# 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 often 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

1

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, Chat GPT has become a very useful resource for Python. Chat GPT can write excellent Python code and also explains all the steps it takes, so I encourage you to use it to help you learn. Although keep in mind it won't be available to you 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.
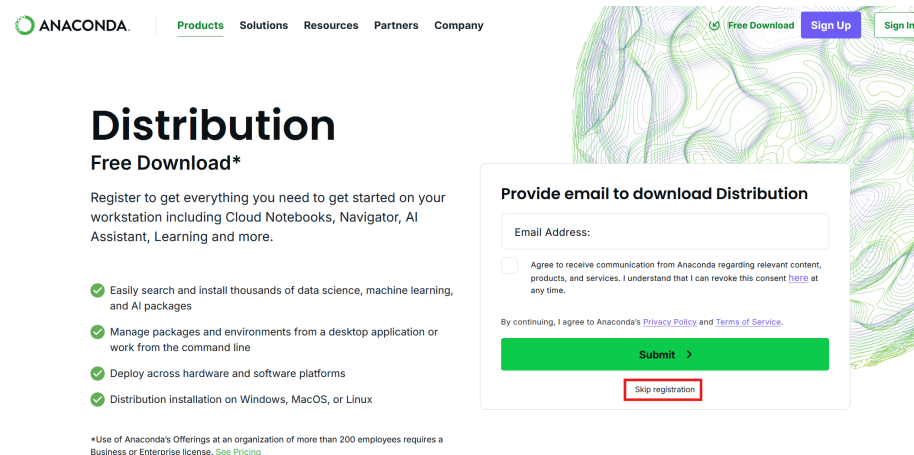
You should see this page:



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:
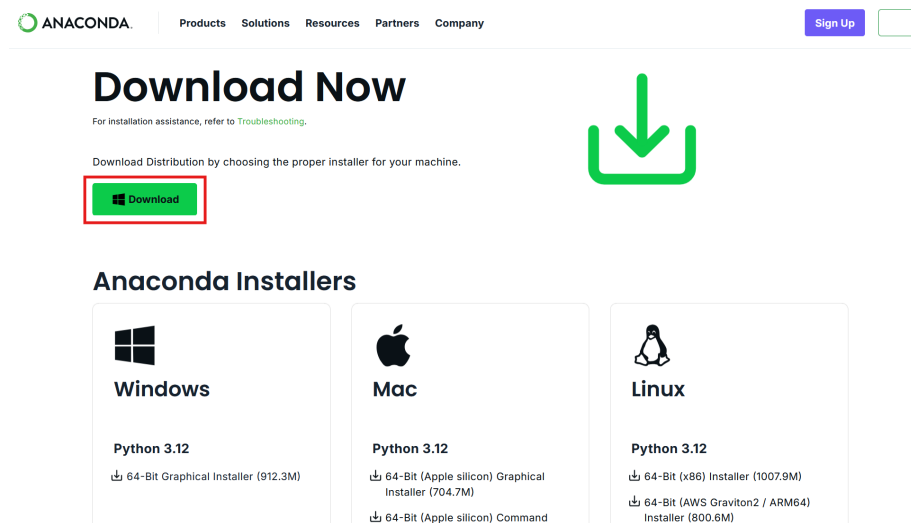
Figure 2.2: Anaconda Download Page

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.

