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.

The first part of this book is used for the "Python crash course" announced during the course Linear Algebra. The crash course consists of two (recommended) lectures that will form a useful basis for various programming assignments in the EOR bachelor program, including the assignment of the (mandatory) course Linear Optimization in the second quartile of the first year.

The two crash course lectures (taught only in English) are outlined below, including links to the lecture materials. To program in Python, we will use the Jupyter Notebook application in which we can execute Python code.

Overview

Lecture 1 (September 9, 12.45–14.30, Cube 242)
General introduction to Jupyter Notebook and *Chapter 3 - Python basics*.
Chapter 4 is not part of the crash course, but you can have a look at it if you are interested.
Right-click and use "Save link as…" for the materials below:
Slides in PDF form.
Exercises of Chapter 3 (see Section 2.2.2 on how to open this file in Jupyter Notebook after storing it on your computer).
Solutions (also Jupyter Notebook file).
Lecture 2 (September 23): Linear algebra with Python (September 23, 14.45-16.40, Cube 241)
Centralized introduction and explanations with time to work on some exercises.

- General introduction to packages; introducing Numpy and Sympy, *Chapter 5 - Linear Algebra*.
- Downloadable materials (right-click and use "Save link as…")
  - Slides in PDF form.
  - Exercises of Chapter 5 (see Section 2.2.2 on how to open this file in Jupyter Notebook after storing it on your computer).

> **Software requirements**
>
> We will use the Jupyter Notebook application during our lectures, which is available on the university computers in the computer room where the lectures take place.
> If you want to use your private laptop, you can install Python and Jupyter Notebook by following the instructions in Chapter 2. Please do this **before** the first lecture.

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.

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. It is also important that you understand basic programming concepts to catch errors that AI might introduce (do not forget that LLMs are merely predicting text and not “consciously” writing a script), or to help improve your AI-prompt writing skills.

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

Software

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 and Jupyter Notebook is by installing Anaconda, which is a software package that includes Python, Jupyter Notebook, and other software applications. You can install Anaconda by visiting https://www.anaconda.com/download.

You should see this page:

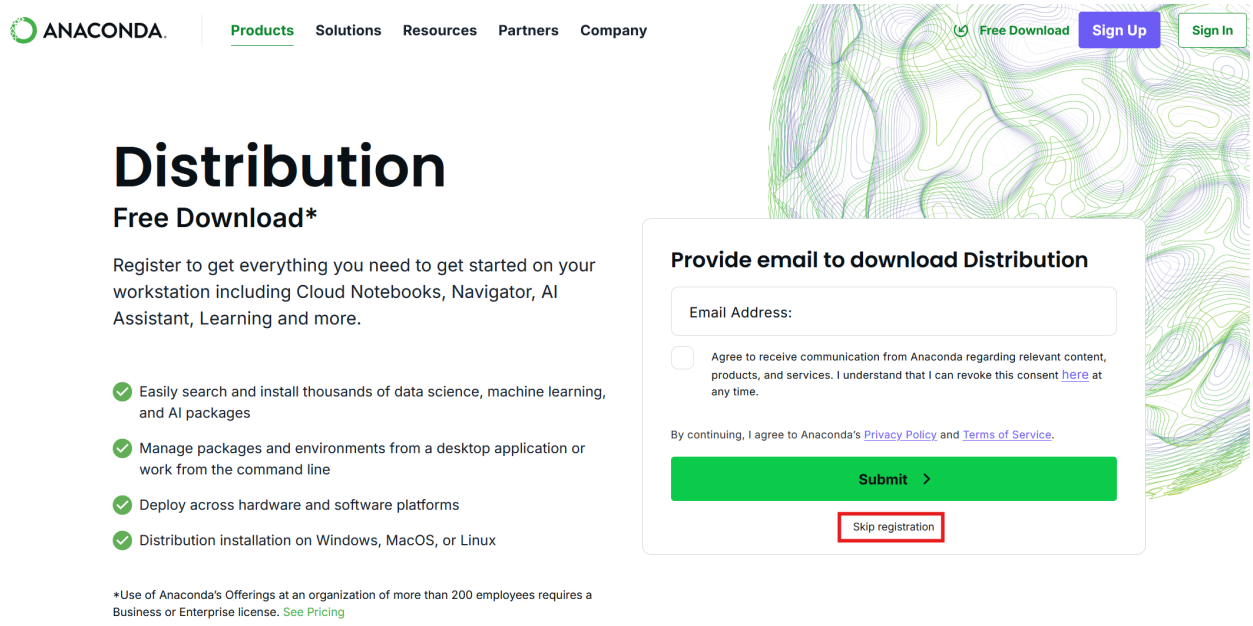


Figure 2.1: Anaconda Download Page

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

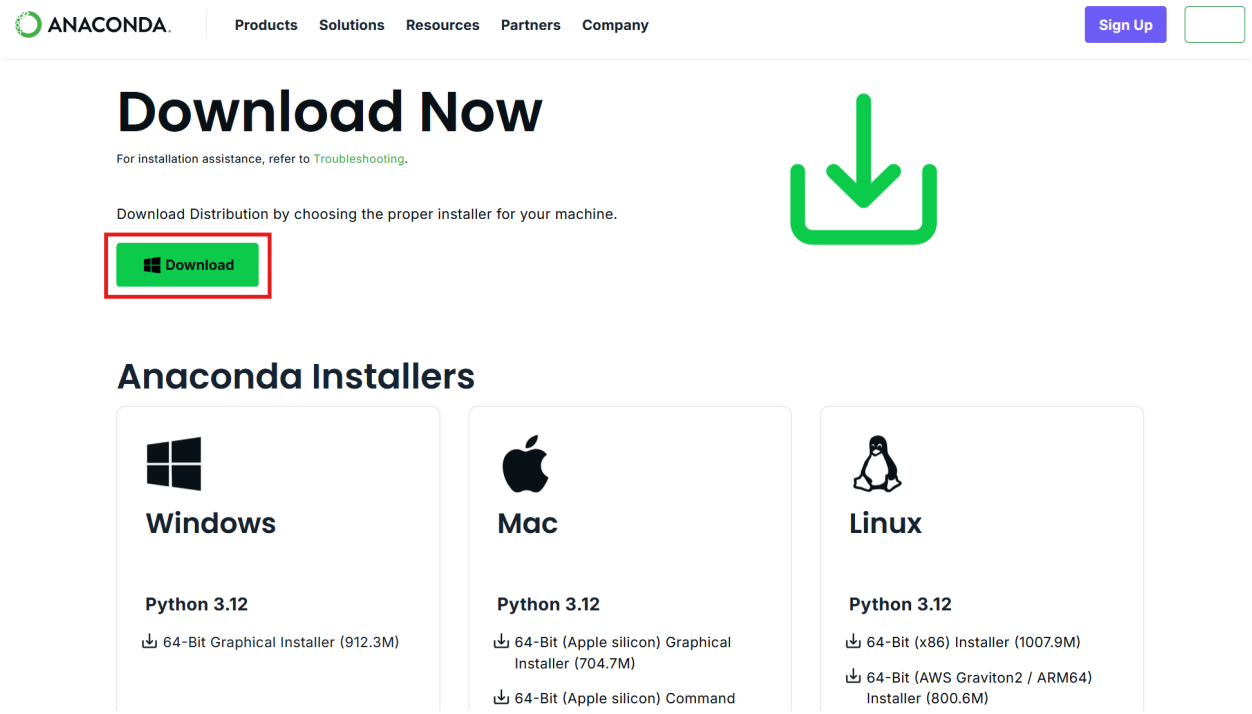


Figure 2.2: Anaconda Download Page

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*. You can open the navigator by searching for it in the Start menu (on Windows).

