Introduction to Data Science

Ethan Meyers

Table of contents

Welcome

This book gives an introduction to Data Science using the Python programming langauge.

1 Introduction

In this chapter we will discuss what the field of Data Science is, and give a brief history of how the field developed.

This book is your guide to understanding the exciting and increasingly influential field of data science. Whether you're curious about how data shapes our world or are looking to explore the possibilities of data-driven insights, this book will provide you with a foundational understanding of what data science is and why it matters.

1.1 What is Data Science?

Data science is a dynamic and interdisciplinary field that combines techniques and theories from statistics, computer science, and specialized knowledge in various areas to extract valuable knowledge and insights from data [Chapter 1]. This data can come in many forms, whether neatly organized in databases or existing as unstructured information like text or images.

At its core, data science follows a systematic process for analyzing data. This includes a range of crucial steps, starting with data collection and ensuring the data is in a usable state through data cleaning. Once prepared, the data is explored to uncover initial patterns and relationships (data exploration). Data scientists then apply various modeling techniques to identify deeper insights, which need to be carefully interpreted to draw meaningful conclusions. Finally, the findings are communicated effectively to inform decisions and understanding [Chapter 1].

The field of data science has experienced remarkable growth in recent years This surge in prominence can be attributed to several key factors: - The explosion in the amount of data being generated across all sectors, from social media to scientific research. - Significant advancements in computing power, enabling the processing and analysis of these vast datasets. - The development of increasingly sophisticated analytical tools and techniques that allow for more complex and insightful data exploration.

By delving into data science, you can gain practical analytical skills that are applicable across a wide array of fields [Chapter 1, 62]. You'll learn how to approach real-world data, identify key questions, and use data-driven methods to find answers and understand the world around us [Chapter 1, 62]. As a lighthearted starting point, you might hear the quip that "A Data Scientist is a Statistician who lives in San Francisco" [Chapter 1, 11]. While humorous, this

simple definition hints at the combination of statistical thinking with the technological innovation often associated with data science. Throughout this book, we will move beyond simplistic definitions to explore the rich and multifaceted nature of this vital field.

Key points - Despite the fact that humans have been collecting data for millenia, and doing sophisticated analyses of data for centuries, the field of data science" (or at least the name) is relatively new. -

1.2 A brief history of Data Science

1.2.1 A brief history of data

1.2.2 A brief history of Statistics

1.2.3 A brief history of computation

Computational devices also have a long history.

1.2.4 The creation of the field of Data Science

2 Literate programming and reproducible research

The concept of reproducible research has a rich history, rooted in the broader scientific principle that results should be verifiable/reproducible by others. As science became increasingly computational in the late 20th century, the challenge of ensuring reproducibility grew. Early computational research often involved custom scripts, manual data manipulation, and undocumented workflows, making it difficult for others to replicate results—even when code and data were shared.

A foundational influence was Donald Knuth's idea of "literate programming" in the 1980s. Knuth advocated for writing code and documentation together, so that the logic and reasoning behind analyses were transparent and accessible. This philosophy inspired later tools that integrated narrative and computation.

In the 1990s and early 2000s, as computational analyses became more central to fields like genomics, climate science, and economics, the limitations of traditional publishing became apparent. Researchers like Jon Claerbout and David Donoho were early advocates for reproducible computational research, arguing that published results should include the code and data necessary to regenerate all figures and analyses. This led to the development of reproducible research standards and the first "compendia"—bundled packages of code, data, and documentation.

The emergence of tools such as Sweave (for R and LaTeX), Jupyter Notebooks (originally IPython), and RMarkdown in the 2000s and 2010s marked a turning point. These platforms allowed researchers to combine code, results, and explanatory text in a single, executable document. This integration made it much easier to share analyses and ensure that others could reproduce and build upon published work. More recently, Quarto has extended these ideas, supporting multiple programming languages and flexible publishing formats.

Despite advances in computational tools, the scientific community has faced what is now known as the "reproducibility crisis." Over the past decade, numerous studies have revealed that a significant proportion of published scientific findings cannot be independently reproduced or replicated. This crisis has affected a wide range of disciplines, from psychology and medicine to the natural and computational sciences. Contributing factors include selective reporting, lack of transparency in methods and data, insufficient documentation of code, and pressures to publish novel results quickly.

The reproducibility crisis has highlighted the urgent need for more transparent and verifiable research practices. As science became increasingly computational in the late 20th century, ensuring reproducibility grew more challenging. Early computational research often involved custom scripts, manual data manipulation, and undocumented workflows, making it difficult for others to replicate results—even when code and data were shared.

The reproducibility crisis has prompted widespread calls for reform. Funding agencies, journals, and professional societies now increasingly require that data and code be made available, and that research workflows be documented in detail. The adoption of version control systems like Git, preregistration of studies, and open peer review are among the practices being promoted to address these challenges.

Today, reproducible research is a cornerstone of open science. The evolution of reproducible research reflects both technological advances and a growing recognition of the importance of openness, transparency, and trust in scientific discovery. By embracing reproducible practices, the scientific community aims to restore confidence in published results and accelerate the pace of reliable, cumulative knowledge.

2.1 Jupyter notebooks

In this book we will be using Jupyter notebooks to do our data analyses. These notebooks allow us to interleave "code cells" which contain Python code, with "Markdown cells" which contain written explanations for what our analyses are showing.

Jupyter notebooks are a powerful tool for interactive computing and reproducible research. They provide an environment where you can write and execute code, visualize data, and document your workflow all in one place. This makes it easy to experiment with different analyses, see immediate results, and keep a clear record of your work.

A typical Jupyter notebook consists of a sequence of cells. Code cells let you write and run code in languages such as Python, R, or Julia. When you run a code cell, the output—such as tables, plots, or text—is displayed directly below the cell. Markdown cells, on the other hand, are used for formatted text, explanations, equations (using LaTeX), and images. This combination supports a narrative style of analysis, where you can explain your reasoning alongside the code and results.

Jupyter notebooks are widely used in data science, education, and research because they encourage transparency and reproducibility. You can share your notebooks with others, allowing them to rerun your analyses, modify code, and build upon your work. Notebooks can be exported to various formats, including HTML and PDF, making it easy to present your findings.

Throughout this book, you will learn how to use Jupyter notebooks effectively: running code, documenting your process, visualizing data, and sharing your results. By mastering notebooks, you'll gain a valuable skill for modern data science workflows.

2.1.1 Using Jupyter notebooks

To use Jupyter notebooks, you interact with two main types of cells: code cells and markdown cells.

- **Code cells**: These contain executable code (such as Python). To run a code cell, click on it and press `Shift+Enter` (or click the "Run" button in the toolbar). The output will appear directly below the cell. You can edit and rerun code cells as often as you like.
- **Markdown cells**: These are used for formatted text, explanations, equations, and images. To edit a markdown cell, double-click it. After editing, press `Shift+Enter` to render the formatted text.

Basic workflow: 1. Add a new cell using the "+" button or menu. 2. Choose the cell type (code or markdown) from the toolbar or menu. 3. Write your code or text. 4. Run the cell with `Shift+Enter`.

You can rearrange cells by dragging them, and you can delete or duplicate cells using the cell menu. Notebooks automatically save your work, but you can also save manually (`Ctrl+S`).

Jupyter notebooks support interactive features such as plotting, widgets, and inline visualizations, making them ideal for data exploration and analysis.

3 Python basics

This chapter introduces the fundamental concepts and basic syntax of the Python programming language. All the code covered here is part of the standard Python language and does not require any additional packages or libraries. Thus the Python discussed here forms the foundation for not only analyzing data in Python, but also for writing any kind of Python code.

While the chapter covers several different key concepts and syntax of Python, we focus on a subset of features that are most central for data analysis, rather than covering the full range of Python's capabilities. Becoming proficient in the basic Python covered in this chapter will be important as a basis for writing code in subsequent chapters, so make sure to practice and understand these concepts thoroughly.

By the end of this chapter, you should be comfortable with writing basic Python code, performing simple calculations, and understanding how Python represents and manipulates different types of data. These foundational skills will prepare you for more advanced topics in data analysis that are covered in the rest of the book.

3.1 Expressions

A **Python expression** is **any piece of code that produces a value.**. For example, the following is an expression that simply creates the number 21.

```
21
```

```
21
```

Similarly, an expression could be a series of mathematical operations that evaluate to a number. For example, if we want to add 5 plus 2 and then multiply the result by 6 we can write:

```
6 * (5 + 2)
```

```
42
```

As mentioned above, the defining features of a *python expression* is that it produces a value. Expressions are one of the fundamental building blocks of data analysis and they will appear frequently throughout this book.

Exercise

What would happen if we remove the parenthesis from the expression we ran above and instead run `6 * 5 + 2`. See if you can predict what the result will be and then try it out in Python by running the code in a code cell and see if you get the result you predicted.

Solution

```
6 * 5 + 2
```

```
32
```

The result is 32, which makes sense because in the standard order of mathematical operations, multiplication occurs before addition so we multiply 6 * 5 and get 30, and then we add 2 to get 32.

3.1.1 Mathematical expressions

The expressions shown above were all "mathematical expressions" because they involve calculating numeric quantities. We can also write statements that will do operations on text and other types of data which we will describe more below. But first, let's explore mathematical expressions a bit more. Below is a table of some of the mathematical operations that are part of Python:

Table 3.1: Python mathematical operators

Operation	Symbol	Example	Result
Addition	+	5 + 3	8
Subtraction	-	10 - 4	6
Multiplication	*	7 * 2	14
Division	/	12 / 5	2.4
Exponentiation	**	3 ** 2	9
Remainder	%	10 % 3	1

Exercise

What is the remainder from dividing 365 by 7? Please write some Python code that produces the answer.

Solution

```
365 % 7
```

```
1
```

3.2 Syntax

Syntax is the set of rules that defines how Python code **must** be written. One can think of syntax as the grammar of the Python programming language. In order for Python to be able to run your code, it **must** use the correct syntax. If incorrect syntax is used, then one will get a "syntax error", and the code will not run.

To illustrate this, let's calculate the value of 8 squared (8^2) which hopefully you remember is equal to the value of 64. As shown Table 3.1, if we want to take a value `x` to the power `y` (i.e., to calculate x^y) we use the syntax `x**y`. So, if we wanted to calculate 8^2 we would write the following Python code:

```
8**2
```

```
64
```

Since we have written the correct syntax, the code runs and the result of 64 is calculated as expected.

However, if we accidentially put an extra space between the two `*` symbols, Python will not know how to interpret the expression and we will get a syntax error as shown below:

```
8* *2
```

When there is a syntax error, Python will print out `SyntaxError` and give you an indication where the syntax error has occurred using a `^` symbol.[1] As we can see here, Python is trying to show that the syntax error has occurred due to the extra space between the `*` symbols.

[1]The reason this is a syntax error is because Python interprets a single `*` symbol as a multiplication symbol. Thus it is trying to multiply 8 by another multiplication symbol `*`, which gives an error since one can only multiply two numbers together.

The ability to be able to spot and fix syntax errors is a fundamental skill you will develop as become proficient in analyzing data in Python.

3.3 Assignment statements

An *assignment statement* is a line of code that is used to store a value in a named **variable**. We can then refer back to this variable name to retrieve the value we have stored.

To assign a value to a variable we use the `=` symbol. For example, the following code assigns the value `10` to the variable `a`:

```
a = 10
```

We can then refer back to the variable `a` later in our code to retrive the stored value. For example, if we just write `a` by itself on the last line of our Python code cell, it will print out the value stored in `a`.

```
a
```

```
10
```

As we can see, the value printed out is `10` which is the value we had previously stored in the name `a`.

If we were to assign the name `a` to another value, it will overwrite the previously stored value and `a` will store the new value.

```
a = 21
a
```

```
21
```

We can also do mathematical operations on values stored in variables, such as adding and multiplying variables together. For example, we can assign the variable `h` to store the value 24, and the variable `d` to store the value 7, and then we can multiply these together and store the result in the variable `t`.

```
h = 24
d = 7
t = h * d
t
```

```
168
```

Exercise

In the above code we calculated `t = h * d`. Which of the following do you think will happen to the value stored in `t` if we change the value of h to 3? I.e., if we run the following code, what do you think it will print out?

```
h = 3
t
```

a. The value of `t` will be change to be 21 (i.e., `7 * 3`).
b. The value of `t` will not change and will still contain `168`.
c. Something else will happen (e.g., Python will give an error).

Solution

```
h = 3
t
```

```
168
```

As you can see, the value of `t` did not change. This illustrates an important point that once a value is calculated and stored in a variable it will not change if the variable that were used as part of the calculation are updated!

3.3.1 Variable names

Variable names in Python must follow certain rules:

- Must start with a letter (a-z, A-Z) or an underscore (_), but not a number.
- Can contain letters, numbers, and underscores.
- Cannot contain spaces or special characters (like `@`, `#`, `$`, etc.).
- Cannot be a reserved Python keyword that are part of the Python language (like `for`, `if`, `class`, etc.).

If these rules are not followed, Python will produce a syntax error

It's also important to use meaningful variable names. For example, `t` is technically a valid variable name but it is not descriptive, while `total_hours` is much clearer. Using meaningful names makes your code easier to read and understand.

Exercise

The minimum wage in the United States in 2025 is $7.25. If someone works 40 hours per week for all 52 weeks in a year, what would there yearly earnings be if they are being paid the minimum wage? Please calculate this quantity by creating *meaningful* (i.e., easy to understand) object names for:

1. The minimum wage amount

2. The number of hours worked in a week
3. The number of weeks in a year

Then calculate the total yearly wage and store this result in another meaningful object name, and print out the value stored in this last object.
Hint: Using underscores _ in your object names is highly encouraged to make them more meaningful/readable.

Solution

```
min_wage = 7.25
hours_worked_in_a_week = 40
weeks_in_a_year = 52
yearly_min_wage_earnings = min_wage * hours_worked_in_a_week * weeks_in_a_year
yearly_min_wage_earnings
```

```
15080.0
```

3.4 Comments

Another very useful feature in Python is the ability to add **comments** to your code. Comments are lines in your code that are ignored by Python when your code runs. They are used to explain what your code is doing, make notes to yourself, or leave instructions for others who may read your code in the future.

In Python, you create a comment by starting the line with the # symbol. Anything after the # on that line will be treated as a comment and not executed.

For example:

```
# The code below calculates the number of seconds in a day
seconds_in_a_day = 60 * 60 * 24
```

```
seconds_in_a_day
```

```
86400
```

We will use comments extensively throughout this book to explain what code is doing and to make our code easier to understand. Adding clear comments is a good habit that will help both you and others who read your code in the future, so we strongly encourage you to add comments liberally for all code you write.

3.5 Functions (call expressions)

A **function** is a reusable piece of code that performs a specific task. You can think of a function as a "machine" that takes some input, does something with it, and then gives you an output.

Python comes with many built-in functions that you can use right away, and you can also load in additional functions in packages that other people have written. You can also write own functions, which is a topic we will discuss later in this book.

To use a function, you "call" it by writing its name followed by parentheses. If the function needs information to do its job, you put that information (called "arguments") inside the parentheses.

For example, the `abs()` function take in a number and returns the absolute value of the number.

```
abs(-10)
```

```
10
```

Some functions can take in multiple arguments. When multiple arguments are provided, they are separated by commas within the parentheses. For example, the `min()` function can take several numbers and will return the smallest one:

```
min(10, 2, 87, 5, 90)
```

```
2
```

Another useful function is the `print()` function for displaying multiple pieces of information in a single Jupyter notebook code cell. By default, Jupyter will only display the result of the last line in a code cell. If you want to display multiple values or add custom messages, you can use the `print()` function.

For example, the code below will print the number `2` because `print(2)` is called. The number `3` will also be displayed as output because it's the last expression in the cell. If `print(2)` was not used, only `3` would be displayed. The `print()` function is useful when you want to display multiple values from within a single cell or when you want to output values that are not on the last line.

```
# We need to call print() explicitly here to print the value of
# 2 since it is not on the last line of the code cell

print(2)



# The value of 3 will be printed here without needing to call
# the print() function because it is the last line in the cell

3
```

```
2
```

```
3
```

Exercise

Try using the `print()` function to display both a message and a value in the same output. For example, print the message "The answer is:" followed by the result of `6 * 7`.

Solution

```
print("The answer is:")
6 * 7
```

```
The answer is:
```

```
42
```

```
# We can also print multiple pieces of text on a single line by
# passing multiple arguments to the print() function:

print("The answer is:", 6 * 7)
```

```
The answer is: 42
```

3.6 Data types

Python is able to process many different types of data, referred to as "data types". So far, we have only explored numeric data. Let's continue exploring numerical data in a little more detail and then we will go on to examine other types of data.

3.6.1 Numbers

Python uses two different formats to store numerical data known as "integers" and "floating-point numbers".

- **Integers** (`int`): Whole numbers without a decimal point, such as `5`, `-3`, or `1000`.
- **Floating-point numbers** (`float`): Numbers that have a decimal point, such as `3.14`, `-0.5`, or `2.0`.

We can tell if a number is a floating point number (i.e., a "float") by seeing if there is a decimal point at the end of the number when we print out the number.

```
# This is an integer, which we can tell becaues there is no decimal point
5
```

```
5
```

```
# Although we are dividing two integers, the result is a floating point number
# which we can tell becaues there is a decimal point

10/2
```

```
5.0
```

We can also use the `type()` function to check if a number is an integer or a floating point number.

```
# This is a floating point number

type(5.0)
```

```
float
```

When analyzing the data, usually it does not matter if Python is storing a number as an integer or a floating point number since Python does the math sensibly and converts between integers and floating point numbers as needed. However, internally Python is representing these numbers in quite different ways.

More importantly, one should be aware that there are some limitations to the way Python stores both integers and floats. In particular, both of these types of numbers are represented using a finite amount of memory, so there is a largest number integer that can be represented and a limit to the precision of floating-point numbers. For most practical purposes, these limits are very large, but you may encounter issues with extremely large numbers or with floating-point arithmetic where results are not exactly as expected due to rounding errors.

For example, if we multiply integers that are very large, we can get a `ValueError` which indicates that Python is running into problems representing the result as an integer.

```
# There is a limited size to integers (although the size is pretty large)

1234567 ** 890
```

Similarly, if we try to create a floating-point number with too many decimal points the number will be truncated, although no error is given, so one needs to be careful if very high precision is needed in a calculation.

```
# There is a limited precision to floating point numbers so the last digits are truncated

.12345678901234567890123456789
```

```
0.12345678901234568
```

We can also convert numbers between integers and floating point numbers using the `int()` and `float()` functions. When converting from a floating point number to an integer using the `int()` function, one needs to be aware that the decimal part of the number will be removed (i.e., rounded down to the closest integer)

```
# Convert an integer to a floating point number.  We can see the conversion worked because th

float(5)
```

```
5.0
```

```
# Convert a floating point number to an integer. Note that the decimal part of the number is

int(3.14159)
```

```
3
```

Finally, one should be aware that Python sometimes prints out numbers using scientific notation. Scientific notation is a way of writing very large or very small numbers more compactly, using the letter `e` to indicate "times ten to the power of." For example, `2.5e6` means (2.5 ×10^{6}), or `2,500,000`. Similarly, `3e-09` means (3 ×10^{-9}), or `0.000000003`. Python will automatically use this notation when displaying numbers that are extremely large or small.

```
# The output is in scientific notation
30 / 4000000000
```

```
7.5e-09
```

Exercise

Take the square root of 12 and then square the result; i.e., calculate $(\sqrt{12})^2$. Does Python return the correct result?
Hint: Note that you can calculate the square root of a number by taking a number to the 1/2 power; i.e., $\sqrt{12} = 12^{0.5}$

Solution

```
(12**.5)**2
```

```
11.999999999999998
```

As you can see, there is slight imprecision here so we get a result of 11.999999999999998 rather than a value of 12.

3.6.2 Character strings

A **character string** (or simply "string") is a sequence of characters that are used to represent text, such as words, sentences, or any other sequence of characters. Strings in Python are enclosed in either single quotes (`'...'`) or double quotes (`"..."`). However, your string must start and end with the same quote type; i.e., if the string starts with a single quote it also ends with a single quote, and the same for double quotes.

The following are valid strings in Python:

```
'This is a valid Python string with single quotes'

"This is another valid Python string using double quotes."
```

```
'This is another valid Python string using double quotes.'
```

While using single or double quotes gives the same result, there are cases where it is natural to use one over the other. For example, if your string contains an apostrophe (single quote), it's easier to use double quotes:

```
"This string contains an apostrophe: it's easy to read."
```

```
"This string contains an apostrophe: it's easy to read."
```

And if your string contains double quotes, it is easier to single quotes when creating your string:

```
'She said, "Hello, world!"'
```

```
'She said, "Hello, world!"'
```

We can also perform operations on strings, such as concatenation (joining strings together). For example, to join two strings together, you can use the `+` operator:

```
"water" + "mellon"
```

```
'watermellon'
```

Note that the `+` operator we have used to concatenate strings is the same `+` operator we used to add numbers. This illustrates an important principle that an operator can behave differently depending on the type of data it is used with. In Python, this is called "operator overloading." For numbers, `+` performs addition, while for strings, it performs concatenation (joining the strings together).

Exercise

Above we have seen that the `+` operator can behave differently depending on whether it is operating on numbers or strings. We have also seen that `*` operator is used to multiple two numbers together. Do you think that the `*` operator will also work on strings? Please write down, or say outloud whether you think the `*` operator will work on strings, then see if your prediction is correct by running the following code:

```
'ha' * 5
```

Solution

```
'ha' * 5
```

```
'hahahahaha'
```

As you can see, the `*` operator works on strings by repeating the string the specified number of times. In this case, `'ha' * 5` produces `'hahahahaha'` (which is very amusing).

