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

Now that we have discussed what data science is, let's learn some of the basics of the Python programming language. Once we have learned some of these Python basics we can begin to analyze data!

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 number. For example, if want want to add 5 plus 2 and then multiple 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 multiple 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

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 that 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 **??**, 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
```

```
SyntaxError: invalid syntax (3783800731.py, line 1)
  Cell In[6], line 1
    8* *2
       ^
SyntaxError: invalid syntax
```

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.

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.

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

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 an 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 multiple 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? Please calculate by creating *meaningful* variable names for:

1. The minimum wage amount
2. The number of hours in a week
3. The number of weeks in a year

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

Solution

```
hours = 24
days = 7
total_hours_in_a_week = hours * days
total_hours_in_a_week
```

```
168
```

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 both the numbers `2` and `3` in the same code cell. If we did not use the `print()` function, only the number `3` would be printed since it is the last line in the cell, but the number 2 would not be printed because it is on the last line in the cell.

```
# 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 detial 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
```