Python for Econometrics and Operations Research

Sander Gribling   Pieter Kleer   Johan van Leeuwaarden   Sven Polak

Table of contents

Chapter 1

Welcome

Welcome to the online "book" that serves as an introduction to the programming language Python, that you will see in various courses throughout the Econometrics and Operations Research (EOR) bachelor program.

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.

1.1 What is a programming language?

Without getting into complicated details, a programming language is a way to communicate to a computer, via written text, tasks or operations that you want it to carry out. 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.

In the EOR bachelor program, the goal is often to tell a computer to carry out complicated numerical computations or to visualize numerical data. To some extent, you have already done this in high school using a graphing calculator. In fact, everything that your graphing calculator can do, you can also do with Python, but the advantage of Python is that it can also handle much more complicated tasks.

To use Python in a correct fashion, it is important that you understand the grammar, i.e., "syntax", of the Python programming language. 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", i.e, grammar mistake 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 a (syntax) error but does something you didn't want it to do. Therefore we need to be very careful when writing in a programming language.

1.2 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 mathematical data? While many languages could perform all of these tasks, some languages excel in some of them. Python is by far the most popular programming language when it comes to "data science" tasks, that you will often encounter in the EOR bachelor program. 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, "large-language" models like ChatGPT have become a very useful resource for Python. ChatGPT can write excellent Python code and also explains all the steps it takes, so we encourage you to use it as a tool to help you when you are stuck. You should keep in mind though that throughout the bachelor program, you will not always be allowed to use tools like ChatGPT.

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

Installation

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:

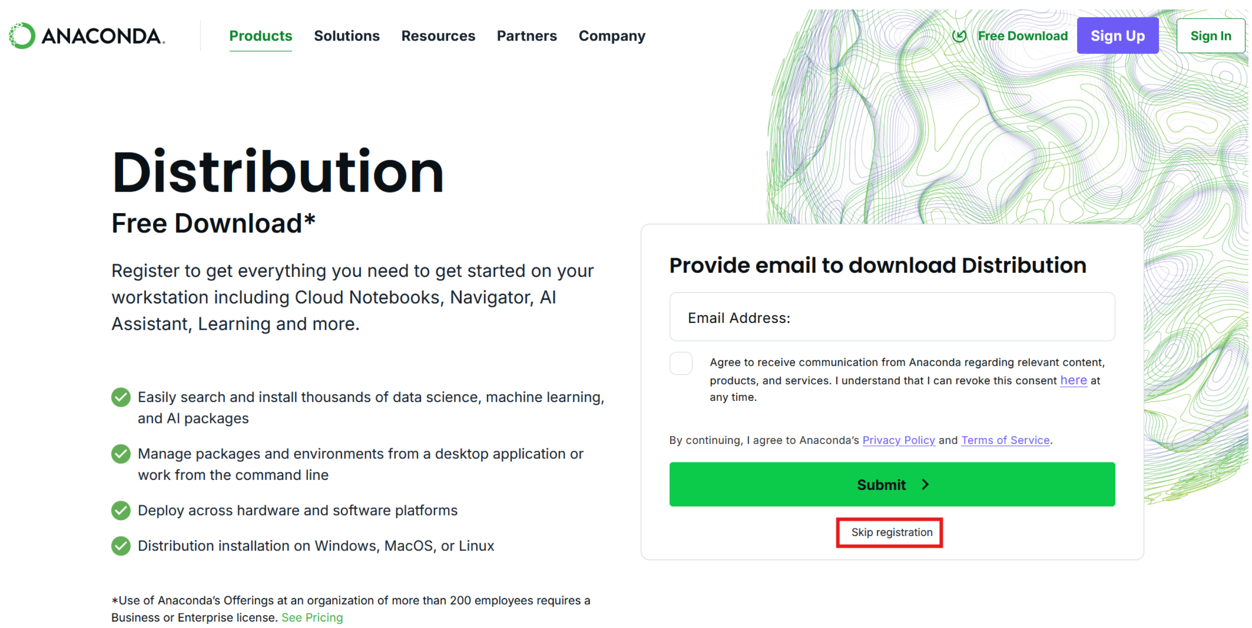


Figure 2.1: Anaconda Download Page

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

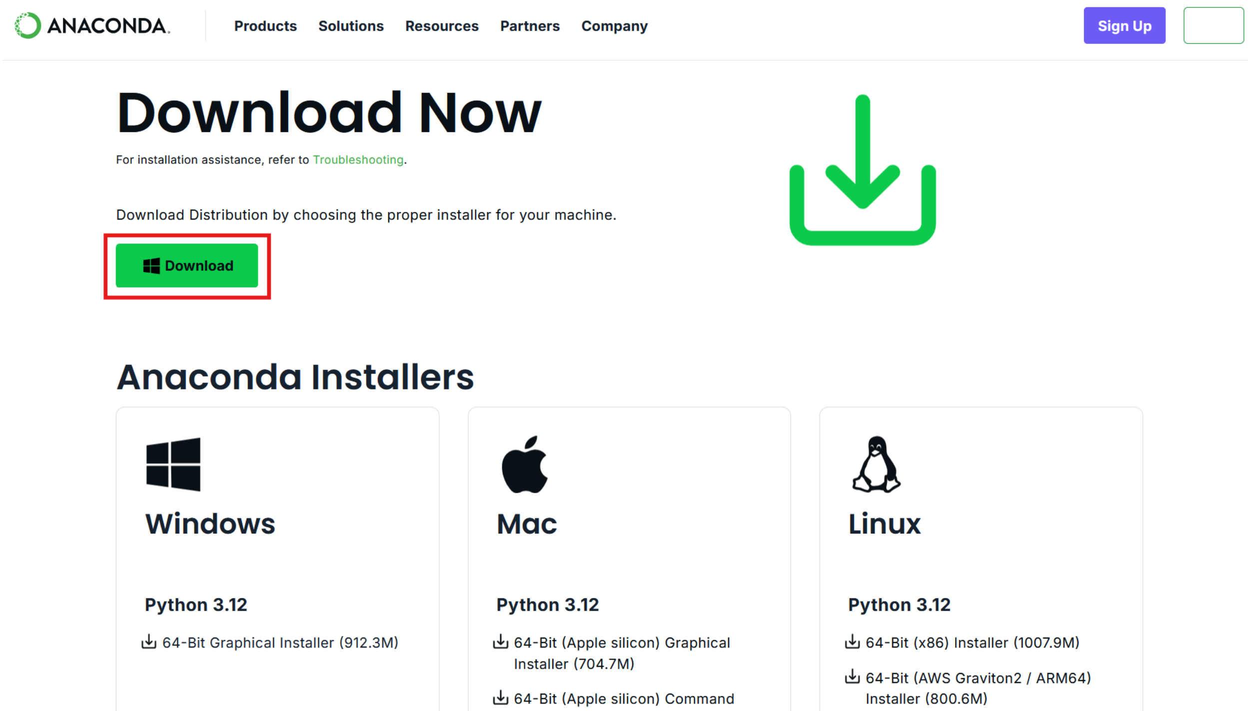


Figure 2.2: Anaconda Download Page

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. You can see all applications that were installed using *Anaconda Navigator*.

