blue point (the actual value) and the red regression line (the predicted value). If the blue point lies on the regression line, then the residual is 0 because the prediction is exactly correct. The farther a blue point is above or below the regression line, the larger the residual and the larger the prediction error.

Working with the existing scatterplot and regression line, we add code to select 4 cars (or 4 rows in the DataFrame), plot them, and plot their residuals:

```
plt.figure(figsize=(6,6))

# scatter plot and regression line
plt.scatter(cars['weight'], cars['mpg'], alpha=0.8, label='All cars')
plt.plot(cars['weight'], reg_line_slope * cars['weight'] + reg_line_intercept, color='red')

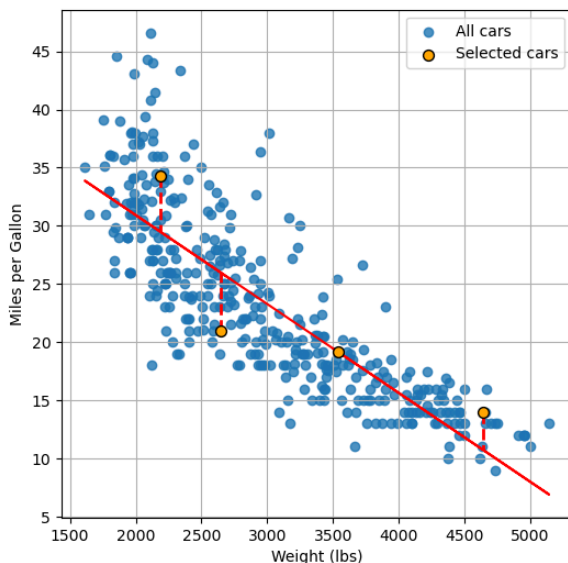
# Note that you don't have to study this block of code.
# It is to demo the residuals for 4 selected cars.
# choose 4 cars to highlight
selected = cars.iloc[[24,315,137,250]]
# plot selected cars
```

```
plt.scatter(selected['weight'], selected['mpg'], color='orange', edgecolor='k', s=50, zord

# plot vertical differences for selected points
for i, row in selected.iterrows():
    x = row['weight']
    y_actual = row['mpg']
    y_pred = reg_line_slope * x + reg_line_intercept
    plt.plot([x, x], [y_actual, y_pred], color='red', linestyle='--', linewidth=2)

# show labels
plt.xlabel('Weight (lbs)')
plt.ylabel('Miles per Gallon')
plt.grid()
plt.legend()
plt.show()
```

Output?



The vertical red dashed lines represent the residuals.

We observe that:

- one of the points is on the regression line and its residual is 0
- one of the points is below the regression line and its residual is negative.
- two of the points are above the regression line and their residuals are positive. Of these two points, the one farther away from the regression line has the larger residual.

Residual Plots

While we can plot the residual for each data point with a red dashed line, it's easy to imagine that all the red lines from all the data points could overlap and create a cluttered plot, making it difficult to make good observations of them.

Instead, we can draw residual plots which display all the residuals in a more organized way. A *residual plot* is a scatterplot of the predictor on the x-axis, and the residuals on the y-axis. The plot also contains a horizontal line at residual = 0 to represent a perfectly accurate prediction. Points above this line have positive residuals and points below this line have negative residuals.

First we find the predicted MPG, then we find the residuals.

```
# find predicted MPG
y_pred = reg_line_slope * cars['weight'] + reg_line_intercept
```

```
# find residuals
residuals = cars['mpg'] - y_pred
```

Output?

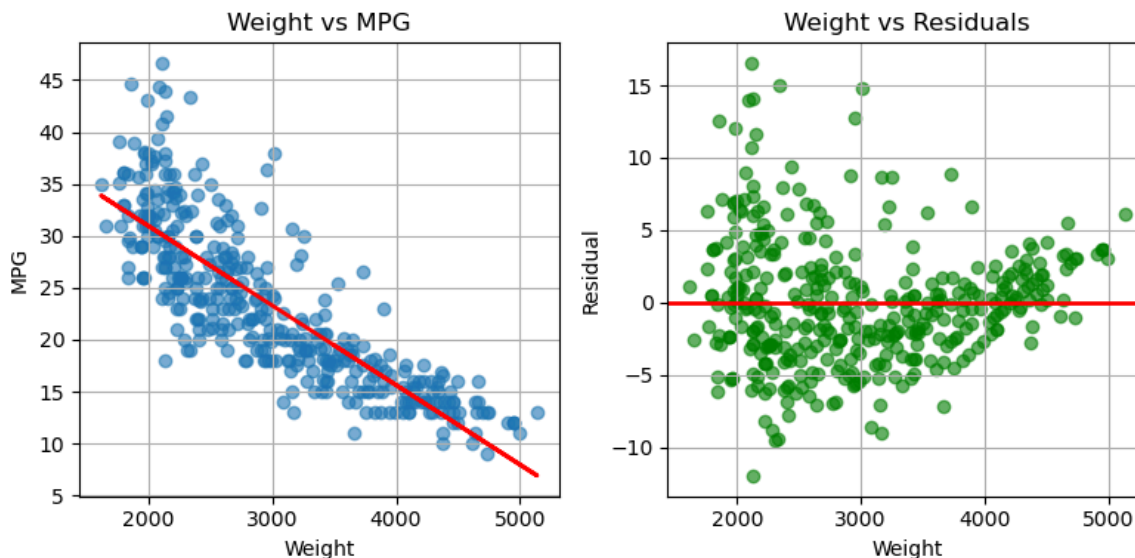
Plot weight vs MPG and weight vs residuals side-by-side.

```
plt.figure(figsize=(8, 4))
# plot weight vs mpg
plt.subplot(1, 2, 1)
plt.scatter(cars['weight'], cars['mpg'], alpha=0.6)
plt.plot(cars['weight'], y_pred, color='red', linewidth=2)
plt.xlabel('Weight')
plt.ylabel('MPG')
plt.title('Weight vs MPG')
plt.grid()

# plot weight vs residuals
plt.subplot(1, 2, 2)
plt.scatter(cars['weight'], residuals, color='green', alpha=0.6)
plt.axhline(0, color='red', linewidth=2)
plt.xlabel('Weight')
plt.ylabel('Residual')
plt.title('Weight vs Residuals')
plt.grid()

plt.tight_layout()
plt.show()
```

Output?



Every point in the original scatterplot has a corresponding point in the residual plot. The x-coordinate stays the same, while the y-coordinate changes from the actual MPG to the residual.

Looking at the Weight vs MPG plot, we observe that:

- At the two ends of the weight range, around 2000 lb and 5000 lb, many of the actual MPG values are above the regression line. This is reflected in the residual plot, with many residuals lying above the zero line.
- In the middle of the weight range, between about 2500 lb to 4000 lbs, many of the actual MPG values are below the regression line. This is reflected in the residual plot, with many of the residuals lying below the zero line.

When the residuals form a curved pattern instead of being randomly scattered around the zero line, it suggests that the relationship between the variables is curved rather than linear. A residual plot can help us detect when a linear regression model may not be a good fit for the data.

We also observed from the Weight vs MPG plot that:

- From about 4000 lb to 5000 lb, all the data points tend to be closer to the regression line. This is reflected in the residual plot, where the data are clustered more closely for the same weight range.
- Below 4000 lb quite a few data points are farther away from the linear regression line. This is reflected in the residual plot, where the data points are spread out more.

Residual plots can be used to detect uneven spread of prediction errors, such as when the residuals are larger for a specific range of x-values.

MSE and RMSE

A residual plot helps us see the pattern and spread of the prediction errors. However, it can still be difficult to judge the overall accuracy of the regression model by looking at many residuals.

To summarize all the residuals with a single number, we use the Mean Squared Error (MSE) and the Root Mean Squared Error (RMSE).

MSE

As each residual is the prediction error for one data point, a summary of all the residuals, or their average, gives us a measure of the overall prediction error.

We simply can't calculate the average of the residuals by summing all the residuals and dividing by the number of residuals. This is because the regression line passes through the center of the data points, so the positive and negative residuals effectively cancel each other out. As a result, the average residual is 0 even when many predictions are far from the actual data.

For example, if two residuals are +5 and -5, then

$$\frac{-5 + 5}{2} = 0$$

The result shows no prediction error, even though both predictions are actually off by 5 units.

