Programming for TiSEM Essential Digital Skills

Pieter Kleer   Christoph Walsh

Table of contents

Chapter 1

About

1.1 Welcome

Welcome to the online "book" for the Programming Module of the TiSEM Minor Essential Digital Skills. We will follow the content in this book during the lectures and it is the basis of the material that will appear on the exam, so you should read through this book carefully. Because this book is new, it is likely that we will make some edits throughout the course.

Before we jump into coding with Python, we will start by discussing what programming is at the most basic level and motivating why we are learning how to code in Python in the first place.

1.2 What is a Programming Language?

Without getting into a complicated details, a programming language is a way to communicate to a computer via written text in a way that the computer can understand you so that you can instruct it to do various operations. This is very different to how we usually interact with a computer, which often involves pointing and clicking on different buttons and menus with your mouse.

Knowing how to program is a very useful skill because you can automate repetitive tasks that would otherwise take you a very long time if you had to them "by hand" (i.e. by clicking things with your mouse). For example, suppose you work in a hotel in a city and you need to check how much your competitors are charging for rooms on different days so that you can adjust prices to stay competitive. Every day you have to go to all the different websites of the competing hotels and take note of the prices in an Excel sheet. With programming, what you could do instead is write code that tells the computer to automatically visit those websites every day, record the hotel room prices, and put them into a

dataset for you. This is a process called *web scraping* and can be done with Python. This is just one example of the many ways programming languages can automate repetitive tasks.

When humans speak to each other and someone makes a grammar mistake, it usually isn't a big deal. We usually know what they mean. But if you make a "syntax error" in a programming language, it won't understand what you mean. The computer will throw an error. What is worse still is a "semantic error" which is when the computer runs your code without an error but does something you didn't want it to do. Therefore we need to be very careful when writing in a programming language.

1.3   Why Python?

There are many different programming languages out there: C, C++, C#, Java, JavaScript, R, Julia, Stata, MATLAB, Fortran, Ruby, Perl, Rust, Go, Lua, Swift - the list goes on. So why should we learn Python over these other alternatives?

The best programming language depends on the task you want to accomplish. Are you building a website, writing computer software, creating a game, or analyzing data? While many languages could perform all of these tasks, some languages excel in some of them. In this course our goal is to learn basic programming techniques required for data science, and Python is by far the most popular programming language for this task. But it's not only useful for that. It is also often used in web development, creating desktop applications and games, and for scientific computations. It is therefore a very versatile programming language that can complete a very wide range of tasks.

Python is also completely free and open source and can run on all common operating systems. This means you can share your code with anyone and they will be able to run it, no matter what computer they are on or where they are in the world.

There is also a very large active community that creates packages to do a wide-range of operations, keeping Python up to date with the latest developments. For example, excellent community help is available at Stackoverflow, so if you Google how to do something in Python most likely that question has already been answered on Stackoverflow. Funnily enough, a key skill to develop with programming is how to formulate your question into Google to land on the right Stackoverflow page. More recently, ChatGPT and other generative artificial intelligence tools have become very useful resources for Python. These can write excellent Python code and also explain all the steps it takes, so I encourage you to use them to help you learn. However, keep in mind you will not be able to use AI in the exam, so don't become too reliant on it!

These days employers are increasingly looking to hire people with programming skills. Knowing how to program in Python - one of the most commonly used

languages by companies - is therefore a very valuable addition to your CV.

Chapter 2

Getting Started

In this chapter we will learn how to install Python and run our very first command.

2.1 Installing Anaconda

The easiest way to install Python is by installing Anaconda. You can do this by visiting https://www.anaconda.com/download.

You should see this page:
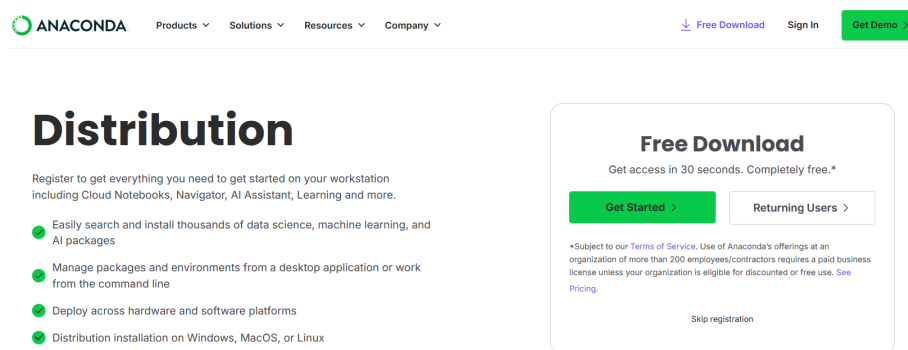
Figure 2.1: Anaconda Download Page

You should click the "Skip registration" button (although feel free to register if you like). You will then see the following page:
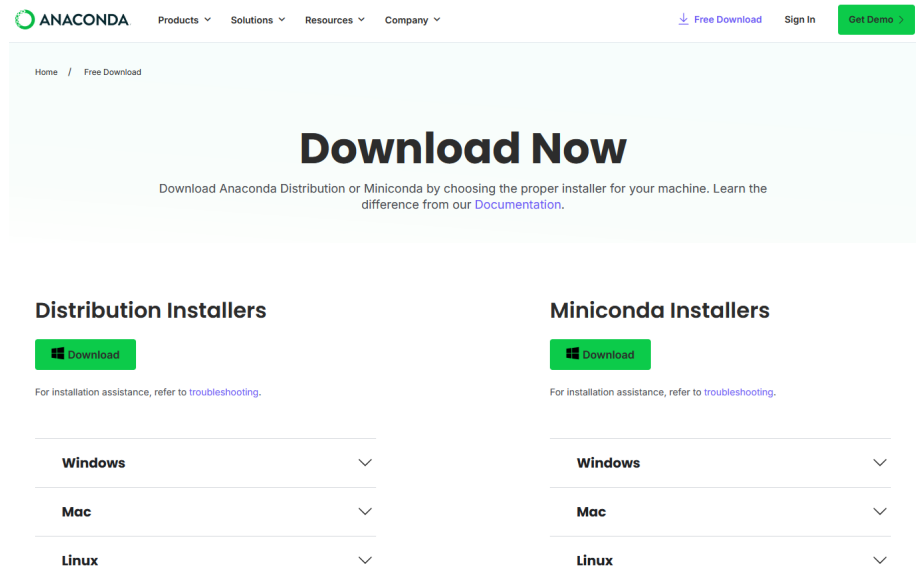
Figure 2.2: Anaconda Download Page

You should then click on the "Download" button. Mac users will see a Mac logo instead.

After downloading the file, click on it to install it. Follow the installation wizard and keep all the default options during installation.

After installation you will see a number of new applications on your computer. These are:

- *Spyder.* This is a computer application that allows you to write Python scripts and execute them to see the output. Such an application is called an Integrated Desktop Environment (IDE). We will see how to use this below.
- *Jupyter Notebook.* This is a web application that allows you to write a notebook (like a report) with text and Python code snippets with output. We will learn how to use this application later in this course.
- *Anaconda Prompt.* This is a way to manage and update packages from the command line. Packages are collections of modules that give Python more functionality, allowing you to perform different types of tasks more easily. All the packages that we will need for this course are installed by default when we install Anaconda, so we will not need to use this in this course.
- *Anaconda Navigator.* This is a graphical user interface for the Anaconda prompt. This essentially allows you to manage your packages without having to learn the different commands required by the Anaconda prompt. We won't need to use this application in this course either.

2.2 Spyder

Open the Spyder program installed by Anaconda. You should see an application that looks like this:
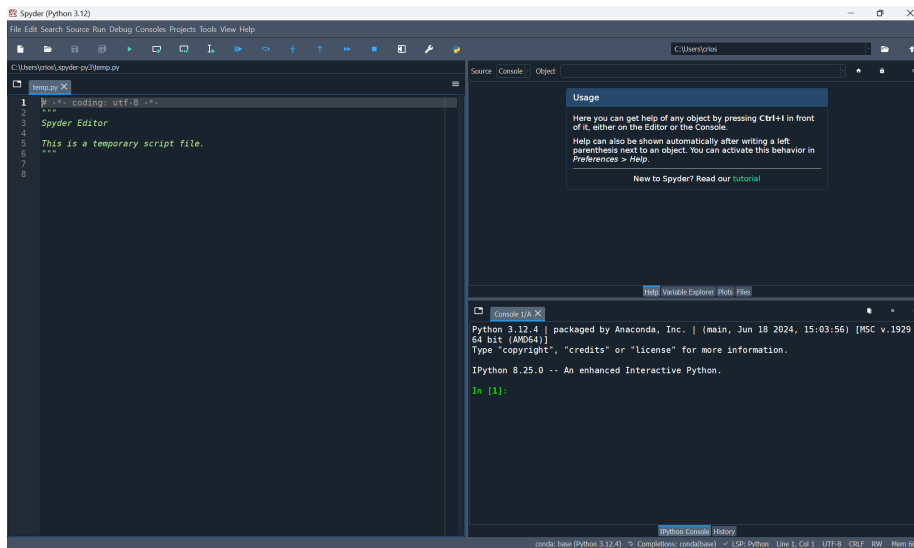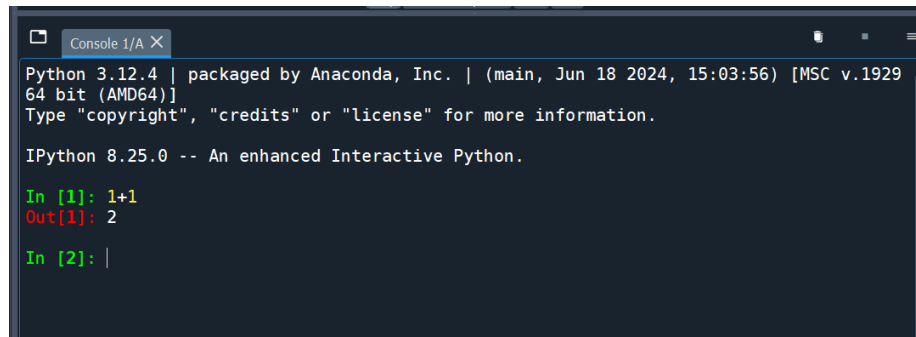
Figure 2.3: Spyder

2.2.1 Python Console

In the bottom right pane you see a console with IPython. IPython is short for *Interactive Python*. We can type Python commands into this console and see the output directly. To find $1 + 1$ in Python, we can use the command `1+1`, similar to how we would do it in Excel or in the Google search engine. Let's try this out in the console. First, click on the console to move the cursor there. Then type `1+1` and press `Enter`. We will see the output `2` on the next line next to a red `Out [1]:`

Figure 2.4: IPython Console

The red `Out [1]` means this is the output from the first line of input (after the green `In [1]`). The second command will have input `In [2]` and output `Out [2]`.

2.2.2 Python Scripts

Typing commands directly into the IPython console is fine if all you want to do is try out a few different simple commands. However, when working on a project you will often be executing many commands. If you were to do all of this in the interactive console it would be very easy to lose track of what you are doing. It would also be very easy to make mistakes.
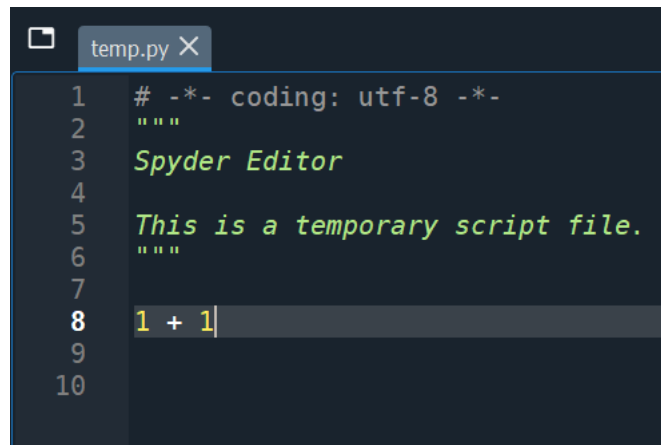
Writing your commands in Python scripts is a solution to this problem. A Python script is a text file with a `.py` extension where you can write all of your commands in the order you want them run. You can then get Spyder to run the entire file of commands. You can also ask it to only run part of the file. This has many advantages over typing commands into the console:

- If you have run 10 commands to calculate something and then afterwards you decide to change what happened in one of the earlier commands, you would often have type all the commands again. In a script you would just need to edit the line with that command. So scripts can save you a lot of time.
- You or anyone else can easily reproduce your work by re-running the Python script.
- By having all the commands in a script you can more easily spot any mistakes you might have.
- It is a way of saving your work.

Therefore it's best practice to write your code in scripts. In the exam, you will also have to supply your script with your answers.

In Spyder, in the left pane you see a file open called `temp.py`. This is an example Python script. We can ignore what is written in the first 6 lines of the script.

We can add our `1 + 1` command to the bottom of the script like this and save it:

Figure 2.5: Python Script

In the Toolbar there are several ways to run this command from the script. For example, you can run the entire file, or run only the current line or selected area. If the cursor is on the line with `1 + 1` and we press the "Run selection or current line" button, then we will see the command and output appear in the IPython console, just like how we typed it there before. Using the script, however, we have saved and documented our work.

If you try run the entire file, you will see `runfile('...')` in the IPython console with the `...` being the path to the Python script you are running. However, you don't see a `2` in the output. This is because when running an entire file, Python does not show the output of each line being run. To see the output of any command we need to put it inside the `print()` function. We can change our line to `print(1 + 1)` to see the output when running the entire file:
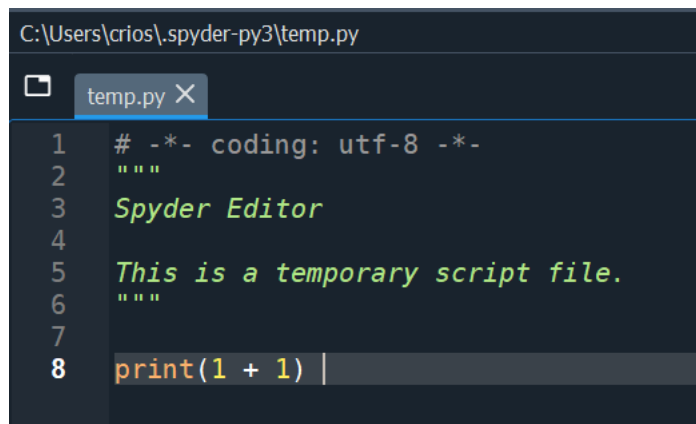
Figure 2.6: Using the print() function

When you run the entire file you should now see a `2` below the `runfile('...')` command.

We now know how to write and run Python scripts! In the next chapter we will learn more Python commands.

2.3   Code Snippets in This Book

In this book, we won't always show screenshots like we did above. Instead we will show code snippets in boxes like this:

```
1 + 1
```

```
2
```

The part that is code will be in color and there will be a small clipboard icon on the right which you can use to copy the code to paste into your script to be able to experiment with it yourself. The output from the code will always be in a separate gray box below it (without a clipboard icon).

Chapter 3

Python as a Calculator

In this chapter we will learn how to use Python as a calculator. In Chapter 2 we already saw how to calculate `1 + 1`. We will now go through some different operations. We will also learn about *functions* and their *arguments* along the way, which we will be be using again and again throughout the rest of this course.

3.1 Addition, Subtraction, Multiplication and Division

We start with the most basic operations. Addition, subtraction, multiplication and division are given by the standard `+`, `-`, `*` and `/` operators that you would use in other programs like Excel. For example:

Addition:

```
2 + 3
```

```
5
```

Subtraction:

```
5 - 3
```

```
2
```

Multiplication:

```
2 * 3
```

```
6
```

Division:

```
3 / 2
```

```
1.5
```

It is also possible to do multiple operations at the same time using parentheses. For example, suppose we wanted to calculate:

$$\frac{2+4}{4\times 2} = \frac{6}{8} = 0.75$$

We can calculate this in Python as follows:

```
(2 + 4) / (4 * 2)
```

```
0.75
```

3.2   Troubleshooting: "Escaping" in Python

Suppose by accident you left out the closing parentheses above. You typed `(2 + 4) / (4 * 2` and Enter. You don't see the output but instead see

```
In [1]: (2 + 4) / (4 * 2
   ...:
```

Python did not run the command, but it also did not give an error. What happened is that because there was no closing parenthesis `Enter` moved to a new line instead of executing the command. That's why we see the `...:`. To "Escape" this situation, you just need to press the `Ctrl+C` button. In general, if anything strange happens in Python and you get stuck, you can always press `Ctrl+C` in the console to escape the current command.

3.3   Exponentiation (Taking Powers of Numbers)

x^n multiplies x by itself n times. For example, $2^3 = 2 \times 2 \times 2 = 8$. In Python we use `**` to do this:

```
2 ** 3
```

```
8
```

Be very careful **not** to use `^` for exponentiation. This actually does a very different thing that we won't have any use for in this course.[1]

3.4   Absolute value

Taking the absolute value turns a negative number into the same number without a minus sign. It has no effect on positive numbers.

---

[1]In fact, there exist many root finding methods. A very famous one is Newton's method developed by Isaac Newton, a famous sciencist that you might have heard of. The reason why there are so many root finding methods is that some work better than others on a given function f. There are other ways to do root finding in Python that allow you to specify a root finding method yourself, but this is a more advanced topic beyond the scope of this course.

In mathematical notation we write $|x|$ for the absolute value of x. The formal definition is:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

Here are some examples:

- $|-2| = 2$
- $|3| = 3$.

This is what the function looks like when we plot it for different x:

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-10, 10, 400)

# Define the absolute value function
y = np.abs(x)

# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x, y, label='y = |x|')

# Add labels and title
plt.title('Absolute Value Function')
plt.xlabel('x')
plt.ylabel('y')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

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

Absolute Value Function

We'll learn how to make plots like this later in this course, but if you want to see the code generating it you can click on the button "Show code generating the plot below".

In Python we can calculate absolute values with:

```
abs(-2)
```

```
2
```

```
abs(3)
```

```
3
```

Taking the absolute value in Python involves using what is called a *function*. Functions are used by calling their names and giving the *arguments* to the function in parentheses. When we do `abs(-2)`, `abs` is the name of the function and `-2` is the argument.

In many ways the functions in Python work a lot like the functions in Excel, just they might have different names or be used a bit differently. For example, in Excel you write `=ABS(-2)` to take the absolute value of -2. The argument is the same, and the function name only differs in that in Excel you need to use capital letters whereas in Python you use lowercase letters (in addition, Excel requires you to put an `=` before the function name).

When using functions it is helpful to read their help pages. You can look at this by typing `help(abs)` in the Console and pressing `Enter`. We then see:

```
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

This tells us that `abs()` takes a single argument and returns the absolute value.[2]

We will be using many different functions and it's a good habit of to look at their help pages. The help pages will be available to you in the Exam.

3.5 Square Roots

The square root of a number x is the y that solves $y^2 = x$. For example, if $x = 4$, both $y = -2$ and $y = 2$ solve this. The principal square root is the positive y from this.

Here is what the square root function looks like for different x:

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the x range (positive values because np.sqrt() not defined for negative
# values)
x = np.linspace(0, 10, 400)

# Define the square root function
y = np.sqrt(x)

# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x, y, label='y = √x')

# Add labels and title
plt.title('Square Root Function')
plt.xlabel('x')
plt.ylabel('y')

# Add a grid
plt.grid(True)

# Add a legend
```

---

[2]The forward slash in `abs(x, /)` marks the end of the positional-only arguments in the function. The `abs()` function takes only one argument, so you can think of this slash as meaning there is only one argument to `abs()`. Some functions like the `math.log()` function that we will see below have two arguments (the number we are taking the log of and the base) and the position (order) of the arguments we provide matter. Other functions, such as the `print()` function we have already encountered, allow you to provide arguments by a *keyword* (see `help(print)` for these).

```
plt.legend()
```

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

Square Root Function
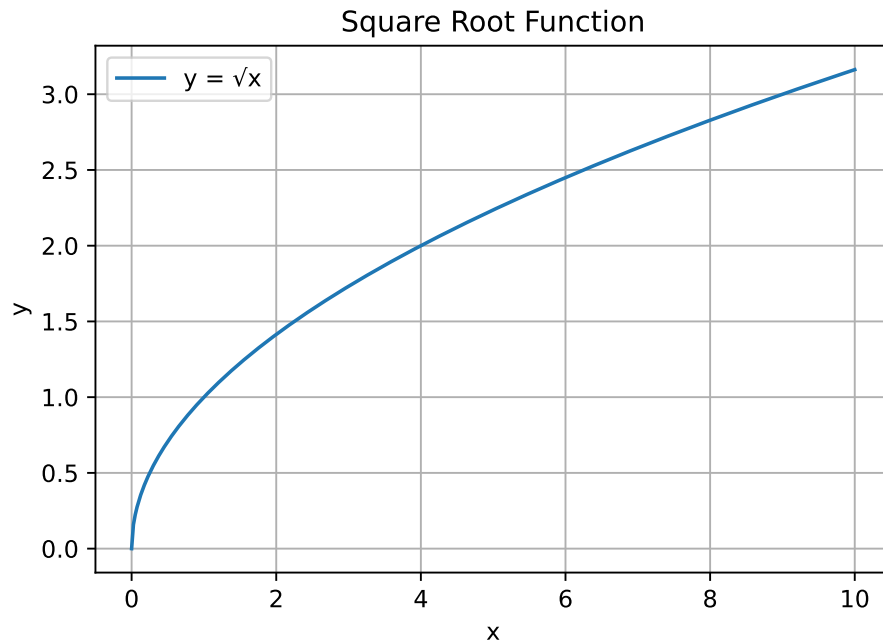
The principal square root a number is equal to the number exponentiated by $\frac{1}{2}$:

$$\sqrt{x} = x^{\frac{1}{2}}$$

```
9 ** (0.5)
```

```
3.0
```

We can follow a very similar approach to above to get the cubed root of a number, such as: $\sqrt[3]{8} = 8^{\frac{1}{3}} = 2$:

In Python:

```
8 ** (1/3)
```

```
2.0
```

Python also has a square root function, but it is not built in. We need to load this function by loading the *module* `math`. A module is a collection of additional functions and other objects that we can load in our Python script. The module `math` contains many mathematical functions, including the `sqrt()` function.

To load the `math` module, we need to include `import math` in our script before executing any of its functions. To run the `sqrt()` function from the `math` module, we need to type `math.sqrt()`. This "dot" notation means we use the `sqrt()` function within the `math` module.

To get $\sqrt{9}$ then we can do:

```python
import math
math.sqrt(9)
```

```
3.0
```

To view the help page `math.sqrt()`, we can use `help(math.sqrt)`.