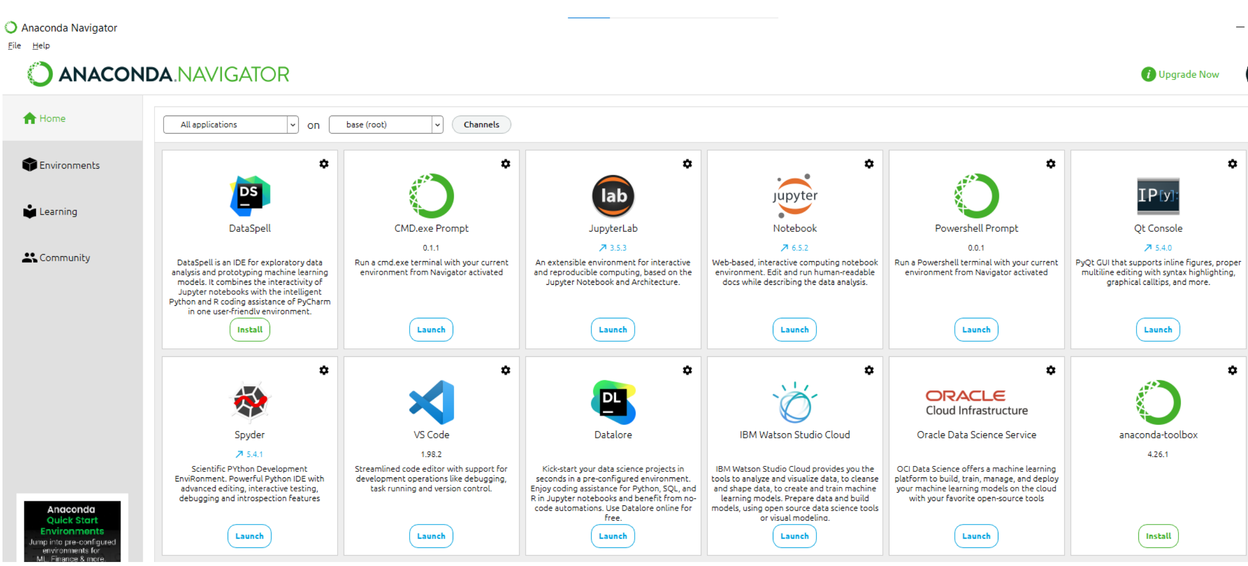


Figure 2.3: Anaconda Navigator

We highlight two applications:

- *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 on.
- *Spyder/VS Code.* These are computer applications that allows you to write Python scripts and execute them to see the output. Such an application is called an Integrated Desktop Environment (IDE).

2.2 Jupyter Notebook

You can open the Jupyter Notebook application either by pressing 'Launch' in the Anaconda Navigator, or you can search for the application directly on your (university) computer via the Start menu.

The application will open as a tab in a web browser, and you should then see a list of folders.

Figure 2.4: Jupyter Notebook application

You can navigate to a folder and then create a new notebook by clicking on 'New' in the top-right and then selecting 'Python 3 (ipykernel)' under Notebook.

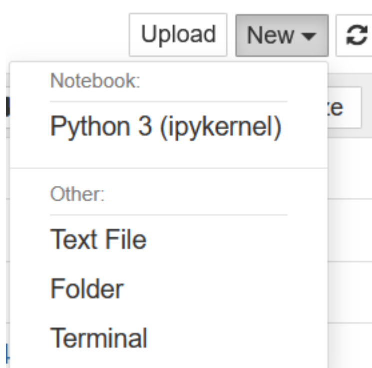


Figure 2.5: Creating notebook

The new notebook will open in another tab and is stored in the folder in which you created it, typically under the name 'Untitled.ipynb'. You can change the name of the file either in the folder in which you stored it, or via `File -> Rename` in the top-left corner. You should see the empty notebook as below.

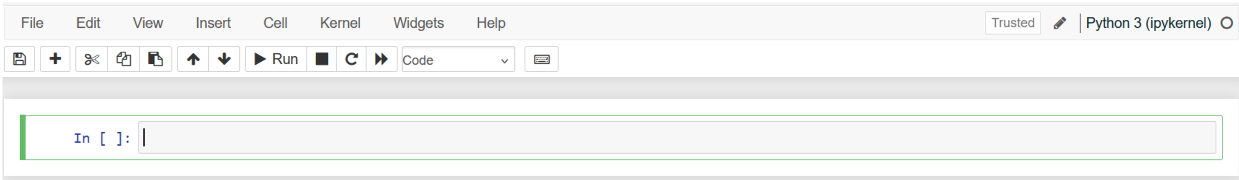


Figure 2.6: New notebook

In the bar you can type Python code. Let us execute our first code, which is a simple calculation $1 + 1$. 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. Type 1+1 in the code bar and click *Run* (or hit Shift + Enter on your keyboard). We will see the output 2 on the next line next to a red Out [1]:

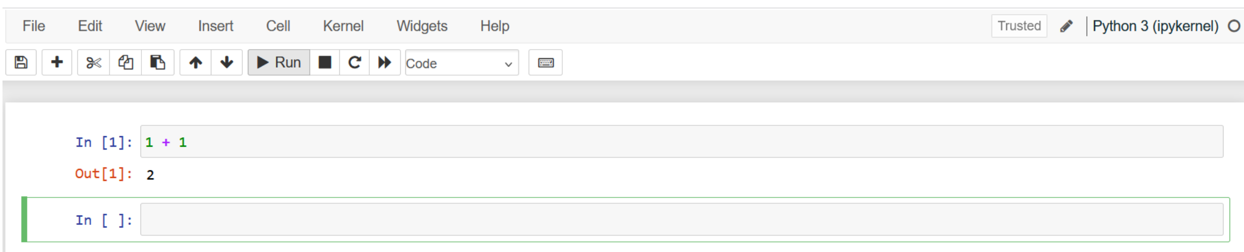


Figure 2.7: First Python code

The red Out [1] means this is the output from the code snippet In [1] executed in this notebook. If we continue typing code in the second bar, it will be called In [2] and its output Out [2]. However, the same happens if we would re-run the first code snippet with `1+1`: It will also be called In [2] and its output Out [2]. The index keeps track of how many code snippets have been executed in the notebook.

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 own Notebook 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 basics

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 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:

```
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:

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

We can calculate this in Python as follows:

```
(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

```
2 ** 3
```

```
8
```

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

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 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
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:

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

```
9
```

Finally, you cannot set $x = 2$ with the command `2 = x`. That will result in an error. The name must be on the left of `=` and the 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) $\frac{((b+c) \cdot a - c^2)^2}{a+b}$

If you want to print multiple expressions within the same code snippet, you can use the `print()` function of Python for each of the expressions.

```
x = 2
y = 3
print(x + y)
print(x - y)
```

```
5
-1
```

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]
z
```

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

We can also create lists with fractional numbers.

```
z = [3.1, 9, 1.9, 7]
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]`.

```
z[1]
```

```
9
```

Something strange is happening here… The first number in the list is 3.1, but `z[1]` returns 9, which is the second number in the list.

The former happens because Python starts counting at index 0 (intead of 1). In other words, if we want to get the first number in the list, the number at position 0, we should use `z[0]` instead.

Below we index all numbers (with indices $0, 1, 2$ and 3) separately.

```
z[0]
```

3.1

```
z[1]
```

9

```
z[2]
```

1.9

```
z[3]
```

7

> **Exercise 3.3**
>
> Consider the list $a = [1, 4, 2, 5, 6, 3, 7]$.
>
> 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 uneven positions.

3.4 For-loop

Suppose we want to compute the sum of the numbers in the list a of Exercise 3.3. We could do this manually by indexing every number and adding them one by one.

```
a = [1, 4, 2, 5, 6, 3, 7]

a[0] + a[1] + a[2] + a[3] + a[4] + a[5] + a[6]
```

28

In fact, we can also define a new variable to store this number in. Let us call this variable `total_sum`.

```
a = [1, 4, 2, 5, 6, 3, 7]

total_sum = a[0] + a[1] + a[2] + a[3] + a[4] + a[5] + a[6]
total_sum
```

28

If the list is a is very long, for example containing thousands of elements, then it becomes very tedious to compute the total sum with the approach above. Such long lists are not uncommon in real-life data.