3.6.2.1 String conversions

We can also convert strings into numbers and numbers into strings. To convert strings into numbers we can again use the `int()` and `float()` functions, but this time we are passing a string as the argument to these functions.

```
int("42")       # Converts the string "42" to the integer 42
```

```
42
```

```
float("3.14")  # Converts the string "3.14" to the float 3.14
```

```
3.14
```

We can see that the output from running these functions are numbers since the output is not in quotes.

We can convert a number into a string using the `str()` function.

```
str(2.5)       # Converts the float 2.5 to the string "2.5"
```

```
'2.5'
```

We can see that the output from running this function is a string since the output is in quotes.

Exercise

Do the following two lines of code produce the same result?

- `10 + 20`
- `int("10" + "20")`

Explain your reasoning then try it in Python to verify your answer is correct.

Solution

```
print(10 + 20)

print(int("10" + "20"))
```

```
30
1020
```

As we can see, the result of running these two pieces of code are different. The first line of code produce the value of 30 since we are simply adding the integers `10` and `20` together. The second line of code first concatenates the strings `"10"` and `"20"` together to create the string `"1020"` and then converts it to the integer `1020`, which is clearly different from the integer `30`.

3.6.2.2 f-strings

An **f-string** (short for "formatted string literal") is a way to embed the values of variables or expressions inside a string. To create an f-string, put the letter `f` before the opening quote, and then include curly braces `{}` around the variables or expressions you want to insert.

For example:

```
name = "Methuselah"
age = 969

f"My name is {name} and I am {age} years old."
```

```
'My name is Methuselah and I am 969 years old.'
```

Exercise

Create three variables: `name`, `age`, and `favorite_color`, and assign them your own name, age and favorite color. Then, use an f-string to print a sentence like:
`"My name is <name>, I am <age> years old, and my favorite color is <favorite_color>."`

Solution

My solution (at the time of writing this book) is below.

```
name = "Ethan"
age = 45
favorite_color = "red"

f"My name is {name}, I am {age} years old, and my favorite color is {favorite_color}."
```

```
'My name is Ethan, I am 45 years old, and my favorite color is red.'
```

3.6.2.3 String methods

A **method** is a function that is attached to a piece of data [2]. There are a number of "string methods" which allow you to perform specific operations on strings, such as changing their case, finding substrings, or replacing text.

You call a method by writing the string (or variable containing a string), followed by a dot (`.`), the method name, and parentheses. For example, the `.upper()` method returns a copy of the string with all letters converted to uppercase:

```
"hello".upper()
```

[2] Or to be more precise, a method is a function attached to an object

```
'HELLO'
```

Here is a table of some particularly useful string methods. Each method returns a new string that is modified as described below.

Method	Example	Description	Result
`.upper()`	`"hello".upper()`	Converts all characters to uppercase	`'HELLO'`
`.lower()`	`"HELLO".lower()`	Converts all characters to lowercase	`'hello'`
`.strip()`	`"  hello  ".strip()`	Removes leading and trailing whitespace	`'hello'`
`.replace(a, b)`	`"ha".replace("a", "o")`	Replace all occurrences of `a` with `b`	`'ho'`
`.count(x)`	`"banana".count("a")`	Counts the number of occurrences of `x`	`3`
`.zfill(n)`	`"42".zfill(5)`	Pads the string with zeros to reach length `n`	`'00042'`
`.find(x)`	`"hello".find("e")`	Returns the index of the first occurrence of `x`	`1`

Exercise

Suppose we have the string `my_sentence = "The quick brown fox jumps over the lazy dog"`. Please use string methods to do the following: 1. Count how many times the letter `e` appears in this sentence. 2. Find the index of the first occurance of the letter `z`.

Solution

```
my_sentence = "The quick brown fox jumps over the lazy dog"

print(my_sentence.count("e"))

my_sentence.find("z")
```

```
3

37
```

Exercise

Suppose again we have the string `my_sentence = "The quick brown fox jumps over the lazy dog"`. Please use the `.replace()` method to replace the word "dog" with the word "canine". Does the string in the `my_sentence` variable change? If not, how could you update the string in the `my_sentence` variable to so that it contains the string "The quick brown fox jumps over the lazy canine"?

Solution

```
my_sentence = "The quick brown fox jumps over the lazy dog"

print(my_sentence.replace("dog", "canine"))

# notice that the variable my_sentence still has the original string
print(my_sentence)

# to update the string in the variable my_sentence we can do the following
my_sentence = my_sentence.replace("dog", "canine")

print(my_sentence)
```

```
The quick brown fox jumps over the lazy canine
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy canine
```

3.6.3 Booleans

A **Boolean** is a data type that can have only two possible values: `True` or `False`. Booleans are used to represent truth values and are very useful for making decisions in your code.

You can create Boolean values directly by writing `True` or `False` (note the capital letters):

```
# Create the Boolean True
True
```

```
True
```

The Boolean `True` is also the same as the integer 1, the Boolean `False` is the same as the integer 0. This means we can do arithmetic on Booleans such as:

```
True + False + True
```

```
2
```

We will use the fact that Booleans can be treated the integers 1 and 0 later in some of our analyses.

3.7 Comparisons

Comparison operators are used to compare values and produce Boolean results. For example, we can assess whether one number is greater than another number:

```
5 > 3
```

```
True
```

If we want to compare whether two values are the same, we use two equal signs `==`. For example, we can see that indeed strings that are created using single quotes are the same as strings created using double quotes by running the following code.

```
"Octothorpe" == 'Octothorpe'
```

```
True
```

Here are some common comparison operators:

Table 3.3: Python comparison operators

Operator	Description	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	7 > 2	True
<	Less than	3 < 1	False
>=	Greater than or equal to	4 >= 4	True
<=	Less than or equal to	2 <= 5	True

Exercise

Is the string `"99"` equal to the integer `99` in Python? Also is `1` equal to `True`? Use the equal to operator (`==`) to do these comparisons and see what result you get.

Solution

```
print("99" == 99)

1 == True
```

```
False

True
```

As we can see, the string `"99"` is not equal to the integer `99` which once again showing strings and integers are not the same thing.
Conversely, the integer `1` is equal to the Boolean `True` again showing that these are identical.

3.8 Data structures

Python provides several built-in data structures that allow you to store and organize collections of data. The most common ones are:

- **Lists**: Ordered, mutable collections of items.
- **Tuples**: Ordered, immutable collections of items.
- **Dictionaries**: Unordered collections of key-value pairs.

We will introduce each of these data structures in the following sections.

3.8.1 Lists

A **list** is an ordered collection of items that can be changed. Lists can contain any type of data, including numbers, strings, or even other lists. Lists are created by placing items inside square brackets `[]`, separated by commas.

For example:

```
my_list = [1, 2, 3, "a", "b", "c", True]

my_list
```

```
[1, 2, 3, 'a', 'b', 'c', True]
```

We can access individual items in a list by their position (called the "index") using square brackets. In Python, indexing starts at 0, so the first item is at index 0, the second at index 1, and so on.

For example, to get the first item in `my_list`, we would use:

```
my_list[0]
```

```
1
```

This returns `1` since the first element in the list (i.e., the element at position 0) is the integer 1.

Likewise, we can get the 6th element (remembering that indexing starts at 0) using:

```
my_list[5]
```

```
'c'
```

We can also change the value of an item in a list by assigning a new value to a specific index. For example, `my_list[0] = 100` will change the first item in the list to 100.

```
my_list[0] = 100

my_list
```

```
[100, 2, 3, 'a', 'b', 'c', True]
```

The fact that we can change items of a list is what makes lists "mutable." This means you can update, add, or remove elements after the list has been created.

Another example of how we can change a list is to use the `append()` method which adds new items to the end of a list.

```
my_list.append("zzz")

my_list
```

```
[100, 2, 3, 'a', 'b', 'c', True, 'zzz']
```

3.8.2 Tuples

A **tuple** is similar to a list in that it is an ordered collection of items, but unlike lists, tuples are **immutable**—meaning their contents cannot be changed after creation. Tuples are created by placing items inside parentheses `()`, separated by commas.

```
my_tuple = (8, 9, "y", "z", False)

my_tuple
```

```
(8, 9, 'y', 'z', False)
```

Similar to lists, we can access individual items in a tuple using square bracket indexing. For example, `my_tuple[2]` will return the third element of the tuple (remember, indexing starts at 0):

```
my_tuple[2]
```

```
'y'
```

However, unlike lists, tuples are "immutable" meaning we can not cannot change the values stored in the tuple after they are created. In particular, if you try to assign a new value to an element of a tuple, Python will produce an error:

```
my_tuple[0] = 100
```

```
TypeError: 'tuple' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[54], line 1
----> 1 my_tuple[0] = 100
TypeError: 'tuple' object does not support item assignment
```

3.8.3 Dictionaries

A **dictionary** is data structure is useful for storing data in a way that allows you to quickly look up values based on a specific key; i.e., -you can think of a dictionary as a "lookup table," where you use a `key` to quickly find the `value` associated with it.

Key	Value
name	Alice
age	30
city	New York

We can do this in Python using:

```
my_dict = {"name": "Alice", "age": 30, "city": "New York"}

my_dict
```

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
```

Note: Keys in a dictionary must be unique and are typically strings or numbers. Values can be of any data type, including numbers, strings, lists, or even other dictionaries.

Accessing Values:

You can access the value associated with a key by using square brackets `[]` with the key inside:

```
print(my_dict["name"])  # Output: Alice
print(my_dict["age"])   # Output: 30
```

If you try to access a key that doesn't exist, Python will raise a `KeyError`.

Adding or Modifying Key-Value Pairs:

Dictionaries are mutable, meaning you can change them after they are created. You can add a new key-value pair to a dictionary or modify an existing one by assigning a value to a key:

```python
# Adding a new key-value pair
my_dict["occupation"] = "Engineer"
print(my_dict)

# Modifying an existing value
my_dict["city"] = "San Francisco"
print(my_dict)
```

Exercise

Create a dictionary to store the number of legs for different animals (e.g., 'dog': 4, 'spider': 8, 'ant': 6). Then, add a new animal, 'cat', with 4 legs to the dictionary. Finally, print the number of legs for 'spider'.

Solution

```python
# Create the animal_legs dictionary
animal_legs = {
    "dog": 4,
    "spider": 8,
    "ant": 6
}
print(f"Original dictionary: {animal_legs}")

# Add 'cat' with 4 legs
animal_legs["cat"] = 4
print(f"Dictionary after adding 'cat': {animal_legs}")

# Print the number of legs for 'spider'
print(f"A spider has {animal_legs['spider']} legs.")
```

3.8.4 Sequences

In Python, a **sequence** refers to an ordered collection of items. Several of the data structures we have alreayd seen, including lists, tuples, and strings, are all sequences. When data is stored in a sequence, we can operate on the data in a consistent manner.

Let's explore some operations that can be performed on any sequence. We'll use a list as an example, but these apply to strings and tuples as well.

```
my_list_sequence = [10, 20, 30, 40, 50]
my_string_sequence = "Hello"
```

Common Sequence Operations:

1. **Indexing**: Accessing an item by its position. Indexing starts from 0 for the first item. `python print(my_list_sequence[0]) # Output: 10 print(my_string_sequence[1]) # Output: 'e'`
2. **Slicing**: Extracting a part of the sequence. Slicing `my_sequence[start:end]` extracts items from `start` up to (but not including) `end`. `python print(my_list_sequence[1:3]) # Output: [20, 30] (items at index 1 and 2) print(my_string_sequence[0:2]) # Output: 'He' (characters at index 0 and 1)`
3. **Length (`len()` function)**: Getting the number of items in a sequence.

```
print(len(my_list_sequence))  # Output: 5
print(len(my_string_sequence)) # Output: 5
```

```
5
5
```

4. **Concatenation (`+` operator)**: Combining two sequences of the same type.

```
list1 = [1, 2]
list2 = [3, 4]
combined_list = list1 + list2
print(combined_list)  # Output: [1, 2, 3, 4]

string1 = "Py"
string2 = "thon"
combined_string = string1 + string2
print(combined_string) # Output: 'Python'
```

```
[1, 2, 3, 4]
Python
```

Note: You cannot concatenate sequences of different types directly (e.g., a list and a string).

5. **Repetition (`*` operator)**: Repeating a sequence a certain number of times.

```
repeated_list = [0, 1] * 3
print(repeated_list)  # Output: [0, 1, 0, 1, 0, 1]
```

```
repeated_string = "Ja" * 7    # Laughing in spanish
print(repeated_string) # Output: 'JaJaJaJaJaJa'
```

```
[0, 1, 0, 1, 0, 1]
JaJaJaJaJaJa
```

These operations provide powerful ways to manipulate and work with ordered data in Python.

Exercise

Given a list `numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`: 1. Extract the sub-list `[3, 4, 5]` using slicing and store it in a variable called `sub_list`. 2. Create a new list called `doubled_sub_list` by concatenating `sub_list` with itself. 3. Print `sub_list` and `doubled_sub_list`.

Solution

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# 1. Extract the sub-list [3, 4, 5]
sub_list = numbers[3:6]
print(f"Sub-list: {sub_list}")

# 2. Create a new list by concatenating sub_list with itself
doubled_sub_list = sub_list + sub_list
print(f"Doubled sub-list: {doubled_sub_list}")
```

Note: dictionaries are **not** sequences because their items are not stored in a specific order and they are accessed by keys rather than by position (index).

3.9 Summary

In this chapter, we introduced the fundamental concepts and basic syntax of Python, focusing on the core features most relevant for data analysis. You learned about expressions, assignment statements, variable naming, comments, functions, and the main data types: numbers, strings, and Booleans. We also covered comparison operators and introduced basic data structures such as lists, tuples, and dictionaries. Mastering these basics will provide a strong foundation for more advanced topics in Python and data science. We strongly encourage you to practice everything we covered in this chapter by completing the exercises that are in the text and the exercises below, and make sure that you understand all of the solutions that are given. We

will use the material covered in this chapter throughout the rest of the book, so we highly recommend that you become comfortable with this material before preceeding to the rest of the book.

3.10 Exercises

3.10.1 Warm-up exercises

Exercise

Try to predict which of the following lines of code produce errors, and try to explain why you believe they will produce errors. Then run these lines in Python to see if your predictions were correct.

1. `5 > = 2`
2. `True**5`
3. `"The cat" == "The Cat"`

Solution

1. `5 > = 2` will produce a `SyntaxError`. The correct operator is `>=` (greater than or equal to), not `> =` with a space. `python # 5 >= 2 # This would be True`
2. `True**5` will evaluate to `1`. `True` is treated as `1` in arithmetic operations, so `1**5` is `1`.

```
print(True**5)
```

```
1
```

3. `"The cat" == "The Cat"` will evaluate to `False` because strings are case-sensitive. The 'c' in "cat" is lowercase, while 'C' in "Cat" is uppercase.

```
print("The cat" == "The Cat")
```

```
False
```

Exercise

Try to predict what the following line of code will evaluate to. Then run it in Python to see if your prediction was correct. Explain the result.

```
True + True + True == 3
```

Solution

```
print(True + True + True == 3)
```

```
True
```

The code evaluates to `True`. In Python, `True` is equivalent to the integer `1` and `False` is equivalent to `0`. So, `True + True + True` is the same as `1 + 1 + 1`, which is `3`. Then, `3 == 3` is `True`.

Exercise

If you run the code `"apple" > "banana"`, it will return `False`. Why is this the case? Experiment with different strings and see if you can figure out how string comparison works.

Solution

String comparison in Python is done lexicographically (alphabetical order). This means Python compares strings character by character, which corresponds to their alphabetical order.
In `"apple" > "banana"`: - Python first compares 'a' (from "apple") with 'b' (from "banana"). - Since 'a' comes before 'b' alphabetically, "apple" is considered "less than" "banana". - Therefore, `"apple" > "banana"` is `False`.
You can test this with other strings:

```
print("zebra" > "apple") #  True, because lowercase 'z' comes after lowercase 'a'
print("car" > "cat")  # False, because 'r' comes before 't' in alphabetic order
print("Apple" > "apple") # False because lowercase letters come before upper case letters
```

```
True
False
False
```

This shows that string comparisons are case-sensitive as well.

Exercise

1. Create two variables, `num1` and `num2`, and assign them the values `15` and `4` respectively.
2. Calculate their sum, difference, product, and quotient (division). Store each result in a separate variable.
3. Try to predict whether each of the variables is a float or an int. Then check your answers using the `int()` and `float()` functions.

Solution

```
# 1. Create variables
num1 = 15
num2 = 4

# 2. Perform calculations
sum_result = num1 + num2
difference_result = num1 - num2
product_result = num1 * num2
quotient_result = num1 / num2

# 3. Print results
print(f"Sum: {sum_result} is an {type(sum_result)}")
print(f"Difference: {difference_result} is an {type(difference_result)}")
print(f"Product: {product_result} is an {type(product_result)}")
print(f"Quotient: {quotient_result} is an {type(quotient_result)}")
```

```
Sum: 19 is an <class 'int'>
Difference: 11 is an <class 'int'>
Product: 60 is an <class 'int'>
Quotient: 3.75 is an <class 'float'>
```

The quotient will be a float (`3.75`) because division `/` always results in a float. All the other results are integers.

Exercise

1. Create a list named `colors` containing the strings “red”, “green”, “blue”, and “yellow”.
2. Print the second element of the list.
3. Change the third element of the list to “purple”.

4. Print the entire updated list.

Solution

```
# 1. Create a list
colors = ["red", "green", "blue", "yellow"]

# 2. Print the second element (index 1)
# Remember that list indexing starts from 0
print(f"The second color is: {colors[1]}")

# 3. Change the third element (index 2) to "purple"
colors[2] = "purple"

# 4. Print the updated list
print(f"Updated list of colors: {colors}")
```

```
The second color is: green
Updated list of colors: ['red', 'green', 'purple', 'yellow']
```

3.10.2 Intermediate exercises

Exercise

1. Create a dictionary named `user_profile` with the following key-value pairs:
   - `"name"`: "Alex"
   - `"age"`: 28
   - `"hobbies"`: A list containing "reading", "hiking", and "coding"
2. Add a new key `"city"` with the value "Toronto" to the `user_profile` dictionary.
3. Access the user's name and their second hobby.
4. Print a message using an f-string: "Alex's second hobby is hiking." using the values from the dictionary.

Solution

```
# 1. Create the dictionary
user_profile = {
    "name": "Alex",
    "age": 28,
    "hobbies": ["reading", "hiking", "coding"]
}

# 2. Add city
user_profile["city"] = "Toronto"
print(f"Updated profile: {user_profile}")

# 3. Access name and second hobby
user_name = user_profile["name"]
# Hobbies is a list, so we access its elements by index
second_hobby = user_profile["hobbies"][1]

# 4. Print the message
print(f"{user_name}'s second hobby is {second_hobby}.")
```

```
Updated profile: {'name': 'Alex', 'age': 28, 'hobbies': ['reading', 'hiking', 'coding'], 'c
Alex's second hobby is hiking.
```

Exercise

You are given a string: `raw_data = "  Product_ID:12345, ProductName:SuperWidget, Price:0050.75  "`. Please do the following:

1. Remove the leading and trailing whitespace from `raw_data`.
2. Extract the `Product_ID` (including the number), `ProductName` (including the name), and `Price` (including the number) into separate variables. You might need string slicing and/or the `.find()` method.
3. Convert the `Price` to a floating-point number.
4. Print the extracted information in a formatted way, like:

   ```
   Product ID: 12345
   Product Name: SuperWidget
   Price: $50.75
   ```

(Hint: You might need to use `.replace()` or other string methods to clean up the extracted parts before printing.)

Solution

```
raw_data = "  Product_ID:12345, ProductName:SuperWidget, Price:0050.75  "

# 1. Remove whitespace
cleaned_data = raw_data.strip()
print(f"Cleaned data: '{cleaned_data}'")

# 2. Extract parts
# Find positions of commas and colons to help with slicing
id_end = cleaned_data.find(",")
product_id_full = cleaned_data[0:id_end] # "Product_ID:12345"

name_start = id_end + 2 # Skip ", "
name_end = cleaned_data.find(",", name_start)
product_name_full = cleaned_data[name_start:name_end] # "ProductName:SuperWidget"

price_start = name_end + 2 # Skip ", "
price_full = cleaned_data[price_start:] # "Price:0050.75"

# Further extract actual values
product_id = product_id_full.split(":")[1]
product_name = product_name_full.split(":")[1]
price_str = price_full.split(":")[1]

# 3. Convert price to float
price_float = float(price_str)

# 4. Print formatted
print(f"Product ID: {product_id}")
print(f"Product Name: {product_name}")
print(f"Price: ${price_float:.2f}") # Format to 2 decimal places
```

```
Cleaned data: 'Product_ID:12345, ProductName:SuperWidget, Price:0050.75'
Product ID: 12345
Product Name: SuperWidget
Price: $50.75
```

This solution uses `split(':')` for simplicity after initial slicing. More robust parsing

could use regular expressions, but that's beyond this chapter. The `.2f` in the f-string formats the float to two decimal places.

3.10.3 Advanced exercises

4 Descriptive statistics and plots

Welcome to the world of data exploration! Now that you've got a handle on Python basics, we're ready to dive into the exciting realm of understanding data. This chapter is all about introducing you to fundamental methods for visualizing and summarizing the information hidden within your datasets.

