# Python for Decision Analytics and Optimization

Pieter Kleer

# Table of contents

# Chapter 1

# Welcome

In this online book we will learn some elementary Python programming tools, as well as the implementation of (linear) optimization problems in Python.

To install Python we will use the Anaconda distribution, which contains all relevant Python software that we will need. In particular, we will use the Spyder editor, or integrated development environment (IDE), to construct our Python programs.


Figure 1.1: Anaconda

To solve (linear) optimization problems we will use Gurobi, a state-of-the-art software package for solving optimization problems that is used extensively in businesses and academia. We will use this software within Python.


Figure 1.2: Gurobi

In the next chapter we describe how you can install the relevant software, after which we explain some Python basics in Chapter 3. In Chapter 4, we will explain how to implement optimization problems in Python and solve them with the Gurobi solver.

# Chapter 2

# Software

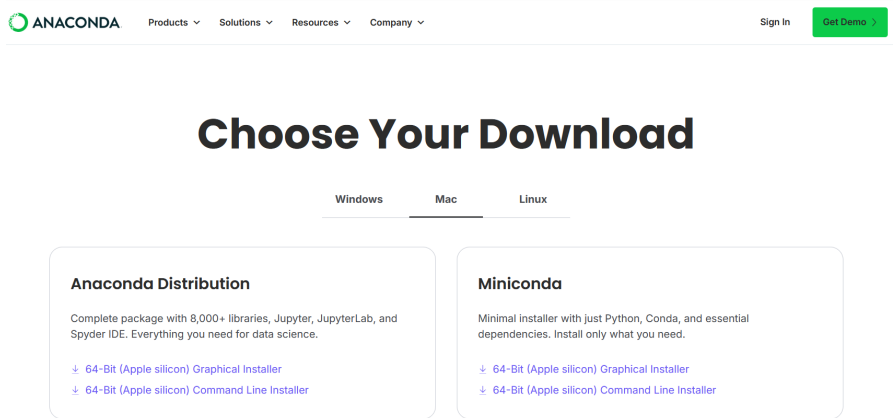
In this chapter we will learn how to install Python and run our very first command. You are of course also free to use your own Python installation. In that case it is best to check the Gurobi documentation to see how to properly install this program.

## 2.1 Anaconda

The easiest way to install Python and the Spyder editor at once is by installing Anaconda, which is a software package that includes Python, Spyder, and various other software applications. In the block below, you can find an outline for installing Anaconda.

**Anaconda installation steps**

1. Go to https://www.anaconda.com/download
2. Click on "Get started" and sign up with an e-mail address of your choice; verify your e-mail address as instructed.
3. Go back to the above Anaconda webpage (either by signing in or via the verification e-mail) and download the *Graphical Installer* installation file (both for MacOS and Windows).




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

After installation, you can see all the newly installed applications using the *Anaconda Navigator*. You can

open the navigator by searching for the Anaconda Navigator in the Start menu (on Windows) or the equivalent on MacOS.