A much better way is to use a *for-loop*, which lets us go through each element in the list one at a time. Here's how we could compute the sum of the numbers in the list a using a for-loop:

```
a = [1, 4, 2, 5, 6, 3, 7]
total_sum = 0

for i in [0,1,2,3,4,5,6]:
    total_sum = total_sum + a[i]

print(total_sum)
```

```
28
```

Let's break down what is happening here:

- We first define the list $a = [1, 4, 2, 5, 6, 3, 7]$. The goal is to add up the numbers in this list.
- We define the variable `total_sum` with initial value 0. This variable will be the "running total" of the numbers in a that we are adding up. After the full code has been executed, this variable will contain the sum of all numbers in a, and its value is printed at the end.
- In the for-loop, we telll Python to repeatedly carry out the indented line of code (`total_sum = total_sum + a[i]`) for the index values of i in the list $[0, 1, 2, 3, 4, 5, 6]$, in the order given as in the list. This list contains the indices of all the positions of the list a, which are not to be confused with the numbers $a[i]$ at those positions.
- In the indented line of code we *overwrite* the value of `total_sum` with its current value plus the number at position i in a, i.e., the number `a[i]` . In the table below this process is illustrated for all the values of i.

`i`	`a[i]`	`total_sum` after this iteration
0	1	0 + 1 = 1
1	4	1 + 4 = 5
2	2	5 + 2 = 7
3	5	7 + 5 = 12
4	6	12 + 6 = 18
5	3	18 + 3 = 21
6	7	21 + 7 = 28

In the first *iteration* of the for-loop ($i = 0$), we have the initial number 0 for `total_sum` so adding `a[0]` results in a new value of `total_sum` being $0 + 1 = 1$.

In the second iteration, with now `total_sum` equal to 1, we add the number $a[1]$, which results in the new value of `total_sum` being 1 (current number of `total_sum`) plus `a[1]` (which is 4), resulting in a new running total of $1 + 4 = 5$.

If we would be interested in only computing, e.g., the sum of the first three numbers in a, we could replace the index list $[0, 1, 2, 3, 4, 5, 6]$ by $[0, 1, 2]$.

```
a = [1, 4, 2, 5, 6, 3, 7]
total_sum = 0

for i in [0,1,2]:
    total_sum = total_sum + a[i]

total_sum
```

```
7
```

Exercise 3.4

Create the list $a = [1, 4, 2, 5, 6, 3, 7]$.

i) Compute the sum of the numbers at the even indices using a for-loop.
ii) Compute the product of the numbers in a using a for-loop.

If you want to execute more lines of code in every iteration of the for-loop, you should indent all of them. In the code below we compute the running total `total_sum` and also use the `print()` command of Python to print the value of the running total after every addition. This results in all the values in the right column of the above table being printed.

```
a = [1, 4, 2, 5, 6, 3, 7]
total_sum = 0

for i in [0,1,2,3,4,5,6]:
    total_sum = total_sum + a[i]
    print(total_sum)
```

```
1
5
7
12
18
21
28
```

3.5 Conditional statements

In many programming situations, we want the computer to make decisions based on certain conditions. For example, if a number is negative, we might want to handle it differently than if it were positive. In Python, we can do this using *conditional statements*, also known as *if/else statements*.

Let's look at a basic example:

```
x = 5

if x > 0:
```

```
    print("x is positive")
else:
    print("x is not positive")
```

```
x is positive
```

Here is what this code does:

- `x = 5` assigns the number 5 to the variable x.
- `if x > 0:` checks whether x is greater than zero, i.e, Python checks whether the condition `x > 0` is true or false. If the condition is true, it executes the indented block underneath: In this case we use the `print()` command of Python to print a piece of text (which can be done by putting it in quotations).
- If the condition is not true (i.e., $x \leq 0$), then Python executes the indented code under `else:` .

We can also have multiple conditions using `elif` , which stands for "else if":

```
x = 0

if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

```
x is zero
```

This checks the conditions one by one from top to bottom and executes the first indented code block where the condition is true. If you want more than three conditions, you should start with an `if` statement, then `elif` statements, and finish with an `else` .

Finally, if you want to execute multiple lines of code for one or more of the conditions, you should indent all those lines under the respective conditions.

```
x = 0

if x > 0:
    print("x is positive")
    print(x)
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
    print(x)
```

```
x is zero
```

💡 Exercise 3.5

Create the list $a = [1, 4, -4, 0, 5, -3, -7]$ in Python.
Use a for-loop in combination with the code above to check for every number in a whether it is positive, zero, or negative. If a number is positive you should print the message `"The number is positive"`, if it is zero `"The number is zero"` and if it is negative `"The number is negative"`.
The output of your piece of code should be as follows.