We'll be focusing on descriptive statistics, which are like trusty tools that help us paint a clear picture of our data. We'll explore how to describe both categorical data (think types and labels) and quantitative data (think numbers and measurements).

Throughout this chapter, we'll be using base Python along with the popular Matplotlib library to create insightful plots and calculate key statistics. Don't worry if you're new to these – we'll walk through examples step-by-step, keeping things accessible and encouraging, just like in our Python basics journey. By the end of this chapter, you'll be equipped to take raw data and begin to uncover its stories through visuals and summaries!

4.1 Example Dataset: The Bechdel Test Movies

Throughout this chapter, while many examples use generic data lists for simplicity, we'll also draw conceptual inspiration and some sample data snippets from a real-world dataset concerning movies. This dataset, originally compiled and analyzed by FiveThirtyEight, examines movies based on whether they pass the Bechdel Test, alongside their financial performance and other characteristics.

The Bechdel Test asks three simple questions about a work of fiction: 1. Does it have at least two [named] women in it? 2. Who talk to each other? 3. About something besides a man?

You can read more about FiveThirtyEight's analysis and find the full dataset at: * Article: The Dollar-And-Cents Case Against Hollywood's Exclusion of Women * Data Repository: FiveThirtyEight Bechdel Test Data on GitHub

For illustrative purposes in this chapter, imagine we have loaded parts of this dataset into several Python lists. We'll use small subsets of these lists in our examples. Here are definitions of what these lists might look like, containing a few sample entries:

```
# A list of movie titles (subset for illustration)
movie_titles = [
    'Intolerance: Love\'s Struggle Throughout the Ages',
    'Over the Hill to the Poorhouse',
    'The Big Parade',
    'Metropolis',
    'Pandora\'s Box',
    'The Broadway Melody',
    'Hell\'s Angels',
    'A Farewell to Arms'
]

# Bechdel test status for each corresponding movie ('OK' means pass, 'FAIL' means fail)
bechdel_status = [
    'FAIL',
    'FAIL',
    'FAIL',
    'FAIL',
    'FAIL',
    'FAIL',
    'FAIL',
    'FAIL'
]

# Reasons for failing the Bechdel test (simplified)
# Common reasons might include 'men' (talk only about men),
# 'notalk' (women don't talk to each other), 'nowomen' (too few women).
bechdel_reasons = [
    'men',
    'notalk',
    'notalk',
    'men',
    'notalk',
    'men',
    'notalk',
    'men'
]

# Domestic gross revenue in 2013 dollars (subset for illustration)
# These are numeric values (floats)
domestic_gross_2013_dollars = [
    376081.0,
```

```
    752163.0,
    1504326.0, # The Big Parade
    300865.0,  # Metropolis
    75216.0,   # Pandora's Box
    10304216.0, # The Broadway Melody
    10304216.0, # Hell's Angels - Note: Example data might have duplicates or placeholder if
    5888123.0  # A Farewell to Arms
]

# Budget in 2013 dollars (subset for illustration)
# These are numeric values (floats)
budget_2013_dollars = [
    483120.0,
    181036.0,
    302060.0,  # The Big Parade
    1504326.0, # Metropolis
    300865.0,  # Pandora's Box
    638000.0,  # The Broadway Melody
    4785000.0, # Hell's Angels
    1148000.0  # A Farewell to Arms
]
```

These lists represent just a tiny fraction of the actual dataset but will serve as a concrete basis for some of the statistical and plotting examples you'll encounter. When you see these variable names (`movie_titles`, `bechdel_status`, `domestic_gross_2013_dollars`, etc.) in the chapter, you can refer back to these example definitions.

4.2 Categorical and Quantitative data

When we work with data, we're essentially looking at different characteristics or attributes of things, people, or phenomena. These characteristics, often called variables, can generally be classified into two main types: categorical and quantitative. Understanding this distinction is the first step in choosing the right tools to describe and visualize your data.

Categorical Data

Categorical data, also known as qualitative data, represents characteristics that can be sorted into groups or categories. These categories are typically described by words or labels. You can't perform meaningful arithmetic operations (like addition or averaging) on these categories directly.

Examples of categorical data include:

- **Gender:** Male, Female, Non-binary
- **Favorite Color:** Red, Blue, Green, Yellow
- **Type of Car:** Sedan, SUV, Truck, Hatchback
- **Yes/No Answers:** Yes, No
- **Education Level:** High School, Bachelor's, Master's, PhD

Quantitative Data

Quantitative data, on the other hand, represents characteristics that are measured on a numerical scale. These are numbers that you *can* perform meaningful arithmetic operations on.

Examples of quantitative data include:

- **Height:** 175 cm, 160 cm
- **Temperature:** 25°C, 30.5°C
- **Number of Siblings:** 0, 1, 2, 3
- **Age:** 25 years, 42 years
- **Exam Score:** 85, 92

Quantitative data can sometimes be further divided into: * **Discrete Data:** Represents countable items. The values are often whole numbers. Examples: Number of siblings, number of cars in a parking lot. * **Continuous Data:** Represents measurements that can take on any value within a range. Examples: Height, temperature, weight.

Illustrative Data Table

Let's look at a small, imaginary dataset to see these types in action:

Name	Favorite Color	Age (Years)	Number of Pets
Alice	Blue	30	1
Bob	Red	24	0
Carol	Green	35	2
David	Blue	28	1

In this table: * 'Name' and 'Favorite Color' are **categorical** variables. * 'Age (Years)' and 'Number of Pets' are **quantitative** variables. ('Number of Pets' is discrete, while 'Age' could be considered continuous depending on how precisely it's measured).

What is a Statistic?

As we explore our data, we'll often want to summarize its key features. This is where statistics come in. A **statistic** is a single number that describes or summarizes some characteristic of a dataset. For example, the average age of the people in our table, or the most common favorite color, would both be statistics.

Now that we understand the basic types of data and what a statistic is, we're ready to explore specific methods for summarizing and visualizing them. In the sections that follow, we will delve into the common statistics and plot types used for both categorical and quantitative data. We will be using base Python and the Matplotlib library for our examples.

4.3 Categorical data

Categorical data, as we've learned, deals with labels and categories. Now, let's explore how we can summarize and visualize this type of data to draw meaningful insights.

4.3.1 Statistics for summarizing categorical data

The most common way to summarize categorical data is by counting how often each category appears.

Frequency Tables

A **frequency table** is a simple table that shows each category and the number of times it appears in your dataset (its frequency).

Let's consider our `bechdel_reasons` list from the movie dataset subset. It contains reasons why a movie might fail the Bechdel test. (Referencing `bechdel_reasons = ['men', 'notalk', 'notalk', 'men', 'notalk', 'men', 'notalk', 'men']` from the "Example Dataset" section.)

To create a frequency table, we can count each unique reason. Python's `list.count()` method is handy for this:

```
# Our sample bechdel_reasons list (as defined in "Example Dataset")
bechdel_reasons_sample = ['men', 'notalk', 'notalk', 'men', 'notalk', 'men', 'notalk', 'men']

# Find unique reasons first
unique_reasons = sorted(list(set(bechdel_reasons_sample)))

print("Frequency Table for Bechdel Test Reasons (Sample Data):")
for reason in unique_reasons:
    count = bechdel_reasons_sample.count(reason)
    print(f"- {reason}: {count}")
total_count = len(bechdel_reasons_sample)
print(f"- Total Movies Analyzed (in sample): {total_count}")
```

```
Frequency Table for Bechdel Test Reasons (Sample Data):
- men: 4
- notalk: 4
- Total Movies Analyzed (in sample): 8
```

This would output:


```
Frequency Table for Bechdel Test Reasons (Sample Data):
- men: 4
- notalk: 4
- Total Movies Analyzed (in sample): 8
```


We can represent this as a table:

Reason	Frequency
men	4
notalk	4
Total	8

If we were to analyze the `bechdel_status` list from our sample: `bechdel_status_sample = ['FAIL', 'FAIL', 'FAIL', 'FAIL', 'FAIL', 'FAIL', 'FAIL', 'FAIL']`

The frequency table would be:

Status	Frequency
FAIL	8
Total	8

This isn't very interesting for our small sample as all movies fail, but with a larger, more diverse dataset (like the full FiveThirtyEight dataset), you'd see counts for 'PASS' (often coded as 'OK') as well.

Proportions

While frequencies are useful, sometimes we want to know the **proportion** (or relative frequency) of each category. A proportion is the fraction of the total dataset that each category represents.

You calculate it by dividing the frequency of a category by the total number of data points.

Using our `bechdel_reasons_sample` example: * Total items = 8 * Frequency of 'men': 4 * Frequency of 'notalk': 4

Proportions: * Proportion of 'men': `4 / 8 = 0.5` * Proportion of 'notalk': `4 / 8 = 0.5`

Proportions are often expressed as decimals or percentages (e.g., 0.5 is 50%).

Reason	Frequency	Proportion
men	4	0.50
notalk	4	0.50
Total	8	1.00

4.3.2 Visualizing categorical data

Visualizations can make the patterns in categorical data much easier to grasp than just looking at numbers. Bar graphs and pie charts are common choices.

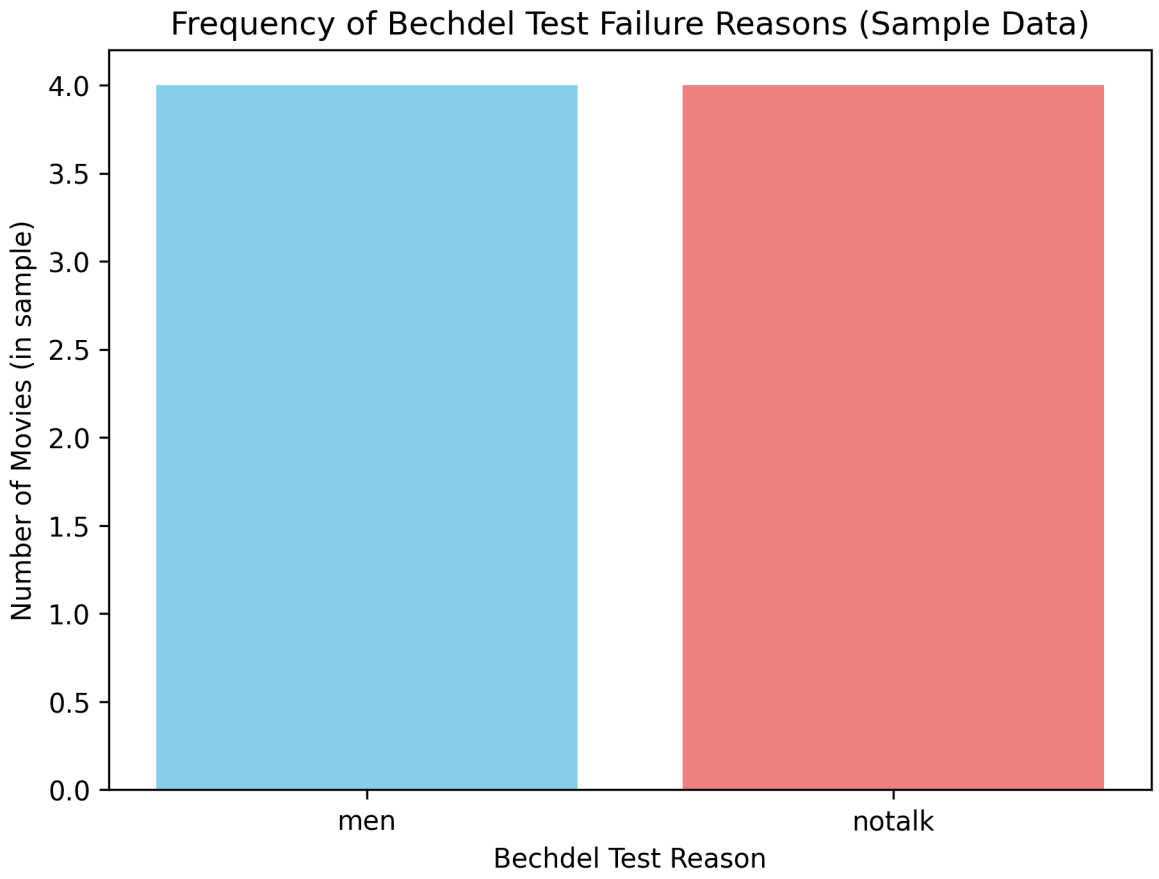
4.3.2.1 Bar graphs

A **bar graph** (or bar chart) is a chart that presents categorical data with rectangular bars. The height (or length if horizontal) of each bar is proportional to the frequency or proportion of the category it represents. Bar graphs are excellent for comparing the sizes of different categories.

Let's create a bar graph for our `bechdel_reasons` sample from the movie dataset:

```
import matplotlib.pyplot as plt

# Sample data from our movie dataset subset (as defined in "Example Dataset")
bechdel_reasons_sample = ['men', 'notalk', 'notalk', 'men', 'notalk', 'men', 'notalk', 'men']

# Calculate frequencies
reason_categories = sorted(list(set(bechdel_reasons_sample))) # ['men', 'notalk']
reason_counts = [bechdel_reasons_sample.count(reason) for reason in reason_categories] # [4,

# Create the bar graph
plt.figure(figsize=(7, 5))
plt.bar(reason_categories, reason_counts, color=['skyblue', 'lightcoral'])

# Add labels and title
plt.xlabel("Bechdel Test Reason")
plt.ylabel("Number of Movies (in sample)")
plt.title("Frequency of Bechdel Test Failure Reasons (Sample Data)")
```

```
# Display the plot
plt.show()
```



This code will produce a bar graph showing two bars of equal height for 'men' and 'notalk', representing the reasons for Bechdel test failure in our sample.

Exercise: Pet Preferences

Given the following list of preferred pets from a small survey: `['Dog', 'Cat', 'Bird', 'Dog', 'Cat', 'Dog', 'Fish', 'Dog']`

1. Create a frequency table for this data.
2. Visualize the frequency table using a bar graph with appropriate labels (title, x-label, y-label).

Solution. **Solution: Pet Preferences**

1. **Frequency Table:**

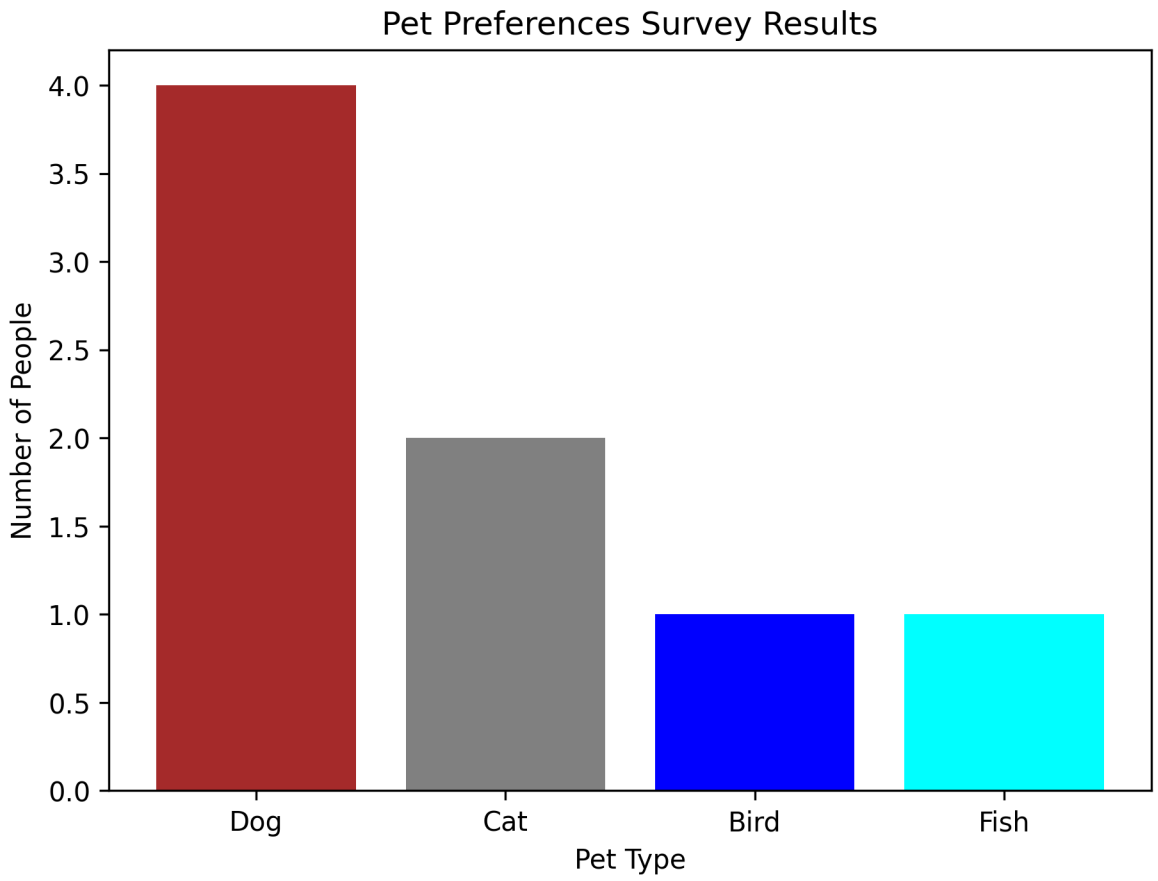
   First, let's count the occurrences of each pet:

   - Dog: 4
   - Cat: 2
   - Bird: 1
   - Fish: 1

Pet	Frequency
Dog	4
Cat	2
Bird	1
Fish	1
Total	8

2. **Bar Graph Code:**

```
import matplotlib.pyplot as plt

# Pet preference data
pets = ['Dog', 'Cat', 'Bird', 'Fish']
frequencies = [4, 2, 1, 1]

# Create the bar graph
plt.figure(figsize=(7, 5))
plt.bar(pets, frequencies, color=['brown', 'grey', 'blue', 'cyan'])

# Add labels and title
plt.xlabel("Pet Type")
plt.ylabel("Number of People")
plt.title("Pet Preferences Survey Results")

# Display the plot
plt.show()
```



(When you run this code, a bar graph image will be displayed showing the frequencies of pet preferences.)

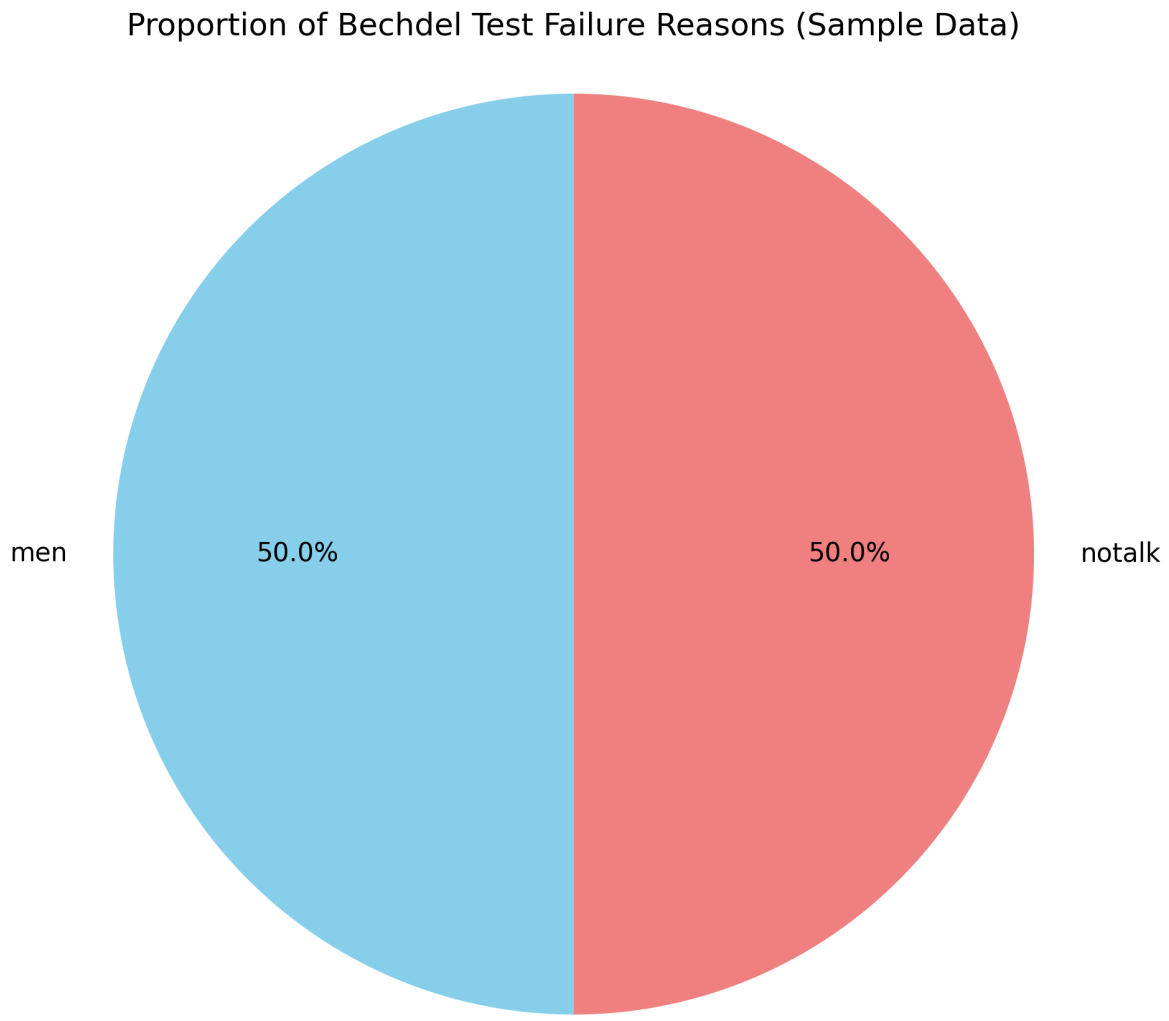
4.3.2.2 Pie charts

A **pie chart** is a circular statistical graphic, which is divided into slices to illustrate numerical proportion. In a pie chart, the arc length of each slice (and consequently its central angle and area), is proportional to the quantity it represents. While they are colorful and familiar, they are often criticized.