Instead we'll use the same method as the calculation of the standard deviation:

- square the error values to remove the negative sign
- find the average of all the squares

This average is the *Mean Squared Error* or the *MSE*. A smaller MSE means the predictions are generally closer to the actual value.

MSE is useful because it summarizes the prediction errors in a single number, and it is the value used by many regression models to find the best regression line.

However, because the errors are squared, the units are also squared. For example, if the MPG prediction errors are measured in miles per gallon, then MSE is measured in miles per gallon². Since people normally think of prediction errors in miles per gallon rather than square miles per gallon, MSE is not easy to interpret.

RMSE

Just as with the standard deviation calculation, we can take the square root of the MSE. This converts the squared units back to the original units of the data, making the result easier to interpret. The result is called the *Root Mean Squared Error* or *RMSE*. Its long name describes how it is calculated: square the residuals, then find their mean, and finally take the square root.

```
rmse = np.sqrt(np.mean(residuals**2))
print("RMSE:", round(rmse, 2))
actual_mpg_mean = np.mean(cars["mpg"])
print("Average actual MPG:", round(actual_mpg_mean, 2))
print("RMSE as a percentage of average MPG:", round(rmse / actual_mpg_mean * 100), "%")
```

Output?

```
RMSE: 4.32
Average actual MPG: 23.45
RMSE as a percentage of average MPG: 18 %
```

The RMSE is 4.32. This means that the regression model's predictions differ from the actual MPG values by about 4.32 miles per gallon, on average. A smaller RMSE indicates that the predictions are closer to the actual values and that the regression model is more accurate.

The average MPG of all the cars in the dataset is 23.45 MPG. The RMSE is about 18%, or about one-fifth of the average MPG. This gives us a sense of the size of the prediction error relative to a typical MPG value in the dataset.

The RMSE is most useful for comparing different regression models on the same dataset. In general the model with the smaller RMSE makes more accurate predictions.

Summary

In this chapter, we review how a linear regression model predicts values using a regression line, then we learn how to measure the accuracy of those predictions. We find the residual, which is the difference between the actual and predicted values, and use residual plots to visualize prediction errors. Then we learn how the Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) summarize the overall prediction error. Together, these tools help us evaluate and compare the accuracy of regression models.

15: Classification

In the previous chapter, we learned how regression predicts numerical values, such as miles per gallon or delivery times. In this chapter, we will learn how to predict categories instead of numbers. *Classification* algorithms determine which category a new data point most likely belongs to, by comparing it with previously labeled data.

In this chapter we cover a simple and intuitive classification method: the nearest neighbor algorithm. We will work with datasets in which each observation belongs to one of only two categories. This is called *binary classification*.

Gather Data

To investigate what a nearest neighbor is, we work with the `cdc` dataset that we used in previous chapters. We will determine whether the height and weight of a person can be used to predict the person's gender.

We read in data from the `cdc.csv` file and inspect it.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

url = 'https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
health = pd.read_csv(url)
display(health.head())
print("Number of rows:", health.shape[0])
```

Output?

	genhlth	exerany	hlthplan	smoke100	height	weight	wt desire	age	gender
0	good	0	1	0	70	175	175	77	m
1	good	0	1	1	64	125	115	33	f
2	good	1	1	1	60	105	105	49	f
3	good	1	1	0	66	132	124	42	f
4	very good	0	1	0	61	150	130	55	f

Number of rows: 20000

Prepare Data

We create a new DataFrame from the columns `height`, `weight`, and `gender`.

```
data = health[['height', 'weight', 'gender']].copy()
data.head()
```

Output?

	height	weight	gender
0	70	175	m
1	64	125	f
2	60	105	f
3	66	132	f

	height	weight	gender
4	61	150	f

Because the classification algorithm works with numerical data, we need to represent the gender categories as numbers instead of letters. We will replace `m` with 0 and `f` with 1.

We arbitrarily assign `m` to 0 and `f` to 1, but we can just as easily assign the 0 and 1 values the opposite way. The numbers do not mean that one category is larger or more important than the other. They simply provide two distinct labels that the algorithm can use to identify the two classes.

the DataFrame has a map method that can be used to map values in a column to new values

```
data['gender'] = data['gender'].map({'m': 0, 'f': 1})
data.head()
data.head()
```

Output?

	height	weight	gender
0	70	175	0
1	64	125	1
2	60	105	1
3	66	132	1
4	61	150	1

The `gender` feature has two categories or two *classes*: 0 or 1. In a classification problem, the possible output categories are called *classes*. Therefore, the gender feature has two classes: class 0 and class 1.

We inspect the `height` and `weight` data by finding their maximum and minimum.

```
print("Maximum height:", data['height'].max())
print("Minimum height:", data['height'].min())
print("Maximum weight:", data['weight'].max())
print("Minimum weight:", data['weight'].min())
```

Output?

```
Maximum height: 93
Minimum height: 48
Maximum weight: 500
Minimum weight: 68
```

The height and weight are measured on different scales. Height ranges from 48 to 93 inches, while weight ranges from about 68 to 500 pounds. Because the nearest neighbor algorithm calculates the distance between data points, the larger values for weight would have a much greater influence on the distance than the smaller values for height.

Therefore, to give each feature equal importance, we convert both height and weight to standard units. After this conversion, both features are measured on the same scale, so neither one dominates the distance calculation.

We bring in the `standard_units` function from Chapter 12, then we use the function to convert the data.

```
def standard_units(array):
    return (array - np.mean(array))/np.std(array)

data['height'] = standard_units(data['height'])
data['weight'] = standard_units(data['weight'])
data.head()
```

Output?

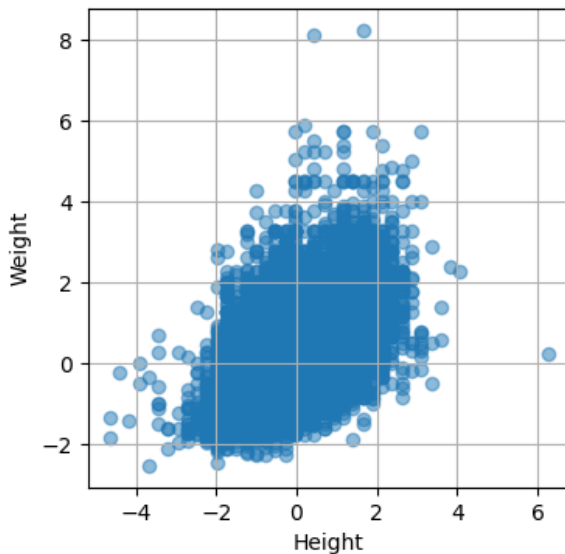
	height	weight	gender
0	0.682792	0.132661	0
1	-0.771453	-1.114845	1
2	-1.740950	-1.613847	1
3	-0.286704	-0.940194	1
4	-1.498576	-0.491092	1

Exploratory Data Analysis (EDA)

First we look at the relationship between `height` and `weight` by plotting them in a scatterplot.

```
plt.figure(figsize=(4,4))
plt.scatter(data['height'], data['weight'], alpha=0.5)
plt.xlabel('Height')
plt.ylabel('Weight')
plt.grid()
plt.show()
```

Output?



There is a positive correlation between `height` and `weight`. As the height increases, the weight tends to also increase. However, the points are fairly spread out, so people with the same height can have a wide range of weights. As a result, height alone cannot predict a person's weight with high accuracy.

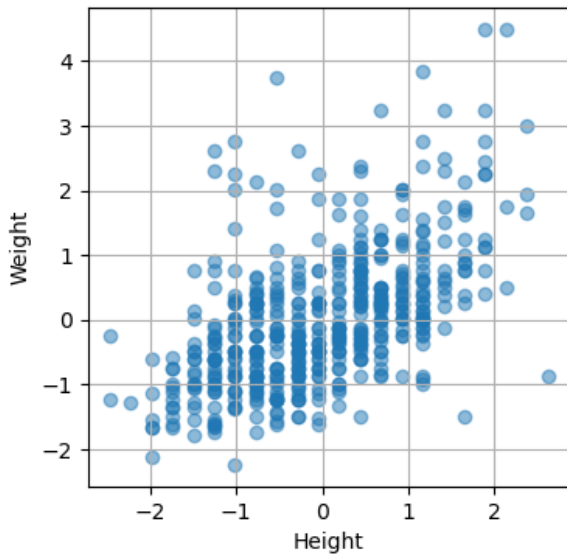
In addition, because the dataset contains 20,000 observations, the scatterplot is so crowded that it is difficult to see any overall pattern. For illustration, we randomly select 500 observations from the dataset. This smaller sample still shows the relationship between the variables while making the individual data points visible.

The nearest neighbor algorithm can later be applied to the full dataset, but we for now use a smaller sample to make the scatterplots and decision boundaries much easier to visualize.