If you only want to use the `sqrt()` function from the `math` module, you could alternatively import the function the following way:

```python
from math import sqrt
sqrt(9)
```

```
3.0
```

This way you don't need to type `math.sqrt()` every time you want to take the square root, and only need to type `sqrt()`. However, it is generally preferred practice to import the `math` module using `import math` and use the function with `math.sqrt()`. This makes the code clearer and easier to understand.

3.6 Exponentials

A very important function in mathematics and statistics is the exponential function. The definition of $\exp(x)$, or e^x, is given by:

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n$$

Note: you don't need to know or remember this definition for the exam. You only need to know how to calculate the exponential function in Python.

This is what the function looks like:

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-2, 2, 400)

# Define the exponential function
y = np.exp(x)

# Create the plot
```
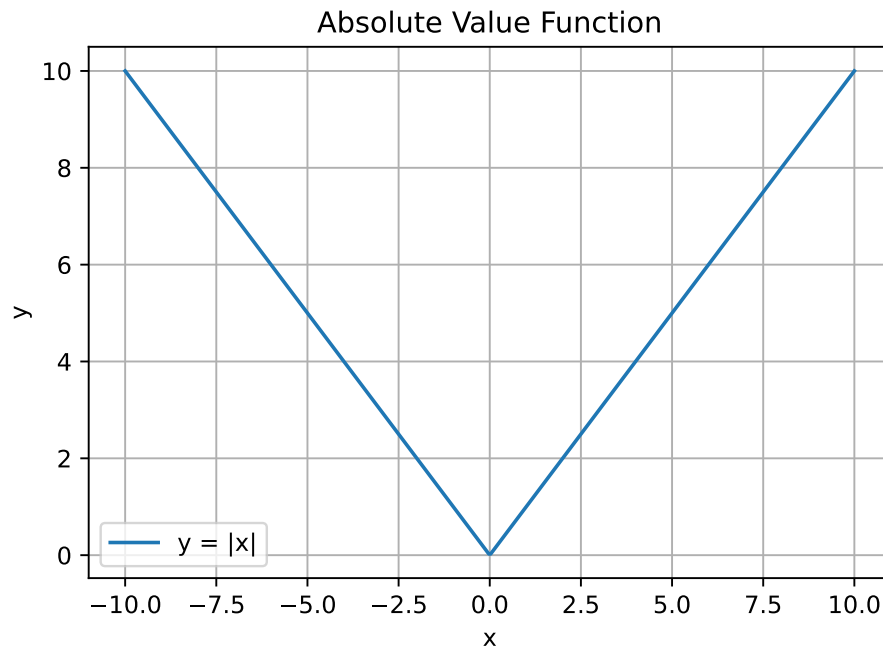
```python
plt.figure(figsize=(6, 4))
plt.plot(x, y, label='y = e^x')

# Add labels and title
plt.title('Exponential Function')
plt.xlabel('x')
plt.ylabel('y')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

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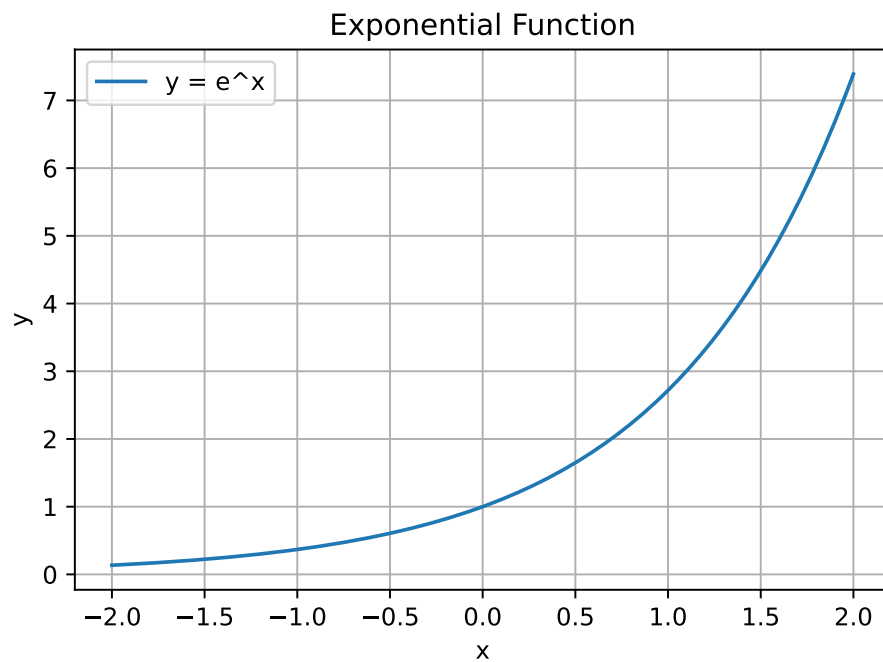

In Python we can use the **exp()** function from the `math` module to calculate the exponential of any number:

```python
math.exp(1)
```

```
2.718281828459045
```

3.7   Logarithms

Another common mathematical function is the logarithm, which is like the reverse of exponentiation.

The log of a number x to a base b, denoted $\log_b (x)$, is the number of times we need to multiply b by itself to get x. For example, $\log_{10} (100) = 2$, because $10 \times 10 = 100$. We need to multiply the base $b = 10$ by itself twice to get to $x = 100$.

A special logarithm is the natural logarithm, $\log_e (x)$, which is the logarithm to the base $\exp(1) = e^1 \approx 2.7183$. This is also written as $\ln(x)$.

This is what the function looks like:

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the x range (positive values only, since ln(x) is undefined for non-positive x)
x = np.linspace(0.1, 10, 400)  # Start from 0.1 to avoid log(0), which is undefined

# Define the natural logarithm function
y = np.log(x)

# Create the plot
plt.figure(figsize=(6, 4))

plt.plot(x, y, label='y = ln(x)')

# Add labels and title
plt.title('Natural Logarithm Function')
plt.xlabel('x')
plt.ylabel('y')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

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

Natural Logarithm Function
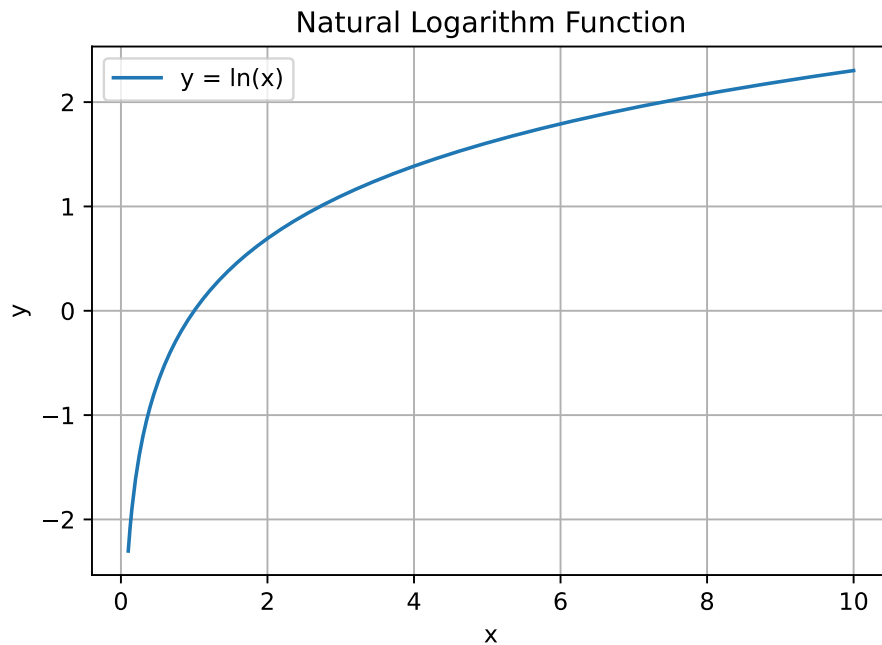
In Python we use the `log()` function from the `math` module to calculate the natural logarithm:

```python
import math
math.log(1)
```

```
0.0
```

What if we want to calculate the logarithm to a base other than e? If we look at the help page for `log()` using `help(math.log)`, we see:

```
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.
```

We can see that the `log()` function can take 2 arguments:

- `x`: the number we want to take the log of.
- `base`: the base with respect to which the logarithms are computed. The default is `math.e` which equals the value of $e \approx 2.718$. Because this argument is contained in a square bracket, it means it is an optional argument. If we don't provide it, it will use the default.

This is the first time that we have seen a function with more than one argument. Earlier when we used the `math.log()` to calculate the natural logarithm we only used one argument because we used the *default setting* for the base. But when we want to use a base other than e, we need to specify it.

Calculate $\log_{10}(100)$ in Python is then done as follows:

```
import math
math.log(100, 10)
```

```
2.0
```

We write both arguments into the `math.log()` function, separated by commas.

The `math` module contains many more mathematical functions. To see all the functions available in the `math` module, we can use the command `dir(math)`. You will see many familiar mathematical functions, such as the trigonometric functions `sin()`, `cos()`, and `tan()`.

3.8   Integer Division and The Modulus Operator

When we divide 7 by 3 we get $2\frac{1}{3}$. We could alternatively say that "7 divided by 3 equals 2 with remainder 1". When programming it is often useful to get these numbers. The tutorial exercises will have several examples of this!

We can perform "integer division" with the `//` operator. This always returns the fraction rounded *down* to the nearest whole number:

```
7 // 3
```

```
2
```

To get the remainder we use the modulus operator `%`:

```
7 % 3
```

```
1
```

Together then 7/3 is 2 with remainder 1.

One thing to note is that integer division with negative numbers doesn't round to the integer closest to zero, but always down. So:

```
-7 // 3
```

```
-3
```

and:

```
7 // -3
```

```
-3
```

both give -3 and not -2

Chapter 4

Variables and Data Types for Single Values

In this chapter we will learn about variables and data types for single values. In Chapter 5 we will learn about data types that can contain multiple values.

4.1 Variables

In Python we can assign single values to *variables* and then work with and manipulate those variables.

4.1.1 Assigning Values to Variables

Assigning a single value to a variable is very straightforward. We put the name we want to give to the variable on the left, then use the = symbol as the assignment operator, and put the value to the right of the =. The = operator binds a value (on the right-hand size of =) to a name (on the left-hand side of =).

To see this at work, let's set $x = 2$ and $y = 3$ and calculate $x + y$:

```
x = 2
y = 3
x + y
```

```
5
```

In Spyder there is a "Variable Explorer" in the top-right pane to see the variables we have created:
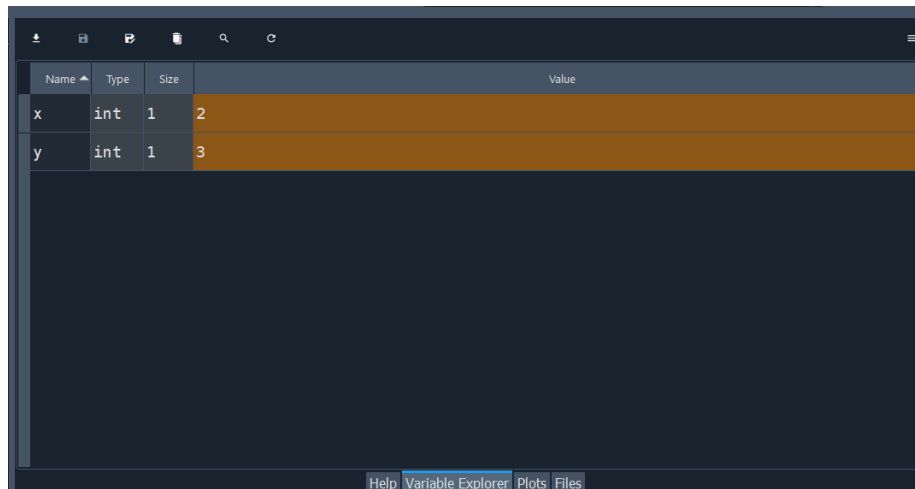
Figure 4.1: Variable Explorer in Spyder

We can see that `x` has a value 2 and `y` has a value 3.

When we assign $x = 2$, in our code, the value is not fixed forever. We can assign a new value to `x`. For example, we can assign the number 6 to `x` instead:

```
x = 6
x + y
```

```
9
```

Finally, you cannot set $x = 2$ with the command `2 = x`. That will result in an error. The name must be on the left of `=` and the value must be on the right of `=`.

4.1.2 Rules for Naming Variables

Variable names can be multiple letters long and can contain underscores (`_`). Underscores are useful because variable names cannot contain spaces and so we can use underscores to represent spaces. Variable names can contain numbers but they cannot start with one. For example `x1` and `x_1` are legal names in Python, but `1x` is not. There are 35 keywords that are reserved and cannot be used as variable names because they are fundamental to the language. For example, we cannot assign a value to the name `True`, because that is a keyword. Below is the list of all keywords.[1] We will learn what many of these keywords are later in this course and how to use them.

---

[1]In fact, there exist many root finding methods. A very famous one is Newton's method developed by Isaac Newton, a famous sciencist that you might have heard of. The reason why there are so many root finding methods is that some work better than others on a given function f. There are other ways to do root finding in Python that allow you to specify a root finding method yourself, but this is a more advanced topic beyond the scope of this course.

```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue',
```

4.2 Common Data Types for Single Values

4.2.1 Integers

You may have noticed that the "Variable Explorer" in Spyder had a "Type" column. For `x` and `y` this was `int` which means "integer". Integers are whole numbers that can also be negative. We can also check the type of a variable using the `type()` function:

```
type(x)
```

```
int
```

4.2.2 Floating-Point Numbers

Numbers that are not whole numbers have the type `float`, which stands for floating-point number:

```
type(1.2345)
```

```
float
```

All the operations we learned about in Chapter 3 also work with floating-point numbers. For example:

```
1.2 * 3
```

```
3.5999999999999996
```

You will notice that we don't get 3.6 like we expected, but instead something very very close but slightly different to 3.6. This is because of how floating-point numbers are represented internally by the computer. The number is split into an integer with a fixed degree of precision and an exponential scaler. For example 1.2 is the same as $12 \times 10e^{-1}$, so the computer needs two integers: 12 and -1 (the exponent) to represent 1.2. Because this process involves some approximations when we perform arithmetic operations on them we can lose some accuracy. However, for most purposes 3.5999999999999996 is close enough to 3.6.

4.2.3 Strings

Python can also work with text in the form of *strings*. Text in Python needs to be wrapped in quotes. These can be either single quotes (`'`) or double quotes (`"`), provided they match.

```
type('Hello world')
```

```
str
```

```
type("This is a string")
```

```
str
```

This `str` means it is a "string" which is a sequence of individual characters.

One thing to be careful with strings is that if you have a string that contains double quotes you have to wrap it in single quotes and vice versa:

```
quote = 'Descartes said "I think, therefore I am" in 1637'
apostrophe = "Don't wrap this with single quotes!"
```

If you find yourself in the unusual situation with a string with both single and double quotes, you can wrap them in triple single quotes (`'''`):

```
quote_with_apostrophe = '''As they say, "Don't judge a book by its cover"'''
```

Another thing to be careful with strings is that numbers surrounded by quotes are strings and not numbers:

```
type('1.2')
```

```
str
```

We can use some of the operators for numbers on strings, but they do very different things. The `+` operator combines strings:

```
a = 'Hello, '
b = 'world'
a + b
```

```
'Hello, world'
```

And the `*` operator repeats strings:

```
a = 'Hello! '
a * 3
```

```
'Hello! Hello! Hello! '
```

4.2.4 Boolean Values

In programming it is often useful to work with variables that are either true or false. Therefore Python has a special data type for this called the Boolean data type. This is named after George Boole who was a mathematics professor in Ireland in the 1800s.

The Boolean values are either `True` or `False`. The words must be capitalized and spelled exactly this way. These are two of Python's keywords.

```
a = True
b = False
type(a)
```

```
bool
```

`True` and `False` are 2 of the keywords that cannot assign values to. Try `2 = True` yourself and see the error that you get.

4.3 Logical and Comparison Operators

4.3.1 Logical Operators

Boolean values have their own operations: `and`, `or` and `not`. These are called *logical operators*. These work as follows:

- `a and b` is `True` if both `a` and `b` are `True`. Otherwise it is `False` (if either or both of `a` or `b` are `False`). Here are all the possible combinations:

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

- `a or b` is `True` if either `a` or `b` (or both) are `True`. Otherwise it is `False` (if both `a` and `b` are `False`). Here are all the possible combinations:

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

- `not a` is `True` if `a` is `False` and is `False` if `a` is `True`. The `not` operator flips the value. Here are all the possible combinations:

a	not a
True	False
False	True

Let's try them out on two specific values `a` and `b`, where `a` is `True` and `b` is `False`:

```
a = True
b = False
a and b
```

```
False
```

This is `False` because we need both `a` and `b` to be `True`.

```
a or b
```

```
True
```

This is `True` because at least one of `a` or `b` is `True`.

```
not a
```

```
False
```

This is `False` because `a` is `True`. It flips the value.

4.3.2 Comparison Operators

Python has operators to check if one number is equal to, not equal to, greater than (or equal to), or less than (or equal to) another number. It checks the (in)equality and returns `True` or `False` depending on the result.

To check if $a = b$, we use the `==` operator:

```
a = 3
b = 2
a == b
```

```
False
```

$a \neq b$, so we get `False`. Be careful to use two equal symbols and not one. If we did `a = b`, it would just reassign to `a` the value of `b` (2):

```
a = 3
b = 2
a = b
a
```

```
2
```

To check if $a \neq b$, we use the `!=` operator (which is supposed to look like the $\neq$ symbol):

```
a = 3
b = 2
a != b
```

```
True
```

This is `True`, because `a` and `b` are not equal.

To see if $a > b$, we use `>` and to see if $a \geq b$ we use `a >= b`:

```
a = 3
b = 2
a >= b
```

```
True
```

We get `True` because $a \geq b$.

To see if $a < b$, we use `<` and to see if $a \leq b$ we use `a <= b`:

```
a = 3
b = 2
a <= b
```

```
False
```

We get `False` because $a \not\leq b$.

4.4 Type Conversion

We can sometimes convert objects between `int`, `float`, `str` and `bool`. Sometimes this conversion is not so intuitive so you need to be careful and know how it works.

If we assign $x = 1$, it will automatically be made an `int`:

```
x = 1
type(x)
```

```
int
```

But we can convert `x` to a `float` using the `float()` function. Let's assign `y` to be `x` as a `float`:

```
y = float(x)
type(y)
```

```
float
```

```
y
```

```
1.0
```

We can see that `y` is not `1` but `1.0`. The `.0` helps us recognize that this is a float.

We can also convert the integer to a string:

```
z = str(x)
type(z)
```

```
str
```

```
z
```

```
'1'
```

The quotes around the 1 helps us recognize that this is a string.

Finally we can also convert from a float to a string:

```
str(y)
```

```
'1.0'
```

If you have an integer stored as a string, we can convert it back to an integer:

```
int('1')
```

```
1
```

Or you can convert it to a float:

```
float('1')
```

```
1.0
```

And if a float is stored as a string, we can convert it back to a float:

```
float('1.5')
```

```
1.5
```

However, it is not possible to convert `'1.5'` to an integer - that will return an error. Similarly you cannot convert strings with characters to integers or floats.

We can also convert floats to integers:

```
int(1.0)
```

```
1
```

If we try to convert a float that isn't a whole number to an integer it will always take the closest integer to zero. For positive numbers this means it always rounds down:

```
int(1.1)
```

```
1
```

```
int(1.9)
```

```
1
```

And for negative numbers it always rounds up:

```
int(-2.1)
```

```
-2
```

```
int(-2.9)
```

```
-2
```

The Boolean values `True` and `False` can be converted to integers, floats and strings. `True` becomes `1`, `1.0` and `'True'` and `False` becomes `0`, `0.0` and `'False'`, respectively. We can also convert integers `1` and `0` back to Boolean:

```
bool(1)
```

```
True
```

```
bool(0)
```

```
False
```

However, if we try convert strings to Boolean we get some unintuitive results. For example:

```
bool('0')
```

```
True
```

```
bool('False')
```

```
True
```

Non-empty strings always return `True`. Only empty strings return `False`:

```
bool('')
```

```
False
```

This is an example of when a programming language does something unintuitive. Therefore when writing a longer program you really need to be sure what each line is doing, otherwise your program will do something unexpected.

Chapter 5

Data Types for Multiple Values

5.1 Introduction

In Chapter 4 we learned about the `int`, `float`, `str` and `bool` data types, which all had single values. But we often have to deal with many values. For example, suppose you wanted to analyze the past daily sales of a company in recent years. It would not be very convenient to assign each of the hundreds of values of sales to different variables and work with them. Python has other data types available to deal with multiple values: lists, tuples, dictionaries, sets and frozen sets. These will be the focus of this chapter.

In later chapters we will also see that there are Python modules that have other data types. In this chapter we will focus on the data types that come built in.

5.2 Lists

A very common way to store a sequence of values (which could be integers, floats, strings or Booleans), is in a list. You can create a list by putting the values in between square brackets, separated by commas:

```python
a = [2, 4, 6]
a
```

```
[2, 4, 6]
```

Lists can also be composed of floats, Booleans, or strings, or even a combination of them:

```python
a = [1, 1.1, True, 'hello']
```

Lists can even have other lists as elements:

```
a = [1, 2, [3, 4]]
```

Although we see 4 numbers, this list actually has only 3 elements:

```
len(a)
```

```
3
```

where the `len()` function returns the length of its argument. The `[3, 4]` is actually considered one element and is itself a list. This kind of a list is called a nested list.

5.2.1 List Operations

If we use the `+` operator on lists it just creates a longer list with one appended to the other:

```
a = [2, 4, 6]
b = [8, 10]
a + b
```

```
[2, 4, 6, 8, 10]
```

If we use the `*` operator it repeats the list:

```
a = [2, 4, 6]
a * 3
```

```
[2, 4, 6, 2, 4, 6, 2, 4, 6]
```

We cannot use the `-` and `/` operators on lists. In these ways lists behave like strings.

5.2.2 List Indexing

Lists are ordered so the order in which we place the elements matters. To extract a particular value from a list based on its position in the list we can use a method called *indexing*. If we want the first element of the list we can extract it with `a[0]`:

```
a[0]
```

```
2
```

The `0` here is the index. It means to take element 0 from the list. In Python and several other programming languages (like C and C++), counting starts at 0 instead of 1. So element 0 is actually the 1st element. This is something that might take some getting used to, so be careful when using indexing.

Even though the list has 3 elements, the last element is extracted with `a[2]`:

```
a[2]
```

```
6
```

If you try to do `a[3]` you get an `IndexError` saying the list index is out of range.