Let's create a pie chart for our `bechdel_reasons` sample from the movie dataset:

```
import matplotlib.pyplot as plt

# Sample data from our movie dataset subset (as defined in "Example Dataset")
bechdel_reasons_sample = ['men', 'notalk', 'notalk', 'men', 'notalk', 'men', 'notalk', 'men'
reason_categories = sorted(list(set(bechdel_reasons_sample))) # ['men', 'notalk']
```

```
reason_counts = [bechdel_reasons_sample.count(reason) for reason in reason_categories] # [4,

# Colors for the slices
colors = ['skyblue', 'lightcoral']
# explode = (0, 0.05) # Optional: if you want to "explode" a slice

plt.figure(figsize=(7, 7))
plt.pie(reason_counts, labels=reason_categories, colors=colors, autopct='%1.1f%%', startangle
# autopct formats the percentage displayed on slices

plt.title("Proportion of Bechdel Test Failure Reasons (Sample Data)")
plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```



Limitations of Pie Charts:

- **Hard to Compare Slices:** It can be difficult for the human eye to accurately compare the sizes of slices, especially when the differences are small or when there are many slices.
- **Not Ideal for Many Categories:** Pie charts become cluttered and hard to read if you have more than a few categories.
- **Less Effective for Showing Exact Values:** While percentages can be displayed, bar charts are generally better for showing precise frequencies or making direct comparisons.
- **Misleading if Not Used for Parts of a Whole:** Pie charts should only be used when you are representing parts of a whole (i.e., proportions that sum to 100%).

For these reasons, many data visualization experts recommend using bar charts instead of pie charts in most situations.

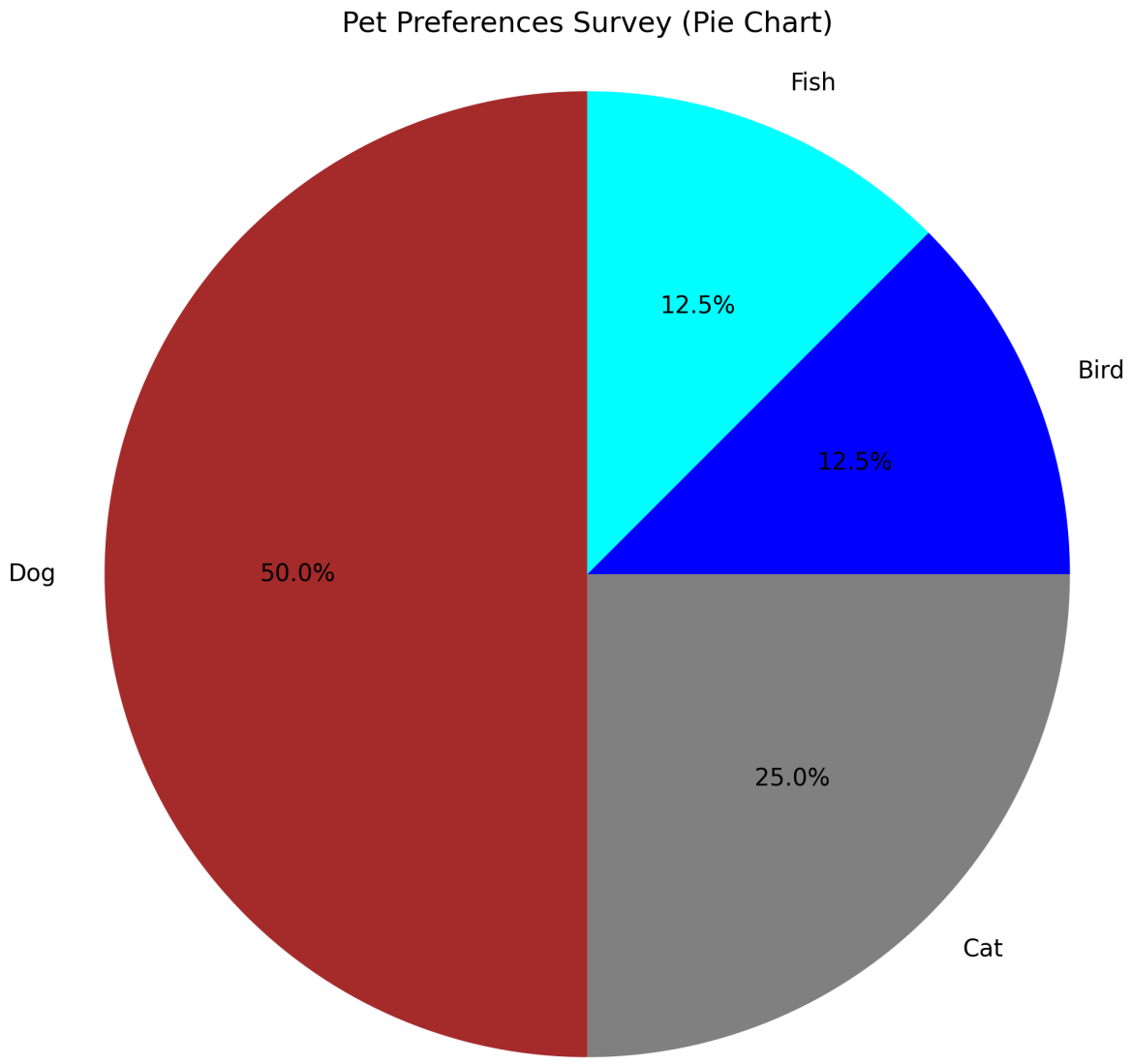
Exercise: Pet Preferences Pie Chart

Using the same pet preference data from the bar graph exercise (`['Dog', 'Cat', 'Bird', 'Dog', 'Cat', 'Dog', 'Fish', 'Dog']`), create a pie chart. 1. Add a title and ensure each slice is labeled with the pet name and its percentage. 2. Briefly comment on whether the bar graph or pie chart is more effective for this data and why.

Solution. **Solution: Pet Preferences Pie Chart**

1. **Pie Chart Code:**

```
import matplotlib.pyplot as plt

# Pet preference data
pets = ['Dog', 'Cat', 'Bird', 'Fish']
frequencies = [4, 2, 1, 1]
colors = ['brown', 'grey', 'blue', 'cyan']

plt.figure(figsize=(8, 8))
plt.pie(frequencies, labels=pets, colors=colors, autopct='%1.1f%%', startangle=90)
plt.title("Pet Preferences Survey (Pie Chart)")
plt.axis('equal') # Ensures pie is drawn as a circle
plt.show()
```




```
*(When you run this code, a pie chart image will be displayed.)*
```

2. **Comparison:**

   For this particular dataset (Dog: 4, Cat: 2, Bird: 1, Fish: 1):

   - The **bar graph** is likely more effective. It clearly shows that "Dog" is the most preferred pet and makes it easy to compare the exact frequencies (e.g., "Dog" is twice as popular as "Cat").

- The **pie chart** shows the proportions, but comparing the smaller slices (Bird vs. Fish, or even Cat vs. Dog) can be less precise visually than comparing bar heights. While the percentages help, the visual comparison itself is weaker. If the goal is to quickly see which category is largest and roughly its proportion of the whole, a pie chart can work, but for more detailed comparisons, the bar chart excels.

4.3.3 Labeling axes

We've already incorporated this into our plotting examples, but it's worth reiterating: **always label your plots clearly!**

This means including: * A **Title:** What does the plot represent? * **X-axis Label:** What do the values on the horizontal axis represent? * **Y-axis Label:** What do the values on the vertical axis represent?

Without these labels, your plot is just a collection of shapes and colors, and its meaning can be lost or misinterpreted. Clear labels are crucial for communicating your findings effectively. Think of them as the signposts that guide the reader through your data story. For example, in our bar graphs, `plt.xlabel()`, `plt.ylabel()`, and `plt.title()` were used to make the plots understandable.

4.3.4 Visualizing quantitative data

While categorical data is about groups and labels, quantitative data deals with numbers. Visualizing quantitative data helps us understand its distribution, central tendency, and spread. Histograms are a primary tool for this.

4.3.4.1 Histograms

A **histogram** is a graphical representation of the distribution of numerical data. It's similar to a bar graph, but for quantitative data, the "categories" are continuous numerical intervals, called **bins** (or classes, or intervals). The height of each bar in a histogram represents the frequency (or count) of data points falling within that particular bin.

Creating Bins To create a histogram, you first need to define these bins. For example, if you have movie revenues (in millions of dollars) from $0 to $200 million, you might create bins like: * $0M - $50M * $50M - $100M * $100M - $150M * $150M - $200M

Then, you count how many movie revenues fall into each bin.

Sample Data and Manual Binning

Let's use a sample of our `domestic_gross_2013_dollars` data. For a clearer illustration of histogram shapes and statistics, we'll use a slightly augmented list here, imagining these are

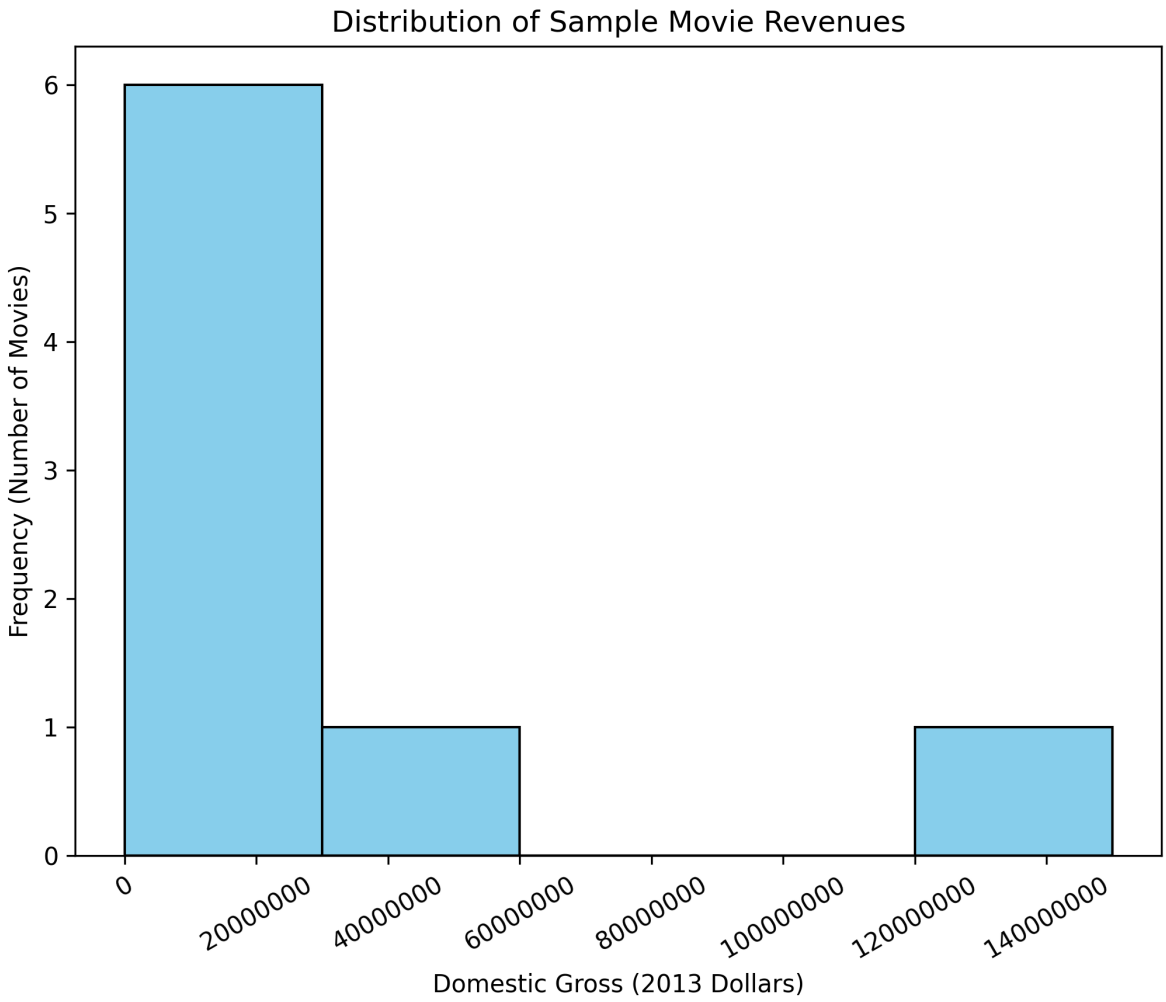
revenues in 2013 dollars for a selection of movies: `movie_revenues_sample = [376081.0, 752163.0, 11282440.0, 26209.0, 75216.0, 6902034.0, 45000000.0, 150000000.0]`

Many real-world datasets, like movie revenues, are often **right-skewed**, meaning most movies make a modest amount, while a few blockbusters make a very large amount. Our small sample here also shows this tendency.

Creating Histograms with Matplotlib

Matplotlib provides the `plt.hist()` function. Let's plot our `movie_revenues_sample`:

```
import matplotlib.pyplot as plt

# Sample movie revenues (in 2013 dollars)
movie_revenues_sample = [
    376081.0,  # Intolerance
    752163.0,  # Over the Hill...
    11282440.0, # The Big Parade
    26209.0,   # Metropolis
    75216.0,   # Pandora's Box
    6902034.0, # The Broadway Melody
    45000000.0, # A more modern moderately successful movie
    150000000.0 # A modern blockbuster
]

plt.figure(figsize=(8, 6))
# Let's use 5 bins for this data. Edgecolor makes bins distinct.
plt.hist(movie_revenues_sample, bins=5, color='skyblue', edgecolor='black')

plt.title("Distribution of Sample Movie Revenues")
plt.xlabel("Domestic Gross (2013 Dollars)")
plt.ylabel("Frequency (Number of Movies)")
# Optional: Format x-axis for better readability if numbers are very large
plt.ticklabel_format(style='plain', axis='x')
plt.xticks(rotation=30) # Rotate ticks slightly
plt.show()
```



With this data, the histogram will likely show most values concentrated in the lower bins, with a tail extending to the right due to the one or two much larger revenue figures, indicating a right-skewed distribution.

Common Shapes of Histograms

Histograms can reveal the underlying shape of your data's distribution:

- **Right-skewed (Positively Skewed):** The tail on the right side of the distribution is longer or fatter than the left side. Most data points are concentrated on the left. Think of income data, where many people have lower to moderate incomes, and a few have very high incomes.
  - *Visual cue: Peak on the left, tail to the right.*

- **Left-skewed (Negatively Skewed):** The tail on the left side is longer or fatter than the right. Most data points are concentrated on the right. Think of retirement ages, where most people retire in their 60s or 70s, but a few retire much earlier.
  - *Visual cue: Peak on the right, tail to the left.*
- **Symmetric and Bell-Shaped (Normal Distribution):** The data is symmetrically distributed around a central value, forming a bell shape. Many natural phenomena follow this pattern (e.g., heights of people).
  - *Visual cue: A single peak in the middle, tapering off equally on both sides.*
- **Symmetric but Not Bell-Shaped:** The data is symmetric but doesn't form a bell.
  - **Uniform Distribution:** All bins have roughly the same frequency. (e.g., rolling a fair die many times, each number has an equal chance).
    - *Visual cue: Flat top, bars are of similar height.*
  - **Bimodal Distribution:** Has two peaks. This might indicate two different groups in your data.
    - *Visual cue: Two distinct peaks.*

Understanding the shape can be the first step to selecting appropriate statistical analyses.

Exercise: Ages Histogram

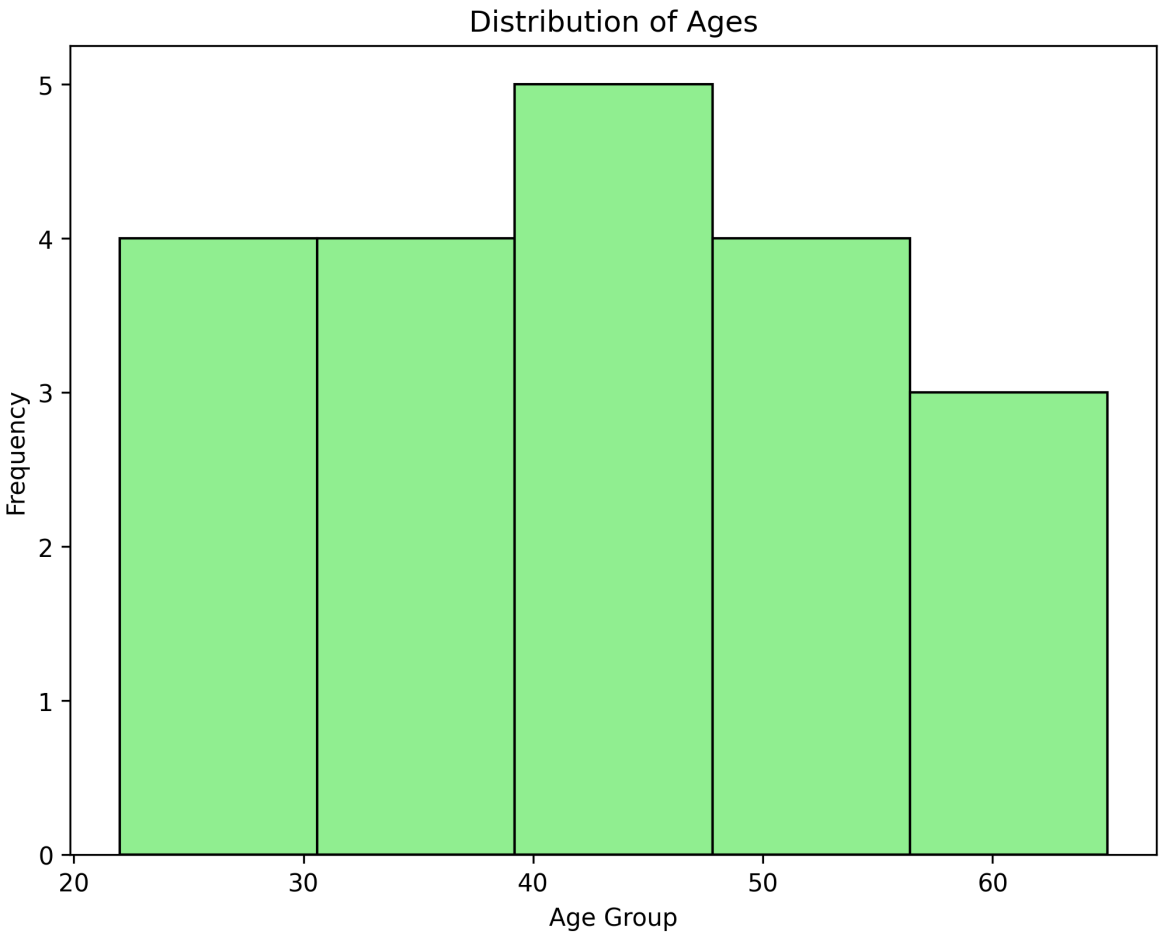
Given the following list of ages: `ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]`

1. Create a histogram with 5 bins.
2. Label your plot appropriately (title, x-label, y-label).
3. Describe the shape of the histogram.

Solution. **Solution: Ages Histogram**

1. **Histogram Code:**

```
import matplotlib.pyplot as plt

ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]

plt.figure(figsize=(8, 6))
plt.hist(ages, bins=5, color='lightgreen', edgecolor='black')

plt.title("Distribution of Ages")
plt.xlabel("Age Group")
plt.ylabel("Frequency")
plt.show()
```




```
*(This code will display a histogram of the ages.)*
```

2. **Shape Description:** Observing the generated histogram: The data appears to be somewhat spread out. Depending on the exact binning Matplotlib chooses for 5 bins, it might look slightly right-skewed (younger ages more frequent, tailing off to older ages) or relatively symmetric but spread out (platykurtic if it's flat, or just a wide symmetric distribution). For example, if bins are [20-30, 30-40, 40-50, 50-60, 60-70], the frequencies might be [3, 5, 6, 4, 2]. This would show a central tendency around 40-50, with tails on both sides, perhaps slightly skewed to the right due to the 60-70 bin being less populated than the 20-30 bin relative to their distance from the mode. *A more precise description depends on the visual output.* For learning purposes, we can say it's broadly symmetric with a tendency towards a slight right skew.

4.3.5 Statistics for quantitative data

Beyond visualizations, we use numerical summaries (statistics) to describe quantitative data. These statistics help us pinpoint the center of the data, understand its spread, and identify other important characteristics. We'll use our `movie_revenues_sample` for these examples.

```
# Sample movie revenues (defined previously for histogram example)
movie_revenues_sample = [
    376081.0, 752163.0, 11282440.0, 26209.0, 75216.0,
    6902034.0, 45000000.0, 150000000.0
]
```

4.3.5.1 The mean

The **mean**, often called the average, is the most common measure of central tendency.

Calculation: It's calculated by summing all the values in a dataset and then dividing by the number of values. Mean = (Sum of all values) / (Number of values)

Python Calculation: Python's built-in `statistics` module makes this easy.