```
# create a sample of 500 rows from data
sample = data.sample(500, random_state=1)
```

```
# plot a scatterplot of the sample data
plt.figure(figsize=(4,4))
plt.scatter(sample['height'], sample['weight'], alpha=0.5)
plt.xlabel('Height')
plt.ylabel('Weight')
plt.grid()
plt.show()
```

Output?



The smaller sample preserves the overall pattern of the data. Although there are fewer points, the scatterplot still shows the same positive relationship between height and weight as the full dataset.

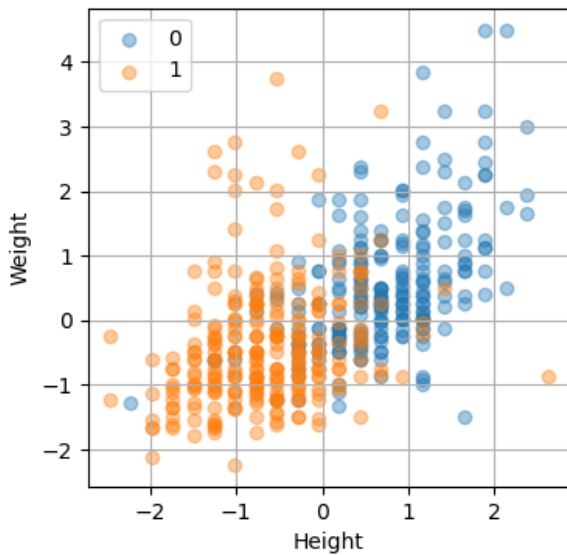
The scatterplot above doesn't tell us which data points belong to class 0 (male) and which belong to class 1 (female). To distinguish the two classes, we use `groupby` to separate the data into two groups based on the `gender`. We then plot each group using a different color, making it easier to see the two categories.

```
# group by gender: the result will be separated into two groups
groups = sample.groupby('gender')

plt.figure(figsize=(4,4))
for gender, group in groups:
    plt.scatter(group['height'], group['weight'], alpha=0.4, label=gender)

plt.xlabel('Height')
plt.ylabel('Weight')
plt.legend()
plt.grid()
plt.show()
```

Output?



Now we can see a pattern with the two *classes*:

- Males (class 0) generally have larger heights and weights. Most of the male data points are above 0 on the height axis and above -0.5 on the weight axis.
- Females (class 1) generally have smaller heights and weights. Most of the female data points lie between -2 and +1 on the height axis and below 0 on the weight axis.
- The two classes overlap considerably. Some males have similar heights and weights as some females, and many of the blue and orange points appear in the same regions of the scatterplot. Therefore it is not always possible to determine a person's gender from height and weight alone.

Because the two classes overlap, we need a method to decide which class a new observation most likely belongs to. The nearest neighbor algorithm does this by looking at the classes of nearby observations.

Nearest Neighbor

Suppose we have a new person named Kai whose height is -1.2 standard units and whose weight is -0.8 standard units. We would like to predict whether Kai belongs to class 0 (male) or class 1 (female).

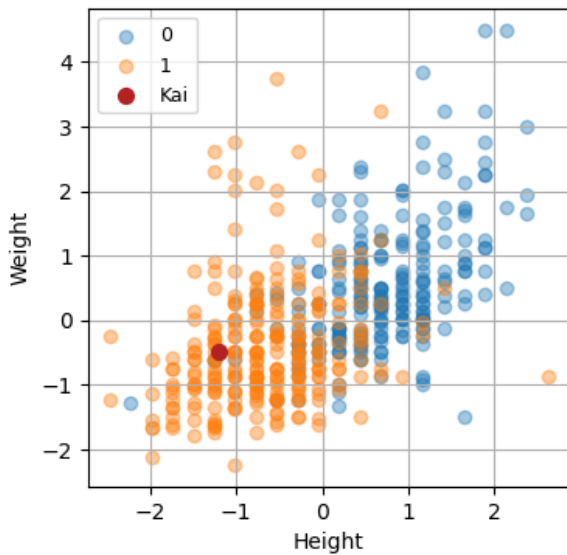
We can plot Kai's height and weight in the scatterplot to see where this new data point is placed.

- If Kai's data point is near the blue cluster, then we would predict that Kai belongs to class 0.
- If Kai's data point is near the orange cluster, then we would conclude that Kai belongs to class 1.

```
kai_height = -1.2
kai_weight = -0.5

plt.figure(figsize=(4,4))
for gender, group in groups:
    plt.scatter(group['height'], group['weight'], alpha=0.4, label=gender)
plt.scatter(kai_height, kai_weight, color='firebrick', s=50, label='Kai')
plt.xlabel('Height')
plt.ylabel('Weight')
plt.legend(fontsize=9)
plt.grid()
plt.show()
```

Output?



Kai's data point lies within the cluster of orange points. Since most of the nearby observations belong to class 1, we predict that Kai belongs to class 1 (female).

The reasoning above is an example of the *nearest neighbor classification* method. We predict the class of the new data point based on class of the *nearest* labeled data point. The idea is that data points that are close together are often more similar than data that are far apart. Therefore, if the nearest neighbor belongs to one class, then the new data is likely to be in the same class.

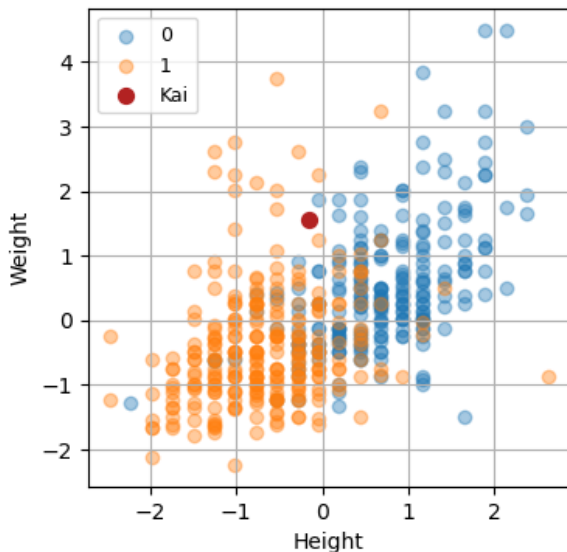
When a new data point clearly lies within a cluster of one class, as in the example above, then it's easy to determine which class the new data belongs to. But what if Kai's height is -0.1 standard units and weight is 1.3 standard units?

We plot Kai's data point in the scatterplot:

```
kai_height = -0.15
kai_weight = 1.55

plt.figure(figsize=(4,4))
for gender, group in groups:
    plt.scatter(group['height'], group['weight'], alpha=0.4, label=gender)
plt.scatter(kai_height, kai_weight, color='firebrick', s=50, label='Kai')
plt.xlabel('Height')
plt.ylabel('Weight')
plt.legend(fontsize=9)
plt.grid()
plt.show()
```

Output?



Now it is no longer clear which class Kai belongs to. Kai's data point lies between some blue and some orange data points, so we cannot decide the class simply by looking at the scatterplot.

To make a prediction, we need to determine which labeled data points are closest to Kai. We do this by calculating the distance between Kai and the other data points.

Find the Nearest Neighbor

From the scatterplot we see that the two nearest data points to Kai's data point are above and below Kai's data point, and both are slightly to the right. However, it is difficult to tell by eye which one is the nearest neighbor. We therefore measure the distance from Kai to each of these data points.

To calculate the distance, we first locate these data points in the dataset.

```
# earch for the two data points with height between -0.5 and 0 and weight between 1 and 2
condition = ((sample['height'] >= -0.5) & (sample['height'] <= 0) &
             (sample['weight'] >= 1) & (sample['weight'] <= 2))

# find the index of the closest data points to Kai's data point
closest_data_index = sample.loc[condition].index
closest_data_index
```

Output?

```
Index([3833, 9270], dtype='int64')
```

We found two indices for the two closest data points and save the two data points in two variables.