```
The number is positive
The number is positive
The number is negative
The number is zero
The number is positive
The number is negative
The number is negative
```

Let us now look at an example from mathematics. Suppose we want to compute the roots x of a quadratic equation of the form:

$$ax^2 + bx + c = 0$$

Here a, b and c are known given numbers, and the goal is to find one or more x's that satisfy the above equation.

From high school math (e.g., "Wiskunde B" if you went to a Dutch high school), we know that the solutions are given by the *abc-formula*:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

and that the expression under the square root, called the *discriminant* $D = b^2 - 4ac$, determines how many (real) roots exist:

- If $D > 0$, the equation has two real roots.
- If $D = 0$, the equation has exactly one real root.
- If $D < 0$, there are no real roots (in the real number system).

💡 Exercise 3.6

Create variables $a = 3, b = 2$ and $c = -1$. Create a variable `D` for the discriminant (in terms of a, b and c).

i) Use conditional statements to determine how many roots the quadratic formula $ax^2 + bx + c$ has, based on the three possibilities for the discriminant. For each possibility, print an appropriate message in the indented code block. For the chosen a, b and c, the function has two roots (so this case should be printed in your code).
ii) Use conditional statements to print the roots x of the quadratic formula $ax^2 + bx + c$, based on the three possibilities for the discriminant (in the third case, do not print the roots, but a message saying there are no roots). *Hint: If you want to print two variables `y` and `z` you can use `print(x,y)` or use `print(x)` and `print(y)` on different indented lines.* For the chosen a, b and c, your output should show the roots -0.9106 and 0.244 (possibly with more or less decimals).

You can play around with your code by choosing different numbers for a, b and c, and see if you get different output cases for both questions above.

Chapter 4

Math basics

In this chapter we will see some of the basic math functionality that Python has to offer. Many of these tasks can be carried out by your graphing calculator as well, but Python can also handle much more difficult problems that you will see in the course of you academic career.

We start with the basics of defining a function, such as a quadratic formula.

4.1 Python function

If we want to compute a certain mathematical expression for many different variables, it is often convenient to use a Python function for this.

For example, consider the quadratic function $f(x) = x^2 + 2x^2 - 1$. Say we want to know the values of $f(-3)$, $f(-2.5)$, $f(1)$ and $f(4)$. What we would like to do is to 'automate' the computation of a function value, so that we do not have to write out the whole function everytime.

For this we can use a Python function for this as follows.

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

What does the code above do? First of all the syntax to tell Python we want to define a function called `f` that takes as input a number `x` is `def f(x):`.

We next have to tell Python what the function is supposed to compute. On the second line, with one tab indented, we have the `return` statement. Here we write down the expression that the function should return (or compute), which in our case is the function value $f(x) = x^2 + 2x^2 - 1$.

We can now compute the function value $f(x)$ for any value of x. What happens is that Python *calls* the function f with input the chosen value of x, and then returns the function value $f(x)$, i.e., the expression in the `return` statement.

```
f(-2)
```

```
-1
```

```
f(1)
```

```
2
```

Note that you can also name the function differently, for example we could also have done `def quadratic_function(x):`. You should then use the name `quadratic_function` too in the command well you call the Python function to compute the function value $f(x)$.