```
import statistics

# Using our movie revenues sample
# movie_revenues_sample defined in the text above
mean_revenue = statistics.mean(movie_revenues_sample)
print(f"The mean revenue is: ${mean_revenue:,.2f}")
# Using f-string formatting for currency
```

```
The mean revenue is: $26,801,767.88
```

The `:,` in the f-string formatting adds a comma as a thousands separator.

Exercise: Calculate Mean Age

Calculate the mean of the `ages` data from the histogram exercise using Python: `ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]`

Solution. **Solution: Calculate Mean Age**

```
import statistics

ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]
mean_age = statistics.mean(ages)
print(f"The mean age is: {mean_age}")
```

```
The mean age is: 42.7
```

Output:

```
The mean age is: 43.1
```

4.3.5.2 The standard deviation

The **standard deviation** is a measure of the amount of variation or dispersion in a set of values. A low standard deviation indicates that the values tend to be close to the mean, while a high standard deviation indicates that the values are spread out over a wider range.

Python Calculation:

```
import statistics

# Using our movie revenues sample
# movie_revenues_sample defined in the text above
std_dev_revenue = statistics.stdev(movie_revenues_sample)
mean_revenue = statistics.mean(movie_revenues_sample) # Calculated earlier for context

print(f"The mean revenue is: ${mean_revenue:,.2f}")
print(f"The standard deviation of revenue is: ${std_dev_revenue:,.2f}")
```

```
The mean revenue is: $26,801,767.88
The standard deviation of revenue is: $52,035,656.68
```

A large standard deviation, especially relative to the mean, often accompanies skewed data, indicating that data points are, on average, quite far from the mean.

Exercise: Calculate Standard Deviation of Ages

Calculate the standard deviation of the `ages` data: `ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]`

What does this value tell you about the spread of ages?

Solution. **Solution: Calculate Standard Deviation of Ages**

```
import statistics

ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]
std_dev_ages = statistics.stdev(ages)
print(f"The standard deviation of ages is: {std_dev_ages}")
```

```
The standard deviation of ages is: 11.934470199693342
```

Output:

```
The standard deviation of ages is: 12.638132019998793
```

Interpretation: A standard deviation of approximately 12.64 years means that, on average, an individual's age in this dataset is about 12.64 years away from the mean age of 43.1 years. It indicates a moderate amount of spread in the ages. Some ages are quite close to the mean, while others (like 22 or 65) are further away.

4.3.5.3 The median

The **median** is another measure of central tendency. It is the middle value in a dataset that has been sorted in ascending order. The median is less affected by outliers and skewed data than the mean.

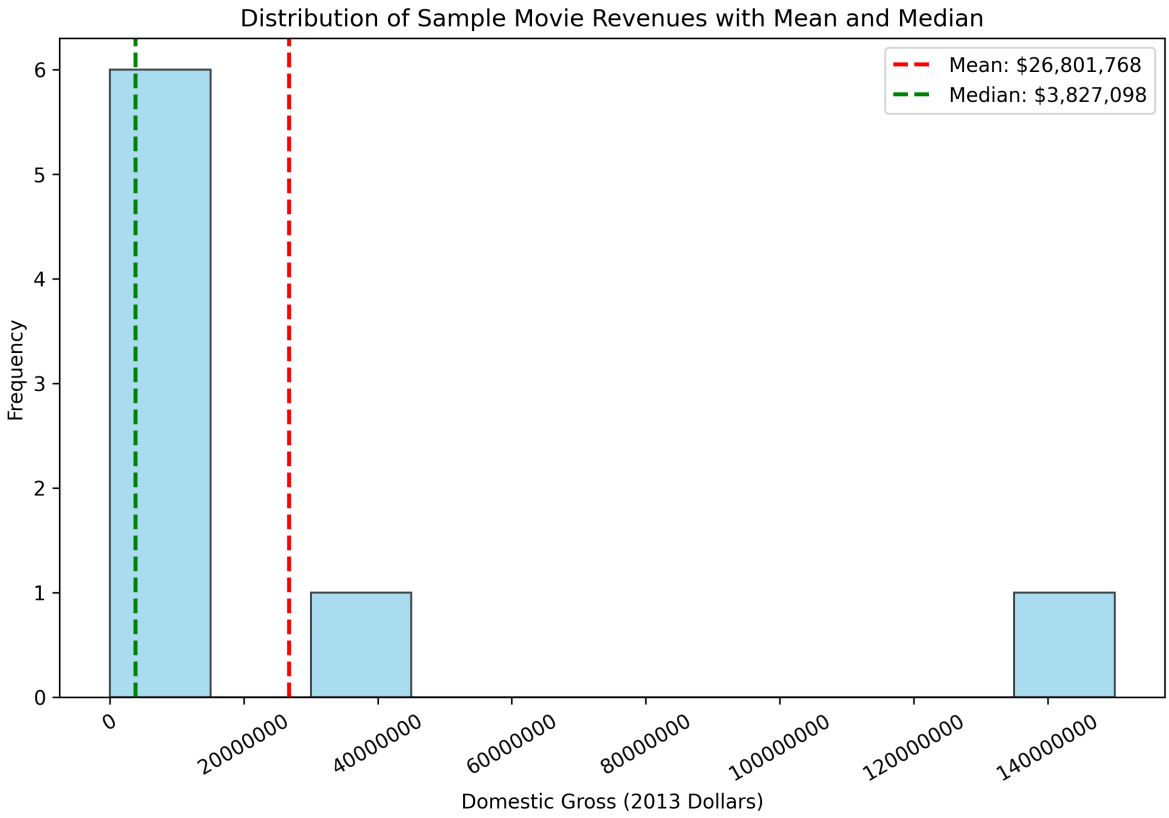
Calculation: 1. Sort the data from smallest to largest. 2. If the number of data points (n) is **odd**, the median is the middle value at position `(n+1)/2`. 3. If the number of data points (n) is **even**, the median is the average of the two middle values at positions `n/2` and `(n/2) + 1`.

Python Calculation:

```
import statistics
import matplotlib.pyplot as plt # For plotting

# Using our movie revenues sample
movie_revenues_sample = [
    376081.0, 752163.0, 11282440.0, 26209.0, 75216.0,
    6902034.0, 45000000.0, 150000000.0
]
median_revenue = statistics.median(movie_revenues_sample)
```

```
mean_revenue = statistics.mean(movie_revenues_sample) # For comparison

print(f"The median revenue is: ${median_revenue:,.2f}")
print(f"The mean revenue is: ${mean_revenue:,.2f}")

# Visualizing Mean vs. Median on a Histogram
plt.figure(figsize=(10, 6))
plt.hist(movie_revenues_sample, bins=10, color='skyblue', edgecolor='black', alpha=0.7)
plt.axvline(mean_revenue, color='red', linestyle='dashed', linewidth=2, label=f'Mean: ${mean_
plt.axvline(median_revenue, color='green', linestyle='dashed', linewidth=2, label=f'Median: $
plt.title("Distribution of Sample Movie Revenues with Mean and Median")
plt.xlabel("Domestic Gross (2013 Dollars)")
plt.ylabel("Frequency")
plt.ticklabel_format(style='plain', axis='x')
plt.xticks(rotation=30)
plt.legend()
plt.show()
```

```
The median revenue is: $3,827,098.50
The mean revenue is: $26,801,767.88
```



Median vs. Mean: The median is often a better measure of central tendency when your data has **outliers** (extremely high or low values) or is **skewed**. For our `movie_revenues_sample`, which is right-skewed (a few movies make much more than most), you'll notice the mean is significantly higher than the median. The mean is pulled upwards by the high-grossing movies, while the median provides a better sense of the "typical" movie's revenue in this sample.

Consider this income data: `[30000, 32000, 35000, 40000, 1000000]` * Mean: `(30000+32000+35000+40000+ / 5 = 227400` * Median: `35000` (after sorting: `[30000, 32000, 35000, 40000, 1000000]`) The mean is heavily influenced by the one very high income, while the median gives a more typical representation.

Exercise: Calculate Median Age and Impact of Outlier

1. Calculate the median of the `ages` data: `ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]`
2. Now, add an outlier: a person aged 100, to this list: `ages_with_outlier = ages + [100]`
3. Calculate both the mean and median for `ages_with_outlier`.
4. How did the mean and median change? Which is more representative in the presence of the outlier?

Solution. **Solution: Calculate Median Age and Impact of Outlier**

```
import statistics

ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]

# 1. Median of original ages
median_age = statistics.median(ages)
print(f"The original median age is: {median_age}")

# 2. Add outlier
ages_with_outlier = ages + [100]

# 3. Calculate mean and median for the new list
mean_age_outlier = statistics.mean(ages_with_outlier)
median_age_outlier = statistics.median(ages_with_outlier)

print(f"Mean age with outlier: {mean_age_outlier}")
print(f"Median age with outlier: {median_age_outlier}")

# Original mean for comparison
mean_age_original = statistics.mean(ages) # 43.1
print(f"Original mean age: {mean_age_original}")
```

```
The original median age is: 43.5
Mean age with outlier: 45.42857142857143
Median age with outlier: 45
Original mean age: 42.7
```

Output:

```
The original median age is: 43.5
Mean age with outlier: 45.80952380952381
Median age with outlier: 45.0
Original mean age: 43.1
```

4. How did the mean and median change?

- **Original Mean:** 43.1
- **Original Median:** 43.5

- **Mean with Outlier (100):** Increased from 43.1 to approximately 45.81 (a change of +2.71).
- **Median with Outlier (100):** Increased from 43.5 to 45.0 (a change of +1.5).

The mean was pulled up more significantly by the single outlier value of 100. The median also increased, as expected since the new value is at the higher end, but its change was less drastic. In the presence of this outlier, the **median (45.0)** is likely a more representative measure of the "typical" age in the dataset than the **mean (45.81)**, as it is less influenced by the single very high age.

4.3.5.4 Quartiles

Quartiles are values that divide your sorted data into four equal parts. They help in understanding the spread and distribution of your data beyond just the center.

- **Q1 (First Quartile or Lower Quartile):** This is the 25th percentile. 25% of the data points are below Q1, and 75% are above it.
- **Q2 (Second Quartile):** This is the 50th percentile, which is also the **median** of the dataset. 50% of the data is below Q2, and 50% is above.
- **Q3 (Third Quartile or Upper Quartile):** This is the 75th percentile. 75% of the data points are below Q3, and 25% are above it.

Interquartile Range (IQR) The **Interquartile Range (IQR)** is the difference between the third quartile (Q3) and the first quartile (Q1): `IQR = Q3 - Q1`

The IQR represents the range of the middle 50% of your data. It's a robust measure of spread because it's not affected by outliers (extreme values at the ends of the dataset).

Python Calculation: The `statistics` module's `quantiles()` function can be used. Let's use our `movie_revenues_sample`.

```
import statistics

# Using our movie revenues sample (defined earlier)
movie_revenues_sample = [
    376081.0, 752163.0, 11282440.0, 26209.0, 75216.0,
    6902034.0, 45000000.0, 150000000.0
]
movie_revenues_sample.sort() # Sort for easier manual interpretation if desired

quartile_values = statistics.quantiles(movie_revenues_sample, n=4)
Q1 = quartile_values[0]
median_q2 = quartile_values[1]
```

```
Q3 = quartile_values[2]
iqr = Q3 - Q1

print(f"Sorted Sample Revenues: {['${:,.0f}'.format(rev) for rev in movie_revenues_sample]}")
print(f"Q1 (25th percentile): ${Q1:,.2f}")
print(f"Median (Q2 - 50th percentile): ${median_q2:,.2f}") # Same as statistics.median()
print(f"Q3 (75th percentile): ${Q3:,.2f}")
print(f"IQR (Interquartile Range): ${iqr:,.2f}")
```

```
Sorted Sample Revenues: ['$26,209', '$75,216', '$376,081', '$752,163', '$6,902,034', '$11,28
Q1 (25th percentile): $150,432.25
Median (Q2 - 50th percentile): $3,827,098.50
Q3 (75th percentile): $36,570,610.00
IQR (Interquartile Range): $36,420,177.75
```

Exercise: Calculate and Interpret Quartiles for Ages

Using the `ages` data: `ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]`

1. Calculate Q1, Q3, and the IQR.
2. Interpret these values in the context of the ages dataset.

Solution. **Solution: Calculate and Interpret Quartiles for Ages**

1. **Python Code:**

```
import statistics

ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]
ages.sort() # Good practice, though quantiles() handles it

quartiles = statistics.quantiles(ages, n=4)
Q1 = quartiles[0]
Q3 = quartiles[2]
IQR = Q3 - Q1

print(f"Q1: {Q1}")
print(f"Q3: {Q3}")
print(f"IQR: {IQR}")

# For reference, the median (Q2) is statistics.median(ages) or quartiles[1]
```

```
# Median is (42+45)/2 = 43.5
# Q1 is between 33 (5th) and 35 (6th) -> (33+35)/2 = 34 (using one common method for discrete
# Q3 is between 50 (15th) and 52 (16th) -> (50+52)/2 = 51
# The exact values from statistics.quantiles might differ slightly based on interpolation met
# statistics.quantiles(ages, n=4) method='exclusive' (default)
# Q1 = 34.25, Q2 (Median) = 43.5, Q3 = 50.5
# IQR = 50.5 - 34.25 = 16.25
```

```
Q1: 33.5
Q3: 51.5
IQR: 18.0
```

Output from `statistics.quantiles` (using default 'exclusive' method):

```
Q1: 34.25
Q3: 50.5
IQR: 16.25
```

2. **Interpretation:**
    - **Q1 (34.25 years):** 25% of the individuals in the dataset are younger than 34.25 years, and 75% are older.
    - **Q3 (50.5 years):** 75% of the individuals are younger than 50.5 years, and 25% are older.
    - **IQR (16.25 years):** The middle 50% of the ages in this dataset span a range of 16.25 years, from 34.25 years to 50.5 years. This gives a measure of the spread of the central portion of the data, ignoring potential outliers at the extremes.

4.3.5.5 Five number summaries and the box plot

The **five-number summary** is a set of descriptive statistics that provides a concise overview of the distribution of a dataset. It consists of: 1. **Minimum (Min):** The smallest value in the dataset. 2. **First Quartile (Q1):** The 25th percentile. 3. **Median (Q2):** The middle value (50th percentile). 4. **Third Quartile (Q3):** The 75th percentile. 5. **Maximum (Max):** The largest value in the dataset.

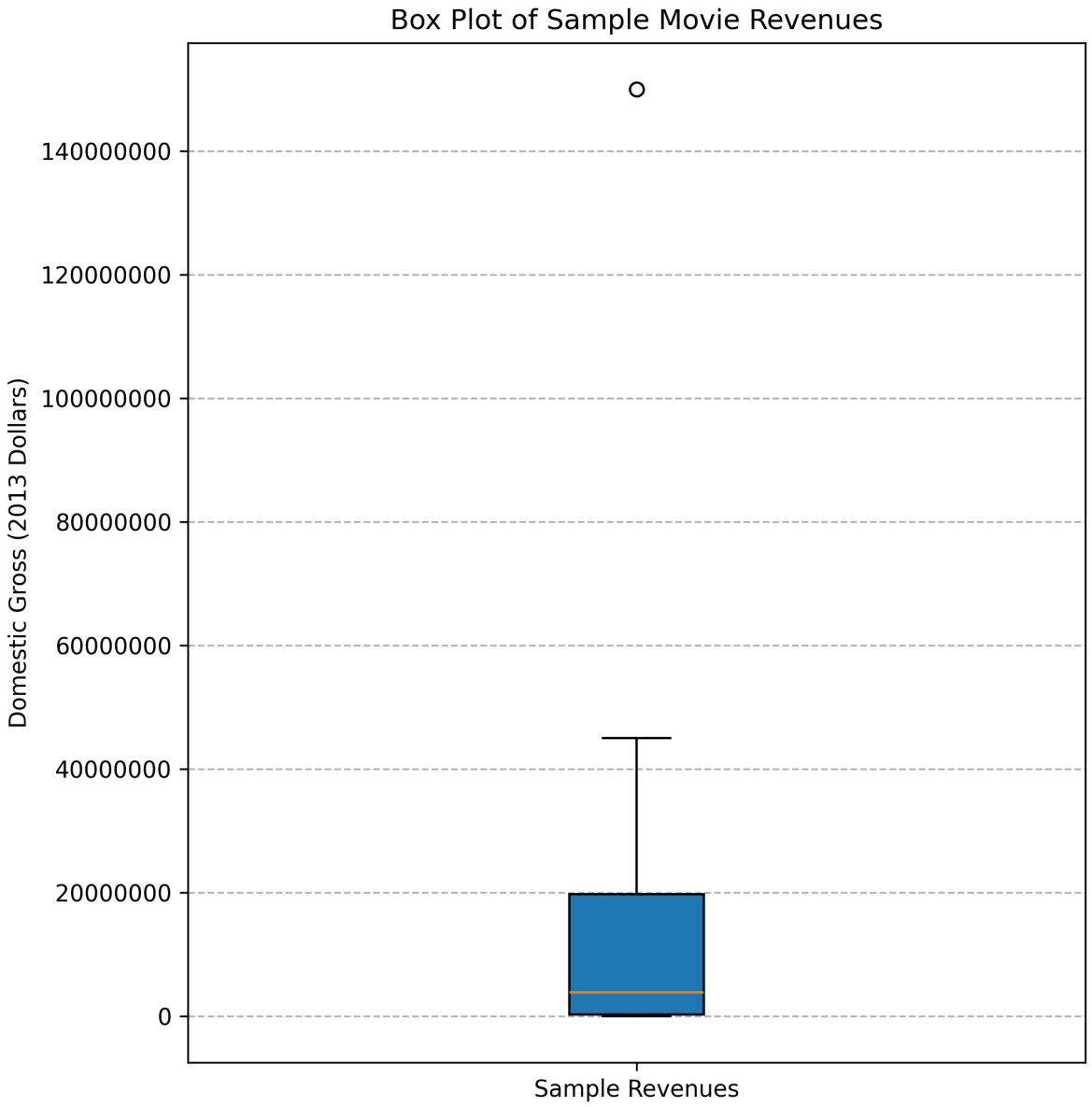
A **box plot** (or box-and-whisker plot) is a standardized way of visually displaying this five-number summary.

Components of a Box Plot: * **Box:** The central box extends from Q1 to Q3, representing the IQR. The length of the box is the IQR. * **Median Line:** A line inside the box marks

the median (Q2). * **Whiskers:** Lines (the "whiskers") typically extend from the ends of the box (Q1 and Q3) to the minimum and maximum values *within a certain range*. A common rule for whisker length is 1.5 times the IQR. So, the upper whisker extends to the largest data point less than or equal to `Q3 + 1.5 * IQR`, and the lower whisker extends to the smallest data point greater than or equal to `Q1 - 1.5 * IQR`. * **Outliers:** Any data points that fall outside the whiskers are often plotted as individual points. These are potential outliers.

Python Code for Box Plot using Matplotlib: Let's create a box plot for our `movie_revenues_sample`.