## 2.2 Spyder

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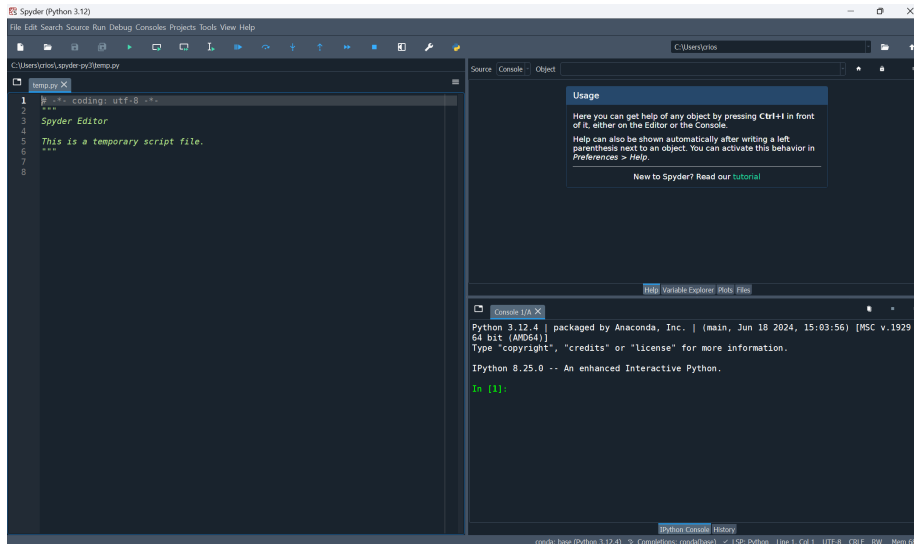


Figure 2.3: Spyder
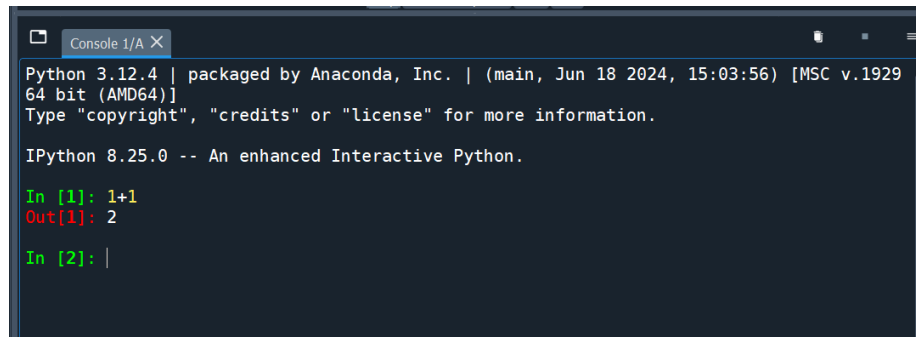
### 2.2.1 Python Console

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]`:

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.

Figure 2.4: IPython Console

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:
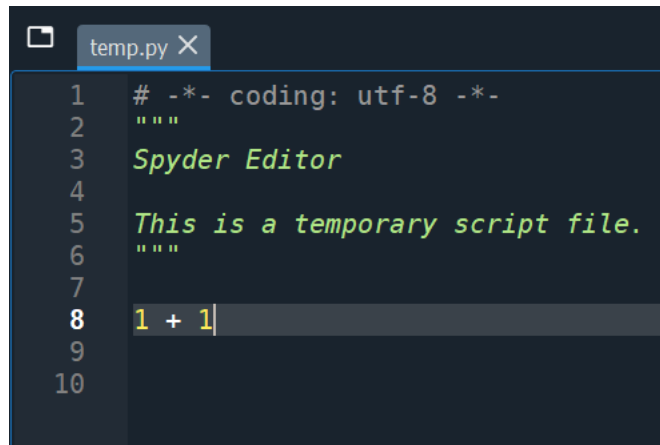
- If you have run 10 commands to calculate something and then afterwards you decide to change what happened in one of the earlier commands, you would often have type all the commands again. In a script you would just need to edit the line with that command. So scripts can save you a lot of time.
- You 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:

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.
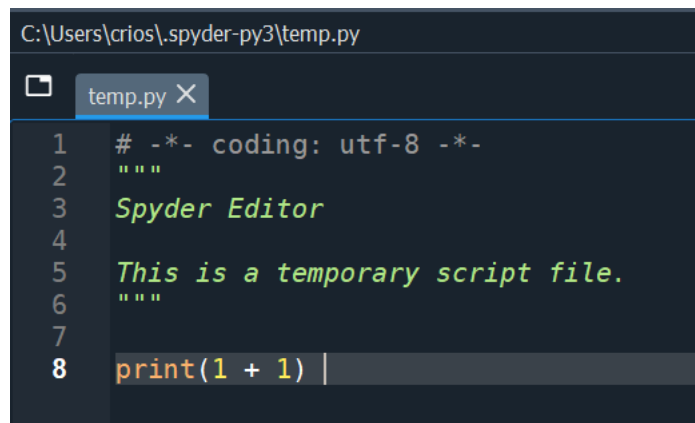
Figure 2.5: Python Script

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.