```
closest_1 = sample.loc[closest_data_index[0]]
closest_2 = sample.loc[closest_data_index[1]]
```

Output?

We now use the distance formula to find the distance between Kai's data point and `closest_1` and find the distance between Kai's data point and `closest_2`.

In two-dimensional space, the distance between two points (x_1, y_1) and (x_2, y_2) is:

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The formula is derived from the [Pythagorean theorem](#), a foundational idea in geometry.

Although the formula looks complicated at first, we can break it down step by step in the an example below.

```
# Illustration of the distance between two points in a 2D space

# create 2 sample points
point_A = (0, 0)  # (x1, y1)
point_B = (3, 4)  # (x2, y2)

# Calculate the rise and run
# - the rise is the difference along the y-axis between the points
# - the run is the difference along the x-axis between the points
run = point_A[0] - point_B[0]  # x1 - x2
rise = point_A[1] - point_B[1]  # y1 - y2

# Calculate the distance
distance = np.sqrt(run**2 + rise**2)
print("Distance between Point A and Point B:", distance)

# Plot
plt.figure(figsize=(3,3))

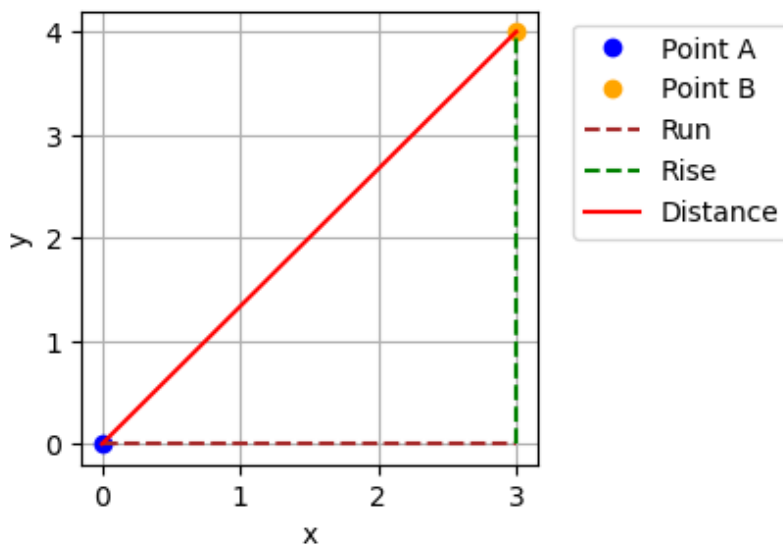
plt.plot(point_A[0], point_A[1], 'o', color='blue', label="Point A")
plt.plot(point_B[0], point_B[1], 'o', color='orange', label="Point B")

plt.plot([point_A[0], point_B[0]], [point_A[1], point_A[1]], color='brown',
         linestyle='--', label="Run")
plt.plot([point_B[0], point_B[0]], [point_A[1], point_B[1]], color='green',
         linestyle='--', label="Rise")
plt.plot([point_A[0], point_B[0]], [point_A[1], point_B[1]], color='red',
         linestyle='-', label="Distance")

plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```

Output?

Distance between Point A and Point B: 5.0



Using the distance formula, we calculate the distance between `closest_1` and Kai's data point and the distance between `closest_2` and Kai's data point.

```
distance_1 = np.sqrt((closest_1['height'] - kai_height)**2 + (closest_1['weight'] - kai_we
distance_2 = np.sqrt((closest_2['height'] - kai_height)**2 + (closest_2['weight'] - kai_we
print("Distance to closest_1:", round(distance_1, 2))
print("Distance to closest_2:", round(distance_2, 2))
```

Output?

```
Distance to closest_1: 0.35
Distance to closest_2: 0.31
```

The nearest neighbor is `closest_2`. We can print the exact height and weight values for point 2.

```
# find the index of closest_2 and print height and weight using that index
idx = closest_2.name
print("closest_2 index:", idx)
print("closest_2 height:", round(data.at[idx, 'height'],2))
print("closest_2 weight:", round(data.at[idx, 'weight'],2))
```

Output?

```
closest_2 index: 9270
closest_2 height: -0.04
closest_2 weight: 1.26
```

We locate `closest_2` in the scatterplot using its height and weight. Since that data point is orange, we predict that Kai also belongs to class 1 (female).

K-Nearest Neighbors

In the example above, the nearest neighbor `closest_2` is not the only data point that's close to Kai. The second nearest data point `closest_1` is almost the same distance away. Rather than relying on just one nearby data point, we can make a more reliable prediction by considering several of the closest neighbors. This approach is called the *k-nearest neighbors* or *kNN* algorithm.

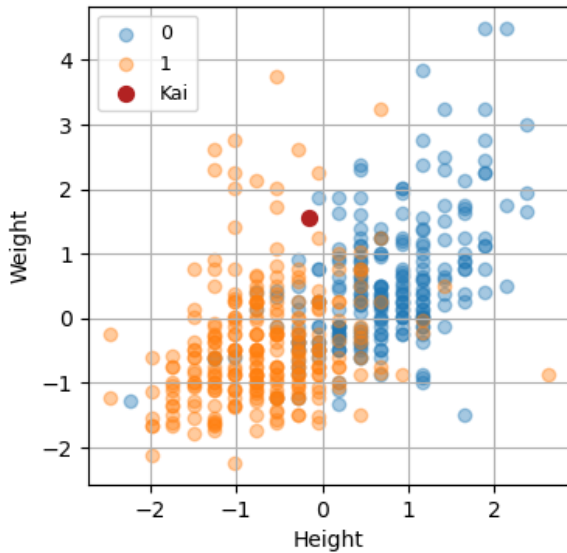
Instead of assigning Kai to the class of a single nearest neighbor, the k-nearest neighbors algorithm looks at the k closest labeled data points. The predicted class is the one that appears most often among these neighbors.

We display the scatterplot with Kai's data point again:

```
kai_height = -0.15
kai_weight = 1.55

plt.figure(figsize=(4,4))
for gender, group in groups:
    plt.scatter(group['height'], group['weight'], alpha=0.4, label=gender)
plt.scatter(kai_height, kai_weight, color='firebrick', s=50, label='Kai')
plt.xlabel('Height')
plt.ylabel('Weight')
plt.legend(fontsize=9)
plt.grid()
plt.show()
```

Output?



We notice that the closest neighbors to Kai's data point have height between -0.6 and 0.2 and weight between 1 and 2 . We locate them in the `sample` dataset and show them in the plot.

```
condition = ((sample['height'] > -0.6) & (sample['height'] < 0.2) &
             (sample['weight'] > 1.25) & (sample['weight'] < 2))

subset_rows = sample.loc[condition]
print("Subset of rows based on locations closest to Kai:")
subset_rows
```

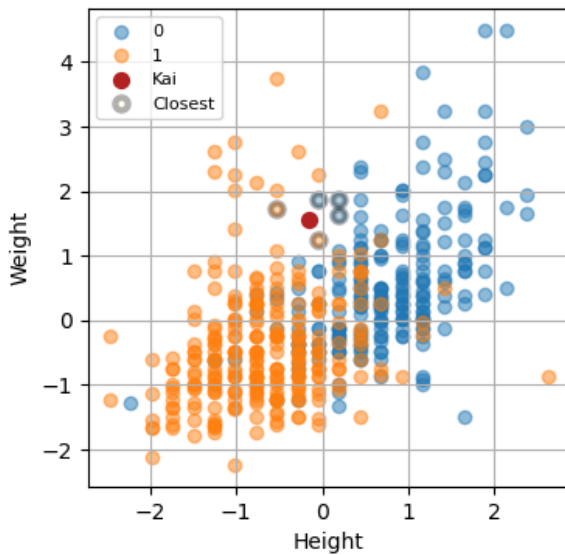
Output?

Subset of rows based on locations closest to Kai:

	height	weight	gender
15716	0.198044	1.629668	0
3833	-0.044330	1.879169	0
11902	-0.529079	1.729469	1
9270	-0.044330	1.255416	1
11659	0.198044	1.879169	0

```
plt.figure(figsize=(4,4))
for gender, group in groups:
    plt.scatter(group['height'], group['weight'], alpha=0.5, label=gender)
plt.scatter(kai_height, kai_weight, color='firebrick', s=50, label='Kai')
plt.scatter(subset_rows['height'], subset_rows['weight'], color = 'cornsilk', alpha=0.3, l
plt.xlabel('Height')
plt.ylabel('Weight')
plt.legend(fontsize=8)
plt.grid()
plt.show()
```

Output?



We see that out of the five closest neighbors to Kai's data point, two are class 1 and three are class 0.