We can also use negative indexing to extract elements from the end of a list. For example, to get the last element of a list `a` we can use `a[-1]`, to get the second-last element we can use `a[-2]`, and so on:

```
a = [1, 2, 3, 4, 5, 6]
a[-1]
```

```
6
```

```
a[-2]
```

```
5
```

Indexing like this also works for strings, and many other objects with multiple values. For example, to get the first character in a string we can get the value at index 0:

```
a = 'hello'
a[0]
```

```
'h'
```

We can also use indexing to change values in a list:

```
a = [2, 4, 6]
a[0] = 8
a
```

```
[8, 4, 6]
```

Because lists have this property, we say they are *mutable*. This is unlike strings which are immutable. You can't change a character in a string using indexing (try it out with the commands `a = 'hello'` and `a[0] = g`).

5.2.3 List Slicing

To get all elements starting from 1 up to but not including index 3 (the 2nd and 3rd element) we can do:

```
a = [1, 2, 3, 4, 5]
a[1:3]
```

```
[2, 3]
```

To get all elements starting from index 2 (the 3rd element onwards) we can do:

```
a[2:]
```

```
[3, 4, 5]
```

To get all elements up to but not including index 2 (the 1st and 2nd element) we can do:

```
a[:2]
```

```
[1, 2]
```

Finally, the following just returns the original list:

```
a[:]
```

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

5.2.4   List Methods

Lists, like many objects in Python, have *methods*. A method in Python is like a function but instead of using the object as an argument to the function, we apply the function *to the object*. We'll see what we mean by this with an example. Suppose we wanted to add another number to our list at the end, like the number 8. Instead of recreating the entire list with `a = [2, 4, 6, 8]`, we can *append* 8 to the end of the list using the `append()` method. Methods are invoked by placing them after the object separated with a . like this:

```
a = [2, 4, 6]
a.append(8)
a
```

```
[2, 4, 6, 8]
```

Notice that we didn't need to assign the output of `append()` to an object with `=`. It altered `a` *in place*. This is what the method does.

To remove an element from a list we can use the `pop()` method. For example, to remove the 2nd element (element with index 1), we can do:

```
a = [2, 4, 6]
a.pop(1)
a
```

```
[2, 6]
```

Another list method is `reverse()` which reverses the ordering of the list:

```
a = [2, 6, 4]
a.reverse()
a
```

```
[4, 6, 2]
```

To sort a list ascending we can use `sort()`:

```
a = [1, 3, 2]
a.sort()
a
```

```
[1, 2, 3]
```

To see the full list of methods available for your list, you can use the command `dir(a)`.

Using the `sort()` method changes our original list. Sometimes we want to see the sorted version of a list but then go back to the original ordering. In this case you shouldn't use the `sort()` method on the list but use the `sorted()` function to create a sorted version of the list. The `sorted()` function returns its input sorted:

```
a = [1, 3, 2]
b = sorted(a)
b
```

```
[1, 2, 3]
```

5.2.5   Iterating over Items in a List

A useful feature of a list is that we can iterate over each element, performing the same operation or set of operations on each element one by one. For example, suppose we wanted to see what the square of each element in the list was. We can use what is called a `for` loop to do this. Here is how to code it:

```
a = [2, 4, 6, 8]
for i in a:
    print(i ** 2)
```

```
4
16
36
64
```

In words, what is happening is "for all i in the list a, print i^2". We use `i` as a sort of temporary variable for each element in `a`. The next line then prints `i ** 2` which squares `i`. You will notice that the `print()` command is indented with 4 spaces. This is to tell Python that this command is part of the loop. When there is code under the `for` loop that is not indented, Python interprets this as not being part of the loop.

To understand this, compare the following two snippets, which are almost the same except the first `print('hello')` is indented and the second is not:

```
a = [2, 4, 6, 8]
for i in a:
    print(i ** 2)
    print('hello')
```

```
4
hello
16
```

```
hello
36
hello
64
hello
```

```
a = [2, 4, 6]
for i in a:
    print(i ** 2)
print('hello')
```

```
4
16
36
hello
```

The first code prints `'hello'` 3 times, and the second only once, even though the code looks almost the same except for the indentations. This is because in the first case, the indentation tells Python that the second `print()` call is part of the loop. In each iteration of the loop, we have to print the square of i and print hello. The loop iterates 3 times, so we see `'hello'` 3 times.

In the second case, the lack of indentation tells Python that the `print('hello')` is not in the loop. Python first finishes the loop (squaring each element of a and printing it). It only then gets to the next part of the code and prints `'hello'`.

Therefore it is very important to be careful with indentation with Python. You should indent with 4 spaces (and not a tab) for content in a loop.

Another thing to note here is that a `for` loop is a situation where the code is no longer running line-by-line from top to bottom. The code goes to the end of the loop and if there are iterations remaining to be done it goes back to the start of the loop. Only when it has completed all the iterations does it go to the next line after the loop.

5.2.6   List Comprehensions

Suppose we wanted to save the square of each element of `a` into a new list called `b`. One way to do that would be to create an empty list called `b` with `b = []`. This is a list with no elements. Then we could use the `for` loop to append the values to `b`, like this:

```
a = [2, 4, 6]
b = []
for i in a:
  b.append(i ** 2)
b
```

```
[4, 16, 36]
```

This works just fine, but the code is a bit "clunky". Moreover, if your list is very very large it would run very slowly. A cleaner and faster way to do this kind of operation is by using *list comprehensions.*

```
a = [2, 4, 6]
b = [i ** 2 for i in a]
b
```

```
[4, 16, 36]
```

This is a very neat and compact way to create the new list. It also reads similar to how we would describe what is happening: "make a list which is i^2 for all elements i in the list a".

5.2.7 List Membership

To see if an element is contained somewhere in a list, we can use the `in` operator:

```
a = [2, 4, 6]
4 in a
```

```
True
```

```
5 in a
```

```
False
```

4 is in `a` so we get `True`, but 5 is not so we get `False`.

5.2.8 Copying Lists

One thing to note about lists, which may be unexpected, is that if we create a list `a` and set `b = a`, we are actually telling Python that `a` and `b` refer to the same object, not just that they have the same values. This has the consequence that if we change `a` that `b` will also change. For example:

```
a = [2, 4, 6]
b = a
a[0] = 8
b
```

```
[8, 4, 6]
```

We set `b = a` but otherwise perform no operations on `b`. We change the first element of `a` (element 0) to 8, and the first element of `b` changes to 8 as well!

Often when we are programming we don't want this to happen. We often want to copy a list to a new one to perform some operations and leave the original list unchanged. What we can do instead is set `b` equal to `a[:]` instead of `a`. This way `b` won't change when `a` changes:

```
a = [2, 4, 6]
b = a[:]
a[0] = 8
b
```

```
[2, 4, 6]
```

Another way is to use the `copy()` method:

```
a = [2, 4, 6]
b = a.copy()
a[0] = 8
b
```

```
[2, 4, 6]
```

Because there are two different ways of copying objects with different conse-
quences, we have two different terms for them:

1. *Deep copy:* This copies `a` to `b` and recursively copies all of its elements,
   resulting in a completely independent object.
2. *Shallow copy:* This copies `a` to `b` but does not recursively copy its elements.
   Instead it only copies the references to the elements in `a` (like the address
   for where in the computer's memory those elements are stored). This
   means that changes to elements of `a` will affect the elements of `b`.

The `b = a` example is a shallow copy and the `b = a[:]` example is like a deep
copy. However, it is not a full deep copy. Using `a[:]` or `a.copy()` only works if
our list is not nested. This method only takes a deep copy of the outermost list.
If we copy with `a[:]` or `a.copy()` to `b` with a nested listed and then change an
element inside one of the nested lists, then the copied object will change as well.
For example:

```
a = [2, 4, [6, 8]]
b = a[:]
a[2][1] = 5
b
```

```
[2, 4, [6, 5]]
```

`b` changes as well! The same happens with the `copy()` method:

```
a = [2, 4, [6, 8]]
b = a.copy()
a[2][1] = 5
b
```

```
[2, 4, [6, 5]]
```

To make a full deep copy which recursively copies the entire object, we can use
the `deepcopy()` function from the `copy` module:

```
import copy
a = [2, 4, [6, 8]]
b = copy.deepcopy(a)
a[2][1] = 5
b

[2, 4, [6, 8]]
```

5.3 Tuples

A `tuple` is another data type that is quite similar to a list. One important difference, however, is that they are *immutable*. We cannot change individual values of a tuple after they are created, and we cannot append values to a tuple.

We can create a tuple in Python using parentheses instead of square brackets:

```
a = (2, 4, 6)
```

Indexing and many other operations that work for lists also work with tuples. We index them the same way as lists (using square brackets like `a[0]`) and we can iterate over the items with `for` loops in the same way. However the list of methods for tuples is much shorter. We cannot append or pop values because the tuples are immutable.

5.3.1 Tuple Assignment

One useful thing we can do with tuples is *tuple assignment*. Suppose we have a list `x = ['a', 'b', 'c']` and we wanted to create 3 objects from this: `x_0 = 'a'`, `x_1 = 'b'` and `x_2 = 'c'`. One way to do this is:

```
x = ['a', 'b', 'c']
x_0 = x[0]
x_1 = x[1]
x_2 = x[2]
```

But a much more elegant way to do this is using tuple assignment:

```
x = ['a', 'b', 'c']
(x_0, x_1, x_2) = x
```

This assigns `'a'` to `x_0`, `'b'` to `x_1` and `'c'` to `x_2` all in one line.

This is especially useful if you have a function that returns multiple objects and we want to assign each output to a different variable. For example, the function `divmod(a, b)` gives the quotient and remainder from dividing `a` with `b`. It essentially calculates `a // b` and `a % b` and returns a tuple with both objects:

```
divmod(7, 3)

(2, 1)
```

This means 7 divided by 3 is 2 with remainder 1. We can use tuple assignment with the output to get:

```
(quotient, remainder) = divmod(7, 3)
quotient
```

```
2
```

```
remainder
```

```
1
```

5.4   Dictionaries

Another common built-in data type is a dictionary. A dictionary maps *keys* to *values*, where the keys can be an immutable data type (usually an integer or string) and the values can be any type, for example, single values, lists, or tuples. For example, a company might have supplier IDs for its suppliers and a dictionary mapping those IDs to the actual company name. In this case, the company IDs are the keys and the company names are the values.

We could create a simple dictionary like this as follows:

```
suppliers = {100001 : 'ABC Ltd.', 100002 : 'EFG Ltd.'}
```

Dictionaries are created within curly brackets with the structure `{key1 : value1, key2 : value2, key3 : value3}`.

To find a company name using the company ID we provide the key in the place we would supply an index for a list or tuple:

```
suppliers[100001]
```

```
'ABC Ltd.'
```

Dictionaries are unordered, so we cannot do `suppliers[0]` to find the first supplier. There is no first value in a dictionary.

We can also add new keys and values to the dictionary:

```
suppliers[100003] = 'HIJ Ltd.'
suppliers
```

```
{100001: 'ABC Ltd.', 100002: 'EFG Ltd.', 100003: 'HIJ Ltd.'}
```

We can also modify values:

```
suppliers[100003] = 'KLM Ltd.'
suppliers
```

```
{100001: 'ABC Ltd.', 100002: 'EFG Ltd.', 100003: 'KLM Ltd.'}
```

To get all the keys in a dictionary we can use the `keys()` method:

```
suppliers.keys()
```

```
dict_keys([100001, 100002, 100003])
```

And to get all the values in a dictionary we can use the `values()` method:

```
suppliers.values()
```

```
dict_values(['ABC Ltd.', 'EFG Ltd.', 'KLM Ltd.'])
```

Using a `for` loop with a dictionary implicitly iterates over the keys. So we can loop over the keys of a dictionary in the following way:

```
for key in suppliers:
    print('Supplier with ID ' + str(key) + ' is ' + suppliers[key])
```

```
Supplier with ID 100001 is ABC Ltd.
Supplier with ID 100002 is EFG Ltd.
Supplier with ID 100003 is KLM Ltd.
```

Finally, to create an empty dictionary we can use `{}`.

5.5 Sets and Frozen Sets

A *set* is another way to store multiple items into a single variable. Sets are unordered and unindexed. This means you cannot extract individual elements using their index like a list, nor by their key like a dictionary.

You can create a set by placing items (like integers or strings) inside curly brackets (`{}`) separated by commas:

```
myset = {'apple', 'banana', 'cherry'}
myset
```

```
{'apple', 'banana', 'cherry'}
```

Sets cannot have duplicate items. It only keeps the unique values. For example, suppose we provide `'cherry'` twice:

```
myset = {'apple', 'banana', 'cherry', 'cherry'}
myset
```

```
{'apple', 'banana', 'cherry'}
```

It only keeps the first `'cherry'`.

You are, however, able to add and remove elements to a set.

```
myset = {'apple', 'banana', 'cherry'}
myset.add('pear')
myset
```

```
{'apple', 'banana', 'cherry', 'pear'}
```

```
myset = {'apple', 'banana', 'cherry'}
myset.remove('apple')
myset
```

```
{'banana', 'cherry'}
```

Converting a list to a set is useful if you want to get the list of unique elements:

```
fruits = ['apple', 'apple', 'apple', 'banana', 'cherry', 'banana']
set(fruits)
```

```
{'apple', 'banana', 'cherry'}
```

We can iterate over sets just like lists (with `for i in myset`). We can also perform set operations on pairs of sets.

For example, for two sets A and B, we can find $A \cap B$ (the set of elements contained in both sets) using:

```
set_a = {1, 2, 4, 6, 8, 9}
set_b = {2, 3, 5, 7, 8}
set_a.intersection(set_b)
```

```
{2, 8}
```

The numbers 2 and 8 are the only numbers in both sets.

To find $A \cup B$ (the set of elements contained in either set) we can do:

```
set_a.union(set_b)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

This gets all the numbers appearing in the two sets (dropping duplicates).

To find $A \setminus B$ (the set of elements in A not contained in B) we can do:

```
set_a.difference(set_b)
```

```
{1, 4, 6, 9}
```

1, 4, 6 and 9 are in A and not in B. The number 2, for example, is not here because that is also in B.

To create a set that is immutable (so that you cannot add or remove items), you can use the **frozenset()** function:

```
myset = frozenset([1, 2, 3])
```

You can still use the same operations on frozensets as normal sets, except you cannot modify them once they are created.

Chapter 6

Defining Functions and Conditional Execution

6.1 Introduction

We have used a number of Python functions so far, such as the absolute value function and the square root function. In this chapter we will learn how we can create our own functions. We will also learn how to use conditional statements inside functions.

6.2 Structure of a Function

We will start off learning how to program a very basic function. Consider the function that returns its input plus one. Mathematically the function would be represented as:

$$f(x) = x + 1$$

So $f(0) = 1$, $f(1) = 2$ and $f(2) = 3$ and so on. In Python we can create this function with:

```python
def add_one(x):
    y = x + 1
    return y
```

The `def` tells Python we are creating a function. We then provide the functions name (here `add_one`). After that we put in the function's arguments in parentheses, separated by commas. Here there is only one argument so we just write `x`. Then like with a `for` loop we add a `:` and add the body of the function below

it indented by 4 spaces. Here the only thing the function does is create `y` which is `x + 1`. We then get the function to return the output, which is `y`.

Let's try it out:

```
add_one(2)
```

```
3
```

We get the expected output!

One thing to note about this function is that the `y` that is assigned `x + 1` in the function is never stored in our environment. The `y` only exists within the function and is deleted after the function ends. We cannot access it outside. We say that `y` is a *local* variable (it is local to the function). It is possible to define *global* variables within a function that can be accessed after the function is called, but for our purposes doing so is generally not very good practice and so we will not cover that here.

We could also shorten our code by doing the calculation on the same line as the `return` command:

```python
def add_one(x):
    return x + 1
```

6.3   Commenting in Python

As we start to write longer programs that include functions, it's s good idea to start *annotating* your code to help other people understand its purpose (and also you when you look back at your own code after a couple of days!). We can do this by adding *comments*. In Python we can add a comment by using the `#` character. Everything after the `#` character is ignored by the Python interpreter, so what we write after it don't need to be "legal" Python commands. We can add a comment to describe what a function does like this:

```python
# This function returns the input plus one:
def add_one(x):
    return x + 1
```

We can also add comments to the same line as code we want to run provided we put it *after* the command. Like this:

```python
2 ** 3  # this command calculates 2 to the power of 3
```

```
8
```

6.4 Conditional Execution

6.4.1 If-Else Statements

Conditional statements, or "if-else statements", are very useful and extremely common in programming. In an if-else statement, the code first checks a particular true/false condition. If the condition is true, it performs one action, and if the condition is false, it performs another action.

A simple example of this is the absolute value function we saw in Chapter 3. Let's define precisely what that function does:

$$|x| = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

If $x < 0$ it returns $-x$, so that the negative number turns positive. Otherwise (if $x = 0$ or it is positive), it keeps the value of x the same.

Although Python already has an absolute value function (`abs()`), let's create our own function (called `my_abs()`) that does the same thing. To do this we use conditional statements (`if` and `else`). Here's how it works:

```python
def my_abs(x):
    if x < 0:
        y = -x
    else:
        y = x
    return y
```

The function first checks if $x < 0$. If it is true, it performs the operation under `if` (sets $y = -x$) and skips past the `else` statement and returns y. If it is false (i.e. $x \not< 0$) then it skips past the operation under the `if` statement and instead does the operation under the `else` statement (sets $y = x$) before returning y.

Let's try it out using some different values:

```python
[my_abs(i) for i in [-2, 0, 3]]
```

```
[2, 0, 3]
```

Just like with the `add_one()` function above we can shorten this function definition. We could alternatively do:

```python
def my_abs(x):
    if x < 0:
        return -x
    else:
        return x
```

The function first checks if $x < 0$. If it is true, it returns $-x$ and ends. It doesn't go any further. If $x \not< 0$ then it skips the operation under `if` and does the operation under `else` (returns x).

This gives the same output:

```
[my_abs(i) for i in [-2, 0, 3]]
```

```
[2, 0, 3]
```

This means the `return` part of a function doesn't have to be at the end of a function. But you should be aware that once a function returns a value it does not continue executing the remaining statements.

For example, consider the following code:

```
def bad_add_one(x):
    return x
    y = x + 1
    return y
[bad_add_one(i) for i in [1, 2, 3]]
```

```
[1, 2, 3]
```

This is very similar to the first `add_one()` function we defined above. The only difference is that we write `return x` as the first command in the function's body. Although the code sets $y = x + 1$ and returns y, the output is always the same as the input. This is because the function returns x at the top, which means the rest of the function is never executed.

6.4.2   If-Else If-Else Statements

Sometimes we want to do one thing if a certain condition holds, another thing if a different condition holds, and something else in the remaining cases. An example of this is the "sign" function, which tells you the sign in front of a value:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{otherwise} \end{cases}$$

If the value is negative, we get -1. If it's zero we get 0. If it's positive (the remaining case), we get $+1$.

To do this in Python, we could nest several if-else statements:

```
def sign(x):
    if x < 0:
        return -1
    else:
        if x == 0:
```

```
            return 0
        else:
            return 1
[sign(i) for i in [-2, 0, 3]]
```

```
[-1, 0, 1]
```

The function does the following:

- If $x < 0$, return -1.
- Otherwise proceed to the next if-else:
    - If $x = 0$, return 0.
    - Otherwise (if $x > 0$), return 0.

Although this works, this is quite complicated and difficult to follow. Some of the `return` statements are indented 4 times, for what should be such a simple function. For these kinds of situations we can make use of the `elif` statement. Here is an alternative way to make this function using `elif`:

```
def sign(x):
    if x < 0:
        return -1
    elif x == 0:
        return 0
    else:
        return 1
[sign(i) for i in [-2, 0, 3]]
```

```
[-1, 0, 1]
```

In words, what the code does in this case is:

- If the 1st check is true ($x < 0$), the function returns -1 and it done.
- If the 1st check is false ($x \not< 0$), the function checks $x = 0$. If that is true it returns 0 and it done.
- If the 1st and 2nd checks are false ($x \not< 0$ and $x \neq 0$), the function returns 1 and it done.

It actually does exactly the same as the first code, but because there is less nesting it is easier to follow and is preferred (especially when you have even more conditions to check!).

6.4.3 While Loops

A while loop is another very common method in programming. A while loop repeats a set of commands whenever a certain condition is true. A while loop also makes it possible to run an infinite loop. To have the sequence of numbers $1, 2, 3, ...$ printed on your screen forever (or until you kill the program) you could do:

```
x = 0
```

```
while True:
    x += 1
    print(x)
```

Note: the `x += 1` here is a shorter way of writing `x = x + 1`. The operations `-=`, `*=` and `/=` also work like this - try them out!

Because `True` will always be `True`, the loop will just keep running forever, adding 1 each time. Eventually the numbers will get so big that `x` will show up as `inf` (infinity), but you would have to let the program run for a very long time before you saw that.

A while loop is also useful if you want to repeat a loop until something happens, but you don't know how many times you need to run it before that happens. One instance when this would occur is if you wanted to numerically approximate a mathematical equation with an iterative algorithm. You want to repeat the iterations until the output starts to stabilize to a certain tolerance (accuracy) level, but you don't know in advance how many iterations this will take. Let's take a look at an example of this now.

The example we will look at is the way the ancient Greeks approximated square roots. Suppose you wanted to find the square root of x. What the ancient Greeks did is start with an initial guess of this y_0, let's say $\frac{x}{2}$. You then find the updated guess y_1 according to the formula:[1]

$$y_1 = \frac{1}{2}\left(y_0 + \frac{x}{y_0}\right)$$

When you have y_1 we can update this for a more accurate approximation with:

$$y_2 = \frac{1}{2}\left(y_1 + \frac{x}{y_1}\right)$$

To write this in general terms, given an initial guess y_0, we update y_n, with $n = 1, 2, \ldots$ according to:

$$y_n = \frac{1}{2}\left(y_{n-1} + \frac{x}{y_{n-1}}\right)$$

You continue updating this way until y_n stops changing very much (for example it changes by less than 0.000001 in an iteration).

Let's work manually with this algorithm to see how well it works. Suppose we want the square root of 2 which we know is approximately equal to 1.414214.

---

[1]In fact, there exist many root finding methods. A very famous one is Newton's method developed by Isaac Newton, a famous sciencist that you might have heard of. The reason why there are so many root finding methods is that some work better than others on a given function f. There are other ways to do root finding in Python that allow you to specify a root finding method yourself, but this is a more advanced topic beyond the scope of this course.

Let's start with a guess $y_0 = \frac{x}{2} = \frac{2}{2} = 1$. This is quite far off the true 1.414214 but we'll go with it anyway. We can update the guess with the formula:

$$y_1 = \frac{1}{2}\left(y_0 + \frac{x}{y_0}\right) = \frac{1}{2}\left(1 + \frac{2}{1}\right) = 1.5$$

This is already a lot closer (0.0858 away). Let's do the next approximation step:

$$y_2 = \frac{1}{2}\left(y_1 + \frac{x}{y_1}\right) = \frac{1}{2}\left(1.5 + \frac{2}{1.5}\right) = 1.416667$$

This is already pretty close (0.00245 away)! Let's do one more:

$$y_3 = \frac{1}{2}\left(y_2 + \frac{x}{y_2}\right) = \frac{1}{2}\left(1.416667 + \frac{2}{1.416667}\right) = 1.414216$$

It's now only 0.0000021 away from the precise answer! That might be close enough for most purposes, and we can always do another iteration to improve its accuracy.

Let's see how to code this in Python:

```python
def my_sqrt(x, tol=0.000001):
    # Arguments:
    #  x   : number to take the square root of.
    #  tol : tolerance level of algorithm.