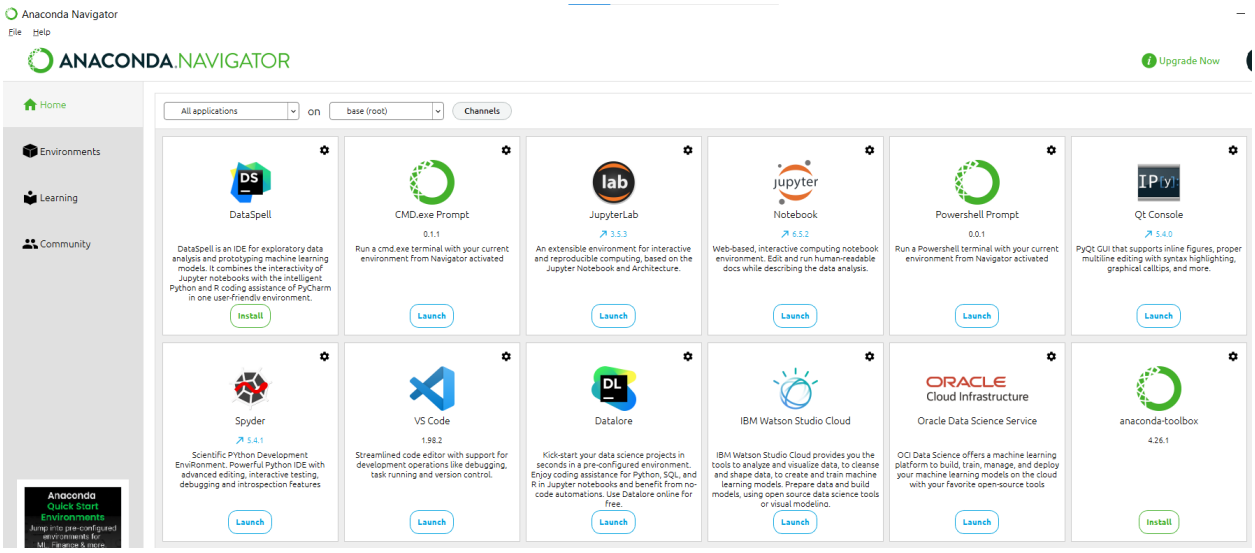



Figure 2.1: Anaconda Navigator

## 2.2 Spyder

To create Python code and scripts, we will be using the Spyder application, which you can open by clicking on *Launch* in the Anaconda Navigator or by searching for the Spyder application on your computer directly. We note that there are many other applications that you can use to create Python code with (such as Visual Studio Code and Jupyter Notebook; two other applications also installed in the Anaconda distribution).

Once you have opened Spyder, you should see an application that looks like this:

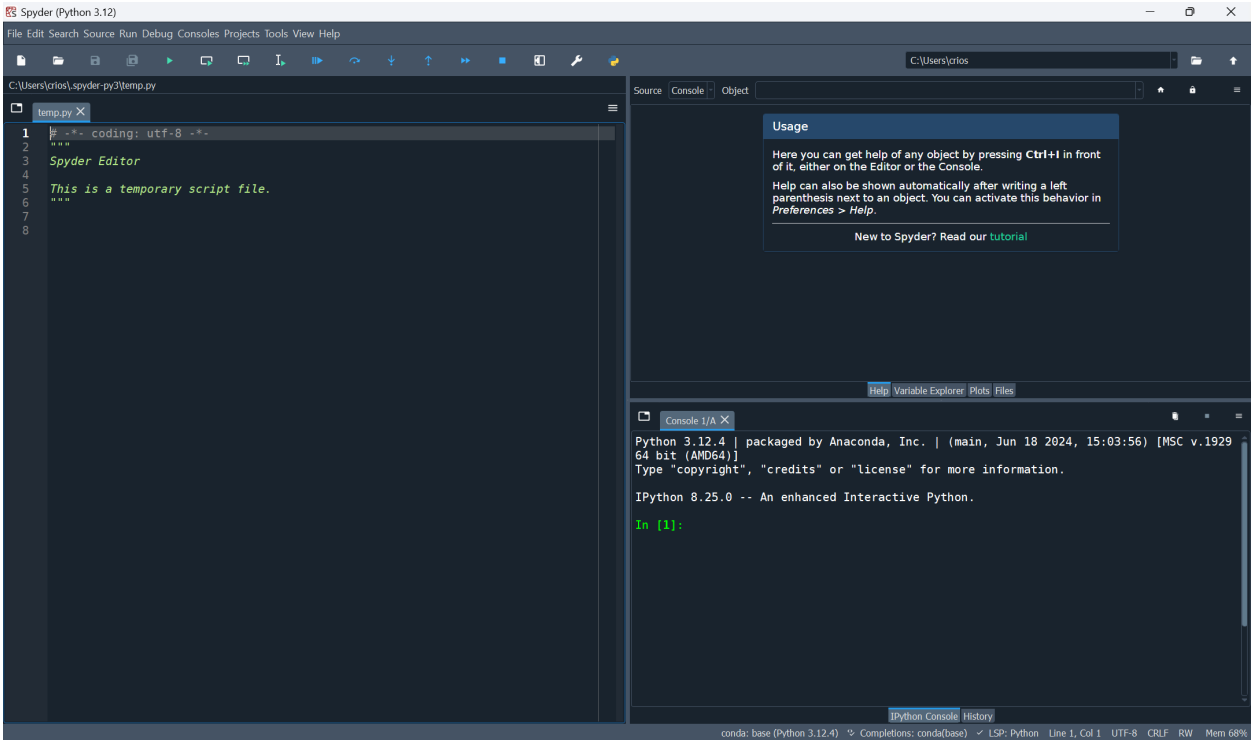



Figure 2.2: Spyder

### 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]`:

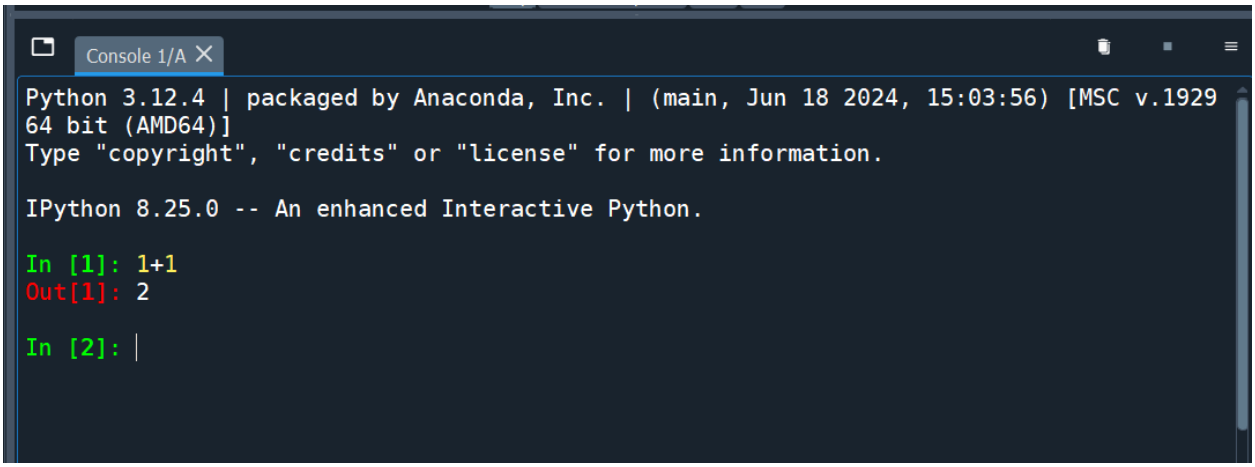



Figure 2.3: IPython Console

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 (like we do at the start of the next chapter). 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 (or only a part of them).

In Spyder, in the left pane you see a file open called `temp.py` when you start up Spyder. This is an example Python script. We can ignore what is written in the first 6 lines of the script. In fact, you can remove these lines if you want. We can add our `1 + 1` command to the bottom of the script like this and save it:

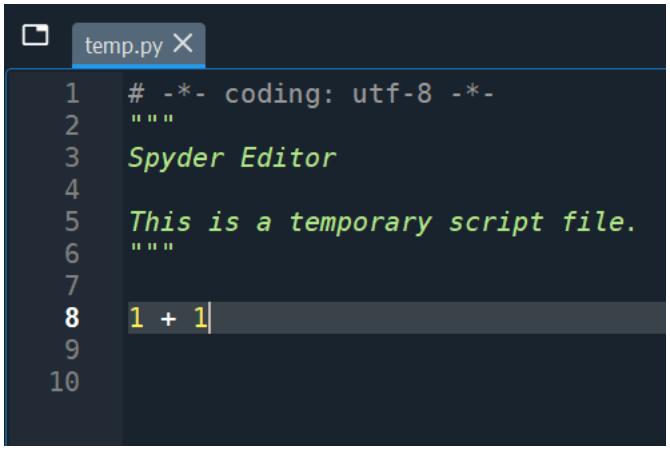



Figure 2.4: Python Script

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:

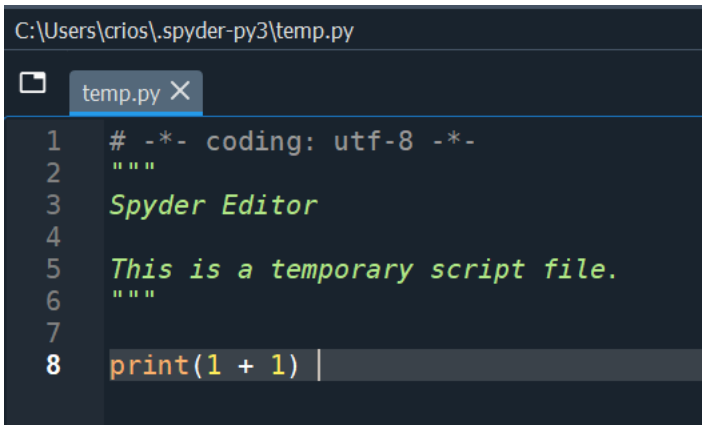



Figure 2.5: Using the print() function

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.

### 2.2.3 Code snippets in this online 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 top part is code and the bottom part in gray is the output of the executed code. There is a small clipboard icon on the right which you can use to copy the code to paste into your own Python file to be able to experiment with it yourself.

## 2.3 Gurobi installation

Based on the above Anaconda installation, we will next explain how to install Gurobi, which is a software package that can solve (linear) optimization problems via Python. We assume in the below that you already have Anaconda/Python installed on your system as in the above steps.

If you get stuck anywhere in the installation process, then you can find more information on Gurobi's Quick Start Guide.

### 2.3.1 Register at Gurobi

Visit the Gurobi website and click on the register button. Open the registration form and make sure that you select the *Academic* account type, and select *Student* for the academic position. Register with your TIAS e-mail account.

### 2.3.2 Download Gurobi

Download Gurobi from the website. Note that you need to login with your Gurobi account before you can download Gurobi. Select the distribution that corresponds to your system (e.g., Windows or macOS),

and select the regular "Gurobi Optimizer", not any of the AMPL variations. Unless mentioned otherwise, download the most recent version.

### 2.3.3 Install Gurobi

Run the installer and follow the installation steps. At some point, the installer may ask whether to add Gurobi to your execution path. This is probably useful to accept.

### 2.3.4 Gurobi license

You cannot use Gurobi without a license, so you need to apply for a license. As a student you can request a free academic license; take the **Named-User Academic** one. After you have obtained the license, you need to activate it for your Gurobi installation. If you open the license details on the Gurobi website, you can see what you need to do: open a Command Prompt (Windows) or Terminal (MacOS) and run the `grbgetkey` command with the code that corresponds to your license. This command will create your license file. Make sure that you remember where the license file is saved. The default location suggested by your device is probably the best choice.

Note that the `grbgetkey` command will check that you are on an academic domain, so you need to perform this step on the **university network** (possibly via a VPN connection). Once installed correctly, you can also run Gurobi without an active VPN connection.

### 2.3.5 Using Gurobi in Python

Gurobi should now be correctly installed, but we also want to be able to use it from Python. Therefore, we need to install Gurobi's Python package. Open the Anaconda prompt application (can also be found in the Anaconda Navigator) and run the following commands one by one (first line + Enter, second line + Enter).

- `conda config --add channels http://conda.anaconda.org/gurobi`
- `conda install gurobi`

> ⚠ Warning
>
> The Anaconda prompt application is NOT the same as the Command Prompt (Windows) or Terminal (MacOS)

If you don't have the Anaconda installation, then you can do something similar with the `pip` command, most likely, for those familiar with it.

### 2.3.6 Test your installation

Now you can test whether everything is setup correctly. Open an interactive Python session, for instance using Spyder. Try the following commands by adding them into a script (make sure there are no spaces at the beginning of the lines) and press Enter.

```
from gurobipy import Model
model = Model()
```

If both these commands succeed, you should see some output related to your academic license.

If the first command fails, then the Gurobi python module has not been installed correctly. If the second command fails, then the license has not been setup correctly (make sure the license file is at the right location).

If at any point, you need more information about Gurobi, then you can always go to the official Gurobi documention online or send the teacher an e-mail.