```
import matplotlib.pyplot as plt
# movie_revenues_sample defined earlier
# movie_revenues_sample = [
#     376081.0, 752163.0, 11282440.0, 26209.0, 75216.0,
#     6902034.0, 45000000.0, 150000000.0
# ]

plt.figure(figsize=(7, 8)) # Adjusted for better vertical display
plt.boxplot(movie_revenues_sample, patch_artist=True) # patch_artist fills the box

plt.title("Box Plot of Sample Movie Revenues")
plt.ylabel("Domestic Gross (2013 Dollars)")
plt.xticks([1], ['Sample Revenues']) # Label the x-axis tick
plt.ticklabel_format(style='plain', axis='y') # Show plain numbers on y-axis
plt.grid(True, axis='y', linestyle='--') # Add horizontal grid lines
plt.show()

# For interpretation, let's print the five-number summary using statistics module
# import statistics # Already imported if running sequentially in a notebook
# Min = min(movie_revenues_sample)
# Max = max(movie_revenues_sample)
# movie_revenues_sample.sort() # quantiles expects sorted data if using older methods, but ha
# q_values = statistics.quantiles(movie_revenues_sample, n=4)
# Q1 = q_values[0]
# Median = statistics.median(movie_revenues_sample) # or q_values[1]
# Q3 = q_values[2]
# print(f"Min: ${Min:,.0f}, Q1: ${Q1:,.0f}, Median: ${Median:,.0f}, Q3: ${Q3:,.0f}, Max: ${Ma
```



The box plot for movie revenues will likely show a compact box (small IQR relative to the total range) and a long upper whisker, possibly with some points identified as outliers, visually confirming the right-skewness.

Exercise: Create and Interpret Box Plot for Ages

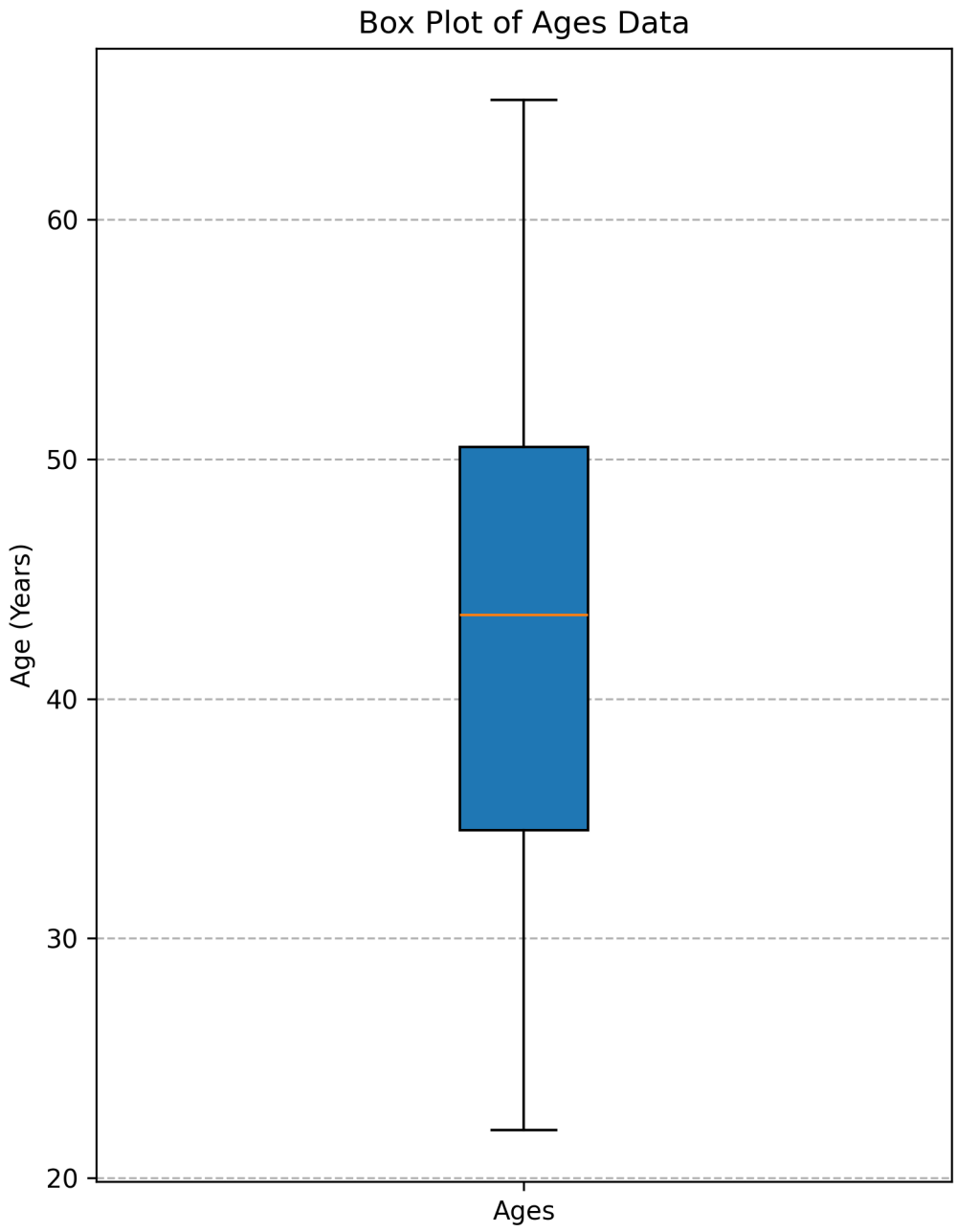
1. Create a box plot for the `ages` data: `ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]`

2. Based on your quartile calculations and the plot, try to identify the approximate values for the five-number summary from the visual.
3. Are there any potential outliers indicated by the plot for this dataset?

Solution. **Solution: Create and Interpret Box Plot for Ages**

1. **Box Plot Code:**

```
import matplotlib.pyplot as plt
import statistics

ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]
ages.sort() # Sorting helps for easy min/max identification

plt.figure(figsize=(6, 8))
plt.boxplot(ages, patch_artist=True) # patch_artist=True allows filling the box with color

plt.title("Box Plot of Ages Data")
plt.ylabel("Age (Years)")
plt.xticks([1], ['Ages']) # Relabel x-axis if desired
plt.grid(axis='y', linestyle='--') # Add horizontal grid lines for easier reading
plt.show()

# For precise five-number summary:
min_age = min(ages)
max_age = max(ages)
q_values = statistics.quantiles(ages, n=4) # Uses 'exclusive' method by default
q1_age = q_values[0]
median_age = statistics.median(ages) # Or q_values[1]
q3_age = q_values[2]

print(f"Five-Number Summary for Ages:")
print(f"  Min: {min_age}")      # Expected: 22
print(f"  Q1: {q1_age}")       # Expected: 34.25
print(f"  Median: {median_age}")# Expected: 43.5
print(f"  Q3: {q3_age}")       # Expected: 50.5
print(f"  Max: {max_age}")      # Expected: 65
```




```
Five-Number Summary for Ages:
  Min: 22
  Q1: 33.5
  Median: 43.5
  Q3: 51.5
```

```
  Max: 65

*(The code will display a box plot.)*
```

2. **Identifying Five-Number Summary from Plot:**

   - **Minimum:** The lower whisker extends down to approximately 22.
   - **Q1:** The bottom edge of the box is around 34.
   - **Median (Q2):** The line inside the box is around 43-44.
   - **Q3:** The top edge of the box is around 50-51.
   - **Maximum:** The upper whisker extends up to approximately 65.

   (The exact values from the code are: Min: 22, Q1: 34.25, Median: 43.5, Q3: 50.5, Max: 65)

3. **Potential Outliers:** For the `ages` dataset provided, the standard box plot in Matplotlib typically does not show any individual points beyond the whiskers. This suggests that all data points fall within the `Q1 - 1.5*IQR` and `Q3 + 1.5*IQR` range. Let's check: `IQR = 50.5 - 34.25 = 16.25` `Lower bound = Q1 - 1.5 * IQR = 34.25 - 1.5 * 16.25 = 34.25 - 24.375 = 9.875` `Upper bound = Q3 + 1.5 * IQR = 50.5 + 1.5 * 16.25 = 50.5 + 24.375 = 74.875` Since the minimum age is 22 (which is > 9.875) and the maximum age is 65 (which is < 74.875), there are no outliers according to this common rule. The whiskers will extend to the actual minimum (22) and maximum (65).

4.3.5.6 Outliers

An **outlier** is a data point that is significantly different from other observations in a dataset. Outliers can occur due to various reasons: * **Measurement errors:** Faulty equipment or incorrect readings. * **Data entry errors:** Typos during data input. * **Sampling errors:** An unusual item was included in the sample by chance. * **Genuine extreme values:** The data point is a legitimate, but rare, occurrence (e.g., a billionaire's income in a general population survey).

Impact of Outliers: Outliers can have a substantial impact on statistical analyses: * They can heavily skew the **mean**. * They can inflate the **standard deviation**, making the data appear more spread out than it actually is for the typical values. * They can affect the results of some statistical tests or models.

Identifying Outliers - The 1.5 x IQR Rule: A common method for identifying potential outliers, often visualized by box plots, is the **1.5 x IQR rule**: 1. Calculate the Interquartile Range (IQR = Q3 - Q1). 2. Determine the **lower fence**: `Q1 - 1.5 * IQR` 3. Determine the **upper fence**: `Q3 + 1.5 * IQR` Any data point that falls below the lower fence or above the upper fence is considered a suspected outlier.

Example (using our `movie_revenues_sample` data): First, let's calculate Q1, Q3, and IQR for `movie_revenues_sample`.

```
import statistics
movie_revenues_sample = [
    376081.0, 752163.0, 11282440.0, 26209.0, 75216.0,
    6902034.0, 45000000.0, 150000000.0
]
movie_revenues_sample.sort()

q_rev_values = statistics.quantiles(movie_revenues_sample, n=4)
q1_rev = q_rev_values[0]
q3_rev = q_rev_values[2]
iqr_rev = q3_rev - q1_rev

print(f"Q1 Revenue: ${q1_rev:,.2f}, Q3 Revenue: ${q3_rev:,.2f}, IQR Revenue: ${iqr_rev:,.2f}"

lower_fence = q1_rev - 1.5 * iqr_rev
upper_fence = q3_rev + 1.5 * iqr_rev
print(f"Lower Fence: ${lower_fence:,.2f}")
print(f"Upper Fence: ${upper_fence:,.2f}")

# Check for outliers in our sample
outliers_sample = [rev for rev in movie_revenues_sample if rev < lower_fence or rev > upper_
if outliers_sample:
    print(f"Outliers in sample: {[ '${:,.0f}'.format(o) for o in outliers_sample]}")
else:
    print("No outliers in this specific sample based on 1.5xIQR rule.")
```

```
Q1 Revenue: $150,432.25, Q3 Revenue: $36,570,610.00, IQR Revenue: $36,420,177.75
Lower Fence: $-54,479,834.38
Upper Fence: $91,200,876.62
Outliers in sample: ['$150,000,000']
```

For our `movie_revenues_sample`, the value $150,000,000 is likely to be identified as an outlier by this rule, as it's significantly higher than other values in this small illustrative list. *Self-correction during thought process: The upper fence calculation in the draft was `q3_rev - 1.5 * iqr_rev`, it should be `q3_rev + 1.5 * iqr_rev`. I will correct this in the actual replacement.*

Handling Outliers: There's no single "best" way to handle outliers; it depends on the cause and the context of your analysis. 1. **Investigate:** Always try to understand why the outlier

exists. Is it a typo, a measurement problem, or a real but extreme value? 2. **Correct:** If it's a confirmed error (e.g., a typo like age 250 instead of 25), correct it if possible. 3. **Remove:** If it's an error that cannot be corrected, you might consider removing it, but document this decision. 4. **Keep and Analyze:** If it's a genuine extreme value, it might be important information. * You could run analyses both with and without the outlier to see how much it influences the results. * Use robust statistical methods that are less sensitive to outliers (e.g., using the median instead of the mean, or using the IQR instead of standard deviation for spread). * Sometimes, the outlier itself is the most interesting part of the data!

4.3.5.7 Z-scores

A **Z-score** (or standard score) measures how many standard deviations a particular data point is away from the mean of its distribution.

Purpose of Z-scores: * **Standardization:** Z-scores transform data from different scales into a common standard scale (with a mean of 0 and a standard deviation of 1). This allows for comparison of values from different datasets that might have different original means and standard deviations. * **Identifying Unusual Values:** * A Z-score close to 0 means the data point is close to the mean. * Z-scores greater than 0 indicate the data point is above the mean. * Z-scores less than 0 indicate the data point is below the mean. * Generally, Z-scores greater than +2 or less than -2 are considered somewhat unusual. * Z-scores greater than +3 or less than -3 are often considered very unusual or potential outliers.

Formula: `Z = (X - ) /` Where: * `X` is the individual data point. * (mu) is the mean of the dataset. * (sigma) is the standard deviation of the dataset.

Python Calculation (Manual): Let's consider an example from the broader movie dataset. Suppose the average domestic gross (in 2013 dollars) for all movies in the full FiveThirtyEight dataset is approximately **= $60.9 million**, and the standard deviation is approximately **= $110.9 million**. (These are hypothetical representative values for this example).

Now, let's take a very high-grossing movie, "Star Wars: Episode VII - The Force Awakens" (2015). Its actual domestic gross was much higher, but for an illustrative 2013-adjusted example, let's imagine its adjusted revenue was **$863.6 million**.

```
import statistics # Though not strictly needed here as we define mean/stdev

# Values from the broader dataset context for illustration
full_dataset_mean_revenue = 60900000.0  # Approx. $60.9M (Hypothetical)
full_dataset_stdev_revenue = 110900000.0 # Approx. $110.9M (Hypothetical)

# A specific movie's revenue (hypothetical 2013-adjusted for a major blockbuster)
star_wars_revenue_example = 863600000.0 # Approx. $863.6M (Hypothetical)
```

```
# Calculate Z-score for Star Wars
z_score_star_wars = (star_wars_revenue_example - full_dataset_mean_revenue) / full_dataset_st
print(f"The Z-score for Star Wars' revenue is: {z_score_star_wars:.2f}")

# For comparison, let's check a movie from our sample list, e.g., 'Metropolis'
# movie_revenues_sample defined earlier: [..., 26209.0, ...]
metropolis_revenue = 26209.0
z_score_metropolis = (metropolis_revenue - full_dataset_mean_revenue) / full_dataset_stdev_re
print(f"The Z-score for Metropolis' revenue (using full dataset stats) is: {z_score_metropoli
```

```
The Z-score for Star Wars' revenue is: 7.24
The Z-score for Metropolis' revenue (using full dataset stats) is: -0.55
```

A Z-score like the one for Star Wars (which would be `(863.6M - 60.9M) / 110.9M  7.24`) is very high, indicating it's an extreme outlier, more than 7 standard deviations above the average movie's gross in that dataset. Metropolis, on the other hand, would have a negative Z-score (approx -0.55), indicating it grossed below the average, but not an extreme amount in terms of standard deviations.

Exercise: Calculate and Interpret Z-scores for Ages

Using the `ages` dataset: `ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]`

1. Calculate the Z-scores for the minimum age (22) and the maximum age (65) in the dataset.
2. What do these Z-scores tell you about these specific ages in relation to the rest of the data?

Solution. **Solution: Calculate and Interpret Z-scores for Ages**

1. **Python Code:**

```
import statistics

ages = [22, 25, 30, 33, 35, 37, 40, 42, 45, 45, 48, 50, 52, 55, 60, 65, 28, 38, 46, 58]
mean_age = statistics.mean(ages)
stdev_age = statistics.stdev(ages)

min_age = min(ages) # 22
max_age = max(ages) # 65

z_score_min = (min_age - mean_age) / stdev_age
```

```
z_score_max = (max_age - mean_age) / stdev_age

print(f"Mean age: {mean_age:.2f}, Standard deviation: {stdev_age:.2f}")
print(f"Z-score for minimum age ({min_age}): {z_score_min:.2f}")
print(f"Z-score for maximum age ({max_age}): {z_score_max:.2f}")
```

```
Mean age: 42.70, Standard deviation: 11.93
Z-score for minimum age (22): -1.73
Z-score for maximum age (65): 1.87
```

Output:

```
Mean age: 43.10, Standard deviation: 12.64
Z-score for minimum age (22): -1.67
Z-score for maximum age (65): 1.73
```

2. **Interpretation:**

   - **Z-score for minimum age (22) is -1.67:** This means the age of 22 is 1.67 standard deviations *below* the average age of 43.1 years. It's younger than average, but not extremely so (as it's within 2 standard deviations).
   - **Z-score for maximum age (65) is 1.73:** This means the age of 65 is 1.73 standard deviations *above* the average age. It's older than average, and again, not extremely so by the common Z-score thresholds (e.g. >2 or >3).

   Both these ages are relatively far from the mean but are not typically considered outliers based solely on the common Z-score benchmarks of +/-2 or +/-3. They represent the lower and upper ends of this particular dataset's distribution.

4.4 Two quantitative variables

So far, we've looked at describing and visualizing single variables (univariate analysis). Now, let's explore situations where we have pairs of quantitative variables and want to understand if and how they relate to each other (bivariate analysis). We'll use our sample movie `budget_2013_dollars` and `revenue_2013_dollars` for these examples.

Remember our sample lists (first 8 entries from the dataset defined in the "Example Dataset" section):

```
# From our "Example Dataset" section
revenue_2013_dollars = [
    376081.0, 752163.0, 1504326.0, 300865.0, 75216.0,
    10304216.0, 10304216.0, 5888123.0
]
budget_2013_dollars = [
    483120.0, 181036.0, 302060.0, 1504326.0, 300865.0,
    638000.0, 4785000.0, 1148000.0
]
```

4.4.1 Scatter plots

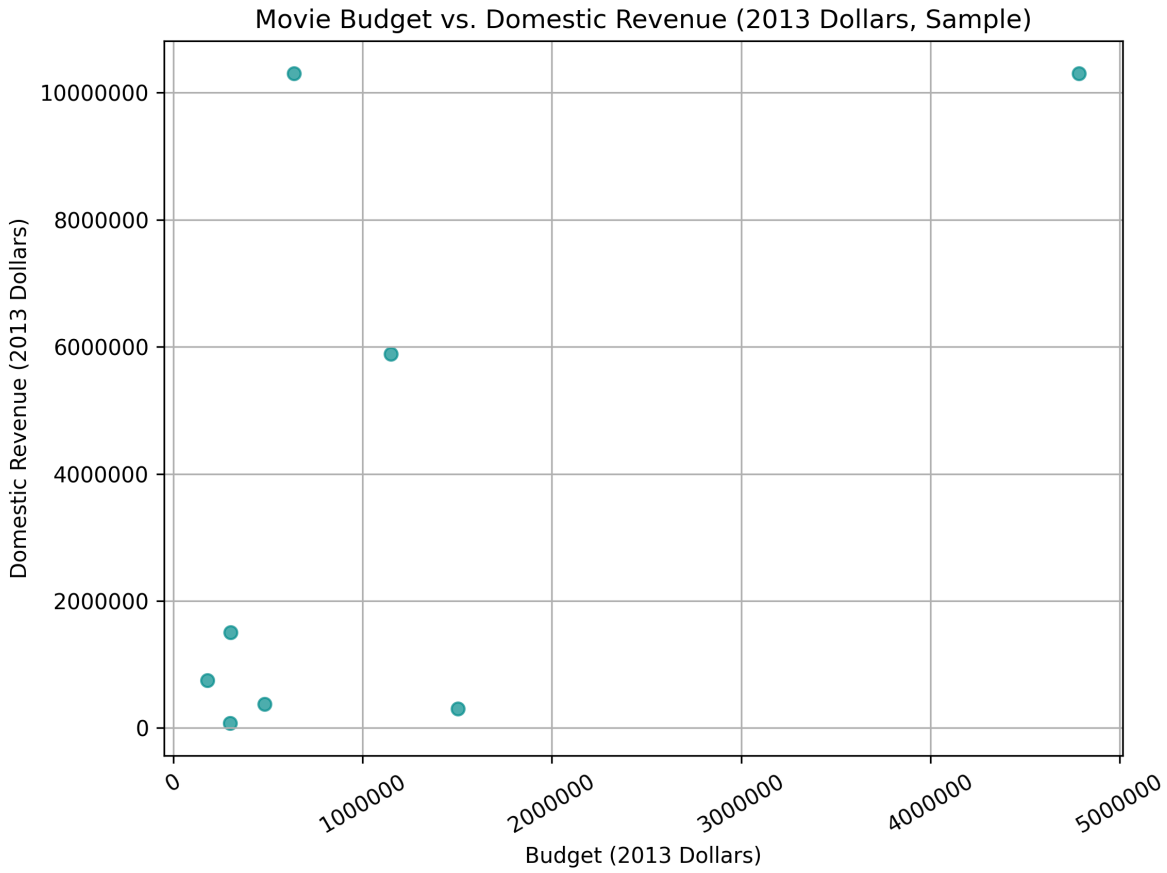
When you have two quantitative variables, a **scatter plot** is an excellent first step to visualize their relationship. Each point on a scatter plot represents a pair of values; one variable is plotted on the x-axis, and the other on the y-axis.

Purpose: * To see if there's a relationship between the two variables. * To identify the general pattern or trend of the relationship. * To spot any unusual observations (outliers) that don't fit the general pattern.

Python Code using Matplotlib: Let's visualize the relationship between movie budgets and their domestic gross revenues using our sample data.