    # Set initial guess:
    y = x / 2

    # Initialize distance:
    dist = tol + 1

    # Update guesses until y changes by less than tol:
    while dist > tol:
        # Previous guess:
        y_old = y
        # Update guess:
        y = (y_old + x / y_old) / 2
        # Calculate distance from last guess:
        dist = abs(y - y_old)
    return y
```

We have written a function that can take 2 arguments: `x`, the number we want to take the square root of, and `tol`, which is the tolerance level for how accurate our approximation should be (a lower number is more accurate). When we write `tol=0.000001` in the function definition it means we say that `0.000001` is the

default value for `tol`. If we don't provide the argument it will use this value, but we can specify a different value if we want.

We now talk about the code in the function. The function first sets $y_0 = \frac{x}{2}$ as the initial guess. It also needs to set `dist = tol + 1` because the while loop checks if `dist > tol`. For this check, `dist` needs to exist locally in the function (`tol` is created from the arguments). And for the while loop to run at least once the `dist` needs to start at a value bigger than `tol`. This is why we add one. Inside the while loop then, because we want to compare how our guess changes, we set `y_old = y` before setting the new `y` according to the approximation formula. Then we get the absolute value of the difference between the new and old guess. We then go back to the top of the loop and we check if `dist` is still bigger than `tol`. If it is, it repeats the steps again. If not, the while loop terminates and we go to the next stage, where `y` is returned as the output.

Let's try it out. First using the default value:

```
my_sqrt(2)
```

```
1.414213562373095
```

We get an approximation that is very close to `math.sqrt(2)`:

```
import math
math.sqrt(2)
```

```
1.4142135623730951
```

Can we specify a looser tolerance as follows:

```
my_sqrt(2, 0.1)
```

```
1.4166666666666665
```

As expected, this is less accurate.

Chapter 7

Introduction to NumPy

7.1 Introduction

Python's built-in data types like lists and tuples are not particularly well-suited for mathematical operations. We will show three examples of computations that we often need to do, and we will see that using lists involves quite a lot of coding to get the tasks done. We will then see that NumPy can do these tasks very efficiently.

7.2 Three Example Problems

Example 1

For the first example, suppose we have a list `x` of numbers and we want to double each of the elements. We can't use `2 * x` because as we learned in Chapter 5 that just repeats the list twice. We have to do something like:

```python
x = [2, 4, 8]
y = []
for i in x:
    y.append(2 * i)
y
```

```
[4, 8, 16]
```

We create an empty list `y`. We then loop over the elements of `x` and append two times the element to `y`. This is very clunky. A better way of doing this is using a list comprehension:

```python
x = [2, 4, 8]
[2 * i for i in x]
```

```
[4, 8, 16]
```

But this is still a bit clunky. We would prefer a method that can just do `2 * x` and get the same output.

Example 2

Another example computation that we often need to do is if we have two lists of numbers `x` and `y` with the same number of elements and and we want to multiply them by each other element-by-element. Expressed in mathematical notation, suppose we have two vectors of numbers x and y:

$$x = [x_1, x_2, \dots, x_n]$$
$$y = [y_1, y_2, \dots, y_n]$$

and we want to calculate z from this which is:

$$z = [x_1 \times y_2, x_2 \times y_2, \dots, x_n \times y_n]$$

We can't do `x * y`. That would return an error. But we could do this using a `for` loop:

```
x = [2, 4, 8]
y = [3, 2, 2]
z = []
for i in range(len(x)):
    z.append(x[i] * y[i])
z
```

```
[6, 8, 16]
```

Because we want to loop over the elements of both `x` and `y`, we have to loop over the indices 0, 1, 2. We could have written `for i in [0, 1, 2]`, but `range(len(x))` does this for us automatically (which is very useful if we have a long list). To see better what range is doing we can do:

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

We can see it creates a list from 0 up to but not including the argument.

We can improve on this code slightly by using the `zip()` function, which combines several *iterables* into one *iterable*.

```
x = [2, 4, 8]
y = [3, 2, 2]
z = []
for i, j in zip(x, y):
    z.append(i * j)
z
```

```
[6, 8, 16]
```

Similarly we can use `zip()` to do the task with a list comprehension:

```
x = [2, 4, 8]
y = [3, 2, 2]
[i * j for i, j in zip(x, y)]
```

```
[6, 8, 16]
```

Even though we have now shortened the command down to one line, this last solution is still pretty clunky and also quite complicated. We would prefer an operation where we can just do `x * y`.

Example 3

For the last example, suppose we want to find the median of a list of numbers. Recall that if the length of the list of numbers is odd, then the median is just the number in the middle when we sort the numbers. If the length of the list of numbers is even, then the median is the average of the two numbers closest to the middle when we sort the numbers.

We could create our own function to do this:

```
def median(x):
    y = sorted(x)
    if len(y) % 2 == 0:
        return (y[len(y) // 2 - 1] + y[len(y) // 2]) / 2
    else:
        return y[len(y) // 2]
```

The function first sorts the list. The `sorted()` function gives the sorted list as the output. We do this instead of `x.sort()` because otherwise the function would sort our input list globally which we may not want it to do. The `if len(y) % 2 == 0:` checks if the length of the list is even. If it is even it takes the average of the element with index `len(y) // 2 - 1` (just left of the middle) and the element with index `len(y) // 2` (just right of the middle). We use `//` to ensure the division returns an integer instead of a float. If the length of the list is odd it returns the element with index `len(y) // 2`. Because `len(y) / 2` is not an integer when `len(y)` is odd we use `//` to round down.

Let's test it out:

```
median([2, 6, 4])
```

```
4
```

```
median([2, 6, 4, 3])
```

```
3.5
```

We get the expected ouptut. However, this is quite complicated. We wouldn't want to have to code this function every time we wanted to do something as common as finding the median.

We will see that functions from the module `numpy` can solve each of these problems (and a lot more) very easily.

7.3   Importing the NumPy Module

We can import the `numpy` module using `import numpy` like with other modules. However it is conventional to load NumPy the following way:

```
import numpy as np
```

This way we can use the functions from NumPy with the shorter `np` instead of having to type `numpy` out in full every time. Doing this shortcut is okay because so many people do it that it's easy for people to read. You can load other modules with shortcuts in a similar way, but you should follow the normal conventions when you can.

NumPy works with *arrays*. An array is like a list but all elements must be of the same type (such as all floats). We can create an array using NumPy's array function. Because we can shorten `numpy` to `np`, we can create an array with the function `np.array()` like this:

```
import numpy as np
x = np.array([2, 4, 8])
x
```

```
array([2, 4, 8])
```

We will now show the power of NumPy by doing all the previous examples with very little code.

7.4   Solving the Example Problems with NumPy

Example 1

To double every number in array:

```
x = np.array([2, 4, 8])
2 * x
```

```
array([ 4,  8, 16])
```

Example 2

To multiply the elements of two arrays element-by-element:

```
x = np.array([2, 4, 8])
y = np.array([3, 2, 2])
x * y
```

```
array([ 6,  8, 16])
```

Example 3

To get the median of an array:

```
x = np.array([2, 6, 4])
np.median(x)
```

```
np.float64(4.0)
```

```
x = [2, 6, 4, 3]
np.median(x)
```

```
np.float64(3.5)
```

The `np.median()` function also works if we just provide a list instead of an `np.array`:

```
np.median([2, 6, 4, 3])
```

```
np.float64(3.5)
```

These are just a few examples of how NumPy can simplify coding drastically.

For many programming tasks we need to do, it's very often many people had to do the same thing before. This means there is often a module available that can do the task. Of course we learn a lot from coding functions from scratch, but in order to complete a task quickly and efficiently it is usually better to use the modules made for the task.

7.5   Matrix Operations

NumPy can also do matrix operations very easily. For example, suppose we had two 3×3 matrices A and B:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 3 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 1 & 2 \\ 3 & 2 & 1 \\ 1 & 3 & 1 \end{pmatrix}$$

and wanted to calculate their product $C = AB$.

Manually, we could calculate each row i and column j of C with $\sum_{k=1}^{3} a_{ik} b_{kj}$.

For example, row 2 and column 1 of C would be:

$$c_{21} = \sum_{k=1}^{3} a_{2k} b_{k1}$$
$$= a_{21} b_{11} + a_{22} b_{22} + a_{23} b_{33}$$
$$= 2 \times 2 + 3 \times 3 + 1 \times 1$$
$$= 4 + 9 + 1$$
$$= 14$$

But doing this for all 9 elements would take a long time, and we could easily make a mistake along the way. Let's use Python to calculate it instead.

If we were to try and do this with only built-in Python commands, it would still be rather complicated. We could define the matrices A and B using nested lists, where each list contains 3 lists representing the rows:

```
A = [
    [1, 2, 3],
    [2, 3, 1],
    [3, 1, 3]
]

B = [
    [2, 1, 2],
    [3, 2, 1],
    [1, 3, 1]
]
```

To calculate $C = AB$ then we follow the same approach as the manual way. We loop through each row i and each column j of the matrix and calculate: $c_{ij} = \sum_{k=1}^{3} a_{ik} b_{kj}$. We do this by starting with a zero matrix and progressively fill it up.

```
C = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
]

for i in range(3):
    for j in range(3):
        for k in range(3):
            C[i][j] += A[i][k] * B[k][j]

for row in C:
    print(row)
```

```
[11, 14, 7]
[14, 11, 8]
[12, 14, 10]
```

This is an example of a triple-nested loop: a loop inside a loop inside a loop.

Using NumPy to do the multiplication is much easier. We can just use the `np.dot()` function:

```python
import numpy as np
A = np.array(A)
B = np.array(B)
np.dot(A, B)
```

```
array([[11, 14,  7],
       [14, 11,  8],
       [12, 14, 10]])
```

NumPy can also do many other matrix operations, such as:

- transposing with the command `np.transpose()`
- inversion with the command `np.linalg.inv()`.

You can therefore use Python to help you with the Mathematics course that you are taking alongside this one!

Chapter 8

Data handling with Pandas

In this chapter, we will go over some of the basics of importing, adjusting and exporting data in Python. For the adjusting part, we will rely on the Pandas package, which is a data analysis package. We start by explaining how to use Pandas data frames, a convenient way to store large datasets. Afterwards, we will explain how you can import data from a file into a data frame, and how to export it to another file.

The Pandas package `pandas` is typically imported under the alias `pd`.

```python
import pandas as pd
```

8.1 Data frames

We will start with a small data set consisting of six persons and some personal information about these people. The data is given in the following dictionary. It contains the names, height, weight, age and dietary preference (i.e., dictionary keys) of everyone.

```python
dataset = {
 'name' : ["Aiden", "Bella", "Carlos", "Dalia", "Elena", "Farhan"],
 'height (cm)' : [185, 155, 190, 185, 160, 170],
 'weight (kg)' : [80, 60, 100, 85, 62, 75],
 'age (years)' : [23, 23, 23, 21, 19, 25],
 'dietary preference' : ['Veggie','Veggie','None','None','Vegan','None']
}
```

```python
print(dataset.keys())
```

```
dict_keys(['name', 'height (cm)', 'weight (kg)', 'age (years)', 'dietary preference'])
```

A dictionary is not a convenient datatype to perform data analysis on. Therefore,

we load the data into a so-called data frame using the `DataFrame()` function from `pandas`.

```
df = pd.DataFrame(dataset)
```

```
print(df)
```

```
     name  height (cm)  weight (kg)  age (years) dietary preference
0   Aiden          185           80           23             Veggie
1   Bella          155           60           23             Veggie
2  Carlos          190          100           23               None
3   Dalia          185           85           21               None
4   Elena          160           62           19              Vegan
5  Farhan          170           75           25               None
```

As you can see here, the keys of the dictionary become the column names of the data frame, and the values are stored in the corresponding column. You can also see the index of the row at the far left. Here we call the data frame `df` (short for data frame), but you can name it whatever you like.

Because we have a relatively small data set, the complete data frame is printed. If the data contains a large number of rows, typically the first and last five rows are printed in the console of Spyder. A data frame is an object of the type `DataFrame` with which you can do all kinds of things.

```
print(type(df))
```

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

8.1.1   Accessing

If you want to print the first or last k rows, you can use the functions `df.head(k)` and `df.tail(k)`, respectively, with `df` being the name of your data frame.

```
# Print first three rows of df
print(df.head(3))
```

```
     name  height (cm)  weight (kg)  age (years) dietary preference
0   Aiden          185           80           23             Veggie
1   Bella          155           60           23             Veggie
2  Carlos          190          100           23               None
```

```
# Print last two rows of df
print(df.tail(2))
```

```
     name  height (cm)  weight (kg)  age (years) dietary preference
4   Elena          160           62           19              Vegan
5  Farhan          170           75           25               None
```

It is also possible to access specific rows or elements from the data frame using indexing.

To extract row i, use `df.loc[i]`.

```
# Extract row 0 (i.e., info of Aiden)
x = df.loc[0]

print(x)
```

```
name                   Aiden
height (cm)              185
weight (kg)               80
age (years)               23
dietary preference    Veggie
Name: 0, dtype: object
```

To extract a specific column, use `df.loc[:,'column_name']`.

```
# Extract age column
x = df.loc[:,'age (years)']

print(x)
```

```
0    23
1    23
2    23
3    21
4    19
5    25
Name: age (years), dtype: int64
```

An alternative that not uses the `loc[]` function is `df[column_name]`.

```
# Extract age column
x = df['age (years)']

print(x)
```

```
0    23
1    23
2    23
3    21
4    19
5    25
Name: age (years), dtype: int64
```

If a column name doesn't contain spaces or special characters (apart from '_') then it's also possible to extract a column using `df.column_name`. Let's try this with the **name** column:

```
# Extract name column
x = df.name
```

```
print(x)
```

```
0       Aiden
1       Bella
2      Carlos
3       Dalia
4       Elena
5      Farhan
Name: name, dtype: object
```

To extract the entry from row i in column 'column_name' we use `df.loc[i,column_name]`.

```
# Extract the name and height from the person on row 0:
x = df.loc[0,'name']
y = df.loc[0,'height (cm)']
```

```
print("The height of", x, "is", y, "cm.")
```

```
The height of Aiden is 185 cm.
```

We can also use slicing to return a specified range of rows. For example, rows i through j can be obtained using `df.loc[i:j]`.

```
# Extract first three rows
x = df.loc[0:2]
```

```
print(x)
```

```
      name  height (cm)  weight (kg)  age (years) dietary preference
0    Aiden          185           80           23             Veggie
1    Bella          155           60           23             Veggie
2   Carlos          190          100           23               None
```

This works differently compared to slicing with lists. Recall there that if we want the first 3 elements of a list `x` then when we need to use `x[0:3]`. This is because the last element of the specified range (3 in this case) is not included when using slicing in lists:

```
# Slicing in list
y = [13,4,5,2,11]
```

```
# Print first, second and third element of y:
print(y[0:3])
```

```
[13, 4, 5]
```

Using slicing, we can also access specific combinations of columns and rows. Suppose we are only interested in the name, height and age of the first, second and third person in the frame. Because the columns have names (i.e., are not

numbers), we index them by a list containing the column names that we are interested in.

```
# Extract block with the first 3 rows and columns name, height and age:
x = df.loc[0:2,['name','height (cm)', 'age (years)']]

print(x)
```

```
     name  height (cm)  age (years)
0   Aiden          185           23
1   Bella          155           23
2  Carlos          190           23
```

It is also possible to return a subset of rows that do not form a consecutive block. You can do this with a Boolean vector indicating for every row whether you want it to be included or not. For example, if we want to return only rows $0, 1, 4$ and 5, we can do the following:

```
rows = [True,True,False,False,True,True]
x = df[rows]

print(x)
```

```
     name  height (cm)  weight (kg)  age (years) dietary preference
0   Aiden          185           80           23            Veggie
1   Bella          155           60           23            Veggie
4   Elena          160           62           19             Vegan
5  Farhan          170           75           25              None
```

Here `rows` is a Boolean list containing entries `True` and `False` with an element being `True` if and only if we want the row to be included (namely $0, 1, 4$ and 5), and `False` otherwise (namely 2 and 3). We can achieve the same result with `df.loc[[0,1,4,5]]`, i.e, by giving a list of the row entries that we are interested in.

The Boolean list approach is convenient, because it can also be used to select rows that satisfy a specified criterion. For example, suppose that we want to only select the rows of persons whose dietary preference is 'None'. This can be achieved as follows.

```
# Boolean vector no_pref indicating whether dietary preference is 'None'
no_pref = df['dietary preference'] == 'None'
print(no_pref)
```

```
0    False
1    False
2     True
3     True
4    False
5     True
```

```
Name: dietary preference, dtype: bool
```

```
# Extract rows for which list no_pref has 'True' entry
x = df[no_pref]
```

```python
print(x)
```

```
     name  height (cm)  weight (kg)  age (years) dietary preference
2  Carlos          190          100           23               None
3   Dalia          185           85           21               None
5  Farhan          170           75           25               None
```

The expression `df['Dietary preference'] == 'None'` checks for every row in the dietary preference column `df['Dietary preference']` whether its entry is `'None'`. If so, it returns `True`, and otherwise `False`. We store these True/False values in the list `no_pref` (short for having no dietary preference). We then use this Boolean list to extract the rows of the data frame consisting of the persons whose dietary preference is `None`.

To extract all the rows where `no_pref` is `False` we can do:

```
x = df[~no_pref]
```

```python
print(x)
```

```
    name  height (cm)  weight (kg)  age (years) dietary preference
0  Aiden          185           80           23             Veggie
1  Bella          155           60           23             Veggie
4  Elena          160           62           19              Vegan
```

For another example, let's get all the rows for people who are at least 23 years old:

```python
x = df[df['age (years)'] >= 23]
```

```python
print(x)
```

```
     name  height (cm)  weight (kg)  age (years) dietary preference
0   Aiden          185           80           23             Veggie
1   Bella          155           60           23             Veggie
2  Carlos          190          100           23               None
5  Farhan          170           75           25               None
```

You can also combine conditions with the logical `&` operator, provided each individual condition is in parentheses:

```python
x = df[(df['age (years)'] >= 23) & (df['dietary preference'] == 'Veggie')]
```

```python
print(x)
```

```
    name  height (cm)  weight (kg)  age (years) dietary preference
0  Aiden          185           80           23             Veggie
```

```
1  Bella            155            60            23            Veggie
```

To do an *OR* condition, we use |. To get all rows with people 23 and over *OR* with vegetarian preferences.

```
x = df[(df['age (years)'] >= 23) | (df['dietary preference'] == 'Veggie')]
```

```
print(x)
```

```
     name  height (cm)  weight (kg)  age (years) dietary preference
0   Aiden          185           80           23            Veggie
1   Bella          155           60           23            Veggie
2  Carlos          190          100           23              None
5  Farhan          170           75           25              None
```

8.1.2 Editing

It is also possible to edit the data frame, both the data in the frame, as well as the column and row names. For example, it might be that we start with data that is not given in a dictionary, but rather in a matrix (which is a list of lists, where each of the inner lists forms a row of the matrix).

```
data = [
[2,4,-1,2],
[5,1,2,9],
[3,7,8,9]
]
```

```
df = pd.DataFrame(data)
```

```
print(df)
```

```
   0  1  2  3
0  2  4 -1  2
1  5  1  2  9
2  3  7  8  9
```

Note that in this case both the rows and columns have their index number as name, so $0, 1$ and 2 for the rows and $0, 1, 2$ and 3 for the columns. The names of the rows are stored in `df.index` and the columns in `df.columns`.

```
# Row names are stored in df.index
print(df.index)
```

```
RangeIndex(start=0, stop=3, step=1)
```

```
# Rename rows
df.index = ['Row0','Row1','Row2']
```

```
# Access the row names
print(df.index)
```

```
Index(['Row0', 'Row1', 'Row2'], dtype='object')
```

If you want to access specific column names, you can use indexing.

```
# Print name of first column
print(df.index[1])
```

```
Row1
```

Here is the complete frame with row names adjusted.

```
# Print data frame
print(df)
```

```
      0  1  2  3
Row0  2  4 -1  2
Row1  5  1  2  9
Row2  3  7  8  9
```

Next, let us adjust the column names.

```
df.columns = ['Col0','Col1','Col2','Col3']
```

```
print(df)
```

```
      Col0  Col1  Col2  Col3
Row0     2     4    -1     2
Row1     5     1     2     9
Row2     3     7     8     9
```

It is also possible to alter the entries within the frame.

```
# Edit entry on row 1, column 2
df.loc['Row1','Col2'] = 10
```

```
print(df)
```

```
      Col0  Col1  Col2  Col3
Row0     2     4    -1     2
Row1     5     1    10     9
Row2     3     7     8     9
```

You can also edit a complete row (or column)

```
# Replace row 2
df.loc['Row2',:] = [-2,-2,-2,-2]
```

```
print(df)
```

```
      Col0  Col1  Col2  Col3
Row0     2     4    -1     2
```

```
Row1    5    1    10    9
Row2   -2   -2    -2   -2

# Replace column 2
df.loc[:,'Col2'] = [-1,-1,-1]

print(df)

       Col0  Col1  Col2  Col3
Row0     2    4    -1    2
Row1     5    1    -1    9
Row2    -2   -2    -1   -2
```

It is also possible to edit the entries of an entire column by applying a function to it using `apply()`. For example, suppose that we want to square all the numbers in the second column. We can do this as follows.

```
def f(x):
    return x**2

# df['Col1'].apply(f) does not overwrite the entries in Col1
# so we have to do this ourselves
df['Col1'] = df['Col1'].apply(f)

print(df)

       Col0  Col1  Col2  Col3
Row0     2    16   -1    2
Row1     5    1    -1    9
Row2    -2   4    -1   -2
```

You can also define and apply simple functions in one go using what are called *lambda* functions. Let's square all the entries in `Col1` again but this time using a *lambda* function:

```
df['Col1'] = df['Col1'].apply(lambda x: x**2)
print(df)

       Col0  Col1  Col2  Col3
Row0     2    256   -1    2
Row1     5    1    -1    9
Row2    -2   16   -1   -2
```

8.1.3 Adding data

It is also possible to add entire new rows and columns.