In KNN, the predicted class is the one that appears most often among these neighbors. Therefore, we predict that Kai belongs in class 0.

Using only one nearest neighbor, Kai is predicted to belong to class 1 because the closest observation belonged to class 1. However, when we consider the five nearest neighbors, 3/5 of the nearby observations belonged to class 0. As a result, the predicted class changed to class 0.

KNN makes predictions based on the *majority vote* of the nearest neighbors rather than relying on a single observation. This makes the prediction less sensitive to an individual data point that may not represent the surrounding neighborhood.

Choosing K

The value of k determines how many nearby data points are used to make a prediction.

- If k is too small, the prediction depends on only a few data points. Some unusual data points can have a large influence on the predicted result.
- If k is too large, the prediction includes many data points that may be too far from the new data point and may not be very similar to it. These more distant data points can outweigh the influence of the nearby data points.

In Kai's example above, using $k=1$ predicted class 1, while using $k=5$ predicted class 0. This shows that different values of k can lead to different predictions.

There is no single best value of k . The choice depends on the dataset and the problem being solved. For binary classification, where there are two categories, an odd value of k is often chosen to reduce the chance of a tie during the majority vote. And in practice, we often try several values of k and choose the one that gives the most accurate predictions.

Decision Boundary

So far we have used the k -nearest neighbor algorithm to predict the class of one new data point. For example, for Kai's data point, we predict that Kai belongs to class 0.

Now imagine many new data points at every location in the scatterplot, covering the entire plot. For each location we apply the k -nearest neighbor algorithm to predict a class, and we end up with predictions for all locations of the scatterplot. When we plot all the predictions for all locations on the plot, we will see that the plot is divided into regions, where all data points of the region belong to class 0 or all data points of the region belong to class 1.

The following code generates a plot showing the regions predicted to belong to class 0 and class 1, when $k=5$.

The code uses a Python machine learning library to generate the plot. Understanding this library is beyond the scope of this course, so you do not need to study the code. Instead, focus on understanding what the resulting plot shows.

```
# create the features and target variable for classification
X = sample[['height', 'weight']] # input features
y = sample['gender']             # label or target variable

# use Python machine learning modules to create a classification model
from sklearn.neighbors import KNeighborsClassifier
from matplotlib.colors import ListedColormap
# the model is created with the number of neighbors k set to 5
knn = KNeighborsClassifier(n_neighbors=5)
# the model is given the input features and the target variable
# so it can learn the relationship between the features and the target variable
# and be able to calculate the distance between new data points and the input features
knn.fit(X, y)

# create a mesh of new data points over the range of the input features
x_min, x_max = X['height'].min() - 0.5, X['height'].max() + 0.5
y_min, y_max = X['weight'].min() - 0.5, X['weight'].max() + 0.5

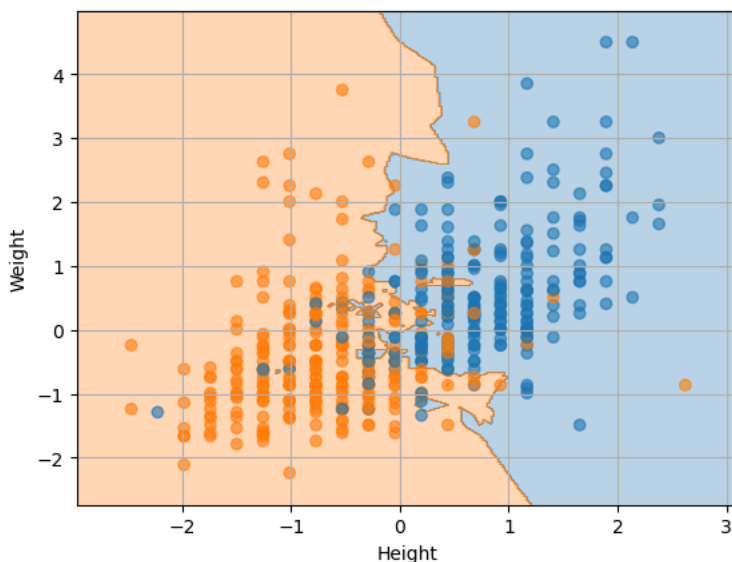
xx, yy = np.meshgrid( np.arange(x_min, x_max, 0.02),
                      np.arange(y_min, y_max, 0.02))

# the model predicts the class for each new data point
grid = pd.DataFrame({'height': xx.ravel(),
                    'weight': yy.ravel()})

Z = knn.predict(grid)
Z = Z.reshape(xx.shape)

# plot the resulting classification regions and the input features
cmap = ListedColormap(['#1f77b4', '#ff7f0e'])
plt.contourf(xx, yy, Z, alpha=0.3, cmap=cmap)
plt.scatter(X['height'], X['weight'], c=y, cmap=cmap, alpha=0.6)
plt.xlabel("Height")
plt.ylabel("Weight")
plt.grid()
plt.show()
```

Output?



In the plot above, we see both the scatterplot of height vs weight data and the predicted regions for each class. The orange regions represent predictions of class 1 and the blue regions represent predictions of class 0. The line separating each orange and blue region is the *decision boundary*. Any new data point on one side of the boundary is predicted to belong to class 0, while any new data point on the other side is predicted to belong to class 1.

A decision boundary is a useful way to visualize how a classification algorithm makes predictions. Instead of predicting the class of one new data point at a time, it shows the predicted class for every possible every possible combination of height and weight.

Automating the Nearest Neighbor Method

Up to now we've identified the nearest neighbors of a new data point by visually inspecting the scatterplot. This works for a small dataset, but it's not practical for larger datasets. Instead, we can use Python to automate the process by:

- calculating the distance between the new data point and every data point in the dataset
- sorting the distances from smallest to largest
- selecting the k data points with the shortest distances

First we write a function to calculate the distance between the new data point and one data point in the dataset

```
# calculate the distance between a new data point and one data point in the dataset
# new_point is a new data point with x and y values (height and weight)
# row is a row or one data point in the dataset with x and y values (height and weight)
def distance_from_point(new_point, row):
    # convert each data point into an array
    new_point = np.array(new_point) # x1, y1
    row = np.array(row)             # x2, y2
    # subtract the 2 arrays
    difference = new_point - row     # x1 - x2, y1 - y2
    # square each difference
    squared_difference = difference ** 2 # (x1 - x2)^2, (y1 - y2)^2
    # add the squared differences and take the square root
    D = np.sqrt(np.sum(squared_difference))
    return D
```

Output?

Next we write a function to go through each row of the `sample` dataset and find the distance between the new data point and the data point in the row. The function stores the distances in a list and return the list.

```
def find_distances(new_point, dataset):
    # create a list to store the distances
    distances = []
    for row in dataset:
        # calculate the distance between the new point and the current point
        a_distance = distance_from_point(new_point, row)
        distances.append(a_distance)
    return distances
```

Output?

Now we use Kai's data point from the example above and find the 5 nearest neighbors.

```
kai_height = -0.15 # same location as before
kai_weight = 1.55
kai_point = [kai_height, kai_weight]

# create a DataFrame with the features (height and weight) from sample
features = sample[['height', 'weight']]
# find all the distances
distances = find_distances(kai_point, features.values)
print(len(distances), "distances calculated")
```

Output?

500 distances calculated

```
results = sample.copy()
results["Distance"] = distances

k = 5
k_nearest_neighbors = results.sort_values(by='Distance').head(k)
print("k nearest neighbors to Kai's point:")
k_nearest_neighbors
```

Output?

k nearest neighbors to Kai's point:

	height	weight	gender	Distance
9270	-0.044330	1.255416	1	0.312963
3833	-0.044330	1.879169	0	0.345715
15716	0.198044	1.629668	0	0.357046
11902	-0.529079	1.729469	1	0.419416
11659	0.198044	1.879169	0	0.479048

The calculated five nearest neighbors to Kai are the same five nearest neighbors that we identified earlier by visually inspecting the scatterplot.

We can also automate the prediction, since we know the five nearest neighbors.

```
# function to find the majority class among the k nearest neighbors
def majority(k_nearest):
    # find the count of 1's
    ones = np.sum(k_nearest.gender == 1)
    # find the count of 0's
    zeros = np.sum(k_nearest.gender == 0)
    # return the larger of the two
    if ones > zeros:
        return 1
    else:
        return 0

print("Predicted gender for Kai:", majority(k_nearest_neighbors))
```

Output?