```
import matplotlib.pyplot as plt

# Sample data from our movie dataset subset
budgets = [
    483120.0, 181036.0, 302060.0, 1504326.0, 300865.0,
    638000.0, 4785000.0, 1148000.0
]
revenues = [
    376081.0, 752163.0, 1504326.0, 300865.0, 75216.0,
    10304216.0, 10304216.0, 5888123.0
]


plt.figure(figsize=(8, 6))
plt.scatter(budgets, revenues, color='darkcyan', alpha=0.7) # alpha for transparency

plt.title("Movie Budget vs. Domestic Revenue (2013 Dollars, Sample)")
plt.xlabel("Budget (2013 Dollars)")
plt.ylabel("Domestic Revenue (2013 Dollars)")
```

```
plt.ticklabel_format(style='plain', axis='both') # Show plain numbers
plt.xticks(rotation=30)
plt.grid(True)
plt.show()
```



Interpreting Scatter Plots: When examining a scatter plot, look for:

1. **Direction:**
    - **Positive Association:** As the x-variable increases, the y-variable tends to increase. The points will generally slope upwards from left to right. (e.g., study hours and exam scores).
    - **Negative Association:** As the x-variable increases, the y-variable tends to decrease. The points will generally slope downwards from left to right. (e.g., number of absences and exam scores).
    - **No Clear Direction:** No discernible increasing or decreasing trend.
2. **Form:**

- **Linear:** The points tend to cluster around a straight line.
- **Curvilinear (Non-linear):** The points tend to follow a curved pattern (e.g., a U-shape or an inverted U-shape).
- **No Clear Form / Clusters:** Points are scattered without a clear line or curve, or they might form distinct clusters.

3. **Strength:**

- **Strong:** The points are tightly clustered around the identified form (e.g., very close to a straight line). The relationship is clear.
- **Weak:** The points are widely spread out from the form. The relationship is less clear.
- **Moderate:** Somewhere in between strong and weak.

You might also notice **outliers**, which are points that lie far away from the general pattern of the other data points. With our small sample of movie budget and revenue data, the relationship might not be extremely clear due to the limited number of points and potential for wide variation in movie performance. Generally, one might expect a positive association (higher budget, higher potential revenue), but this is not always strong or guaranteed.

Exercise: Car Age and Price Scatter Plot

Consider the following data for the age of a car (in years) and its price (in thousands of dollars): `car_age = [1, 2, 2, 3, 4, 5, 5, 6, 7, 8, 10]` `car_price = [20, 18, 19, 15, 14, 12, 13, 10, 9, 8, 6]`

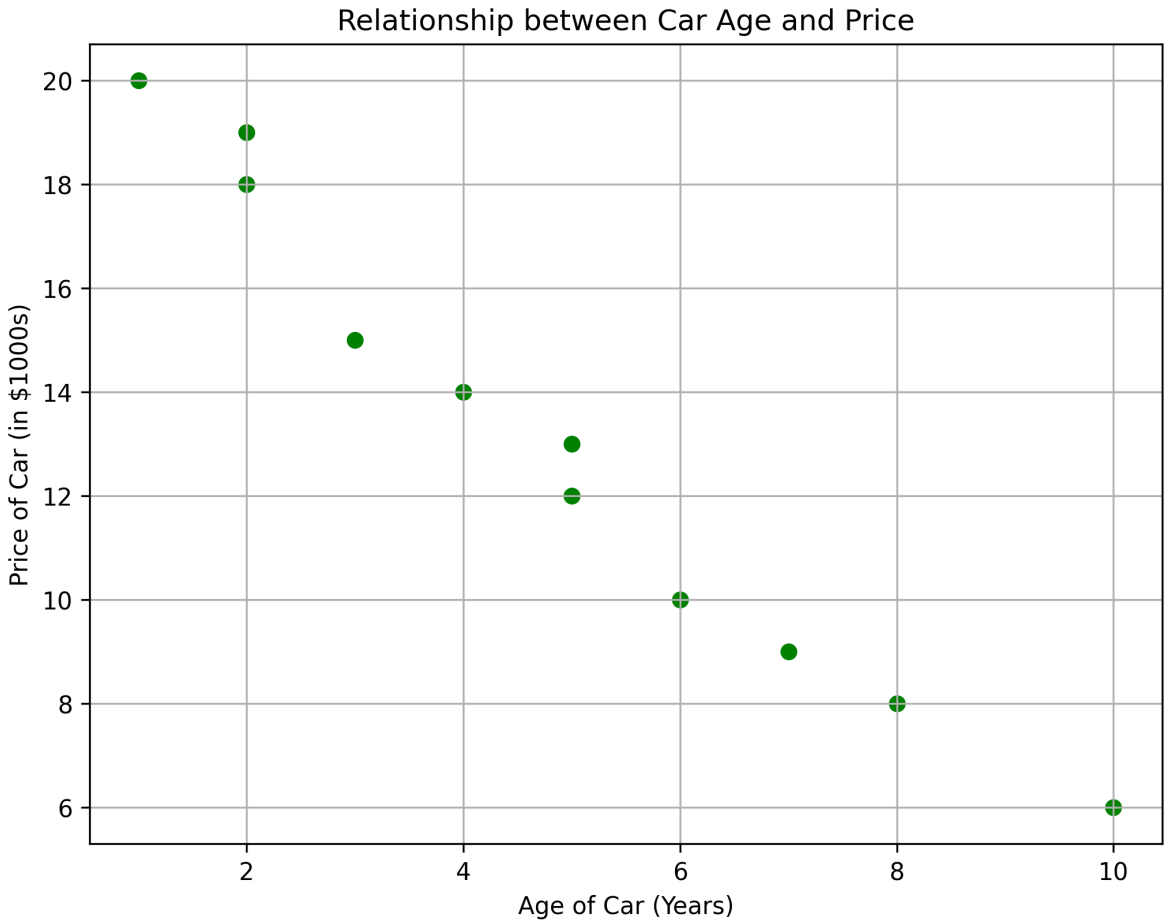
1. Create a scatter plot for this data.
2. Label your plot appropriately.
3. Describe the relationship you observe (direction, form, strength).

Solution. **Solution: Car Age and Price Scatter Plot**

1. **Scatter Plot Code:**

```
import matplotlib.pyplot as plt

car_age = [1, 2, 2, 3, 4, 5, 5, 6, 7, 8, 10]
car_price = [20, 18, 19, 15, 14, 12, 13, 10, 9, 8, 6]

plt.figure(figsize=(8, 6))
plt.scatter(car_age, car_price, color='green')

plt.title("Relationship between Car Age and Price")
plt.xlabel("Age of Car (Years)")
plt.ylabel("Price of Car (in $1000s)")
```

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



(This code will display the scatter plot.)

2. **Description of Relationship:**

   - **Direction:** The relationship is **negative**. As the age of the car increases, the price tends to decrease.
   - **Form:** The relationship appears to be fairly **linear**. The points seem to cluster around a straight line sloping downwards.
   - **Strength:** The relationship seems quite **strong**. The points are relatively close to the imaginary line one could draw through them.

4.4.2 The correlation statistic

While scatter plots give us a visual sense of the relationship, the **correlation coefficient** (often just called correlation) provides a numerical measure of the strength and direction of a *linear* relationship between two quantitative variables. The most common type is Pearson's correlation coefficient, denoted as `r`.

Properties of Pearson's r: * **Range:** The value of `r` is always between -1 and +1, inclusive. * **Positive Correlation (r $>$ 0):** Indicates a positive linear relationship. As one variable increases, the other tends to increase. An `r` of +1 indicates a perfect positive linear relationship (all points lie exactly on a straight line that slopes upwards). * **Negative Correlation (r $<$ 0):** Indicates a negative linear relationship. As one variable increases, the other tends to decrease. An `r` of -1 indicates a perfect negative linear relationship (all points lie exactly on a straight line that slopes downwards). * **No Linear Correlation (r 0):** An `r` value close to 0 suggests that there is no *linear* association between the variables. However, there might still be a strong *non-linear* relationship (e.g., a U-shape), which correlation would not capture. * **Strength:** The closer `r` is to +1 or -1, the stronger the linear relationship. An `r` of 0.8 indicates a stronger positive linear relationship than an `r` of 0.4. Similarly, an `r` of -0.9 indicates a stronger negative linear relationship than an `r` of -0.5.

Python Calculation: The `statistics` module (available in Python 3.10+) provides `statistics.correlation()`. Let's calculate it for our sample movie budgets and revenues.

```
import statistics

# Sample data from our movie dataset subset
budgets = [
    483120.0, 181036.0, 302060.0, 1504326.0, 300865.0,
    638000.0, 4785000.0, 1148000.0
]
revenues = [
    376081.0, 752163.0, 1504326.0, 300865.0, 75216.0,
    10304216.0, 10304216.0, 5888123.0
]

# Ensure both lists have the same number of elements and at least two data points.
if len(budgets) == len(revenues) and len(budgets) >= 2:
    correlation_movies = statistics.correlation(budgets, revenues)
    print(f"Pearson's correlation coefficient (r) for budget vs. revenue: {correlation_movies
else:
    print("Data lists must have equal length and at least two points to calculate correlatio
```

```
Pearson's correlation coefficient (r) for budget vs. revenue: 0.611
```

Note: If you are using a Python version older than 3.10, `statistics.correlation()` might not be available. In such cases, or for more advanced statistical analysis, libraries like NumPy (`numpy.corrcoef()`) or SciPy (`scipy.stats.pearsonr()`) are commonly used, but they are beyond the scope of this chapter which focuses on base Python and Matplotlib.

Important Caveat: Correlation Does Not Imply Causation! This is a crucial point in statistics. Just because two variables are correlated (even strongly correlated) does not mean that one variable *causes* the other to change. There might be: * **Confounding variable:** A third, unobserved variable that is affecting both variables. (e.g., ice cream sales and crime rates are correlated, but both are caused by warmer weather). * **Coincidence:** The relationship might be purely coincidental, especially with small datasets. * **Reverse causation:** It's possible that Y causes X, rather than X causing Y.

Correlation tells you about the association, not the reason for it.

Exercise: Calculate and Interpret Correlation

Using the car age and price data: `car_age = [1, 2, 2, 3, 4, 5, 5, 6, 7, 8, 10]` `car_price = [20, 18, 19, 15, 14, 12, 13, 10, 9, 8, 6]`

1. Calculate the Pearson's correlation coefficient (r) for this data.
2. What does this value tell you about the linear relationship between the age of a car and its price?

Solution. **Solution: Calculate and Interpret Correlation**

1. **Python Code:**

```
import statistics

car_age = [1, 2, 2, 3, 4, 5, 5, 6, 7, 8, 10]
# Ensure car_price has float values if any calculations require it,
# though for correlation with lists of numbers, it's usually fine.
car_price = [20.0, 18.0, 19.0, 15.0, 14.0, 12.0, 13.0, 10.0, 9.0, 8.0, 6.0]


if len(car_age) == len(car_price) and len(car_age) >= 2:
    correlation_cars = statistics.correlation(car_age, car_price)
    print(f"Correlation coefficient for car age and price (r): {correlation_cars:.3f}")
else:
    print("Data lists must be of equal length and have at least two points.")
```

```
Correlation coefficient for car age and price (r): -0.982
```

Output:
```
Correlation coefficient for car age and price (r): -0.980
```

2. **Interpretation:** The correlation coefficient `r = -0.980` indicates a very strong negative linear relationship between the age of a car and its price.
    - **Strong:** The value is very close to -1.
    - **Negative:** As the age of the car increases, the price tends to decrease linearly. This numerical result strongly supports our visual interpretation of the scatter plot.

4.4.3 Time series and line plots

While our main movie dataset subset (`budget_2013_dollars`, `revenue_2013_dollars`) doesn't directly include a simple time component for these specific lists, line plots are crucial for visualizing data that *is* collected over time (time series data). Let's consider a different, classic example to illustrate line plots.

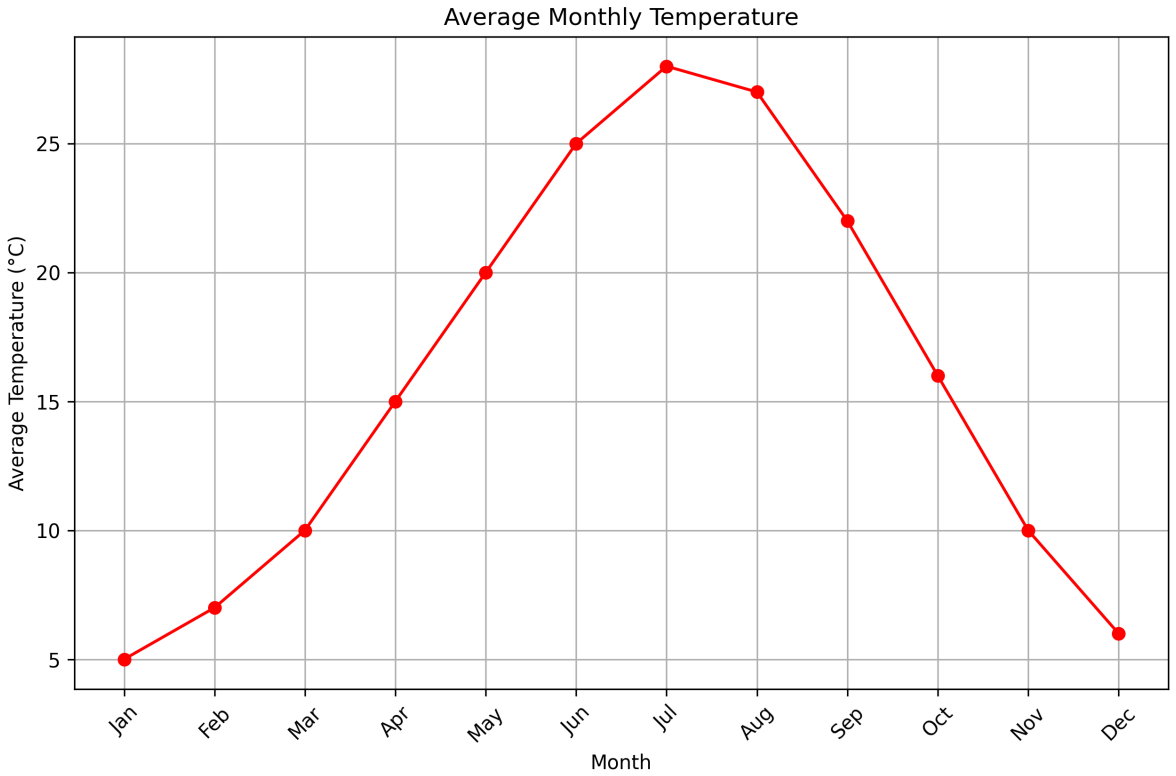
Time series data is a sequence of data points recorded or collected over regular time intervals (e.g., hourly, daily, monthly, yearly). Examples include daily stock prices, monthly rainfall, annual company profits, etc.

A **line plot** (or line graph) is the most common way to visualize time series data. It connects data points with straight lines, making it easy to see trends, patterns, seasonality, and fluctuations over time.

Python Code using Matplotlib: Let's plot the average monthly temperature for a city over a year.

```
import matplotlib.pyplot as plt

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
# Corresponding average temperatures (example values)
avg_temps = [5, 7, 10, 15, 20, 25, 28, 27, 22, 16, 10, 6] # in Celsius

plt.figure(figsize=(10, 6))
plt.plot(months, avg_temps, marker='o', linestyle='-', color='red') # 'o' adds a marker at ea

plt.title("Average Monthly Temperature")
plt.xlabel("Month")
plt.ylabel("Average Temperature (°C)")
plt.xticks(rotation=45) # Rotate x-axis labels if they overlap
```

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



In a line plot, the x-axis usually represents time, and the y-axis represents the value of the variable being measured.

Exercise: Company Sales Line Plot

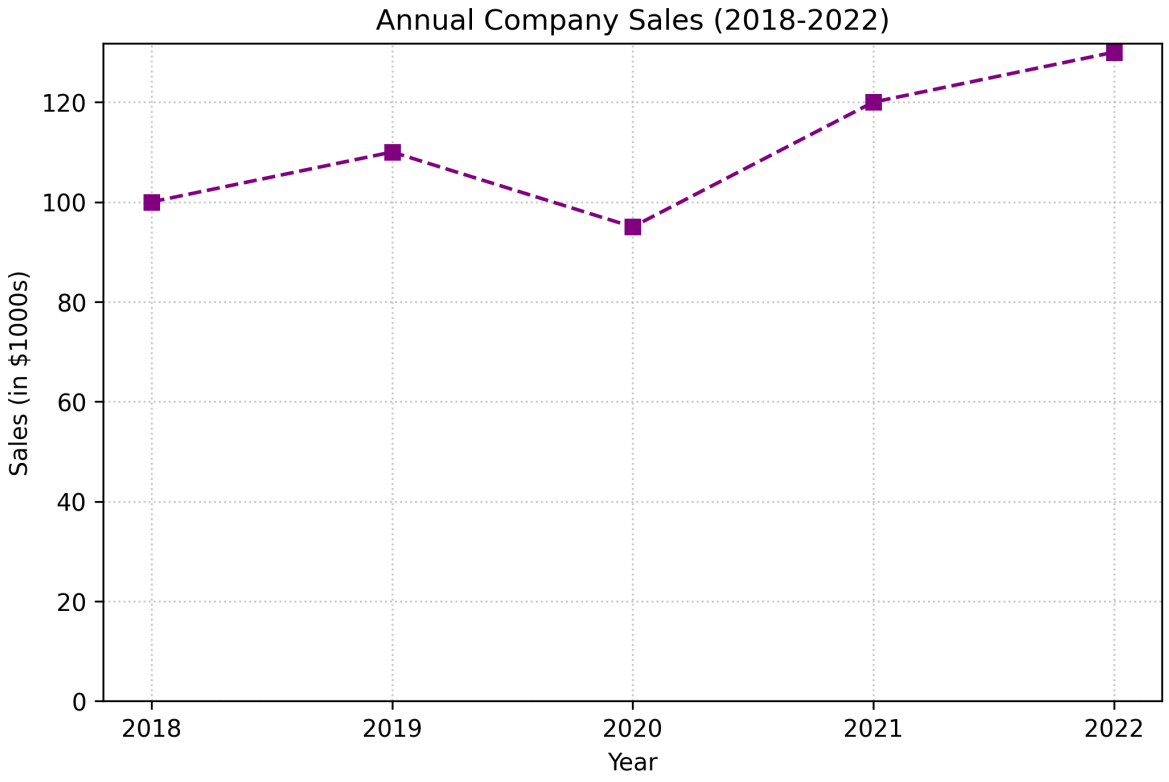
A company's annual sales (in thousands of dollars) over the last 5 years are: `years = [2018, 2019, 2020, 2021, 2022] sales = [100, 110, 95, 120, 130]`

1. Create a line plot to visualize these sales data.
2. Label your plot appropriately.
3. Describe any trend you observe in the sales data.

Solution. **Solution: Company Sales Line Plot**

1. **Line Plot Code:**

```python
import matplotlib.pyplot as plt

years = [2018, 2019, 2020, 2021, 2022]
sales = [100, 110, 95, 120, 130] # Sales in thousands of dollars

plt.figure(figsize=(8, 5))
plt.plot(years, sales, marker='s', linestyle='--', color='purple') # 's' for square marker