```
data = [
[2,4,-1,2],
[5,1,2,9],
```

```
[3,7,8,9]
]
```

```
# Create frame out of data
df = pd.DataFrame(data)
```

```
# Name rows and columns
df.columns = ['Col0','Col1','Col2','Col3']
df.index = ['Row0','Row1','Row2']
```

```
print(df)
```

```
      Col0  Col1  Col2  Col3
Row0    2     4    -1     2
Row1    5     1     2     9
Row2    3     7     8     9
```

Next, we add a row to the data frame. The `loc[]` command uses for this adds the row at the bottom of the current data frame.

```
# Add a row
df.loc['New row'] = [5,5,3,1]
```

```
print(df)
```

```
          Col0  Col1  Col2  Col3
Row0         2     4    -1     2
Row1         5     1     2     9
Row2         3     7     8     9
New row      5     5     3     1
```

The same holds for adding a column, which is done as follows. Note that here we use `[:,'New column']` and not `['New column']`, because the latter would add the new data as a row.

```
df.loc[:,'New column'] = [1,1,1,1]
```

```
print(df)
```

```
          Col0  Col1  Col2  Col3  New column
Row0         2     4    -1     2           1
Row1         5     1     2     9           1
Row2         3     7     8     9           1
New row      5     5     3     1           1
```

It's also possible to add a new column without using the `loc[]` command:

```
df['New column'] = [1,1,1,1]
```

```
print(df)
```

```
          Col0  Col1  Col2  Col3  New column
Row0         2     4    -1     2           1
Row1         5     1     2     9           1
Row2         3     7     8     9           1
New row      5     5     3     1           1
```

You can also add a new column with the `insert()` function. This allows you to specify at which position you want the column to be inserted. The `insert()` function needs three arguments: a position i where the column should be inserted, the column name and the column data, so the syntax is something like `insert(i,column_name,column_data)`.

```
# Insert column with name 'New column' at position 2.
df.insert(2,'Inserted column', [10,10,10,10])

print(df)
```

```
          Col0  Col1  Inserted column  Col2  Col3  New column
Row0         2     4               10    -1     2           1
Row1         5     1               10     2     9           1
Row2         3     7               10     8     9           1
New row      5     5               10     3     1           1
```

Adding a row at a specific position is also possible, but this is more involved and omitted here.

8.2 Mathematical operations

It is also possible to obtain statistical information from numerical columns.

```
data = [
[2,4,-1,2,'hello'],
[5,1,2,9,'bye'],
[3,7,8,9,'hello'],
[3,5,8,9,'hi'],
[31,5,4,9,'hi'],
[3,7,8,5,'hello'],
]

# Create frame out of data
df = pd.DataFrame(data)

# Name rows and columns
df.columns = ['Col0','Col1','Col2','Col3','Col4']
df.index = ['Row0','Row1','Row2','Row3','Row4','Row5']

print(df)
```

```
       Col0   Col1   Col2   Col3    Col4
Row0      2      4     -1      2   hello
Row1      5      1      2      9     bye
Row2      3      7      8      9   hello
Row3      3      5      8      9      hi
Row4     31      5      4      9      hi
Row5      3      7      8      5   hello
```

For example, we can compute the minimum, maximum and average value by using the function `min()`, `max()` and `mean()`, respectively.

```python
# Minimum of the first column
min_col1 = df['Col1'].min()

print(min_col1)

1

# Maximum of the second column
max_col2 = df['Col2'].max()

print(max_col2)

8

# Mean of the zeroth column
mean_col0 = df['Col0'].mean()

print(mean_col0)

7.833333333333333
```

It is also possible to count occurrences of a given `word` (or number) using `value_counts()[word]`. For example, suppose we want to count how often the word `'hello'` appears in the third column:

```python
count_hello = df['Col4'].value_counts()['hello']

print(count_hello)

3
```

We can also to more advanced things like counting the total number of occurrences of every word in the fourth column. By having a quick look at the data, we see that there are three distinct greetings, `'hello'`, `'hi'` and `'bye'`, to be found in the fourth column. A quick way to obtain these greetings in a list is to use the `unique()` function. This function returns a list with all the unique entries found in the specified column.

```python
greetings  = df['Col4'].unique()

print(greetings)
```

```
['hello' 'bye' 'hi']
```

Next, we can loop over the greetings in the list `greetings` and apply the `value_counts()` function to all of them. We store the results in a dictionary whose keys are the greetings in `greetings` and whose values are the number of times every greeting appears in the fourth column.

```python
# Create empty dictionary
occur_count = {}

for i in greetings:
    occur_count[i] = df.loc[:,'Col4'].value_counts()[i]

print(occur_count)

{'hello': np.int64(3), 'bye': np.int64(1), 'hi': np.int64(2)}
```

8.3 Importing and exporting data

Data is typically provided in an external file, for example, a comma-separated values (CSV) file. You can download the data that was given at the beginning of this section here in and you should store in under the name dataset.csv.

```python
df = pd.read_csv('dataset.csv')

print(df)
```

```
     name  height (cm)  weight (kg)  age (years)  dietary preference
0   Aiden          185           80           23              Veggie
1   Bella          155           60           23              Veggie
2  Carlos          190          100           23                None
3   Dalia          185           85           21                None
4   Elena          160           62           19               Vegan
5  Farhan          170           75           25                None
6   Geert          178           80           25              Veggie
```

To test the code above in Spyder, you need to store the Python file that you execute the code in, in the same folder as the file `dataset.csv`.

The function `read_csv` stores the data in the file dataset.csv. The first row of the data is assumed to contain the names of the columns. If the data file does not contain such a first row, we can import the data with the additional argument `header=None`, meaning that we tell Python that there is no first row containing the column names.

```python
df = pd.read_csv('dataset.csv', header=None)

print(df)
```

```
          0             1              2              3                    4
0      name    height (cm)    weight (kg)    age (years)    dietary preference
1     Aiden           185             80             23                Veggie
2     Bella           155             60             23                Veggie
3    Carlos           190            100             23                  None
4     Dalia           185             85             21                  None
5     Elena           160             62             19                 Vegan
6    Farhan           170             75             25                  None
7     Geert           178             80             25                Veggie
```

Note that in the code above, the first row of the data is now included in the
data frame, instead of set to be the names of the columns. The columns are now
indexed by integers like the rows of the frame.

It is also possible to export an (adjusted) data frame to a comma-separated file.
Let us first add another row to the existing frame and then export it to a new
file called new_dataset.csv.

```python
df = pd.read_csv('dataset.csv')

# Highest index in original frame is 5, so 6 is the index
# at which we place the new row
df.loc[6] = ['Geert',178, 80, 25, 'Veggie']

print(df)
```
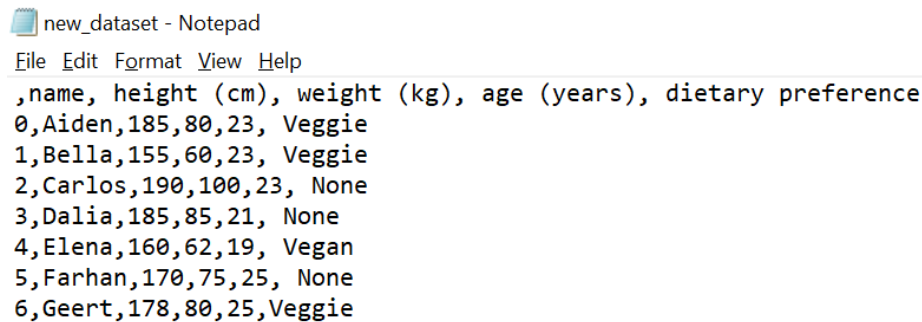
```
       name    height (cm)    weight (kg)    age (years)    dietary preference
0     Aiden           185             80             23                Veggie
1     Bella           155             60             23                Veggie
2    Carlos           190            100             23                  None
3     Dalia           185             85             21                  None
4     Elena           160             62             19                 Vegan
5    Farhan           170             75             25                  None
6     Geert           178             80             25                Veggie
```

Now that we have added a new row, we can use the **to_csv()** function tho store
the frame in the new comma-separated file.

```python
df.to_csv('new_dataset.csv')
```

The folder in which you have stored the original **dataset.csv** file, as well as the
Python file in which the code is executed, should now contain a new file called
new_dataset.csv. If you open the file in, e.g., Notepad (Windows) or Excel, you
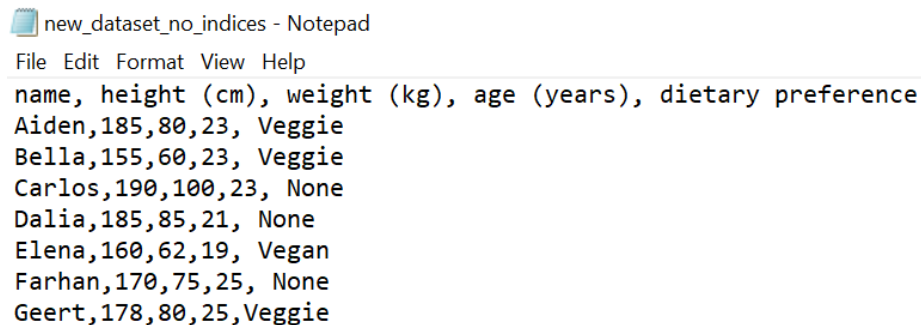will see something like the following figure.

Figure 8.1: New data frame in .csv file

On the first line the column names can be found, and on the following lines the data from the frame. However, Python also exported the row indices $0, 1, \ldots, 6$. If you don't want these indices to be included (they were also not contained in the original .csv file), you can use the argument `index=False` in `to_csv()`.

```
df.to_csv('new_dataset_no_indices.csv', index=False)
```

This time the resulting file does not have the row indices at the beginning of every line.

Figure 8.2: New data frame without row indices in .csv file

Instead of storing the data in a new file, we can also overwrite the original dataset.csv file.

```
df.to_csv('dataset.csv',index=False)
```

You should, however, be careful with overwriting files in this way. Always make sure you have a copy of the data stored somewhere else, in case something goes wrong!

Chapter 9

Mathematics and plotting

In this chapter we will see some mathematical algorithms from the `scipy` package (or module) as well as how to visualize data, e.g., some of the figures we have seen in earlier chapters, using `matplotlib`.

SciPy is a package that can be used to perform various mathematical tasks and algorithms, making it very important for data analysis purposes. The Matplotlib package is essential in Python to create insightful visual representations of your data and the analysis you performed on it.

We first discuss two fundamental mathematical tasks: finding a root of a mathematical function and minimizing a mathematical function. After that, we will explain how to visualize data and mathematical functions.

9.1   Root finding

Consider the function $f(x) = x^2 + 2x - 1$. A visualization of this function is given below. We will learn how to create this figure ourselves in Section 9.3.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-3, 3, 600)

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Create the plot
plt.figure(figsize=(6, 4))
```

```python
plt.plot(x, f(x), label='$f(x) = x^2 + 2x - 1$')

# Add labels and title
plt.title('Plot of the function f on the interval [-3,3]')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

# Set range
plt.xlim(-3,3)
plt.ylim(-4,14)

# Add a legend
plt.legend()

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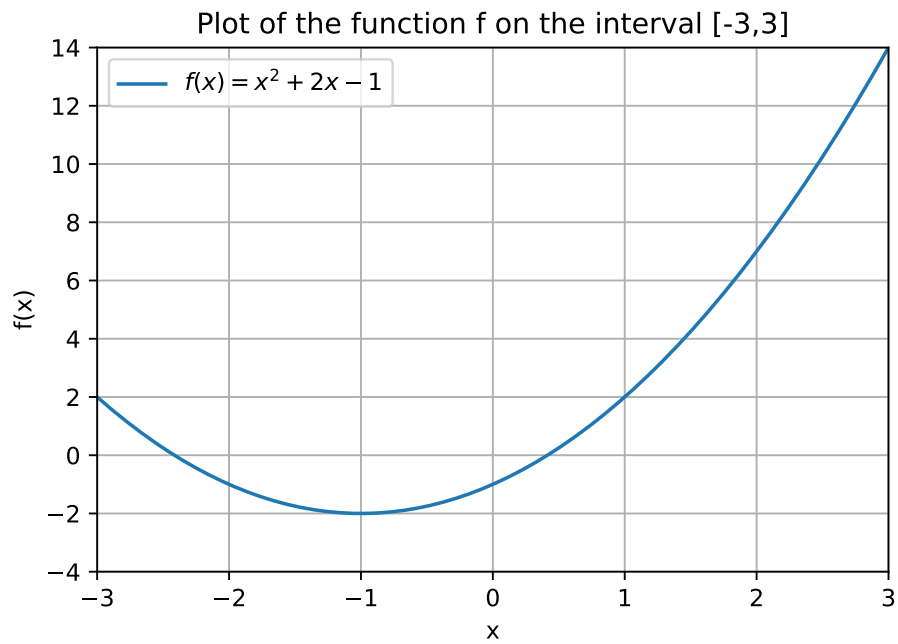

Plot of the function f on the interval [-3,3]

> **i** Root finding
>
> A root x of a function $f : \mathbb{R} \to \mathbb{R}$ is a point that satisfies the equation $f(x) = 0$.

In our case, we want to solve the equation $x^2 + 2x - 1 = 0$. You might remember from your high school math course that, for given numbers a, b and c, the roots of the quadratic function $f(x) = ax^2 + bx + c$, are given by

$$x_\ell = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_r = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

where the subscript ℓ is used to denote the fact that this will be the "left" root and r to denote the "right" root.

For $f(x) = x^2 + 2x - 1$, we have $a = 1, b = 2$ and $c = -1$. Plugging in these values in the formula above gives $x_\ell = -1 - \sqrt{2} \approx -2.4142$ and $x_r = -1 + \sqrt{2} \approx 0.4142$. Although this is an easy way to find the roots of a quadratic function, we want to be able to find roots of any function f if they exist.

The easiest way to find a root of a general function is to use the the the function `fsolve()` from the (sub)package `scipy.optimize`. This package contains many functions to carry out algorithmic tasks on mathematical functions. To avoid having to write `scipy.optimize` every time we will import this package under the name `optimize`, just as we did with NumPy where we used the line `import numpy as np`.

Below is the code snippet that carries out the root finding procedure.

```python
# Import package containing fsolve()
import scipy.optimize as optimize

# Define the function f
def f(x):
    return x**2 + 2*x - 1

# Using fsolve() to find root of f
guess = 3
f_zero = optimize.fsolve(f,guess)

# Print the found root
print("A root of the function f is given by", f_zero)
```

```
A root of the function f is given by [0.41421356]
```

In chronological order, this code does the following:

- Import the `optimize` (sub)package from the `scipy` package.

- Define $f(x) = x^2 + 2x - 1$ as a Python function (Chapter 6).
- Use `optimize.fsolve(f,guess)` so that Python knows that we want to use the function `fsolve()` from `optimize`.
- The argument that is returned by `fsolve()` is assigned to the variable `f_zero`.
- Print a message containing the root of f that was found.

The use of `fsolve()` requires some additional explanation. It takes two input arguments, the first one being a (mathematical) function `f` and the second one an initial guess for where a root of f might be, which we store in the variable `guess`. Note that you cannot reverse the order of the input arguments: `f_zero = optimize.fsolve(guess,f)` does not work!

We could also have used `optimize.fsolve(f,3)` instead of separately defining `guess = 3` and then using `guess` as input argument. In coding, it is common practice to always define input data, the number 3 in this case, in a separate variable. Especially if the number would be used in multiple places in your code, this is useful.

It is important to observe that a Python function itself can be an input argument of another Python function! That is, the function `f` is an input argument of the function `fsolve()`. It is important to know that, to use a function f as an input `fsolve()`, it is required to define f as a Python function. That is, it is not possible to use the command `optimize.fsolve(x**2 + 2*x - 1,guess)`.

The second input argument `guess` is needed by `fsolve()` to execute the underlying mathematical root finding method that is used to find a root of f. We do not go into the actual mathematical method that is being carried out in the background by Python when we execute `fsolve()`[1], but the idea is that the method starts at the guess that we provide and then gradually finds its way to a root of f by doing various calculations.

In fact, the choice of initial guess can influence the outcome of `fsolve()`. That is, a different initial guess can lead the underlying mathematical method to a different root of the function. This is illustrated in the code below, where we find the other root $-2.4142....$

```
guess = -2
f_zero = optimize.fsolve(f,guess)

print("A root of the function f is given by", f_zero)

A root of the function f is given by [-2.41421356]
```

---

[1]In fact, there exist many root finding methods. A very famous one is Newton's method developed by Isaac Newton, a famous sciencist that you might have heard of. The reason why there are so many root finding methods is that some work better than others on a given function f. There are other ways to do root finding in Python that allow you to specify a root finding method yourself, but this is a more advanced topic beyond the scope of this course.

As you might have noted, the output of `optimize.fsolve(f,guess)` is a list containing one number, for example, the last piece of code returned `[-2.41421356]`. This happends because `fsolve()` can also be used to find roots of higher-dimensional functions, in which the case the root is a vector of numbers (stored in a list).

If you instead only want to output the number `-2.41421356`, i.e., the value of the element in the string, you can use `optimize.fsolve(f,guess)[0]` instead. This means that what is stored in `f_zero` is the 0-th (and only) element of the list `optimize.fsolve(f,guess)`.

```
guess = -2
f_zero = optimize.fsolve(f,guess)[0]

print("A root of the function f is given by", f_zero)

A root of the function f is given by -2.414213562373095
```

We can also use root finding to solve other types of equations. Suppose we want to compute an x such that

$$f(x) = 6.$$

Moving the 6 to the left, we see that this is the same as computing an x such that $f(x) - 6 = 0$. Therefore, if we define the function $g(x) := f(x) - 6$, then an x that satisfies $g(x) = 0$ also satisfies $f(x) = 6$, and vice versa. Let us code this as well.

```
def g(x):
    return f(x) - 6

guess = 4
f_zero = optimize.fsolve(g,guess)[0]

print("A number x satisfying f(x) = 6, is given by", f_zero)

A number x satisfying f(x) = 6, is given by 1.82842712474619
```

The function g works as follows: It computes $f(x)$ by running the function f with the input x, and then substracting 6 from it. Note that we could have also defined g by returning `x**2 + 2x - 1 - 6`. It is, however, more convenient to just write `f(x) - 6` here, because we have already defined the function f earlier. Also, if we would change the definition of the function f, the function g is automatically updated as well.

Everything we have seen up till now also allows us to write a general Python function to solve an equation of the form

$$f(x) = c$$

for a given function f and number c. In the example above, we had $c = 6$.

```python
def solve_eq(f,c,guess):
    """
    Input
    ----------
    f : A mathematical function taking as input a variable x,
    c : The right hand side value of the equation f(x) = c,
    guess : The initial guess for fsolve().

    Returns
    -------
    A value x solving f(x) = c.
    """

    def g(x):
        return f(x) - c

    x = optimize.fsolve(g,guess)[0]
    return x
```

The function above takes as input the function f, the number c and an initial guess that `fsolve()` can use. Let us try out `solve_eq()` on some input data. The goal will be to solve the equation

$$3x^2 - 4x + 1 = 5.$$

```python
# We create the function h(x) = 3x^2 - 4x + 1
def h(x):
    return 3*x**2 - 4*x + 1

# Right hand side of the equation h(x) = 5
d = 5

# Our initial guess for fsolve() (we choose 1 here).
initial_guess = 1

# Print the found solution
print("A solution x to h(x) = d is given by", solve_eq(h,d,guess))
```

```
A solution x to h(x) = d is given by 2.0
```

Note that the input arguments `h`, `d` and `initial_guess` need not have the same names as the local variables `f`, `c` and `guess` in the function `solve_eq()`. What is important is that we input the arguments in the correct order in which we want them to be identified with the local variables. That is, by doing `solve_eq(h,d,guess)` Python knows that we want to assign the first input argument `h` to the first local variable `f`, the second input argument `d` to the second local variable `c`, and the third input argument `initial_guess` to the

local variable `guess`.

9.2 Minimization

Another fundamental task in mathematics is to find the minimum value that a function can attain. Recall the function $f(x) = x^2 + 2x - 1$ from the previous section.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x = np.linspace(-3, 3, 600)

# Define the absolute value function
def f(x):
    return x**2 + 2*x -1

# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x, f(x), label='$f(x) = x^2 + 2x - 1$')

# Add labels and title
plt.title('Plot of the function f on the interval [-3,3]')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

# Set range
plt.xlim(-3,3)
plt.ylim(-4,14)

# Add a legend
plt.legend()

# Show the plot
plt.show()
```
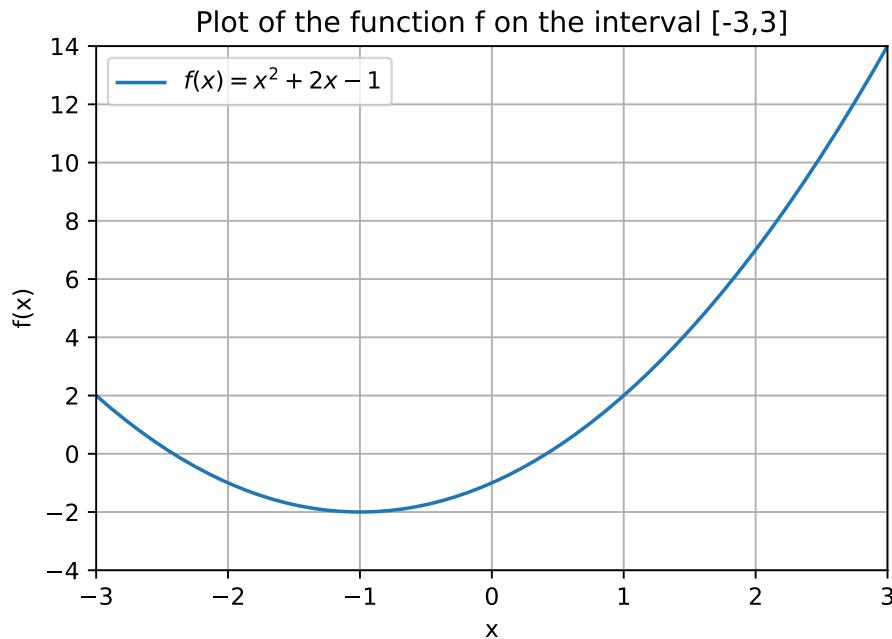
It can be seen that the point x at which the function f is the lowest, i.e., attains its minimum is $x = -1$, and the function values in that point is $f(-1) = -2$. Python has various ways of finding the minimum of a function, the easiest-to-use being `fmin()` from the `optimize` module.