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
```

---

[1]The `keyword` module contains functions for testing if a string is a keyword. The command `keyword.kwlist` returns the full list of keywords.

```
plt.legend()

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

## Absolute Value Function
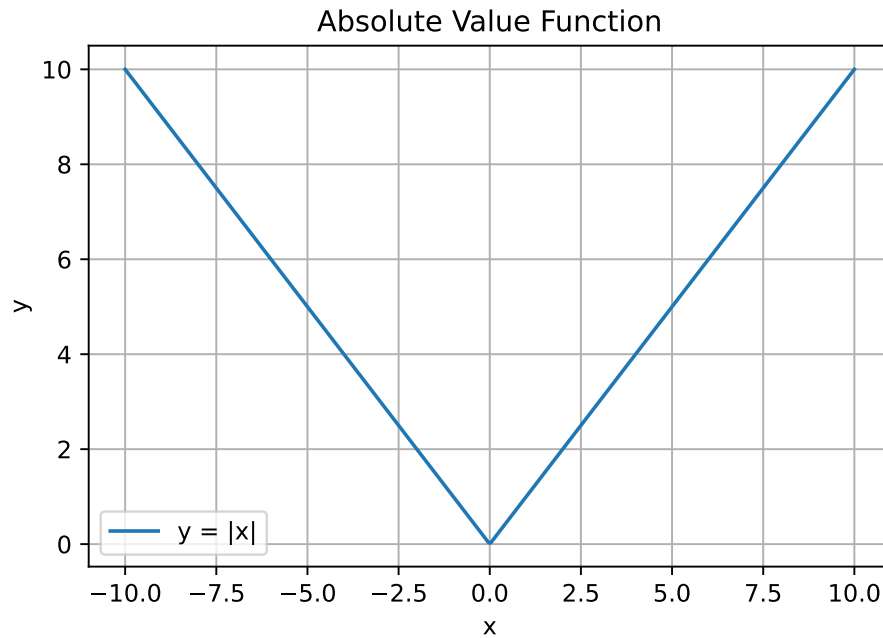


We'll learn how to make plots like this later in this course, but if you want to see the code generating it you can click on the button "Show code generating the plot below".

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.

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
plt.legend()

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

Square Root Function
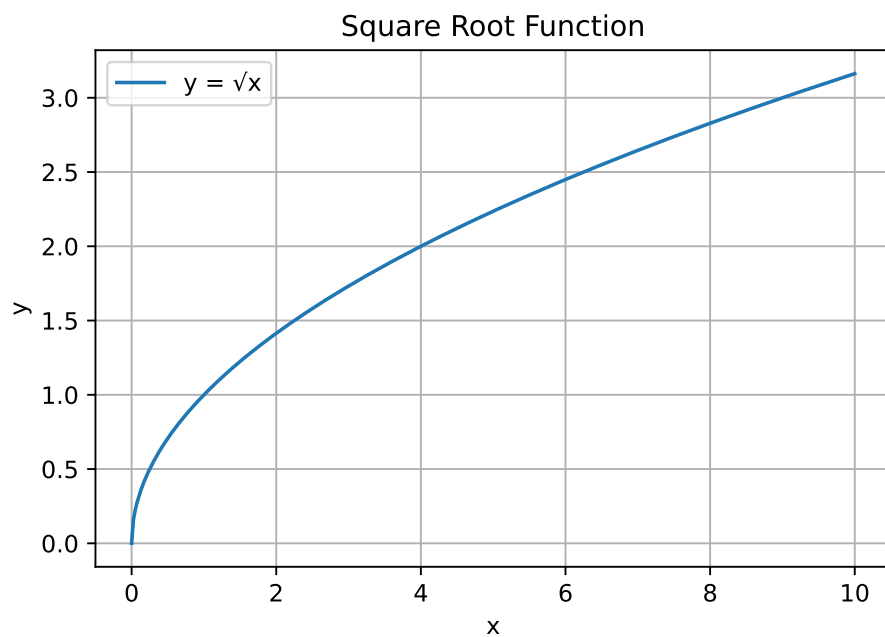


The principal square root a number is equal to the number exponentiated by $\frac{1}{2}$:

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

```
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:

```
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:

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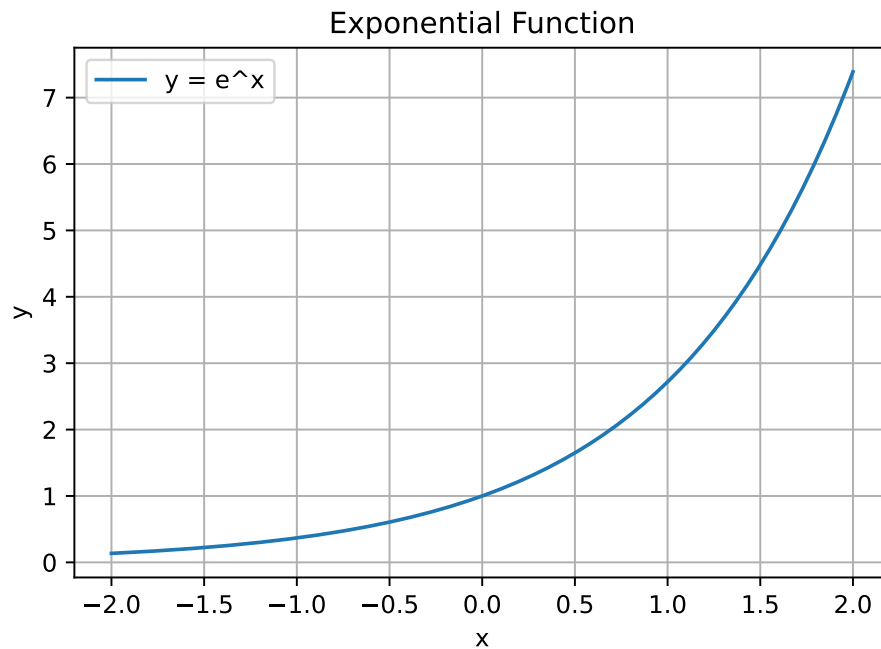
In Python we can use the `exp()` function from the `math` module to calculate the exponential of any number:

```
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-positiv
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
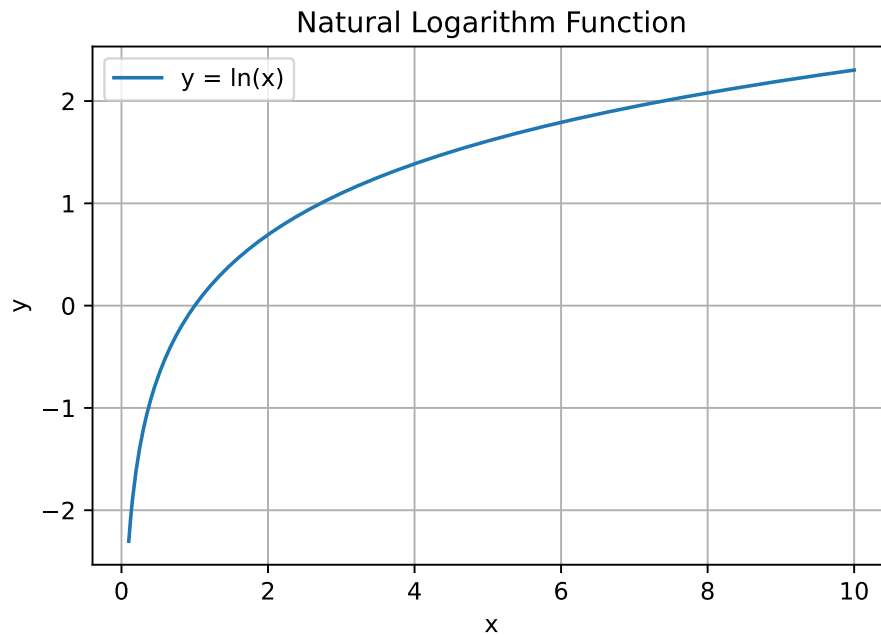


In Python we use the `log()` function from the `math` module to calculate the natural logarithm:

```
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.