# Chapter 3

# Python basics

In this chapter we will learn how to use Python as a calculator and see some basic programming concepts.

## 3.1 Arithmetic operations

We start with the most basic arithmetic 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:

```
print(2 + 3)
```

```
5
```

Subtraction:

```
print(5 - 3)
```

```
2
```

Multiplication:

```
print(2 * 3)
```

```
6
```

Division:

```
print(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:

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

We can calculate this in Python as follows:

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

```
0.75
```

With the `**` operator (two stars) we can raise a number to the power of another number. For example, $2^3 = 2 \times 2 \times 2 = 8$ can be computed as

```
print(2 ** 3)
```

```
8
```

> ⚠ WARNING
>
> Do **not** use `^` for exponentiation. This actually does a very different thing in Python.

> Exercise 3.1
>
> Compute the following expressions using the operator $+, -, *, /$ and $**$:
>
> i) $3 + 5 \cdot 2$
>
> ii) $\frac{(10-4)^2}{3}$
>
> iii) $\frac{((2+3)\cdot 4-5)^2}{3+1}$

## 3.2 Variables

In Python we can assign single numbers, as well as text, to *variables* and then work with and manipulate those variables.

Assigning a single number 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 number to the right of the `=`. The `=` operator binds a number (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
print(x + y)
```

```
5
```

When we assign $x = 2$, in our code, the number is not fixed forever. We can assign a new number to `x`. For example, we can assign the number 6 to `x` instead. The sum of $x$ (which is 6) and $y$ ( which is 3), is now 9:

```
# Original variable assignment
x = 2
y = 3
print(x + y)
```

```
# Overwriting assignment of variable x
x = 6
print(x + y)
```

```
5
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 number must be on the right of `=`.

> Exercise 3.2
>
> Define variables $a, b, c$ with numbers $19, 3$ and $7$, respectively. Compute the following expressions:
> i) $a + b \cdot c$
> ii) $\frac{(a-c)^2}{b}$
> iii) $\frac{((b+c)\cdot a - c^2)^2}{a+b}$

## 3.3 Lists

We can also store multiple variables in one object, a so-called *list*. A list with numbers is created by writing down a sequence of numbers, separated by commas, in between two brackets `[` and `]`.

```
z = [3, 9, 1, 7]
print(z)
```

```
[3, 9, 1, 7]
```

We can also create lists with fractional numbers.

```
z = [3.1, 9, 1.9, 7]
print(z)
```

```
[3.1, 9, 1.9, 7]
```

To access the numbers in the list, we can *index* the list at the position of interest. If we want to get the number at position $i$ in the list, we use the syntax `z[i]`.

```
print(z[1])
```

```
9
```

Something strange is happening here… The left-most number in the list is 3.1, but `z[1]` returns 9. This happens because Python actually starts counting at index 0 (instead of 1).

> Indexing convention in Python
>
> The *left-most number* in a Python list is located at *position* 0. The number next to that at position 1, etc. That is, the $i$-th number in a list with $n$ numbers can be found at position $i - 1$ for $i = 1, \dots, n$

In other words, the "first" number in the list is located at position 0, and we can access it using `z[0]` instead.

Below we index the number of the list at positions $i \in \{0, 1, 2, 3\}$ separately.

```
print(z[0])
```

```
3.1
```

```
print(z[1])
```

```
9
```

```
print(z[2])
```

```
1.9
```

```
print(z[3])
```

```
7
```

Exercise 3.3

Consider the list $a = [11, 41, 12, 35, 6, 33, 7]$.

i) Compute the sum of the numbers at even positions in $a$ (i.e., positions $0, 2, 4$, and $6$).

ii) Compute the result of multiplying the first and last elements of $a$, then subtracting the middle element.

iii) Compute the square of the element at position $2$, divided by the sum of the elements at the odd positions.

## 3.4 Printing text

It is also possible to print text in Python with the `print()` command. To do this, you have to put the desired text in quotation marks, either single `''` or double `""`. It does not matter if you use single or double quotation as long as you use the same type on both end of the text.