Predicted gender for Kai: 0

We have now automated the k-nearest neighbors algorithm. Given any new data point, our code can calculate the distance to every data point in the dataset, identify the k nearest neighbors, and use their majority class to make a prediction.

Summary

In this chapter, we learned how the k-nearest neighbors (KNN) algorithm classifies new data by comparing it with nearby labeled data. We explored distance calculations, choosing the value of k, decision boundaries, and how to automate the process of finding the nearest neighbors and making predictions.

In the next chapter, we will use the same distance calculations and nearest-neighbor search to build a complete k-nearest neighbors classifier.

16: Classifier

In the previous chapter we learned the basic ideas behind classification and the k-nearest neighbors (KNN) algorithm.

In this chapter, we will see how to separate data into training and testing sets. Then, using the same automated steps from the previous chapter, we implement a KNN *classifier* that can make predictions on a variety of datasets, including ones with more than two features.

Training and Testing

Two important steps in working with an algorithm to make predictions are:

- *Training* the algorithm by providing it with labeled data that it will use to make predictions. The labeled dataset is called the *training dataset*.
- *Testing* the algorithm on new data to measure how accurately it makes predictions. The new dataset is called the *testing dataset*.

For KNN, the algorithm is the sequence of steps to find the k nearest neighbors of a new data point and use the majority class among those neighbors to make the prediction.

Before we train and test an algorithm, we begin with a single dataset that has been gathered and prepared. Then we divide this input dataset into two parts: a training dataset and a testing dataset.

Gather Data

We use the same `cdc` dataset as in the previous chapter. The original dataset contains 20,000 rows. Because KNN compares every testing row with every training row, classifying all 20,000 rows takes a long time. We will randomly select 3,000 rows so that the classifier runs more quickly while still using a large and varied sample of the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# read input data from file
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_
health = pd.read_csv(url)
# random sample of 3000 rows
health = health.sample(n=3000, random_state=1)
```

Output?

Prepare Data

We take the same steps to prepare data as in Chapter 15: convert `gender` values to numbers, change `height` and `weight` to standard units.

```
# select only the features we want to use for classification and form a new dataset
data = health[['height', 'weight', 'gender']].copy()
# convert gender to numeric values
data['gender'] = data['gender'].map({'m': 0, 'f': 1})
# convert height and weight to standard units
data['height'] = (data['height'] - data['height'].mean()) / data['height'].std()
data['weight'] = (data['weight'] - data['weight'].mean()) / data['weight'].std()
```

Output?

Create Training and Testing Data

We use random sampling to create the two datasets. First we randomly shuffle all the rows of the dataset, in the same way that we randomly shuffle data for hypothesis testing in Chapter 10. Then we assign some of the rows to the training dataset and the remaining rows to the testing dataset. For this chapter we will use half of the rows for training and half for testing.

```
# shuffle all the rows
shuffled_data = data.sample(frac=1)
# take first half for training set
training_set = shuffled_data.iloc[:len(shuffled_data)//2].copy()
# rename the gender column to _class, a requirement for our classifier
training_set = training_set.rename(columns={'gender': '_class'})
# take second half for testing set
testing_set = shuffled_data.iloc[len(shuffled_data)//2:].copy()
# rename the gender column to _class, a requirement for our classifier
testing_set = testing_set.rename(columns={'gender': '_class'})
```

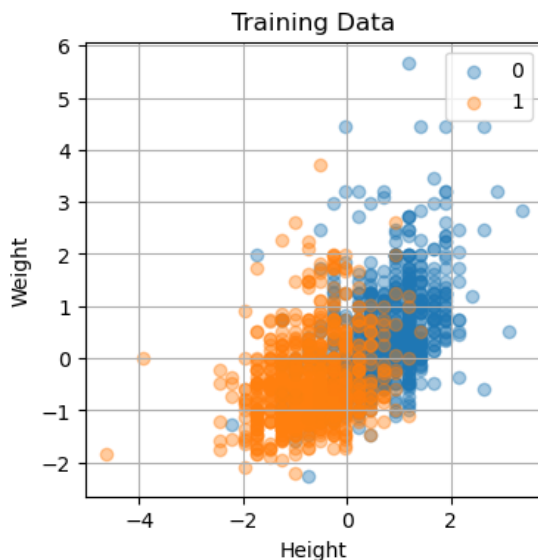
Output?

We confirm the training dataset by plotting the `height` vs `weight`.

```
plt.figure(figsize=(4,4))

groups = training_set.groupby('_class')
for name, group in groups:
    plt.scatter(group['height'], group['weight'], alpha = 0.4, label=name)
plt.xlabel('Height')
plt.ylabel('Weight')
plt.legend()
plt.title('Training Data')
plt.grid()
plt.show()
```

Output?



The scatterplot shows only the training data. When KNN makes a prediction, it searches only the training dataset for the nearest neighbors. The testing data are not used to make predictions because we want them to represent new, unseen data. After the predictions have been made, we compare them with the actual classes in the testing dataset to measure the classifier's accuracy.

Implement a Classifier

We now have the necessary datasets to create and test a simple classifier. We use the training and testing datasets above to train and test the classifier.

To write the code for the classification algorithm, we follow the steps that we've used in Chapter 15:

1. Find the distance between the new data point and all the existing data points in the dataset.
2. Use these distances to find the k points that are closest (the shortest distance) to the new data point.
3. Find the class that occurs the most in the k nearest data points. This will be our predicted class for the new data point.

We bring in the `distance_from_point` function and `find_distances` function from Chapter 15.

```
# calculate the distance between a new data point and one data point in the dataset
# - new_point is a new data point with x, y coordinates on the scatterplot
# - row is a row or one data point in the dataset, with x, y coordinates on the scatterplot
# return the distance between the two data points
def distance_from_point(new_point, row):
    # convert each data point into an array
    new_point = np.array(new_point)    # x1, y1
    row = np.array(row)                 # x2, y2
    # subtract the 2 arrays
    difference = new_point - row        # x1 - x2, y1 - y2
    # square each difference
    squared_difference = difference ** 2    # (x1 - x2)^2, (y1 - y2)^2
    # add the squared differences and take the square root
    D = np.sqrt(np.sum(squared_difference))
    return D

# find the distances between a new data point and all the data points in the dataset
# - new_point is a new data point with x and y values
# - dataset has rows of data features, with x and y values
# return the list of distances
def find_distances(new_point, dataset):
    # create a list to store the distances
    distances = []
    for row in dataset:
        # calculate the distance between the new point and the current point
        a_distance = distance_from_point(new_point, row)
        distances.append(a_distance)
    return distances
```

Output?

We also add one new function `find_k_nearest_neighbors` and bring in the `majority` function from Chapter 15.

```
# find the k nearest neighbors of a new data point in the dataset
# - new_point is a new data point with x and y values (height and weight)
# - dataset is the DataFrame containing the training data
# - k is the number of nearest neighbors to find
# return a DataFrame containing the k nearest neighbors and their distances
def find_k_nearest_neighbors(new_point, dataset, k):
    # extract the features of each row of data
    features = dataset.drop(columns='_class')
    # find all distances
    distances = find_distances(new_point, features.values)
    # create a new DataFrame and add the distances to it
    results = dataset.copy()
    results["Distance"] = distances

    # sort the dataset by 'Distance', then take the top k rows
    k_nearest_neighbors = results.sort_values(by='Distance').head(k)
```

```
    return k_nearest_neighbors

# find the majority class of the k nearest neighbors
# - k_nearest is a DataFrame containing the k nearest neighbors and their distances
# return the predicted class
def majority(k_nearest):
    # find the count of 1's
    ones = np.sum(k_nearest._class == 1)
    # find the count of 0's
    zeros = np.sum(k_nearest._class == 0)
    # return the larger of the two, or the majority class
    if ones > zeros:
        return 1
    else:
        return 0
```

Output?

We have now automated each step of the KNN algorithm. We've written the functions to calculate distances, find the k nearest neighbors, and determine the majority class. The final function, `classify`, is the classifier. When we call the classifier, it calls the other functions in turn to carry out each step of the KNN algorithm and return the predicted class.

```
# k-Nearest Neighbors Classifier. A function that accepts as input:
# - the new data point,
# - the DataFrame of existing data,
# - the number of nearest neighbors
def classify(new_point, dataset, k):
    k_nearest = find_k_nearest_neighbors(new_point, dataset, k)
    return majority(k_nearest)
```

Output?

Test the Classifier

We start by testing the classifier with a new data point.