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

```python
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:

```python
7 // 3
```

2

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

```python
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:

```python
-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:

We can see that x has a value 2 and y has a value 3.
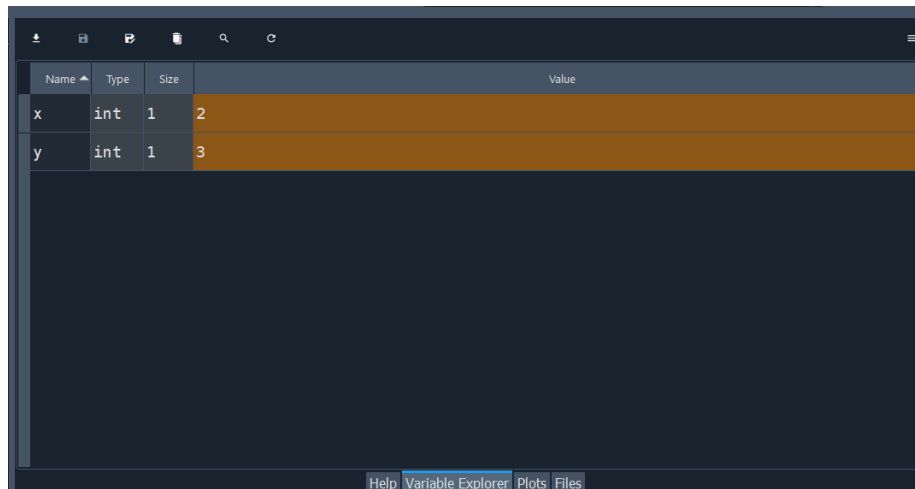
Figure 4.1: Variable Explorer in Spyder

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

## 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.

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

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

---

[1]The keyword module contains functions for testing if a string is a keyword. The command keyword.kwlist returns the full list of keywords.

## 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 $a \not\geq b$.

## 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 stringers. `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:

```python
bool(1)
```

`True`

```python
bool(0)
```

`False`

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

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

`True`

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

`True`

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

```python
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:

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

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

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

```
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.

### 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 I want the first element of the list I 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 langauges (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 last element of a list `a` we use `a[-1]`, to get the 2nd-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*.

### 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:

```python
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:

```python
a[2:]
```

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

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

```python
a[:2]
```

```
[1, 2]
```

Finally, the following just returns the original list:

```python
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:

```python
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, 4, 6]
a.reverse()
a
```

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

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

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

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

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, which 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 that 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 (not tabs) 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 consequences, 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 a deep copy.

However, you should know that 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 nest listed and then change an element inside one of the nested lists, then the copied object will change as well. For example:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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.

## 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:

```
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 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:

```
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 doesn't need to be "legal" Python commands. We can add a comment to describe what a function does like this:

```
# 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:

```
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 at it 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:

```python
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`:

```python
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 check 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 used to approximate 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:

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

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 generate terms, given an initial guess $y_0$, we update $y_n$, with $n = 1, 2, ...$ 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 interaction).

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. 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 way)! 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 need to provide the argument it will use this value, but we can specify a different value.

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.

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 `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 be 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.

# 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:

```
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. Mathematically, 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])

You will notice that even though we provided a list of integers, NumPy converted these to floats. NumPy arrays always work with floats.

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)
```

```
4.0
```

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

```
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])
```

```
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 often 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 $3 \times 3$ 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:

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

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:

```
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 and inversion with `np.linalg.inv()`. You can therefore use Python to help you with the Mathematics course that you are taking alongside this one!