```
print('Hello world!')
```

```
Hello world!
```

```
print("Hello world!")
```

```
Hello world!
```

A piece of text within quotation marks is called a *string* in Python. It is also possible to store text (strings) in a list, and print it from the list. For example, we can store the days of the weeks as text in the list `days` as follows.

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]
```

If we now want to print a day, we should index the list at the corresponding position. For example, to print `"Wednesday"`, we should index the list at position 2.

```
print(days[2])
```

```
Wednesday
```

If you want to print both text(s) and variables within the same `print()` command, you can do this by separating these quantities with a comma.

```
print("The third day of the week is", days[2])
```

```
The third day of the week is Wednesday
```

Exercise 3.4

Create a list called `courses` that contains five course names as text/strings. Then:

i. Print the first course in the list.
ii. Print the last course in the list.
iii. Print a sentence that says: My favorite color is where should be your favorite course indexed from the list.

## 3.5 For-loop

For-loops are a convenient way to avoid unnecessary repetition of different lines of code. Suppose we want to print all the days of the week from the `days` list.

We can do this by using a print statement for every day separately by indexing `days` at all its positions.

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

print("The days of the week are:")
print(days[0])
print(days[1])
print(days[2])
print(days[3])
print(days[4])
print(days[5])
print(days[6])
```

```
The days of the week are:
Monday
Tuesday
Wednesday
Thursday
```

```
Friday
Saturday
Sunday
```

However, looking at the above, the only thing that changes in the last seven lines of code is the position at which we access `days`. We can also print all the days of the week based on the list `days` above using a for-loop.

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in range(0,7):
    print(days[i])
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

With the line `for i in range(0,7):` we tell Python that we would like to carry out a piece of code (the line below it) with different values of the *iteration variable* `i` ranging from 0 to 7, but NOT including 7. In other words, we want to run a piece of code for the values of `i` being $0, 1, 2, 3, 4, 5$ and 6. It is important to realize here that the index 7 is not included as value for `i` in the for-loop (this is what the Python developers decided on).

In Python such a range, called the *iterable*, can be specified with the `range(a,b)` function where $a$ is the smallest value and $b-1$ the largest value of $i$. In the first *iteration* with $i = 0$ Python will now print `days[0]`, in the second iteration with $i = 1$ Python will print `days[1]`, etcetera.

The code we want to execute repeatedly can be found under the lines `for i in range(0,7):` with an *indentation (or tab)*. The indendation is important so that Python known this is the line of code you want to be carried out repeatedly. We can also carry out multiple lines of code in a for-loop, then all these lines should be indented.

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in range(0,7):
    print("Day", i+1, "of the week is:")
    print(days[i])
```

```
Day 1 of the week is:
Monday
Day 2 of the week is:
Tuesday
Day 3 of the week is:
Wednesday
Day 4 of the week is:
```

```
Thursday
Day 5 of the week is:
Friday
Day 6 of the week is:
Saturday
Day 7 of the week is:
Sunday
```

Note that in the first line in the for-loop, here we print three aspects in one `print()` statement: The word `Day`, the value of `i+1`, and the words `of the week is:`. As $i$ ranges from 0 to 6, the values of $i+1$ ranges from 1 to 7.

We note that we only use two print statements for the sake of illustration. The above could have been printed with one `print()` statement as well.

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in range(0,7):
    print("Day", i+1, "of the week is:", days[i])
```

```
Day 1 of the week is: Monday
Day 2 of the week is: Tuesday
Day 3 of the week is: Wednesday
Day 4 of the week is: Thursday
Day 5 of the week is: Friday
Day 6 of the week is: Saturday
Day 7 of the week is: Sunday
```

Exercise 3.5

Take your list `courses` from the previous exercise and consider the list `days` from above. Print the five sentences: "On Monday the course [course0] is being taught", "On Tuesday the course [course1] is being taught", etc. by using a for-loop in which you index the courses and days from the two lists, and where [course0], [course1], etc. are the first, second, etc. course names from the list.

We remark that it is also possible to directly iterate over the values in a list (the iterable).