```
def quadratic_function(x):
    return x**2 + 2*x - 1

quadratic_function(1)
```

```
2
```

Just as your graphing calculator we can plot a Python function, search for its roots, integrate a certain area under the curve and much more! More advanced tasks that Python can handle will introduced in later courses in the EOR bachelor program.

If you want to get a better understanding of the codes in the coming sections, you could already have a look at Chapter 9 of this course document of another course taught at the Tilburg School of Economics and Management. We do not explain the code here, but give it as a teaser what more is possible with Python!

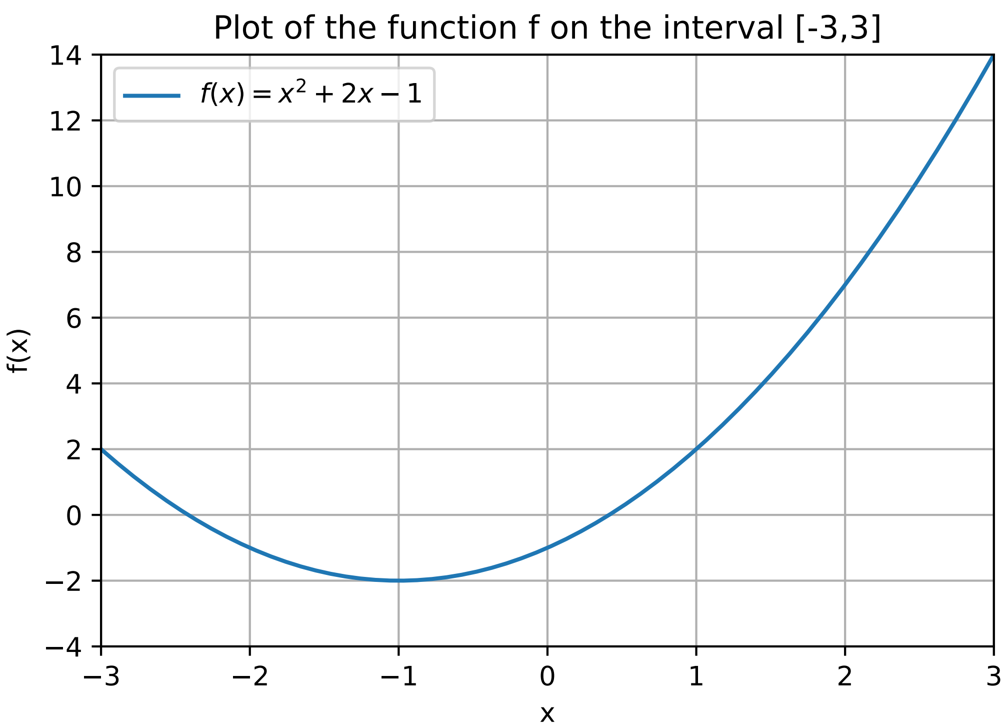
4.2 Plotting

Consider again the function $f(x) = x^2 + 2x - 1$. A visualization of this function is given below. If you want to plot a function in Python you have to make use of functionality from NumPy and Matplotlib which are so-called Python packages.

Packages are functions written by other people to make our live easy, i.e., so that we do not have to write every code file from scratch in Python.

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

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

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

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

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

```
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

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

# Add a legend
plt.legend()

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



4.3 Root finding

Similarly, the SciPy package can be used to carry out various mathematical tasks and algorithms, making it very important for data analysis purposes.

The code below uses a pre-written Python function called `fsolve()` from SciPy to compute the roots of a function f. In other words, `fsolve()` is a mathematical algorithm for finding the root of a function, such as Newton's method, that someone implemented in Python and made available publicly for the whole world to use. If you are interested in the *source code* of this function, you can look it up in the *documentation* of

Python (more specifically, SciPy in this case).

```
import scipy.optimize as optimize

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

guess = 3
f_zero = optimize.fsolve(f,guess)

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

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

The function `fsolve()` takes two inputs: a function of which we want to find a root, and an initial guess (3 in our case) of where the root is.

Entering an initial guess for where the root is located, is in some sense the equivalent of giving a bracket in which the root should lie on your graphing calculator. In fact, there are other root finding functions available in Python that work in this way, i.e., that require you to give an initial bracket, just as you do on your graphing calculator.

Different initial guesses might lead to different roots found by Python. In fact, as you can see the function f has two roots of which the above code finds the right one. We could find the left root by filling in a different initial guess, e.g., -3 instead of 3.