The syntax that is used for this function is similar to that of `fsolve()`. Although we already defined the function f in the previous section, we will redefine it in the code below for sake of completeness.

```python
import scipy.optimize as optimize

def f(x):
    return x**2 + 2*x - 1

guess = 1
minimum = optimize.fmin(f,guess)

print('The minimum of the function f is attained at x = ', minimum)
```

```
Optimization terminated successfully.
        Current function value: -2.000000
        Iterations: 19
        Function evaluations: 38
The minimum of the function f is attained at x =  [-1.]
```

Note that Python outputs some information in the console about the mathemat-

ical optimization procedure that was performed in order to find the minimum of the function. It displays the function value at the miminum that was found, in our case $-2 = f(-1)$, and a number of iterations and function evaluations. These last two pieces of information are not relevant for us, but are useful for an expert who wants to understand better how well the optimization procedure performed. If you want, you can suppress all this information by adding `disp=False` or `disp=0` as an argument to the fmin() function. This is illustrated below.

```python
minimum = optimize.fmin(f,guess,disp=False)

print('The minimum of the function f is attained at x = ', minimum)

The minimum of the function f is attained at x =  [-1.]
```

Also here, if you want to return only the value `-1.0` instead of the list `[-1.0]` you can do the following as we illustrated for `fsolve()` as well.

```python
minimum = optimize.fmin(f,guess,disp=False)[0]

print('The minimum of the function f is attained at x = ', minimum)

The minimum of the function f is attained at x =  -1.0000000000000018
```

9.3   Visualization

In this section we will explain the basics for plotting functions and data, for which we will use the `matplotlib.pyplot` (sub)package. We import it under the name `plt`. You might wonder why we use the name `plt` and not the perhaps more obvious choise `plot`. This is because `plot()` is a command that we will be using, so we do not want to create any conflicts with this function when executing a Python script.

In this section we will explain step-by-step how to generate the figure that we have seen in the previous two sections. We start with plotting the function $f(x) = x^2 + 2x - 1$ for some values of x in a two-dimensional figure.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the x range of x-values
x = np.array([-3,-2,-1,0,1,2,3])

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
```
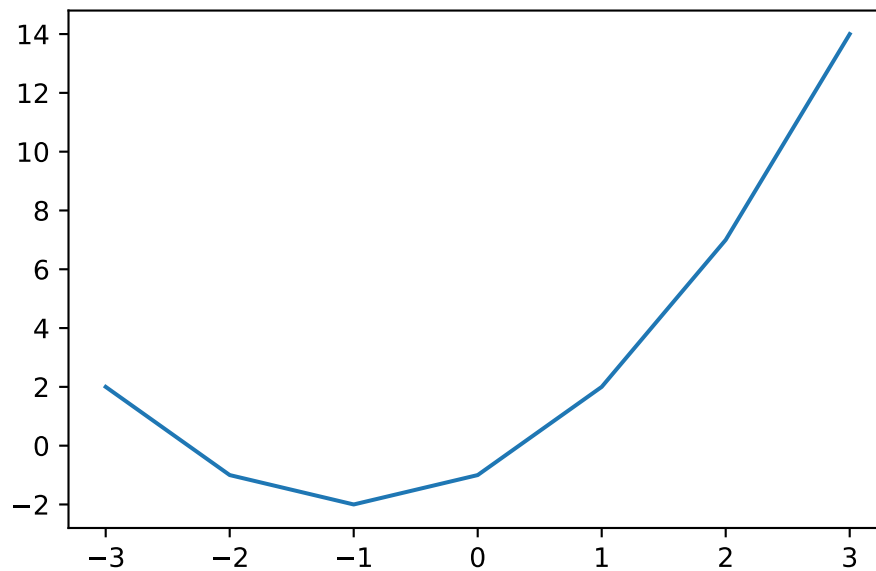
```
y = f(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y)

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


You can view the figure in the Plots pane (or tab) in Spyder.

> If the resolution of the plots in the Plots pane is bad, you can increase it by going to "Tools > Preferences > IPython console > Graphics > Inline backend > Resolution" and set the resolution to, for example, 300 dpi.

> You can get the Plots pane in fullscreen by going to the button with the three horizontal lines in the top right corner and choose "Undock". You can "Dock" the pane again as well if you want to leave the fullscreen mode.
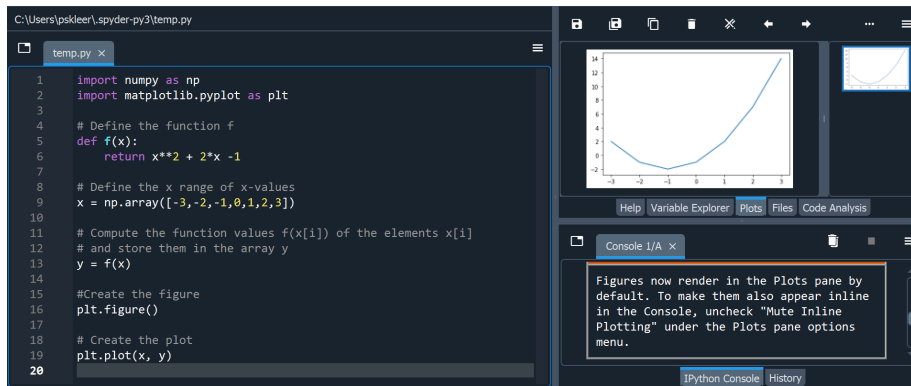
Figure 9.1: IPython Console

We will next explain what the code above is doing. After defining the function f, we create the vector (i.e., Numpy array)

$$x = [x_1, x_2, x_3, x_4, x_5, x_6, x_7] = [-3, -2, -1, 0, 1, 2, 3].$$

We then compute the function values $f(x_i)$ for $i = 1, \ldots, 7$ and store these in the vector y. This might seem a bit strange. We defined the vector f as being a function that takes as input a number x and outputs the number $f(x)$, but now we are inputting a whole vector of numbers x into the function f. Python is capable of handling this, and deals with this by returning the function value for every element of the vector x. That is, it will output the vector

$$[f(x_1), f(x_2), f(x_3), f(x_4), f(x_5), f(x_6), f(x_7)] = [2, -1, -2, -1, 2, 7, 14].$$

We call f a vectorized function: At first glance, it is defined to have a single number as input, but it can also handle a vector as input, in which case it returns the function evaluation for every element of the vector. This typically only works when `x` is defined to be a Numpy array. If we would have defined `x = [-3,-2,-1,0,1,2,3]` as a list of numbers, the code would have given an error (try this yourself!).

> If you use mathematical functions or functions from Numpy, Scipy or Matplotlib, it is best to store numerical input data for these functions in Numpy arrays (and not lists).

Next, we create an (empty) figure using the command `plt.figure()`. Then comes the most important command, `plt.plot(x,y)`, that plots the elements in the vector x against the elements in the vector $y = f(x)$, and connects consecutive combinations (x_i, y_i) and (x_{i+1}, y_{i+1}) with a line segment. For example, we have

$(x_1, y_1) = (-3, 2)$ and $(x_2, y_2) = (-2, -1)$. The left most line segment is formed by connecting these points.

If you only want to plot the points (x_i, y_i), and not the line segments, you can use `plt.scatter(x,y)` instead of `plt.plot(x,y)`.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the x range of x-values
x = np.array([-3,-2,-1,0,1,2,3])

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)

#Create the figure
plt.figure()

# Create the plot
plt.scatter(x, y)
```
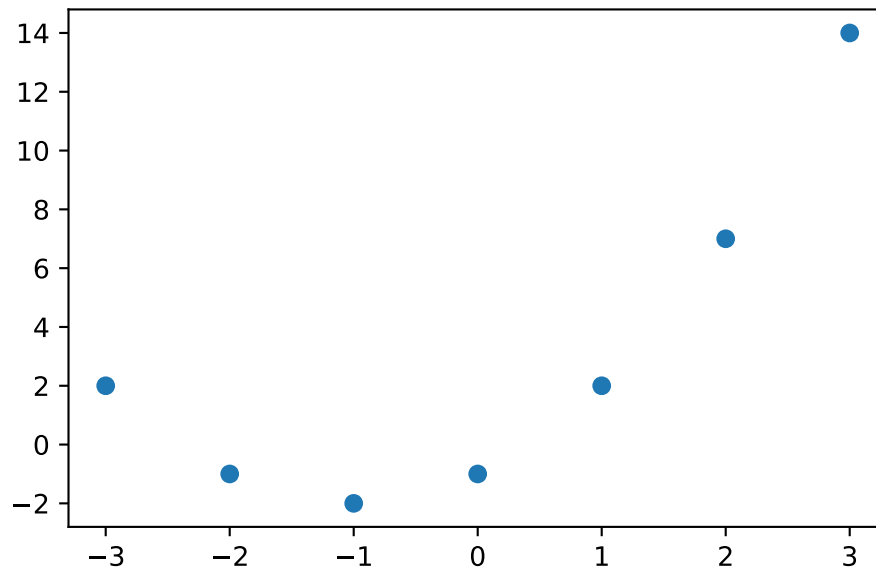

Observe that the (blue) line in the figure that was generated using `plt.plot(x,y)`

is not as "smooth" as in the figures in the previous sections, where the function does not (visibly) have these segments. To get a smoother function line, we can include more points in the vector x.

A quick way to generate a number k of evenly spaced points in the interval $[a, b]$ is the command `np.linspace(a,b,k)` from the Numpy package. It takes as input the bounds of the interval $[a, b]$ and the number of points k that we want to have in it. Consider the following example, where we want to generate $k = 11$ points in the interval $[a, b] = [0, 1]$.

```python
import numpy as np

x = np.linspace(0,1,11)

print(x)

[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

Note that the end points of the interval form the first and last element in the vector x. "Evenly spaced" refers to the fact that the distance between two consecutive points in x is always the same. For this x this common distance is $0.1 = (b - a)/(k - 1)$.

Let us plot again the function f, but this time with 600 elements in x in the interval $[-3, 3]$. We use `plt.plot()` again, instead of `plt.scatter()`. We now obtain a much smoother function line.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y)
```
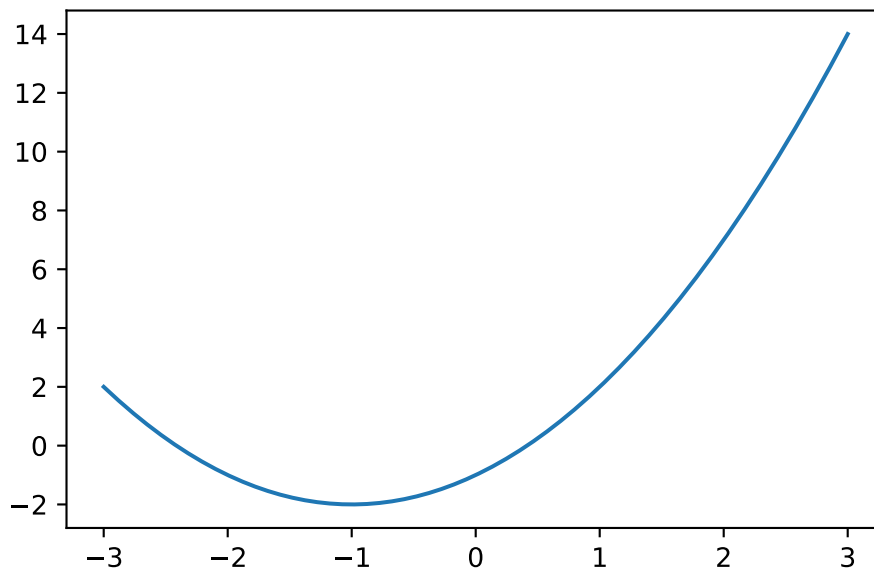
You can add a legend for the line/points that you plot by using the label argument of `plt.plot()`. For example we can add the function description using `plt.plot(x,y,label='$f(x) = x^2 + 2x - 1$')`.

> **i** Note
>
> If you want to add a superscript in a label, you should use the `^` symbol, as opposed to `**` which you should use if you want to do an actual numerical computation with exponentiation.

This is in particular useful if you plot multiple functions in one figure, as the example below illustrates. There we plot the functions f and g, with $g(x) = 3x$ a new function. To have the labels appear in the legend of the figure, you need to add a legend to the figure with `plt.legend()`.

If you want to add labels to the horizontal and vertical axis, you can use the commands `plt.xlabel()` and `plt.ylabel()`.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the function g
def g(x):
```

```
    return 3*x

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)
z = g(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y, label='$f(x) = x^2 + 2x - 1$')
plt.plot(x, z, label='$g(x) = 3x$')

# Create labels for axes
plt.xlabel('x')
plt.ylabel('Function value')

# Create the legend with the specified labels
plt.legend()
```
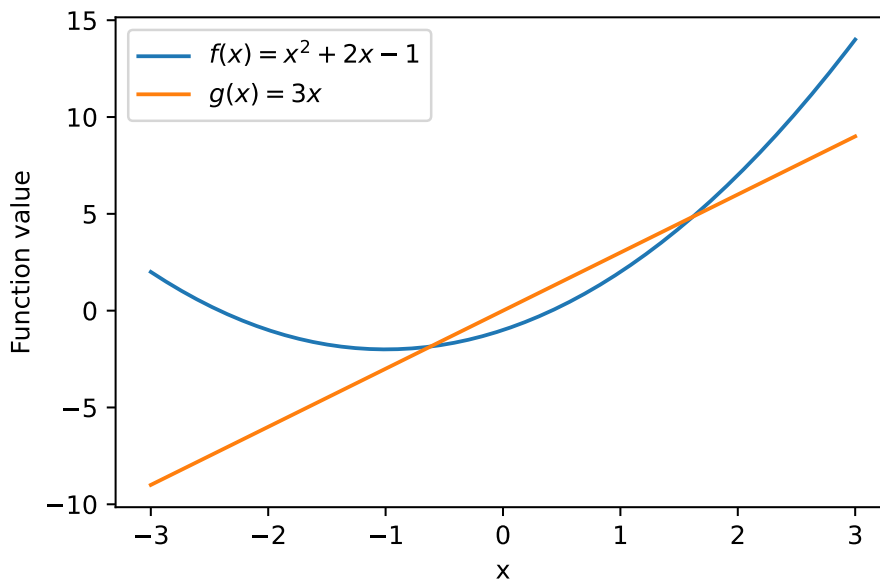

You might observe that the range on the vertical axis changed now that we

added a second function to the plot. When we only plotted the function f, the
vertical axis ranged from -2 to 14, but now with the function g added to it, it
ranges from -10 to 15.

You can fix the range $[c, d]$ on the vertical axis using the command
`plt.ylim(c,d)`, and to fix the range of the horizontal axis to $[a, b]$, you
can use `plt.xlim(a,b)`. In the figure below, we fix the vertical range to
$[c, d] = [-10, 14]$ and the horizontal axis to $[a, b] = [-3, 3]$.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the function g
def g(x):
    return 3*x

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)
z = g(x)

#Create the figure
plt.figure()

# Create the plot
plt.plot(x, y, label='$f(x) = x^2 + 2x - 1$')
plt.plot(x, z, label='$g(x) = 3x$')

# Create labels for axes
plt.xlabel('x')
plt.ylabel('Function value')

# Create the legend with the specified labels
plt.legend()

# Fix the range of the axes
plt.xlim(-3,3)
plt.ylim(-10,14)
```
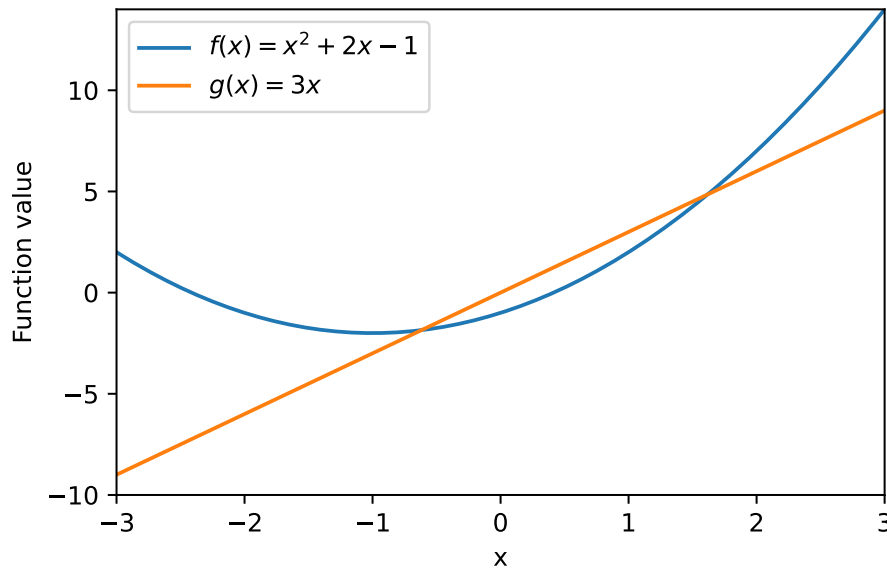
Finally, you can also add a title to the plot using the command `plt.title()` as well as a grid in the background of the figure using `plt.grid()`. These are illustrated in the figure below.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the function f
def f(x):
    return x**2 + 2*x -1

# Define the function g
def g(x):
    return 3*x

# Define the x range of x-values
x = np.linspace(-3,3,600)

# Compute the function values f(x[i]) of the elements x[i]
# and store them in the array y
y = f(x)
z = g(x)

#Create the figure
plt.figure()

# Create the plot
```

```python
plt.plot(x, y, label='$f(x) = x^2 + 2x - 1$')
plt.plot(x, z, label='$g(x) = 3x$')

# Create labels for axes
plt.xlabel('x')
plt.ylabel('Function value')

# Create the legend with the specified labels
plt.legend()

# Fix the range of the axes
plt.xlim(-3,3)
plt.ylim(-10,14)

# Add title to the plot
plt.title('A first plot in Python of two functions')

# Add grid to the background
plt.grid()
```
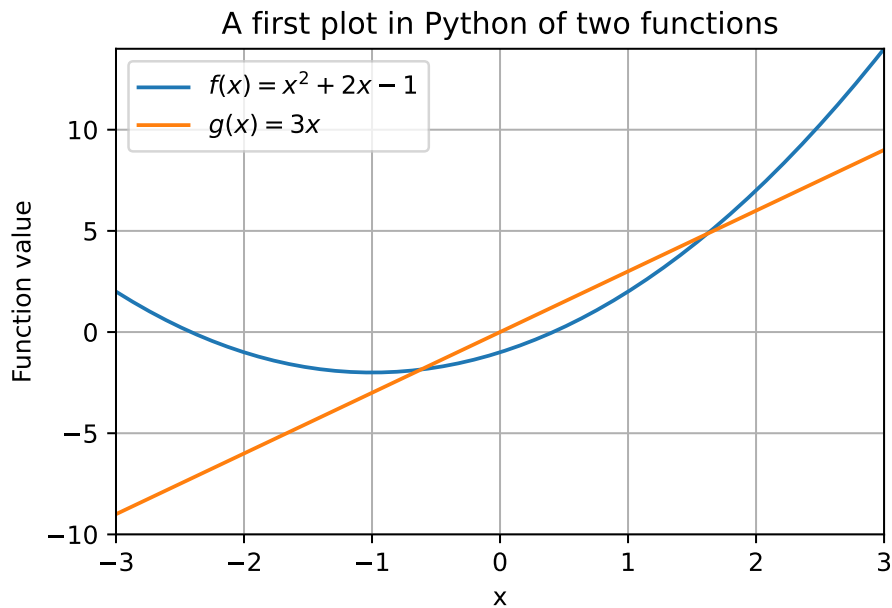

This completes the description of how to plot figures like those we saw in the pre-vious two sections. As a final remark, there are many more plotting options that we do not cover here. For example, with the `plt.xticks()` and `plt.yticks()` commands you can specify the numbers you want to have displayed on the

horizontal and vertical axis, respectively. Also, there are commands to specify line color, width, type (e.g., dashed) and much more! You do not need to know this, but feel free to play around with such commands!

Chapter 10

Object oriented programming

Python is what is called an object oriented programming language. Almost everything in Python is an object on which, e.g., functions can be applied. Recall, for example, the Pandas data frames that we say in the last chapter: We could change elements in the frame, compute summary statistics of its columns, and modify its size.

Objects in Python are generated as instances of a class. Even something simple as a decimal number is an object from a class. To figure out what class an object is from, you can use the `type()` function.

```python
x = 1.42
```

```python
print(type(x)) # x is an object from the class float
```

```
<class 'float'>
```

To illustrate the concept of a class in more detail, we will create a class whose objects model students. The class could, for example, be used to keep track of the student administration at a school or university. You should think of a class as a blueprint: It describes the type of information we keep track of of every student, and how we can retrieve and manipulate it. The basic syntax for creating a class is given by `class class_name:`. For example if we want to create an (empty) class called `Student` we can do the following

```python
class Student:
    # Here the code to create and manipulate objects would be given
    pass
```

The `pass` command here is used because there is no further code in this class. You could use it as a kind of temporary place holder if you know you are going

to create a class, but have not yet code to place into it.

10.1   Attributes

In general, to create an object of a class we need to have some basic information of the object. For example, we can require that every student needs to have a name, age and student number. These pieces of information, needed to create an object of the class, are called attributes. We use a special function in Python called `__init__()`, that can be used to so-called initialize an object. For the student class with attributes name, age and student number (ID), this is done as follows.

```python
class Student:
    # This function initializes an object of the class Student
    # by setting the attributes (name, age and ID) of an object.
    def __init__(self,name,age,student_number):
        self.name = name
        self.age = age
        self.ID = student_number
```

In general, the `__init__()` function has a first input called `self` which you should think of as representing an empty object that we are going to fill with the student's information. The other input arguments of `__init__()` are the inputs for the attributes of the student.

The attributes we set by doing `self.attribute_name = attribute_input`, where `attribute_name` is the name of the attribute (name, age and ID) here and `attribute_input` the input arguments of the function `__init__()` (name, age and student_number here).

Note that the variables we use for the input arguments of `__init__()` do not have to be the same as the names of the attributes themselves. For example, for the student number, we use the input argument `student_number` in `__init__()`, but we set this argument to be the ID attribute (when we do `self.ID = student_number`).

We next create an object (i.e., student) of the class with given inputs for the attributes.

```python
# Creating an object (called stud) of the class Student
student1 = Student("Aidan",19,"2029912")

# Print attributes of the object
print("Name of student is", student1.name)
print("Age of student is", student1.age)
print("Student number of student is", student1.ID)

Name of student is Aidan
```

```
Age of student is 19
Student number of student is 2029912
```

The line `student1 = Student("Aidan",19,"2029912")` creates an object (or instance) of the Student class called `student1`. The input arguments of `Student("Aidan",19,"2029912")` are automatically passed to the (by every class required) `__init__()` function, that is, `name = "Aidan"`, `age=19`, and `student_number="2029912"`. You should think of `student1` becoming the local variable `self` in the `__init__()` function.

