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

Now that we have covered some of the basic Python syntax we are ready to start analyzing some data.

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()
```

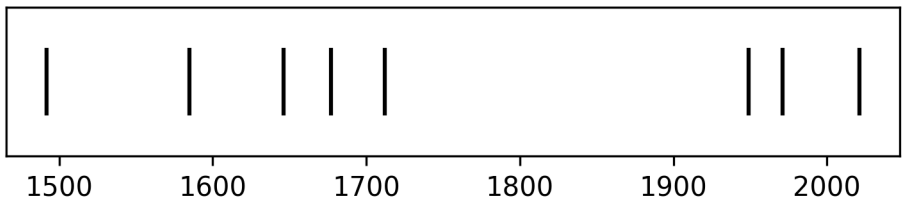

Figure 5.1: Timeline of recent earthquakes on La Palma

```
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.

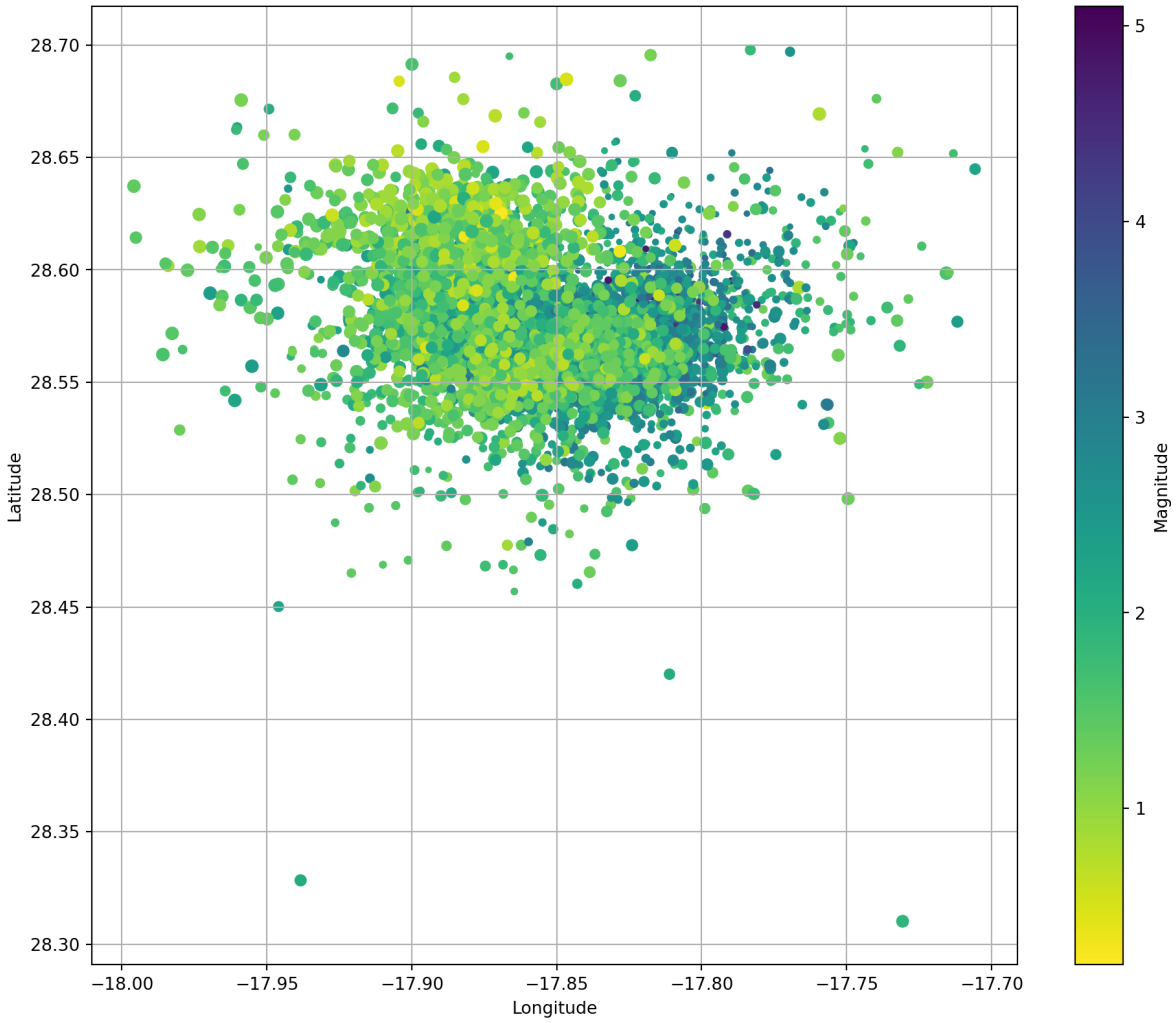

Figure 5.2: Locations of earthquakes on La Palma since 2017.

6 Data tables

7 Data visualization