```
guess = -3
f_zero = optimize.fsolve(f,guess)

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

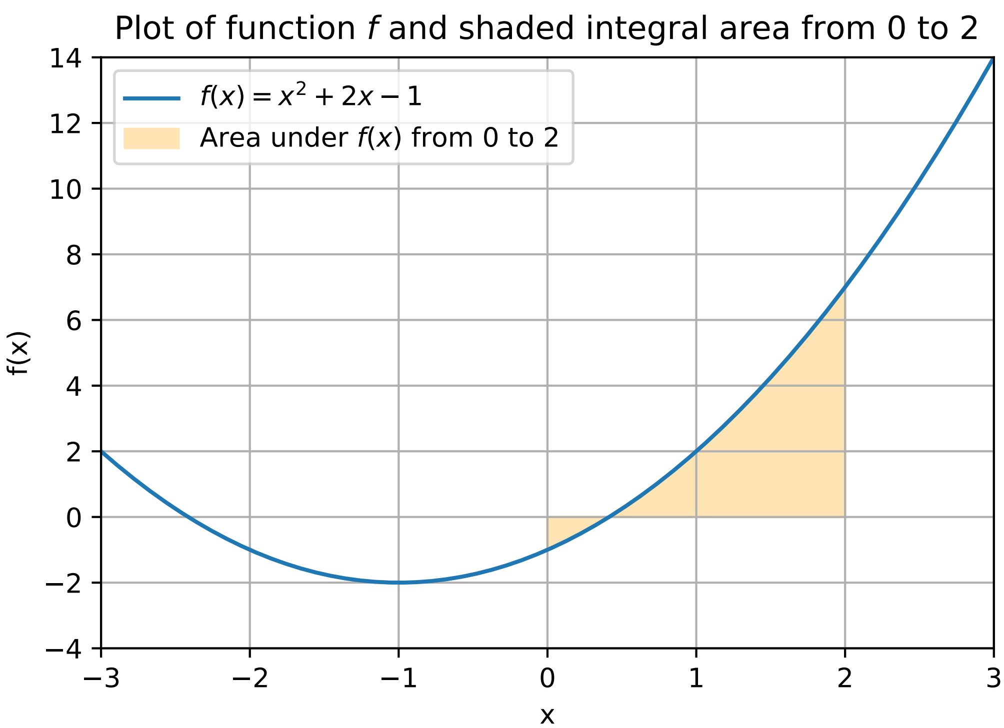
```
A root of the function f is given by [-2.41421356]
```

4.4 Integration

Finally, it is also possible to use built-in functionality from SciPy to integrate a function. Below we integrate the function f from 0 to 2. This integral area is illustrated in the figure below.

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

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

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

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

# Define the interval for shading (0 to 2)
x_fill = np.linspace(0, 2, 300)
plt.fill_between(x_fill, f(x_fill), alpha=0.3, color='orange',
    label='Area under $f(x)$ from 0 to 2')

# Add labels and title
plt.title('Plot of function $f$ and shaded integral area from 0 to 2')
plt.xlabel('x')
plt.ylabel('f(x)')

# Add a grid
plt.grid(True)

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

# Add a legend
plt.legend()

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




```
from scipy.integrate import quad

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

# Perform the integration from 0 to 2
result, error = quad(f, 0, 2)

# Print the result
print("Integral of f(x) from 0 to 2 is:", result)
print("Numerical error in integral computation is at most", error)
```

```
Integral of f(x) from 0 to 2 is: 4.666666666666666
Numerical error in integral computation is at most 5.666271351443603e-14
```

4.5 Why Python and not my calculator?

So far we have illustrated task with Python that you graphing calculator can also carry out. The advantage of Python is that it can handle much more complicated computing tasks and handle much more difficult mathematical functions, that your graphing calculator is not able to handle.

Many of these tasks you will come across in various courses of the EOR bachelor program, already starting with the course Linear Optimization in the second quartile of year 1.

Furthermore, throught the EOR bachelor program you will also see some other programming languages such as R and Matlab. Many of the general programming ideas, such as for-loops and conditional statements, exist in those languages as well, but sometimes the syntax (i.e., the grammar of the programming language) is different than that of Python.