It is also possible to, after an object has been created, change its attributes.

```
# Add last name of student as well
student1.name = "Aidan Amir"
```

```
print(student1.name)
```

```
Aidan Amir
```

10.2 Methods

In a class we define, next to `__init__()`, additional functions that can be used to used to modify or obtain information of the student. Such function in a class are called methods. The `__init__()` function is also strictly speaking a method of the class.

As a first example, suppose we want to check if a student is of adult age (18 years or older). We can do this by adding the method `adult()` to the Student class. The first input of such an additional function always should be `self`. Next to that, there could also be additional arguments (we will give an example of this later).

Note that there is a difference between how we access attributes and how we apply methods: Attributes are accessed using `object_name.attribute_name`, but functions are applied with `object_name.method_name()`, i.e., with brackets () at the end.

```python
class Student:
    # This method initializes an object of the class Student
    # by setting the attributes (name, age, and ID) of an object.
    def __init__(self, name, age, student_number):
        self.name = name
        self.age = age
        self.ID = student_number

    def adult(self):
        if self.age >= 18:
            return print(self.name,"is an adult")
```

```python
        else:
            return print(self.name,"is not an adult")
# Create two Student objects
student1 = Student("Aidan",19,"2029912")
student2 = Student("Bella",17,"2034912")

# Check whether students are adults
student1.adult()
student2.adult()

Aidan is an adult
Bella is not an adult
```

10.2.1 Input arguments

We also give an example of a method that takes additional input arguments (next to `self`). Suppose we have a list of student numbers that have registered for the exam of a course. We create a method `reg_check` that takes as input the list of student numbers and checks whether a student (i.e., object) has registered or not by inspecting whether or not the student number of the student is contained in the list.

```python
class Student:
    # This method initializes an object of the class Student
    # by setting the attributes (name, age, and ID) of an object.
    def __init__(self, name, age, student_number):
        self.name = name
        self.age = age
        self.ID = student_number

    def adult(self):
        if self.age >= 18:
            return print(self.name,"is an adult")
        else:
            return print(self.name,"is not an adult")

    def reg_check(self,registrations):
        for i in registrations:
            if(i == self.ID):
                return print(self.name, "is registered")
        return print(self.name, "is not registered")

student1 = Student("Aidan",19,"2029912")

course_A = ["2034757","2029912","2087645","2091234","2198765"]
course_B = ["2034757","2029915","2087645","2091234","2198765"]
```

```
# Aidan's student number is contained in list of exam course A
student1.reg_check(course_A)

# Aidan's student number is not contained in list of exam course B
student1.reg_check(course_B)

Aidan is registered
Aidan is not registered
```

10.2.2 Updating attributes

We can also include more involved attributes in a class. For example, suppose that we want to be able to keep track of the courses that a student is taking part in. To achieve this, we create an extra attribute `courses` that is initialized as an empty list in `__init__()`. In addition to this, we define a method `addCourse()` that allows us to append a course to the courses list. We temporarily remove the methods `age` and `reg_check` to not make the code too long.

```python
class Student:
    # This method initializes an object of the class Student
    # by setting the attributes (name, age, and ID) of an object.
    def __init__(self, name, age, student_number):
        self.name = name
        self.age = age
        self.ID = student_number
        self.courses = []

    def addCourse(self,course_name):
        return self.courses.append(course_name)
```

Note that the `courses` attribute is not obtained from an input argument of `__init__()`. This is because we define it to be an empty list.

```python
student1 = Student("Aidan",19,"2029912")

# Initially the courses list is empty
print(student1.courses)

# Add course Programming
student1.addCourse("Programming")
print(student1.courses)

# Add course Linear Algebra
student1.addCourse("Linear Algebra")
print(student1.courses)

[]
['Programming']
```

```
['Programming', 'Linear Algebra']
```

Similarly, we can also define a method that can be used to delete a course. Deleting an element from a list can be done using the `remove` function, that is illustrated below.

```python
x = ['Aidan','Bella','Charlie']
x.remove('Bella')

print(x)
```

```
['Aidan', 'Charlie']
```

Next, we give the class with the addition and deletion method for courses.

```python
class Student:
    # This method initializes an object of the class Student
    # by setting the attributes (name, age, and ID) of an object.
    def __init__(self, name, age, student_number):
        self.name = name
        self.age = age
        self.ID = student_number
        self.courses = []

    # This method adds a course to the list courses
    def addCourse(self,course_name):
        return self.courses.append(course_name)

    # This method deletes a course from the list courses
    def delCourse(self,course_name):
        return self.courses.remove(course_name)

student1 = Student("Aidan",19,"2029912")

# Initially the courses list is empty
print(student1.courses)

# Add course Programming
student1.addCourse("Programming")
print(student1.courses)

# Add course Linear Algebra
student1.addCourse("Linear Algebra")
print(student1.courses)

# Delete course Programming
student1.delCourse("Programming")
print(student1.courses)
```

```
[]
['Programming']
['Programming', 'Linear Algebra']
['Linear Algebra']
```

10.2.3 Overview

To summarize, a class consists of an initialization method `__init__()` that is mandatory and used to initialize objects/instances of a class by setting the attributes of an object. Next to that, there can be other methods that either can be used to manipulate and/or to derive properties of an object.

We can summarize the aspects of the Student class as follows.

class	Student
Attributes	name, age, student number, courses
Methods	initialization, check age, check registration
	add course, delete course

```python
# Import packages here if needed
# ...

# Define class with all its methods here
class Student:
    # This method initializes an object of the class Student
    # by setting the attributes (name, age, and ID) of an object.
    def __init__(self, name, age, student_number):
        self.name = name
        self.age = age
        self.ID = student_number
        self.courses = []

    # This method check if student is >= 18
    def adult(self):
        if self.age >= 18:
            return print(self.name,"is an adult")
        else:
            return print(self.name,"is not an adult")

    # This method checks if ID is included in list registrations
    def reg_check(self,registrations):
        for i in registrations:
            if(i == self.ID):
                return print(self.name, "is registered")
        return print(self.name, "is not registered")
```

```python
    # This method adds a course to the list courses
    def addCourse(self,course_name):
        return self.courses.append(course_name)

    # This method deletes a course from the list courses
    def delCourse(self,course_name):
        return self.courses.remove(course_name)

# Put here code to initialize objects.
student1 = Student("Aidan",19,"2029912")

# Print attributes or test methods
# ...
```

It is important to separate your testing code from the class. All the methods that form the class have to be indented under the `class Student:` line. Testing code should come afterwards (when the complete class has been defined).

Furthermore, you have to be aware of the difference in syntax between accessing an attribute of an object, or applying a method to it.

```python
# Print attributes with object_name.attribute_name
# and NOT object_name.attribute_name()
print(student1.name) # Not print(student1.name())

# Apply method with object_name.method_name()
# and NOT object_name.method_name. The brackets () indicate
# that you are applying a method. Additional input arguments
# of the method are put in the brackets.
student1.adult()

# Note that for the line above we don't use the print() statement,
# because the adult() method itself returns a print() statement.

Aidan
Aidan is an adult
```

10.3   Inheritance

In Python you can also build classes on top of each other, so that the new class inherit the attributes and methods of an old class. Suppose we start with a simple class Person, whose objects are people that have a name and age. There is also a method in this class that checks whether someone is an adult (18 years or older).

```python
class Person:
    # Initialize a person
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Check if person is an adult
    def adult(self):
        if(self.age >= 18):
            return print(self.name, "is an adult")
        else:
            return print(self.name, "is not an adult")

person1 = Person("Bella", 19)

# Check if Bella is an adult
person1.adult()

Bella is an adult
```

Suppose we now also want to create a class Student, where every student object has the attributes name, age and student number. We can do this by defining this class from scratch, but we can also make more clever use of the class Person that we already have created. Note that in terms of attributes, every student (having age, name and student number) can be seen as a special case of a person (having age and name), but not the other way around. In this case, we call Person the parent class and Student the child class.

It is possible to define the class Student so that it automatically inherits the attributes and methods of the class Person. To achieve this, we should put the parent class in brackets of the new child class that we want to define, so in our case we should define the Student class using `class Student(Person)`. Let us do this with Python code as well, where we in addition also set the student number as attribute.

```python
class Student(Person):
    # Initialize a student
    def __init__(self, name, age, student_number):
        Person.__init__(self,name,age)
        self.ID = student_number
```

The class Student also needs an `__init__()` method, which takes as input all the attributes that we want to set (age, name and student number). Instead of using `self.name = name` and `self.age = age` to set the name and age of a student, we call the `__init__()` function from the Person class with the line `Person.__init__(self,name,age)` with arguments `name` and `age`.

You could, of course, argue that we might as well have typed `self.name = name` and `self.age = age` here, but you should imagine that objects in Python can

have hundreds of attributes. If you are building classes on top of each other with this number of attributes, calling the initialization method from a parent class is much more efficient than setting all the attributes of an object from a child class explicitly again.

After having created an object from the class Person with the line `Person.__init__(self,name,age)`, theryby setting the name and age of the student, we, in addition, set the student number with `self.ID = student_number`.

```
student1 = Student("Bella",19,"20143576")
```

```
print(student1.name, student1.age, student1.ID)
```

```
Bella 19 20143576
```

We can also check is the student is an adult by using the `adult()` method from the class Person. Because we defined the class Student to be the child of the parent class Person, we can use this method right away also on objects of the class Student.

```
# Check if Bella is an adult
student1.adult()
```

```
Bella is an adult
```

10.4   Mathematical example

In this section we will define a class whose objects are two-dimension rectangles in the two-dimensional plane $\mathbb{R}^2$, and we define some methods do derive properties from these rectangles. An example of a rectangle in $\mathbb{R}^2$, with corner points $(1,3),(4,3),(4,1),(1,1)$ is given below.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the x range
x_coord = [1,4,4,1,1]
y_coord = [1,1,3,3,1]

# Create the plot
plt.figure(figsize=(6, 4))
plt.plot(x_coord, y_coord,linewidth=3)

# Add labels and title
plt.title('A rectangle with corners $(1,3), (4,3), (4,1), (1,1)$')

# Add a grid
```
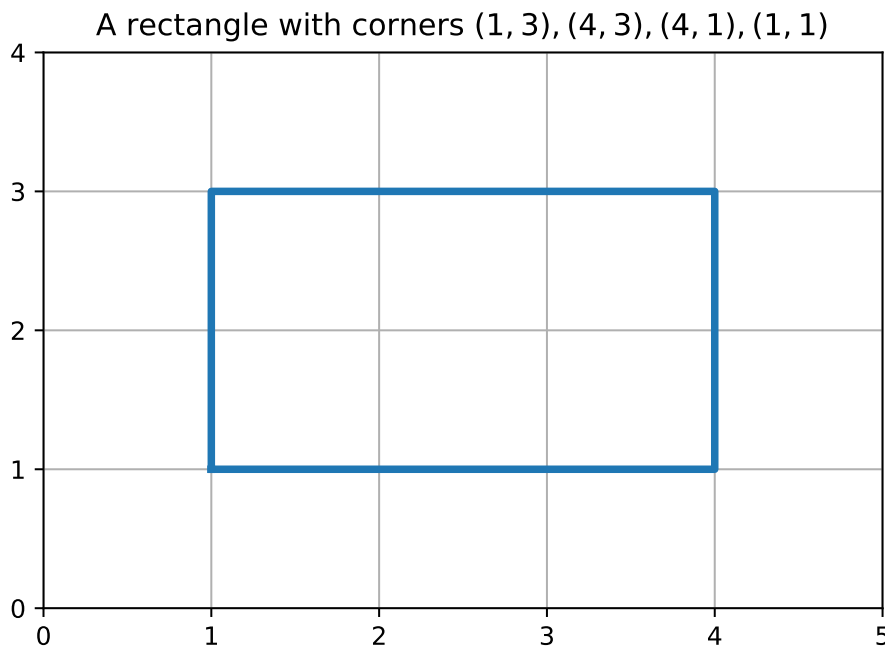
```python
plt.grid(True)

# Set range
plt.xlim(0,5)
plt.ylim(0,4)

# Set ticks
plt.xticks([0,1,2,3,4,5])
plt.yticks([0,1,2,3,4])

# Show plot
plt.show()
```

A rectangle with corners $(1, 3), (4, 3), (4, 1), (1, 1)$

To keep things simple, we assume that the rectangles never appear at an angle, but always with the horizontal and vertical sides parallel to the horizontal and vertical axis, respectively.

We should first think about how we want to represent a rectangle in Python, i.e., what will the attributes of a rectangle object of the class be? There are many possibilities here, for example, we could explicitly choose four corner points, that together form the rectangle, as its attributes. This would require eight input numbers: four x-coordinates and four y-coordinates.

A more efficient way of doing this is to store the coordinates of the upper-

left corner point, the height, and the width of the rectangle (only four input numbers). In the example above the upper-left corner point is $(4, 1)$, the height of the rectangle is 2, and the width is 3. Note that this information is enough to derive the locations of the other corner points, i.e., $(4 + \text{width}, 1), (4, 1 - \text{height}), (4 + \text{width}, 1 - \text{height})$.

```python
class Rectangle:
    # Here corner is the upper-left corner point which
    # should be a list containing the x- and y-coordinate.
    def __init__(self, corner, height, width):
        self.corner= corner
        self.height = height
        self.width = width
```

Note that we assume that the upper left corner is given in a list whose first entry is the x-coordinate of the upper left corner, and whose second entry is the y-coordinate. This is merely a modeling choice; we could also have defined two separate attributes for the x- and y-coordinate.

```python
rectangle1 = Rectangle([1,3],2,3)

# Print upper left corner of the rectangle
print(rectangle1.corner)
```

```
[1, 3]
```

10.4.1   Methods

We will next define three methods that can be used to compute explicitly the corner points of the rectangle, its area and its circumference.

Let us start with the area (width times height) and circumference (two times width plus two times height) of the rectangle. Note that these are relatively straightforward to compute these given that the width and height of a rectangle object are two of its attributes.

```python
class Rectangle:
    # Here corner is the upper-left corner point which
    # should be a list containing the x- and y-coordinate.
    def __init__(self, corner, height, width):
        self.corner= corner
        self.height = height
        self.width = width

    # Compute area
    def area(self):
        return self.width*self.height

    # Compute circumference
```

```python
    def circumference(self):
        return 2*self.width + 2*self.height

rectangle1 = Rectangle([1,3],2,3)

# Compute and print list of corner points of the rectangle
area1 = rectangle1.area()
circumference1 = rectangle1.circumference()

print("The area of the rectange is", area1)
print("The circumference of the rectange is", circumference1)

The area of the rectange is 6
The circumference of the rectange is 10
```

We continue with the corner points method `corners()`; we leave out the area and circumference method from the code to keep the code snippet short.

If we know that the left upper corner point is located at $[x, y]$ then the upper-right corner is $[x + \text{width}, y]$, the lower-right corner $[x + \text{width}, y - \text{height}]$, and the lower-left corner $[x, y - \text{height}]$. We will write a method that returns these points, including the upper-left corner, in a list (resulting in a list of lists as all the corner points themselves will be lists of two elements).

```python
class Rectangle:
    # Here corner is the upper-left corner point which
    # should be a list containing the x- and y-coordinate.
    def __init__(self, corner, height, width):
        self.corner= corner
        self.height = height
        self.width = width

    # Compute corner points: the output has the points in the
    # order [upper-left, upper-right, lower-right, lower-left]
    def corners(self):
        up_right = [self.corner[0] + self.width, self.corner[1]]
        low_right=[self.corner[0]+self.width,self.corner[1]-self.height]
        low_left = [self.corner[0],  self.corner[1] - self.height]
        return [self.corner, up_right, low_right, low_left]
```

Note that the method `corners()` computes the remaining three corner points by using only the upper left corner point, height and width of the rectangle. To illustrate how this is done, let us consider the computation of the upper right corner $[x + \text{width}, y]$ which is stored in the list `up_right` in the code above.

To compute the first coordinate of this point, $x + \text{width}$, we need the first coordinate of the `self` object. Because the attribute `self.corner` is a list of two elements, we can access its first coordinate using `self.corner[0]`. Similary, we can acces its second coordinate with `self.corner[1]`, which is y. This

results in the expression `[self.corner[0] + self.width, self.corner[1]]` for the upper right point $[x + \text{width}, y]$.

```python
rectangle1 = Rectangle([1,3],2,3)

# Compute and print list of corner points of the rectangle
corners1 = rectangle1.corners()
print(corners1)
```

```
[[1, 3], [4, 3], [4, 1], [1, 1]]
```

10.4.2   Plotting method

We conclude this section with a method that can be used to plot a rectangle in a two-dimension plane, like the figure given at the beginning of this section. For the plotting method, we also make use of the method to determine the corner points, so this is included in the class below as well. The following code will be explained below.

```python
import matplotlib.pyplot as plt


class Rectangle:
    # Here corner is the upper-left corner point which
    # should be a list containing the x- and y-coordinate.
    def __init__(self, corner, height, width):
        self.corner= corner
        self.height = height
        self.width = width

    # Compute corner points: the output has the points in the
    # order [upper-left, upper-right, lower-right, lower-left]
    def corners(self):
        up_right = [self.corner[0] + self.width, self.corner[1]]
        low_right=[self.corner[0]+self.width,self.corner[1]-self.height]
        low_left = [self.corner[0],  self.corner[1] - self.height]
        return [self.corner, up_right, low_right, low_left]

    # Plot the rectangle
    def plotting(self):
        plt.figure()

        # Obtain corner points of the rectangle using corners()
        c = self.corners()

        # Define two lists: x-coordinates and y-coordinates
        x_coord = [c[0][0],c[1][0],c[2][0],c[3][0],c[0][0]]
        y_coord = [c[0][1],c[1][1],c[2][1],c[3][1],c[0][1]]
```

```python
        # Define ranges (margin of 3 around the rectangle)
        plt.xlim(self.corner[0]-3,self.corner[0] + self.width + 3)
        plt.ylim(self.corner[1]- self.height - 3,self.corner[1] + 3)

        # Create the plot
        plt.plot(x_coord, y_coord)
        return

rectangle1 = Rectangle([1,3],2,3)

# Corner points of the rectangle
corners1 = rectangle1.corners()
print(corners1)

# Plot the rectangle
rectangle1.plotting()

[[1, 3], [4, 3], [4, 1], [1, 1]]
```
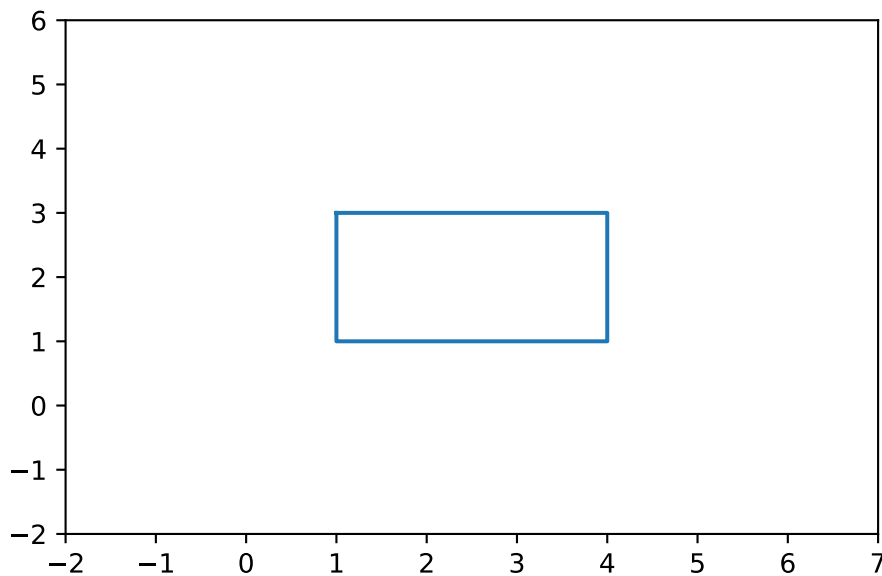

To be able to plot, we first import the `matplotlib.pyplot` package under the usual alias `plt`. We then define the class Rectangle, whose `__init__()` and `corners` methods are the same as before.

We continue with explaining the `plotting()` method. The method first creates an empty figure using `plt.figure()` and determines the corner points of the rectangle using the `corners()` method. They are stored stored in the variable

c. Because `c` is a list of lists, we can access the x-coordinate of the i-th corner point in the list with `c[i][0]`, and the y-coordinate with `c[i][1]`.

Recall that the `plot()` function plots two lists/arrays $x = [x_1, \ldots, x_n]$ and $y = [y_1, \ldots, y_n]$ against each other by plotting the points (x_i, y_i) and then connecting them with line segments. To plot a rectangle, we therefore define

$$x = [x_{\text{upper-left}}, x_{\text{upper-right}}, x_{\text{lower-right}}, x_{\text{lower-left}}, x_{\text{upper-left}}]$$

in the list `x_coord` and

$$y = [y_{\text{upper-left}}, y_{\text{upper-right}}, y_{\text{lower-right}}, y_{\text{lower-left}}, y_{\text{upper-left}}].$$

in the list `y_coord`.

Note that the command `plt.plot(x_coord,y_coord)` first draws the point $(x_{\text{upper-left}}, y_{\text{upper-left}})$, then $(x_{\text{upper-right}}, y_{\text{upper-right}})$, etc. and connects them. It is important that we include the upper-left point at the end again so that the left vertical side of the rectangle is drawn as well.

Finally, we also set the range of the horizontal and vertical axis with the `xlim()` and `ylim()` functions. We include a margin of 3 around the box. To see how this works for the horizontal axis, note that `self.corner[0]` contains the x-coordinate of the upper-left point, and `self.corner[0] + width` that of the upper-right point. Therefore, to create a margin of 3 on both sides of the rectangle, we let the hortizontal axis range from `self.corner[0] - 3` to `self.corner[0] + width + 3`.

10.4.3 Overview

The complete class of the previous subsections, containing all the methods that we set up, is given below.