```
new_point = [2, 2] # x,y coordinate of the new data point in the scatterplot

# run the classifier on the new data point and print the predicted class
print("Predicted class:", classify(new_point, training_set, 5))
```

Output?

Predicted class: 0

Visually checking the location (2,2) in the scatterplot, we see that a prediction of class 0 is expected.

Next we use the testing data with our classifier.

First we save the `_class` (gender) data from `testing_set`, then we remove the `_class` data from the `testing_set`. The remaining columns contain only the feature values that the classifier will use to make its predictions.

```
actual = testing_set["_class"]
test_set = testing_set.drop(columns=['_class'])
```

Output?

Then we test the classifier with every row of the `test_set`. Each prediction for a data point in `test_set` is saved in a list of predicted values.

```
# create a list to store the predicted classes
predicted = []
# repeat for each row in the test set
```

```
for index, row in test_set.iterrows(): # iterrows() returns the index and row of test_set
    # run the classifier on the row and append the predicted class to the list
    predicted.append(classify(row, training_set, 5))
print(len(predicted), "predictions made")
```

Output?

1500 predictions made

Find Prediction Accuracy

We store the `actual` data and the `predicted` data in a DataFrame and print the first five rows.

```
results = pd.DataFrame({'Actual': actual, 'Predicted': predicted})
results.head(5)
```

Output?

	Actual	Predicted
18280	1	1
10229	1	1
17305	0	0
9053	0	0
4971	1	1

Since the data is randomly shuffled before being divided into the training and testing sets, the results above may differ slightly with each run of this example. The predicted classes for individual data points may be different, but the overall accuracy should be similar.

We calculate the percent accuracy by finding the ratio of correct predictions over the total predictions.

```
# add all the rows where Actual is the same as Predicted
correct = (results.Actual == results.Predicted).sum()
# find the ratio of correct predictions over total predictions as a percentage
print("Accuracy:", round(correct / len(results) * 100), "%")
```

Output?

Accuracy: 84 %

The classifier correctly predicted the gender for a high percentage of the people in the testing dataset.

Looking at the female and male ratios in the starting dataset of 3000 people, we find the count and the percentage of `f` and `m` values:

```
display(health["gender"].value_counts())
display(health["gender"].value_counts(normalize=True) * 100)
```

Output?

```
gender
f    1570
m    1430
Name: count, dtype: int64
```

```
gender
f    52.333333
m    47.666667
```

Name: proportion, dtype: float64

We see that if a classifier always predicted female, it would be correct about 52% of the time. Our classifier achieves a much higher accuracy, showing that it has learned patterns in the data rather than simply predicting the most common class.

The classifier correctly predicts the gender for a high percentage of the people in the testing dataset, but it is not perfect because height and weight alone do not completely distinguish males from females. Many people have similar heights and weights, so some data points are difficult to classify correctly.

Test the Classifier with a Different Dataset

In the previous example we use the classifier to predict gender by using height and weight. One advantage of KNN is that the algorithm does not depend on a particular dataset. As long as we provide a training dataset with feature values and a class label, we can use the same classifier to make predictions on completely different data.

To demonstrate this, we will use the cars dataset that we've used in previous chapters. Instead of predicting gender, we will predict whether a car is fuel efficient. The features will be horsepower, weight, and cylinders, and the classifier will indicate whether the car is fuel efficient or not. Note that we do not need to change the classifier itself, we simply provide it with a different dataset.

```
url = "https://raw.githubusercontent.com/DeAnzaDataScience/CIS11/refs/heads/main/datasets_cars"
cars = pd.read_csv(url)
display(cars.head())
print("Number of rows:", cars.shape[0])
```

Output?

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

Number of rows: 392

We create a DataFrame with only the features and target we want.

```
car_data = cars[["horsepower", "weight", "cylinders", "mpg"]].copy()
```

Output?

We want to predict whether a car is fuel efficient. Since fuel efficiency is measured by mpg, we will use mpg to define our classes. However, a classifier can only predict categorical data, and mpg is continuous quantitative data, so we convert mpg into two categories:

- 0: mpg < 25, not fuel efficient
- 1: mpg >= 25, fuel efficient

```
# Convert mpg to binary classification: 0 if < 25, 1 if >= 25
car_data['mpg'] = (car_data['mpg'] >= 25).astype(int)
car_data.head()
```

Output?

	horsepower	weight	cylinders	mpg
0	130.0	3504.0	8	0

	horsepower	weight	cylinders	mpg
1	165.0	3693.0	8	0
2	150.0	3436.0	8	0
3	150.0	3433.0	8	0
4	140.0	3449.0	8	0

In the previous example we used two features, height and weight, to predict gender. In this example, we will use *three* features: horsepower, weight, and cylinders to predict whether a car is fuel efficient. Even though the dataset and the number of features are different, the KNN algorithm does not change. The same classifier works without modifications on completely different datasets, even if they have a different number of features.

Although we first developed the algorithm using two features, the code is not limited to two dimensions. The `distance_from_point` function works with arrays of any length, so the same classifier can be used with datasets that have two features, three features, or many more.

We continue preparing the data:

- change the `mpg` label to `_class` as required by our classifier
- convert all the features to standard units

```
# rename the mpg column to _class
car_data = car_data.rename(columns={'mpg': '_class'})
```

Output?

```
# convert all the features to standard units
car_data['horsepower'] = (car_data['horsepower'] - car_data['horsepower'].mean()) / car_data['horsepower'].std()
car_data['weight'] = (car_data['weight'] - car_data['weight'].mean()) / car_data['weight'].std()
car_data['cylinders'] = (car_data['cylinders'] - car_data['cylinders'].mean()) / car_data['cylinders'].std()
```

Output?

```
car_data.head()
```

Output?

	horsepower	weight	cylinders	_class
0	0.663285	0.619748	1.482053	0
1	1.572585	0.842258	1.482053	0
2	1.182885	0.539692	1.482053	0
3	1.182885	0.536160	1.482053	0
4	0.923085	0.554997	1.482053	0

We divide the data into the training and testing sets.

```
shuffled_data = car_data.sample(frac=1)
training_set = shuffled_data.iloc[:len(shuffled_data)//2].copy()
testing_set = shuffled_data.iloc[len(shuffled_data)//2:].copy()
```

Output?

We save the `_class` column from the `testing_set`. Then we remove the `_class` column from the testing data.

```
# save the actual target or `_class`, then drop `_class` from the testing_set
```

```
actual = testing_set._class
test_set = testing_set.drop(columns='_class')
test_set.head()
```

Output?

	horsepower	weight	cylinders
277	-0.947474	-0.991973	-0.862911
377	-0.973454	-1.192113	-0.862911
139	-0.557775	-0.893080	-0.862911
317	-0.765614	-0.512812	-0.862911
330	0.715245	-0.079567	0.309571

Now that the data has been prepared, we use the classifier to predict the fuel efficiency of the cars.

```
predicted = []
for index, row in test_set.iterrows():
    predicted.append(classify(row, training_set, 5))