```
days = ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]

for i in days:
    print(i)
```

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

```
Sunday
```

In this case, $i$ does not range from 0 to 6, but in iterates directly over the strings in the list `days`. So in the first iteration $i =$ "Monday", in the second iteration $i =$ "Tuesday", etcetera.

## 3.6 Conditional statements

It is also possible in Python have different lines of code executed based on conditions. For this we use an **if/else statement**. This allows the program to execute certain lines of code only when a condition is true.

For example, suppose we store a number in the variable x and want to print a message only if x is greater than 5.

```
x = 8

if x > 5:
    print("x is greater than 5")
```

```
x is greater than 5
```

The line if `x > 5:` checks whether the condition x > 5 is true. If it is true, Python executes the indented line below it. If the condition is false, nothing happens and Python continues with the rest of the code. You can check this yourself by changing the value of $x$ to something strictly smaller than 5. If you run the code snippet, nothing should be printed.

We can also tell Python what to do if the condition is false by adding an **else statement**.

```
x = 3

if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

```
x is not greater than 5
```

Here Python checks the condition. Since 3 > 5 is false, i.e., the condition $x > 5$ is not true, the code under `else:` is executed instead.

Just like with for-loops, **indentation is important**. All lines that belong to the if or else block must be indented so Python knows which code belongs to which condition.

We can also check more than two conditions using `elif` (short for "else if"):

```
x = 5

if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is exactly 5")
```

```
    print("You can execute multiple lines of code when a condition is true")
else:
    print("x is less than 5")
```

```
x is exactly 5
You can execute multiple lines of code when a condition is true
```

In this case Python checks the conditions from top to bottom and executes only the first one that is true. In the snippet above the line `"x is exactly 5"` will be printed. You can play around with this code snippet by changing the values of $x$ and observing that, depending on its value, a different line will be printed.

You can use the following symbols to compare a variable with a given number.

| Operator | Meaning | Example ( `x = 5` ) | Result |
|---|---|---|---|
| `<` | Less than | `x < 7` | `True` |
| `>` | Greater than | `x > 7` | `False` |
| `<=` | Less than or equal to | `x <= 5` | `True` |
| `>=` | Greater than or equal to | `x >= 6` | `False` |
| `==` | Equal to | `x == 5` | `True` |
| `!=` | Not equal to | `x != 5` | `False` |

You can also combine multiple statements into one condition using the `and` and `or` *keyword arguments*. In the code below, we do not finish with an `else` statement; this is not mandatory.

```
x = 5

if x > 5 and x < 10:
    print("x is between 5 and 10")
elif x == 5 or x == -5:
    print("The absolute value of x is 5")
```

```
The absolute value of x is 5
```

You can also combine for-loops and if/else statements. For example, recall the original if/else statement where we checked whether $x$ was greater or equal than $5$.

```
x = 3

if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

```
x is not greater than 5
```

We can also check this for a collection of numbers stored in a list `numbers` by iterating with a for-loop over these numbers and check the conditions for every element in the list.

```
numbers = [3, 7, -3, 4, 10]

for x in numbers:
    if x > 5:
        print("x is greater than 5")
    else:
        print("x is not greater than 5")
```

```
x is not greater than 5
x is greater than 5
x is not greater than 5
x is not greater than 5
x is greater than 5
```

Note that in the above code we use indentation twice! For every iteration of the for-loop we want to check the if/else statement, so all those lines need to be indented. Because it also needs to be clear, for every iteration, which lines correspond to the if-statement, and which to the else-statement, the `print()` statements are indented once more.

Exercise 3.6

Consider the list of temparatures (in degrees Celsius) $[20, 14, 12, 18, 25, 30, 31]$ corresponding to the days of the week Monday through Sunday. For every day you have to print the message "[day] is a [cold/normal/hot] summer day", where a day is cold if the temperature is below 15, normal if it is in the interval $[15, 25]$ (i.e., greater or equal than 15 and smaller or equal than 25, and hot if it is higher than 25.

The output of your code should be:

```
Monday is a normal summer day.
Tuesday is a cold day.
Wednesday is a cold day.
Thursday is a normal summer day.
Friday is a normal summer day.
Saturday is a hot summer day.
Sunday is a hot summer day.
```