```python
import matplotlib.pyplot as plt


class Rectangle:
    # Here corner is the upper-left corner point which
    # should be a list containing the x- and y-coordinate.
    def __init__(self, corner, height, width):
        self.corner= corner
        self.height = height
        self.width = width

    # Compute area
    def area(self):
        return self.width*self.height

    # Compute circumference
```

```python
    def circumference(self):
        return 2*self.width + 2*self.height

    # Compute corner points: the output has the points in the
    # order [upper-left, upper-right, lower-right, lower-left]
    def corners(self):
        up_right = [self.corner[0] + self.width, self.corner[1]]
        low_right=[self.corner[0]+self.width,self.corner[1]-self.height]
        low_left = [self.corner[0],  self.corner[1] - self.height]
        return [self.corner, up_right, low_right, low_left]

    # Plot the rectangle
    def plotting(self):
        plt.figure()

        # Obtain corner points of the rectangle using corners()
        c = self.corners()

        # Define two lists: x-coordinates and y-coordinates
        x_coord = [c[0][0],c[1][0],c[2][0],c[3][0],c[0][0]]
        y_coord = [c[0][1],c[1][1],c[2][1],c[3][1],c[0][1]]

        # Define ranges (margin of 3 around the rectangle)
        plt.xlim(self.corner[0]-3,self.corner[0] + self.width + 3)
        plt.ylim(self.corner[1]- self.height - 3,self.corner[1] + 3)

        # Create the plot
        plt.plot(x_coord, y_coord)
        return
```

Chapter 11

Errors and debugging

While writing Python scripts, you probably have encountered the situation that Python returned an error message, indicating that there is something wrong with your code. When writing a script from scratch this will often happen. Roughly speaking, most of the time that you spend on creating a working Python script goes into correcting errors in the first draft of your code. This process is called debugging.

In this chapter we will discuss different types of errors that Python might return, as well as some general strategies that you might adopt in order to efficienctly debug your code.
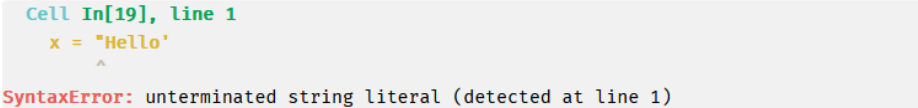
11.1   Error types

The three main types of errors that can occurs in Python are syntax, runtime and logical errors. We will give some examples of each of these three errors in the coming subsections.

11.1.1   Syntax error

A syntax error occurs when Python does not understand the words or characters, i.e., syntax, that you use in your code. In more technical terms, Python is unable to interpret the code that you wrote because you did not use the correct syntax.

```
x = "Hello'
```

```
  Cell In[19], line 1
    x = "Hello'
              ^
SyntaxError: unterminated string literal (detected at line 1)
```
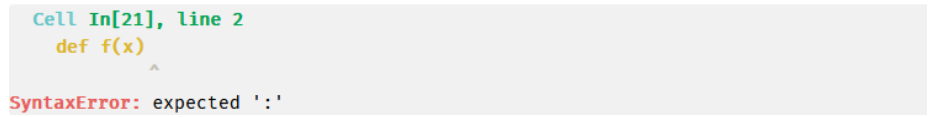
Figure 11.1: SyntaxError

In the example above, we define a string `x`, but on the left we use a double quote and on the right a single quote. This is not correct, we should either use double quotes or single quotes on both sides. The output message indicates that the error occurs on the first line of the code, although the term "unterminated string literal" might not directly make sense to you.

Roughly speaking, Python tries say you started the creation of a string by using the double quotes at the beginning, and is therefore expecting the string to be concluded somewhere as well with double quotes, but this never happens. The code should have been as follows.

```
x = "Hello"
```

Sometimes it is more obvious what is going wrong in your code. In the example above, Python raises an error saying that the colon `:` is missing, which is needed in order to define a function in Python.

```
# Define the function f(x) = x^2
def f(x)
    return x**2
```

```
  Cell In[21], line 2
    def f(x)
            ^
SyntaxError: expected ':'
```

Figure 11.2: SyntaxError

The code should have been as follows.

```
# Define the function f(x) = x^2
def f(x):
    return x**2
```

11.1.2 Runtime error

A runtime error occurs when Python runs (or executes) your code, but cannot create any output because the operations you try to perform on the variables that you defined are not possible. These errors are typically trickier than syntax errors: It might be that on some input data, your code does exactly what you want it to do, but on other input data it is not able to execute your code.

Consider the example below where we define a division function that takes as input two numbers a and b, and returns a/b.

```
def division(a,b):
    return a/b

a = 1
```

```
b = 3
c = division(a,b)

print(c)
```

0.3333333333333333

This function works perfectly fine for the inputs `a = 1` and `b = 3`. However, if we we would take $b = 0$, then Python raises a ZeroDivisionError, which indicates that we are trying to divide by 0, which is mathematically not possible.

```
def division(a,b):
    return a/b


a = 1
b = 0
c = division(a,b)

print(c)
```


Figure 11.3: ZeroDivisionError

Another common runtime error in the context of functions is a TypeError. Such an errors occur when, you do not provide the correct number of input arguments for a function. In the code below, we have forgotten to input the second variable `b` that the function `division()` requires.

```
def division(a,b):
    return a/b


d = 1
e = 3
f = division(d)

print(c)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 6
      4 d = 1
      5 e = 3
----> 6 f = division(d)
      8 print(c)

TypeError: division() missing 1 required positional argument: 'b'
```

Figure 11.4: TypeError

In the context of indexing or accessing data in, e.g., a list, a common runtime error is the IndexError. Such an error occurs when you are, for example, trying to access an element of a string that does not exist. In the example below the list `x` has four elements. Since Python starts counting indices at 0, we have `x[0]` = 2, `x[1]` = 1, `x[2]` = 6 and `x[3]` = 8. Therefore, `x[4]` does not exist and so Python is not able to access it.

```python
x = [2,1,6,8]

print(x[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[26], line 3
      1 x = [2,1,6,8]
----> 3 print(x[4])

IndexError: list index out of range
```

Figure 11.5: IndexError

For scripts with classes, a common runtime error is an AttributeError which occurs if you are trying to access an attribute of an object that does not exist. In the below example, every student object has attributes name and age, but not ID. The code executes correctly up to the last line, but the last line itself gives an error.

```python
class Student:
    def __init__(self,name,age):
        self.name = name
        self.age = age

student1 = Student("Aidan",19)
print(student1.ID)  # ID attribute does not exist
```

```
--------------------------------------------------------------------------------
AttributeError                                    Traceback (most recent call last)
Cell In[27], line 7
      4            self.age = age
      6 student1 = Student("Aidan",19)
----> 7 print(student1.ID)

AttributeError: 'Student' object has no attribute 'ID'
```

Figure 11.6: AttributeError

11.1.3   Logical error

Arguably the most difficult to deal with are logical errors. These are errors that are not raised by Python, but are errors in the logic of the code. For example, a function that does not return the output it is supposed to return.

Consider the example below where the goal is to create a function that substracts a number b from another number a, i.e., the function is supposed to return $a - b$. The programmer made a mistake and instead, the function returns $a + b$. There is nothing wrong with the syntax of the function, nor does Python encounter issues while running the code. However, the function does not what the programmer wants it to do.

```python
def substract(a,b):
    #This function substracts the number b from a
    return a + b


a = 1
b = 2
c = substract(a,b)

print(c) # Should be 1 - 2 = -1

3
```

In the example above, it is fairly easy to conclude that the function is not behaving as it should, because almost all choices of a and b (excluding cases where a and/or b or zero) will return the wrong output.

Sometimes, however, logical errors are harder to spot as a function might still give the correct ouput for many input arguments.

As an example, suppose we want to create a function that can decided whether two intervals $[a, b]$ and $[c, d]$, with $a, b, c, d \in \mathbb{R}$ and $a \leq b, c \leq d$ are overlapping or not. If the first interval is $[a, b]$ lies completelely left of the interval $[c, d]$, i.e., if $b < c$, then the intervals do not overlap.

```python
def overlap(a,b,c,d):
    if b < c:
```

```
        return print("No overlap!")
    else:
        return print("There is overlap.")

a = 1
b = 3
c = 0.5
d = 3.4

overlap(a,b,c,d)

There is overlap.
```

However, this function does not always give the correct output, because it does not account for the fact that the second interval $[c, d]$ might lie completely left of the first interval $[a, b]$, in which case there is also no overlap.

```
def overlap(a,b,c,d):
    # This function check whether two intervals [a,b] and [c,d]
    # have overlap or not.
    if b < c:
        return print("No overlap!")
    else:
        return print("There is overlap.")

a = 1
b = 3
c = -3
d = -2

overlap(a,b,c,d) # Gives wrong output

There is overlap.
```

To avoid logical errors, it is recommended to always test your code and functions on a diverse collection of input data.

11.2   Exceptions

Instead of having Python code crash due to an error, there is also the possibility to "catch" certain errors directly in a function. This allows a programmer to specify what should happen if a certain error occurs, without the whole code crashing. You can do this with a `try-except` construction.

Let's look at an example. We consider the function g that computes the square root of the first k powers of a number x and sums them up, i.e.,

$$g(x, k) = \sqrt{\sum_{i=1}^{k} x^k} = \sqrt{x + x^2 + x^3 + \cdots + x^k}.$$

Note that x can be any number, but k has to be an integer for the summation to be well-defined. For example, for $x = 1.3$ and $k = 3$, we have $g(1.3, 3) = 1.3 + 1.3^2 + 1.3^3 = 5.187$.

```python
import math

# Define the function g(x,k)
def g(x,k):
    sum = 0
    # Here we compute the sum
    for i in range(k):
        sum = sum + x**(i+1) # i + 1, because range() starts at 0
    return math.sqrt(sum)

x = 1.3
k = 3
print(g(x,k))
```

```
2.2774986278810356
```

Multiple things can go wrong with trying to execute this function. For example, we might input a number k which is not an integer. This gives rise to a TypeError, because the `range()` function only works for integer inputs.

```python
x = 1.3
k = 2.1
g(x,k)
```

```
--------------------------------------------------------------------------------
TypeError                                      Traceback (most recent call last
Cell In[96], line 3
      1 x = 1.3
      2 k = 2.1
----> 3 g(x,k)

Cell In[95], line 7, in g(x, k)
      5 sum = 0
      6 # Here we compute the sum
----> 7 for i in range(k):
      8     sum = sum + x**(i+1) # i + 1, because range() starts at 0
      9 return math.sqrt(sum)

TypeError: 'float' object cannot be interpreted as an integer
```

Figure 11.7: TypeError

Or we might input a number x for which the summation $\sum_{i=1}^{k} x^k$ is negative, in which case we cannot take the square root (recall that the square root of a negative number does not exist).

```
x = -10
k = 3
g(x,k)
```

```
--------------------------------------------------------------------------------
ValueError                                      Traceback (most recent call last)
Cell In[9], line 3
      1 x = -10
      2 k = 3
----> 3 g(x,k)

Cell In[7], line 9, in g(x, k)
      7 for i in range(k):
      8     sum = sum + x**(i+1) # i + 1, because range() starts at 0
----> 9 return math.sqrt(sum)

ValueError: math domain error
```

Figure 11.8: ValueError

To inform a user that tries to execute this function what goes wrong, we can use a `try-except` construction. The rough outline of such a construction is as follows.

```
try:
    # Some code that Python should execute
```

```python
except SomeError:
    # If an error of the type SomeError occurs
    # then do ...
except AnotherError:
    # If an error of the type AnotherError
    # occurs, then do ...
```

A simple example, when trying to divide by zero, is given below.

```python
try:
    5/0
except ZeroDivisionError:
    print("You cannot divided by zero.")
```

```
You cannot divided by zero.
```

Python will first try to execute the code under the **try** statement. If this gives rise to an error (i.e., exception), then it will do as is specified under the **except** statement of the error that was raised.

You can use the **try-except** construction also within a function. Let us do this for the function g above.

```python
import math

# Define the function g(x,k)
def g(x,k):
    try:
        sum = 0
        # Here we compute the sum
        for i in range(k):
            sum = sum + x**(i+1) # i + 1, because range() starts at 0
        return math.sqrt(sum)
    except TypeError:
        return print("Parameter k should be an integer.")
    except ValueError:
        return print("Cannot take square root of negative power sum.")
```

```python
x = 1.3
k = 2.1
g(x,k) # Gives TypeError because k not integer.
```

```
Parameter k should be an integer.
```

```python
x = -10
k = 3
g(x,k) #Gives ValueError because power sum is negative.
```

```
Cannot take square root of negative power sum.
```

11.3   Debugging

The process of identifying and fixing errors (also known as bugs), is called debugging. One of the reasons errors are called bugs goes back to a computer that was built at Harvard University in 1947. At some point a moth got into the computer, which were giant at the time (the computers not the moths), and caused a piece of hardware to malfunction.

In general, there is no fixed recipe for how to debug a function that is not working properly, but there are many ways to get more insights in what a function is doing step-by-step, giving you the option to identify where something is going wrong. These are especially useful when you are trying to fix logical errors.

Below we will give some examples of what you can do to get better insights in what is going wrong in your function. This is by no means meant to be an exhaustive list.

11.3.1   Use `print()` statements

One way to get some insight into what your function is doing, is to create print statements in the code. Let us have another look at the example given above.

```python
import math

# Define the function g(x,k)
def g(x,k):
    sum = 0
    # Here we compute the sum
    for i in range(k):
        sum = sum + x**(i+1) # i + 1, because range() starts at 0
    return math.sqrt(sum)

x = 1.3
k = 3
print(g(x,k))
```

```
2.2774986278810356
```

```
--------------------------------------------------------------------------
TypeError                                   Traceback (most recent call last
Cell In[96], line 3
      1 x = 1.3
      2 k = 2.1
----> 3 g(x,k)

Cell In[95], line 7, in g(x, k)
      5 sum = 0
      6 # Here we compute the sum
----> 7 for i in range(k):
      8     sum = sum + x**(i+1) # i + 1, because range() starts at 0
      9 return math.sqrt(sum)

TypeError: 'float' object cannot be interpreted as an integer
```

Figure 11.9: TypeError

Python tells us that there is a math domain error, but it does not give any explicit information about what went wrong. Given that we use the `sum` variable as input for the square root function, it is a good idea to keep track of how the `sum` argument changes throughout the execution of the function, especially what its value is when we try to input it into the square root function. We use a `try-except` construction to not have Python print the whole error code.

```python
import math

# Define the function g(x,k)
def g(x,k):
    try:
        sum = 0
        # Here we compute the sum
        for i in range(k):
            print("Sum at start iteration",i,":",sum) # Print here to see how sum changes
            sum = sum + x**(i+1) # i + 1, because range() starts at 0
        print("Sum after for-loop:", sum) # Print sum at end of for-loop
        return math.sqrt(sum)
    except ValueError:
        return print("Something goes wrong")

x = -10
k = 3
print(g(x,k))

Sum at start iteration 0 : 0
Sum at start iteration 1 : -10
```

```
Sum at start iteration 2 : 90
Sum after for-loop: -910
Something goes wrong
None
```

From the execution it can be seen that the `sum` variable is negative right before we try to use `math.sqrt(sum)`. This is what causes the error: the value of `sum` is outside of the domain of the square root function (the domain is all nonnegative numbers for which the square root is well-defined).

The `None` printed at the end is the result of the final print statement `print(g(x,k))` which cannot be properly executed because `g(x,k)` has not been computed.

11.3.2   Use `assert`

If we already would have had some suspicion that the `sum` variable being negative is what is causing the error in the function g, we could have also checked this quickly by adding the command `assert sum >= 0` after the for-loop, instead of adding the print() statements and the `try-except` construction. If the assertion is not true, then Python raises an AssertionError.

```python
import math

# Define the function g(x,k)
def g(x,k):
    sum = 0
    # Here we compute the sum
    for i in range(k):
        sum = sum + x**(i+1) # i + 1, because range() starts at 0
    assert sum >= 0 # Check if sum is nonnegative
    return math.sqrt(sum)

x = -10
k = 3
print(g(x,k))
```

```
--------------------------------------------------------------------------------
AssertionError                          Traceback (most recent call last)
Cell In[68], line 14
     12 x = -10
     13 k = 3
---> 14 print(g(x,k))

Cell In[68], line 9, in g(x, k)
      7 for i in range(k):
      8     sum = sum + x**(i+1) # i + 1, because range() starts at 0
----> 9 assert sum >= 0 # Check if sum is nonnegative
     10 return math.sqrt(sum)

AssertionError:
```

Figure 11.10: AssertionError

We can also check the assertion with a `try-except` construction to avoid the error being printed.

```python
import math

# Define the function g(x,k)
def g(x,k):
    try:
        sum = 0
        # Here we compute the sum
        for i in range(k):
            sum = sum + x**(i+1) # i + 1, because range() starts at 0
        assert sum >= 0 # Check if sum is nonnegative
        return math.sqrt(sum)
    except AssertionError:
        return print("The sum is negative; cannot compute square root.")

x = -10
k = 3
print(g(x,k))
```

```
The sum is negative; cannot compute square root.
None
```
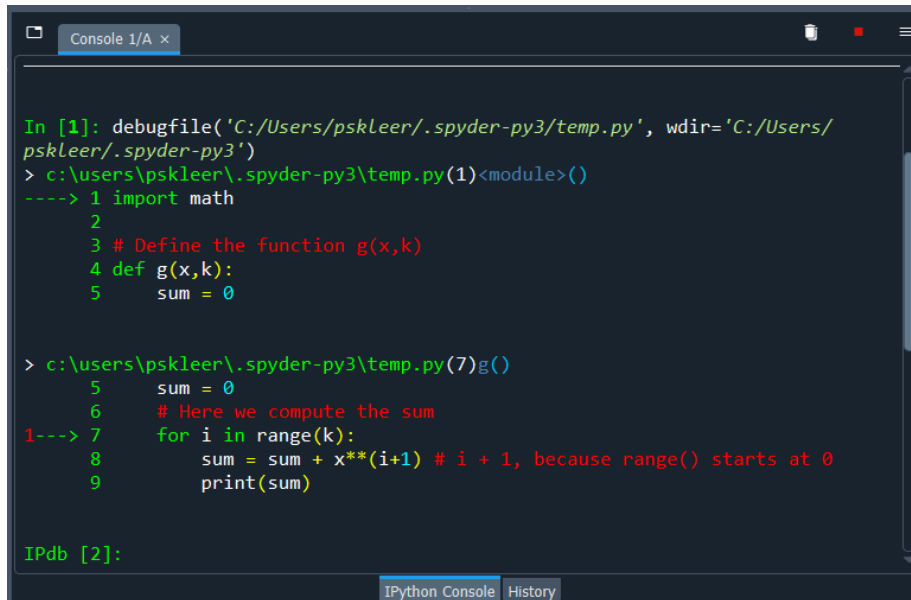
11.3.3 Use Spyder debugger

Spyder also has the option to debug your code. What this means is that Spyder allows you to execute the code line-by-line, therefore making it easier for you to figure out at which specific line your function is running into trouble. We will next explain the very basics of debugging in Spyder.

The first thing you should do is set a breakpoint somewhere. What will happen

is that Python will run your code until the moment it tries to execute the code on the breakpoint line. The breakpoint you set by click just right of the line number on which you want to place the breakpoint. It will then appear as a red circle next to the line number.

Figure 11.11: Setting a breakpoint

After you have placed the breakpoint, you should go to Debug in the toolbar of Spyder, and select Debug (Ctrl+F5).

Figure 11.12: Start debug mode

Then in the console you will see the following appearing, which indicated that you have gone into debug mode. It shows that the current execution halted at line 7, which is the line where we added the breakpoint. This is the start of the for-loop.
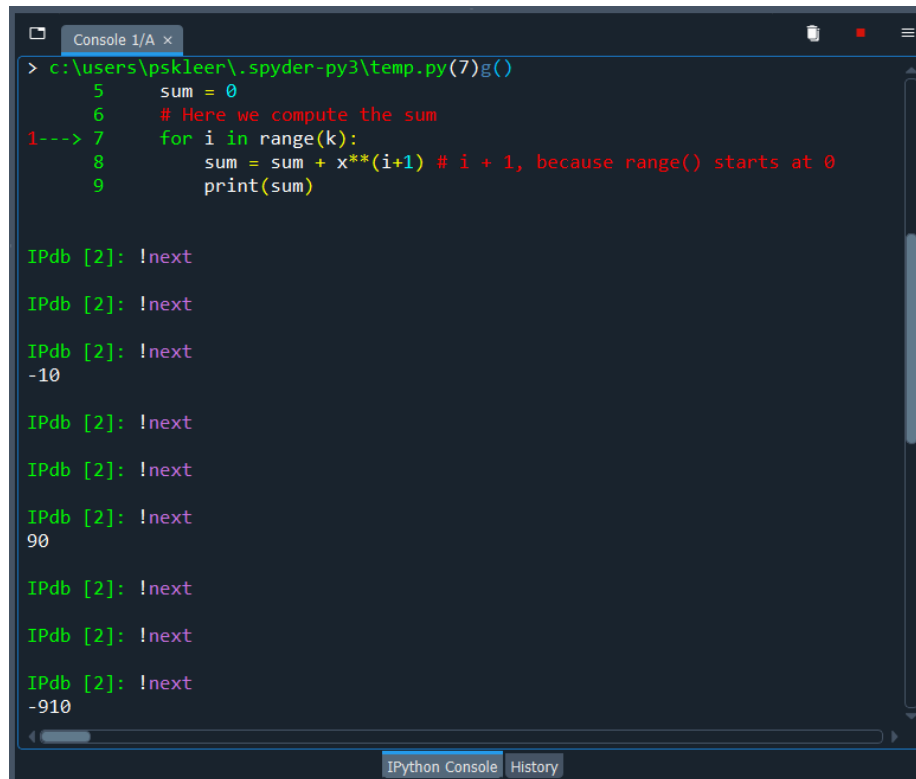
Figure 11.13: Start debug mode

By pressing Ctrl+F10 (the Step option in the Debug tab), you can move one line, i.e., step, down in the execution of the code. The current line that Python is at, is denoted by the blue arrow pointing rightwards. Doing Ctrl+F10 a couple of times, we reach line 9 at which point Python will print -10. This finishes the execution of the for-loop for $i = 0$. If you again press Ctrl+F10, you will see that the arrow jumps back to line 7. This indicates that we again start the for-loop, but now with $i = 1$.

Lines that contain a print statement will have their content appear in the console. Below you can see the output -10, arising from `print(sum)` on line 10 in the round of the for-loop with $i = 0$, 90 from the round of the for-loop with $i = 1$, and -910 from $i = 2$. Recall that $k = 3$ in this example, so the for-loop stops after $i = 2 = k - 1$.

Figure 11.14: Start debug mode

In general, Python will go one-by-one through all the rounds of the for-loop, for the values $i = 0, \dots, k-1$. Afterwards, Python will continue with the return statement on line 10. This line-by-line execution makes it possible to identify the exact moment where your code is doing something wrong.

While doing line-by-line exectution in debugging mode, you can also keep track of how variables in your code change in the variable explorer. This often also gives insights into what your code is doing.