# combine the actual and predicted into a DataFrame,
# and find the number of correct predictions
correct = (actual == predicted).sum()
print("Accuracy:", round(correct / len(actual) * 100))
```

Output?

Accuracy: 84

We see that the classifier has high accuracy on a completely different dataset. This demonstrates that the KNN algorithm is not tied to a particular dataset and can be reused to solve different classification problems.

Summary

In this chapter, we built a KNN classifier step by step by combining several functions into a single classifier that predicts the class of new data. We used training and testing datasets to measure its accuracy and saw that the same classifier can be applied to completely different datasets without changing the algorithm.

KNN is one of many machine learning algorithms. Machine learning is a foundation of modern AI because it enables computers to learn patterns from data and make predictions.

INDEX

- A**
- accuracy
 - 16: Classifier
 - algorithm
 - 15: Classification
 - 16: Classifier
 - 2: Numbers and Strings
 - alternative hypothesis
 - 9: Hypothesis Testing
 - arithmetic calculations
 - 2: Numbers and Strings
 - arrays
 - 4: Sequences and Tables
 - assignment operator
 - 2: Numbers and Strings
 - association
 - 5: Plotting
 - average
 - 12: The Mean
 - average mpg
 - 5: Plotting
- B**
- bootstrap
 - 11: Estimation
- C**
- car dataset
 - 5: Plotting
 - Central Limit Theorem
 - 12: The Mean
 - classification
 - 15: Classification
 - 16: Classifier
 - code example
 - 2: Numbers and Strings
 - conditional statements
 - 7: Conditionals, Iterations, and Probabilities
 - confidence interval
 - 11: Estimation
 - confounding variable
 - 10: Two Sample Tests
 - control group
 - 10: Two Sample Tests
 - correlation
 - 13: Linear Regression
 - correlation_coefficient
 - 13: Linear Regression
- D**
- data analysis
 - 1: Introduction to Data Science
 - 10: Two Sample Tests
 - 15: Classification
 - 6: Table Manipulation
 - 8: Sampling
 - data manipulation
 - 4: Sequences and Tables
 - data preparation
 - 16: Classifier
 - data science
 - 1: Introduction to Data Science
 - 3: Data Types
 - 4: Sequences and Tables
 - 6: Table Manipulation
 - 8: Sampling
 - data trends
 - 1: Introduction to Data Science
 - 5: Plotting
 - data types
 - 2: Numbers and Strings
 - 3: Data Types
 - 4: Sequences and Tables
 - 7: Conditionals, Iterations, and Probabilities
 - data visualization
 - 1: Introduction to Data Science
 - 15: Classification
 - 4: Sequences and Tables
 - 6: Table Manipulation
 - data_analysis
 - 13: Linear Regression
 - data_visualization
 - 13: Linear Regression
 - dataframe
 - 6: Table Manipulation
 - dataframes
 - 4: Sequences and Tables
 - dataset
 - 11: Estimation
 - 15: Classification
 - 6: Table Manipulation
 - 8: Sampling
 - decision making
 - 1: Introduction to Data Science
 - delivery
 - 6: Table Manipulation
 - descriptive statistics
 - 6: Table Manipulation
- E**
- error handling
 - 2: Numbers and Strings
 - estimation
 - 11: Estimation
 - exploratory data analysis
 - 10: Two Sample Tests
- F**
- features
 - 15: Classification
 - float
 - 3: Data Types
 - foundational concepts
 - 7: Conditionals, Iterations, and Probabilities
 - fuel efficiency
 - 5: Plotting
- G**
- groupby
 - 6: Table Manipulation
 - groupby method
 - 5: Plotting
- H**
- hypothesis testing
 - 10: Two Sample Tests
 - 11: Estimation
 - 9: Hypothesis Testing
- I**
- insights
 - 1: Introduction to Data Science
 - int
 - 3: Data Types
- K**
- KNN
 - 16: Classifier
- L**
- line plots
 - 5: Plotting
 - linear regression
 - 14: Root Mean Squared Error
 - linear_regression
 - 13: Linear Regression
 - loops
 - 7: Conditionals, Iterations, and Probabilities
- M**
- machine learning
 - 1: Introduction to Data Science
 - 15: Classification
 - 16: Classifier
 - mean
 - 12: The Mean
 - Mean Squared Error
 - 14: Root Mean Squared Error
 - measures of spread
 - 12: The Mean
 - MSE
 - 14: Root Mean Squared Error
- N**
- negative association
 - 5: Plotting
 - negative_correlation
 - 13: Linear Regression
 - null hypothesis
 - 10: Two Sample Tests
 - 9: Hypothesis Testing
 - numbers
 - 3: Data Types
 - numpy
 - 4: Sequences and Tables

O

observed data

[9: Hypothesis Testing](#)

output

[2: Numbers and Strings](#)

P

pandas

[6: Table Manipulation](#)

parameter

[11: Estimation](#)

percentiles

[11: Estimation](#)

population

[11: Estimation](#)

[8: Sampling](#)

population parameter

[12: The Mean](#)

positive association

[5: Plotting](#)

positive_correlation

[13: Linear Regression](#)

prediction error

[14: Root Mean Squared Error](#)

predictions

[1: Introduction to Data Science](#)

[16: Classifier](#)

print function

[3: Data Types](#)

probability

[7: Conditionals, Iterations, and Probabilities](#)

probability sample

[8: Sampling](#)

Python

[15: Classification](#)

[2: Numbers and Strings](#)

[4: Sequences and Tables](#)

[7: Conditionals, Iterations, and Probabilities](#)

Python functions

[3: Data Types](#)

Q

quantitative data

[5: Plotting](#)

R

random sample

[8: Sampling](#)

random samples

[9: Hypothesis Testing](#)

random variation

[9: Hypothesis Testing](#)

randomness

[7: Conditionals, Iterations, and Probabilities](#)

regression

[13: Linear Regression](#)

[14: Root Mean Squared Error](#)

[15: Classification](#)

residual plot

[14: Root Mean Squared Error](#)

residuals

[14: Root Mean Squared Error](#)

RMSE

[14: Root Mean Squared Error](#)

Root Mean Squared Error

[14: Root Mean Squared Error](#)

S

sample

[12: The Mean](#)

[8: Sampling](#)

sample comparison

[10: Two Sample Tests](#)

sampling

[8: Sampling](#)

sampling variability

[11: Estimation](#)

scatterplot

[14: Root Mean Squared Error](#)

scatterplots

[13: Linear Regression](#)

simulation

[7: Conditionals, Iterations, and Probabilities](#)

[8: Sampling](#)

[9: Hypothesis Testing](#)

standard deviation

[12: The Mean](#)

statistical analysis

[11: Estimation](#)

[6: Table Manipulation](#)

statistical significance

[10: Two Sample Tests](#)

[9: Hypothesis Testing](#)

statistical techniques

[1: Introduction to Data Science](#)

statistics

[12: The Mean](#)

[4: Sequences and Tables](#)

[7: Conditionals, Iterations, and Probabilities](#)

[8: Sampling](#)

strings

[3: Data Types](#)

supervised learning

[16: Classifier](#)

T

test statistic

[9: Hypothesis Testing](#)

testing dataset

[16: Classifier](#)

training dataset

[16: Classifier](#)

treatment group

[10: Two Sample Tests](#)

type function

[3: Data Types](#)

V

variables

[13: Linear Regression](#)

[2: Numbers and Strings](#)

variance

[12: The Mean](#)

GLOSSARY

S

Sample Word 1 | Sample Definition 1

DETAILED LICENSING

Overview

Title: Foundations of Data Science for All

Webpages: 32

All licenses found:

- [CC BY 4.0](#): 81.3% (26 pages)
- [Undeclared](#): 12.5% (4 pages)
- [Public Domain](#): 6.3% (2 pages)

By Page

- Foundations of Data Science for All - [CC BY 4.0](#)
 - Front Matter - [CC BY 4.0](#)
 - TitlePage - [CC BY 4.0](#)
 - InfoPage - [Public Domain](#)
 - Table of Contents - [Undeclared](#)
 - Program Page - [CC BY 4.0](#)
 - Accessibility Statement - [Undeclared](#)
 - Licensing - [CC BY 4.0](#)
 - AI Use - [Undeclared](#)
 - About the Book - [CC BY 4.0](#)
 - About the Authors - [CC BY 4.0](#)
 - 1: Introduction to Data Science - [CC BY 4.0](#)
 - 2: Numbers and Strings - [CC BY 4.0](#)
 - 3: Data Types - [CC BY 4.0](#)
 - 4: Sequences and Tables - [CC BY 4.0](#)
 - 5: Plotting - [CC BY 4.0](#)
 - 6: Table Manipulation - [CC BY 4.0](#)
 - 7: Conditionals, Iterations, and Probabilities - [CC BY 4.0](#)
 - 8: Sampling - [CC BY 4.0](#)
 - 9: Hypothesis Testing - [CC BY 4.0](#)
 - 10: Two Sample Tests - [CC BY 4.0](#)
 - 11: Estimation - [CC BY 4.0](#)
 - 12: The Mean - [CC BY 4.0](#)
 - 13: Linear Regression - [CC BY 4.0](#)
 - 14: Root Mean Squared Error - [CC BY 4.0](#)
 - 15: Classification - [CC BY 4.0](#)
 - 16: Classifier - [CC BY 4.0](#)
 - Back Matter - [CC BY 4.0](#)
 - Index - [Public Domain](#)
 - Glossary - [CC BY 4.0](#)
 - Detailed Licensing - [CC BY 4.0](#)
 - Detailed Licensing - [Undeclared](#)

This page titled [Foundations of Data Science for All \(Nguyen\)](#) is shared under a [CC BY 4.0](#) license and was authored, remixed, and/or curated by Clare Nguyen.