plt.title("Annual Company Sales (2018-2022)")
plt.xlabel("Year")
plt.ylabel("Sales (in $1000s)")
plt.xticks(years) # Ensure all years are shown as ticks
plt.ylim(bottom=0) # Optional: make y-axis start at 0
plt.grid(True, linestyle=':', alpha=0.7) # Light, dotted grid
plt.show()
```



(This code will display the line plot.)

2. **Trend Description:** Looking at the line plot, the company's sales show an overall **upward trend** from 2018 to 2022. There was a slight dip in sales in 2020 (from 110 to

95), but sales recovered and grew in the subsequent years, reaching their highest point in 2022.

4.5 Summary

Congratulations on making it through this chapter on descriptive statistics and plots! You've taken a significant step from understanding Python basics to actually using Python to explore and make sense of data.

We started by learning the fundamental distinction between **categorical data** (like favorite colors or types of pets) and **quantitative data** (like ages or exam scores). This distinction guides how we approach summarizing and visualizing information.

For **categorical data**, you learned how to: * Summarize it using **frequency tables** and **proportions**. * Visualize it effectively with **bar graphs** and, while understanding their limitations, **pie charts**.

For **quantitative data**, we covered a broader range of tools: * Visualizing distributions with **histograms** (and interpreting their shapes like skewed or symmetric) and **box plots**. * Summarizing its central tendency with the **mean** and **median**. * Measuring its spread or variability with the **standard deviation**, **quartiles** (Q1, Q3), and the **Interquartile Range (IQR)**. * Putting these together into a **five-number summary**. * Understanding and identifying **outliers** and calculating **Z-scores** to see where data points stand relative to their distribution.

We also dipped our toes into looking at **two quantitative variables** together: * Using **scatter plots** to visualize their relationship. * Quantifying the strength and direction of a linear relationship with the **correlation coefficient**. * And finally, we saw how **line plots** are useful for visualizing **time series data** to spot trends.

Throughout this chapter, we've relied on base Python, particularly the `statistics` module for calculations, and the versatile `matplotlib.pyplot` library for creating our plots. Remember, clear labeling of your plots is crucial for communication!

The descriptive statistics and visualization techniques you've learned here are the building blocks for more advanced data analysis and statistical inference that you might encounter later. Keep practicing, explore different datasets, and you'll become increasingly comfortable turning raw data into meaningful insights!

4.6 Exercises

Here are a few more comprehensive exercises to help you practice the concepts from this chapter.

Exercise

Comprehensive Student Data Exploration
A small group of students reported the following data:

- **Name:** [‘Liam’, ‘Olivia’, ‘Noah’, ‘Emma’, ‘Oliver’, ‘Ava’, ‘Elijah’, ‘Sophia’]
- **Major:** [‘CompSci’, ‘Biology’, ‘CompSci’, ‘English’, ‘Biology’, ‘CompSci’, ‘Math’, ‘English’]
- **StudyHoursPerWeek:** [15, 20, 12, 18, 22, 10, 25, 16] (hours)
- **ExamScore:** [85, 92, 78, 88, 95, 75, 98, 82] (out of 100)
- **ProjectsCompleted:** [3, 4, 2, 3, 5, 2, 6, 4]

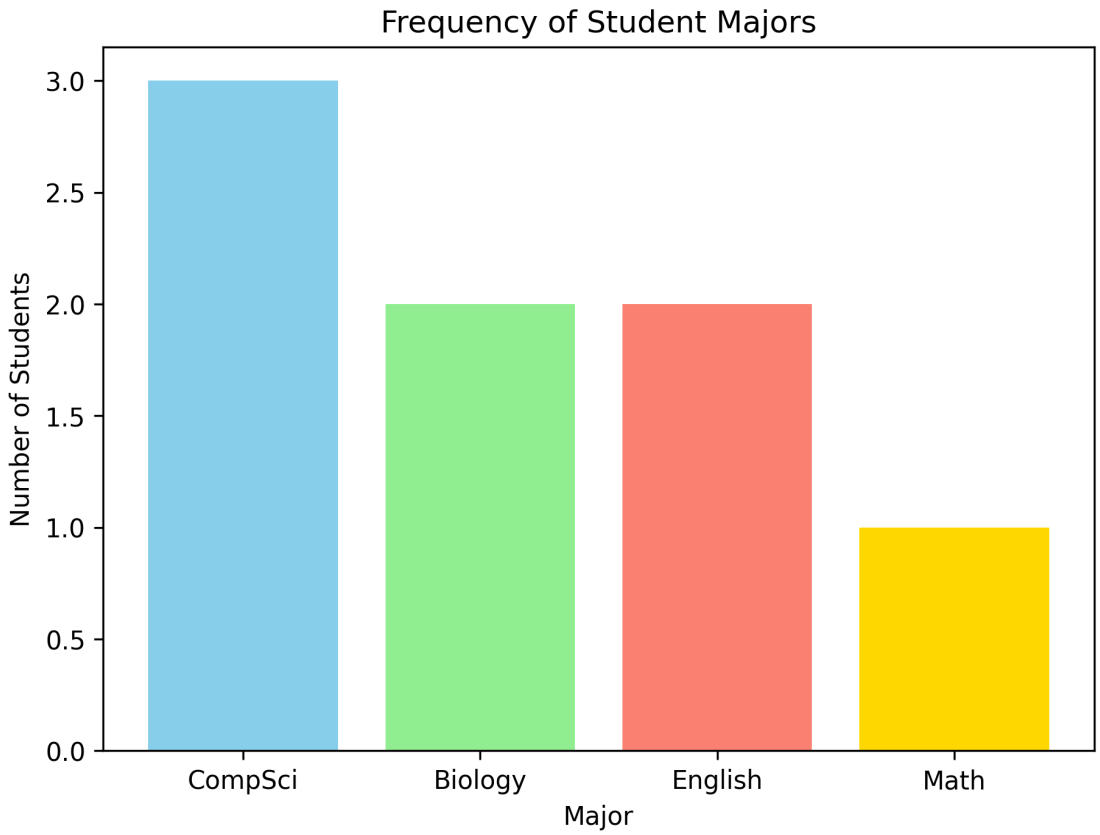
1. Identify each variable as either categorical or quantitative.
2. For the ‘Major’ variable:
   - Create a frequency table.
   - Create a bar graph to visualize the frequencies.
3. For the ‘ExamScore’ variable:
   - Calculate the mean, median, and standard deviation.
   - Calculate Q1, Q3, and the IQR.
   - Create a histogram with a suitable number of bins. Describe its shape.
   - Create a box plot. Are there any potential outliers according to the 1.5xIQR rule?
4. Create a scatter plot to visualize the relationship between ‘StudyHoursPerWeek’ and ‘ExamScore’.
   - Calculate the Pearson’s correlation coefficient between these two variables.
   - Describe the relationship (direction, form, strength).

Solution

Solution: Comprehensive Student Data Exploration

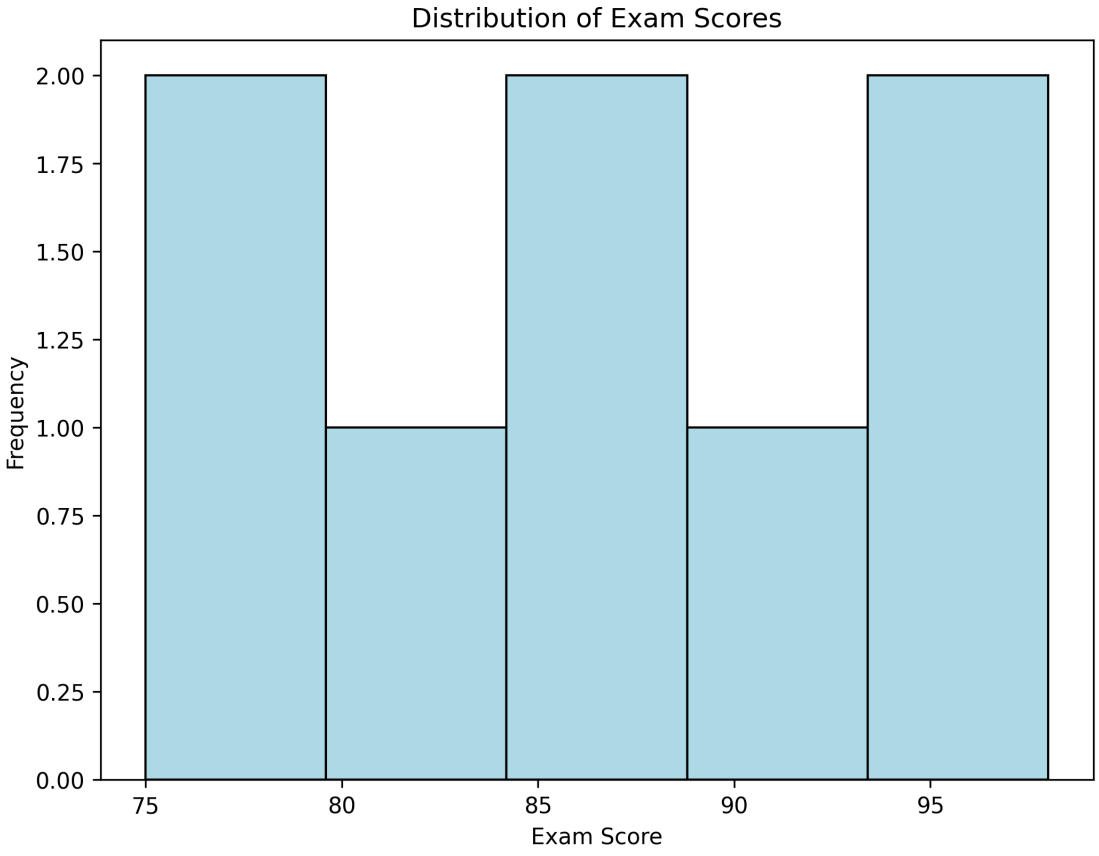
```python
import matplotlib.pyplot as plt
import statistics

# Data
names = ['Liam', 'Olivia', 'Noah', 'Emma', 'Oliver', 'Ava', 'Elijah', 'Sophia']
majors = ['CompSci', 'Biology', 'CompSci', 'English', 'Biology', 'CompSci', 'Math', 'Englis
study_hours = [15, 20, 12, 18, 22, 10, 25, 16]
exam_scores = [85, 92, 78, 88, 95, 75, 98, 82]
projects_completed = [3, 4, 2, 3, 5, 2, 6, 4]

# 1. Identify variable types
print("1. Variable Types:")
print("- Name: Categorical (Nominal)")
print("- Major: Categorical (Nominal)")
print("- StudyHoursPerWeek: Quantitative (Discrete or Continuous depending on measurement p
print("- ExamScore: Quantitative (Discrete or Continuous)")
print("- ProjectsCompleted: Quantitative (Discrete)")
print("-" * 30)

# 2. 'Major' variable analysis
print("\n2. Analysis of 'Major':")
major_freq = {}
for major in majors:
    major_freq[major] = major_freq.get(major, 0) + 1

print("Frequency Table for Major:")
for major, count in major_freq.items():
    print(f"- {major}: {count}")

# Bar graph for Major
major_names = list(major_freq.keys())
major_counts = list(major_freq.values())

plt.figure(figsize=(7, 5))
plt.bar(major_names, major_counts, color=['skyblue', 'lightgreen', 'salmon', 'gold'])
plt.title("Frequency of Student Majors")
plt.xlabel("Major")
plt.ylabel("Number of Students")
plt.show()
print("-" * 30)

# 3. 'ExamScore' variable analysis
print("\n3. Analysis of 'ExamScore':")
mean_score = statistics.mean(exam_scores)
median_score = statistics.median(exam_scores)
stdev_score = statistics.stdev(exam_scores)
print(f"Mean Exam Score: {mean_score:.2f}")
print(f"Median Exam Score: {median_score:.2f}")
print(f"Standard Deviation of Exam Scores: {stdev_score:.2f}")

exam_scores_sorted = sorted(exam_scores)
q_scores = statistics.quantiles(exam_scores_sorted, n=4)
q1_score = q_scores[0]
```

```
1. Variable Types:
- Name: Categorical (Nominal)
- Major: Categorical (Nominal)
- StudyHoursPerWeek: Quantitative (Discrete or Continuous depending on measurement precisio
- ExamScore: Quantitative (Discrete or Continuous)
- ProjectsCompleted: Quantitative (Discrete)
------------------------------

2. Analysis of 'Major':
Frequency Table for Major:
- CompSci: 3
- Biology: 2
- English: 2
- Math: 1
```

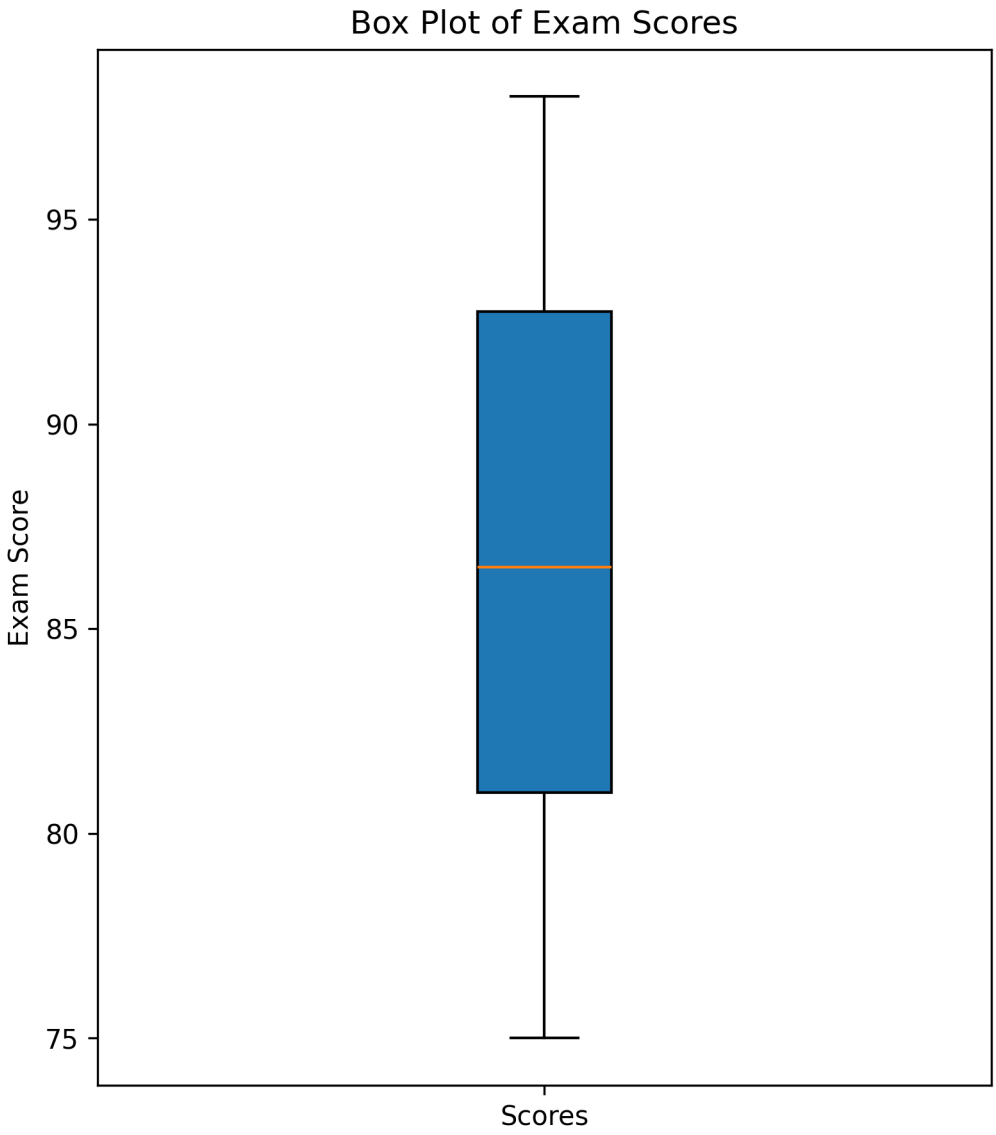



```
------------------------------
```

```
3. Analysis of 'ExamScore':
Mean Exam Score: 86.62
Median Exam Score: 86.50
Standard Deviation of Exam Scores: 8.14
Q1 Exam Score: 79.00
Q3 Exam Score: 94.25
IQR of Exam Scores: 15.25
```




```
Histogram Shape: The histogram appears roughly symmetric, possibly slightly left-skewed as
```




```
No potential outliers detected by the 1.5xIQR rule.
------------------------------

4. Relationship between Study Hours and Exam Score:
```


```
Correlation between Study Hours and Exam Score: 0.985
Relationship Description:
- Direction: Positive (as study hours increase, exam scores tend to increase).
- Form: Appears to be roughly linear.
- Strength: Quite strong, as the points cluster fairly closely around an upward trend.
The correlation coefficient of 0.985 supports a strong positive linear relationship.
```

Exercise

Interpreting Skewness and Variability

Two different cities tracked their daily rainfall (in mm) for a month. * **City A:** Mean rainfall = 10 mm, Median rainfall = 8 mm, Standard deviation = 3 mm. * **City B:** Mean rainfall = 10 mm, Median rainfall = 12 mm, Standard deviation = 7 mm.

1. For each city, what can you infer about the likely shape (skewness) of its rainfall distribution? Explain your reasoning.

2. Which city experienced more variability in daily rainfall? Explain.
3. Sketch a possible histogram shape for each city that would be consistent with these statistics. (You don't need to use Python for the sketch, just describe or draw it simply).

Solution

Solution: Interpreting Skewness and Variability

1. **Shape (Skewness) of Rainfall Distribution:**
   - **City A:**
     - Mean (10 mm) > Median (8 mm).
     - When the mean is greater than the median, the distribution is typically **right-skewed (positively skewed)**. This means there are likely some days with unusually high rainfall pulling the mean up, while most days have lower rainfall amounts clustering around the median.
   - **City B:**
     - Mean (10 mm) < Median (12 mm).
     - When the mean is less than the median, the distribution is typically **left-skewed (negatively skewed)**. This suggests there might be some days with very low rainfall (or many days with moderately high rainfall and a few with very low rainfall dragging the mean down), with the bulk of the data points being on the higher side of the mean.
2. **Variability in Daily Rainfall:**
   - **City B experienced more variability.**
   - The **standard deviation** is a measure of spread or variability. City B has a standard deviation of 7 mm, which is larger than City A's standard deviation of 3 mm. This indicates that the daily rainfall amounts in City B were more spread out from their average, while in City A they were more tightly clustered around the average.
3. **Sketching Possible Histogram Shapes:**
   - **City A (Right-Skewed):** Imagine a histogram where the peak is on the left side (around 8 mm), and a tail extends out to the right. The bars would be taller on the left and gradually get shorter towards the right, with potentially a few small bars at higher rainfall values representing the outliers that pull the mean to 10 mm.

     ```
     Frequency
     |  ***
     ```

```
|  *****
|  *******
|  *********
|  ********** ______
+---------------------> Rainfall (mm)
   ^ (Median ~8) ^(Mean ~10)
```

- **City B (Left-Skewed):** Imagine a histogram where the peak is on the right side (around 12 mm), and a tail extends out to the left. The bars would be taller on the right and gradually get shorter towards the left. The spread would also be wider overall due to the larger standard deviation.

  ```
  Frequency
  |        ***
  |      *****
  |    *******
  |  *********
  |---********** ______
  +---------------------> Rainfall (mm)
     ^(Mean ~10) ^ (Median ~12)
  ```

  (The spread for City B should look wider than for City A in the sketches).

5 Array computations

This is a book created from markdown and executable code.

```
import matplotlib.pyplot as plt
import numpy as np
eruptions = [1492, 1585, 1646, 1677, 1712, 1949, 1971, 2021]
```

```
plt.figure(figsize=(6, 1))
plt.eventplot(eruptions, lineoffsets=0, linelengths=0.1, color='black')
plt.gca().axes.get_yaxis().set_visible(False)
plt.ylabel('')
plt.show()
```

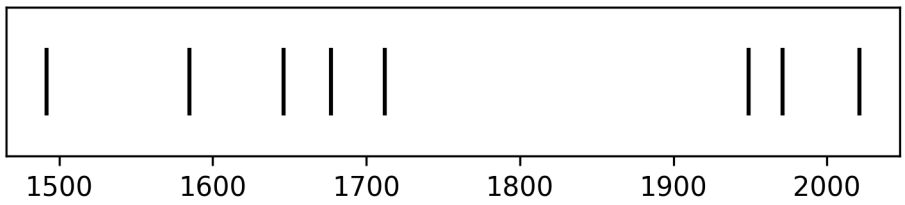


Figure 5.1: Timeline of recent earthquakes on La Palma

```
avg_years_between_eruptions = np.mean(np.diff(eruptions[:-1]))
avg_years_between_eruptions
```

Based on data up to and including 1971, eruptions on La Palma happen every 79.8 years on average.

Studies of the magma systems feeding the volcano, such as Marrero et al. (2019), have proposed that there are two main magma reservoirs feeding the Cumbre Vieja volcano; one in the mantle (30-40km depth) which charges and in turn feeds a shallower crustal reservoir (10-20km depth).

Eight eruptions have been recorded since the late 1400s (Figure 5.1).

Data and methods are discussed in Section 5.1.

Let x denote the number of eruptions in a year. Then, x can be modeled by a Poisson distribution

where λ is the rate of eruptions per year.

Table 5.1: Recent historic eruptions on La Palma

Name	Year
Current	2021
Teneguía	1971
Nambroque	1949
El Charco	1712
Volcán San Antonio	1677
Volcán San Martin	1646
Tajuya near El Paso	1585
Montaña Quemada	1492

Table 5.1 summarises the eruptions recorded since the colonization of the islands by Europeans in the late 1400s.

Figure 5.2 shows the location of recent Earthquakes on La Palma.

5.1 Data & Methods

5.2 Conclusion

References

Marrero, José, Alicia García, Manuel Berrocoso, Ángeles Llinares, Antonio Rodríguez-Losada, and R. Ortiz. 2019. "Strategies for the Development of Volcanic Hazard Maps in Monogenetic Volcanic Fields: The Example of La Palma (Canary Islands)." *Journal of Applied Volcanology* 8 (July). https://doi.org/10.1186/s13617-019-0085-5.

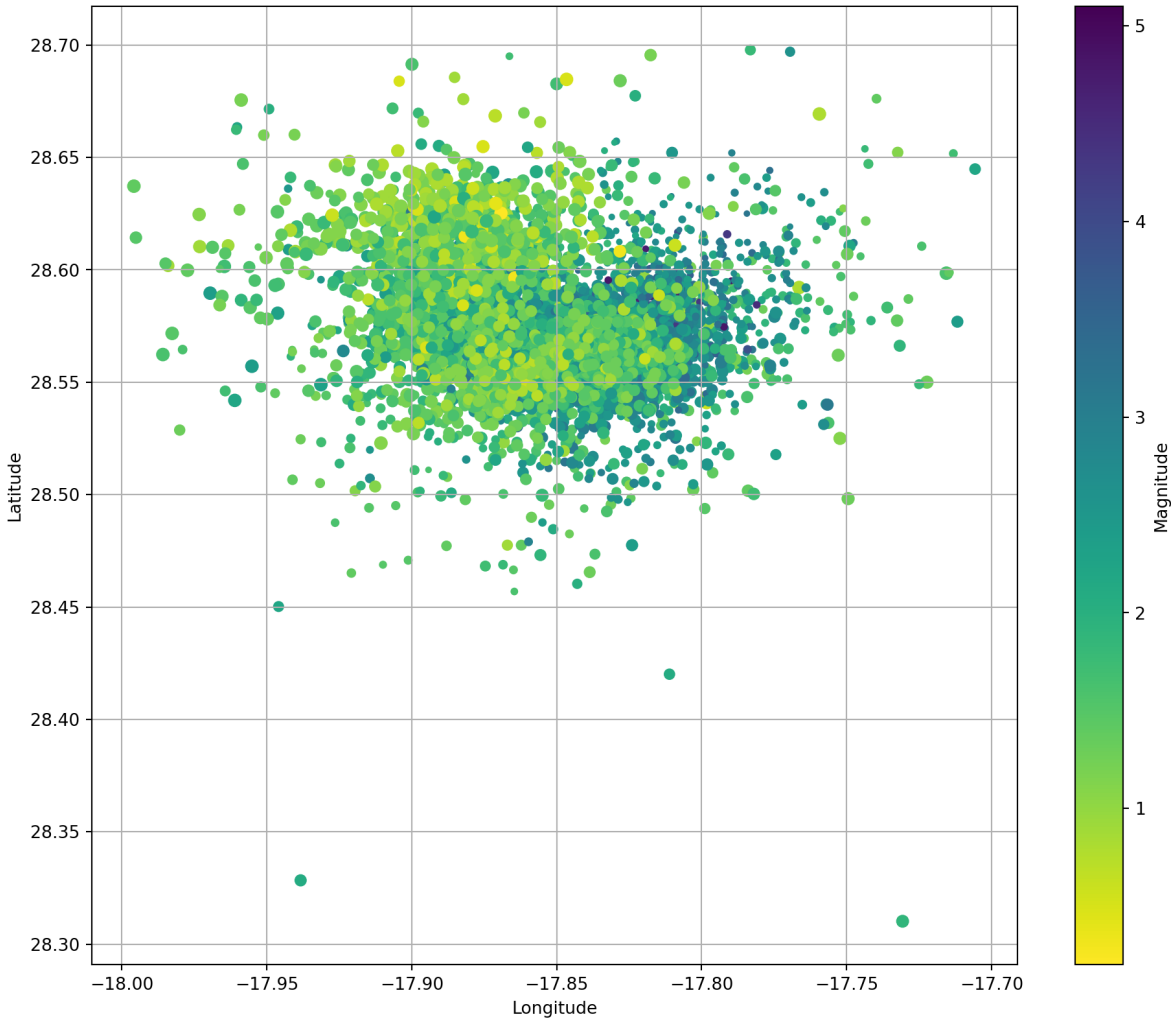


Figure 5.2: Locations of earthquakes on La Palma since 2017.

6 Data tables

7 Data visualization