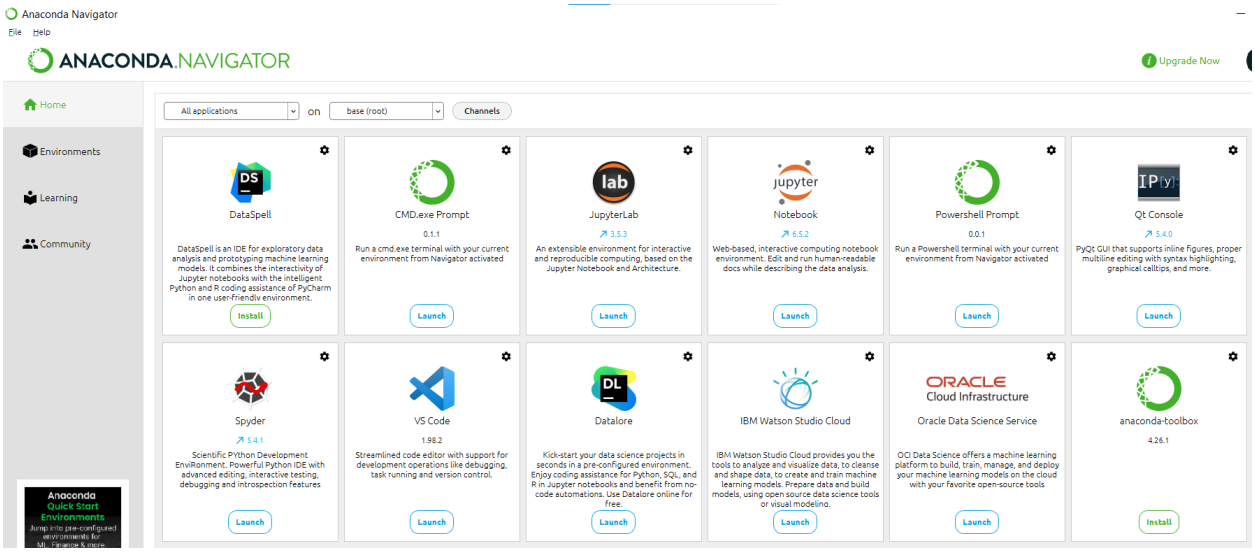


Figure 2.3: Anaconda Navigator

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

2.2.1 Creating new notebook

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.

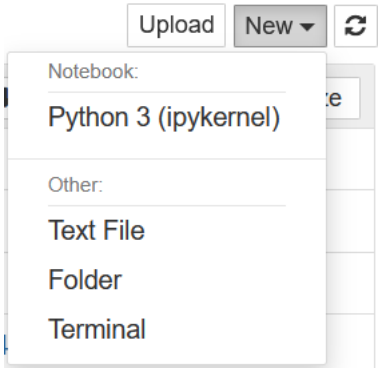


Figure 2.5: Creating notebook

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.

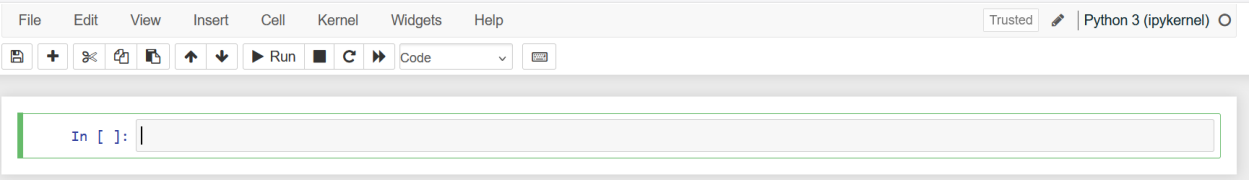


Figure 2.6: New notebook

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]:

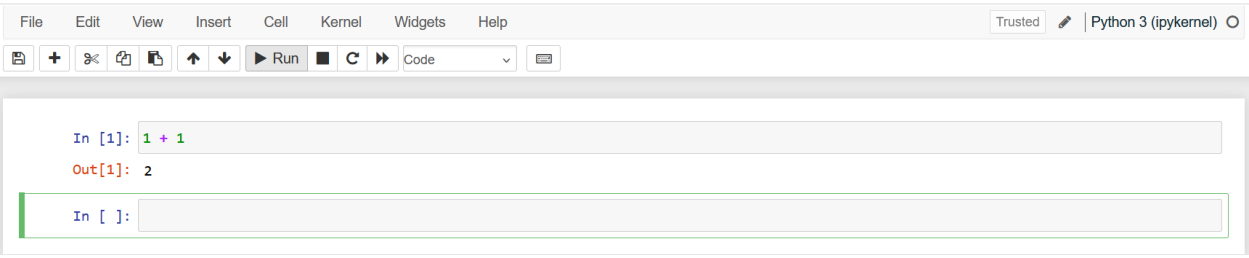


Figure 2.7: First Python code

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.

2.2.2 Opening existing notebook

The extension of a Jupyter Notebook file is `.ipynb` and sometimes denoted as Jupyter Source File. If you have stored such a file on your computer, you can open in it Jupyter Notebook as follows:

- Open Jupyter Notebook (see above) and navigate to the folder where you stored the file.
- Click on the .ipynb file and then it should open in a new tab of the browser.

> ⚠ WARNING about files on university computer
>
> If you store a file on a (TiU) university computer, for example in the Downloads folder, it will typically be deleted when you log out. To avoid this, either:
> - Copy the file onto a USB drive.
> - E-mail the file to yourself.
> - Store it on the M: drive (that is denoted by the drive that has your name) of the university computer, whose files are not deleted.
>
> Next time you want to use a file, put it again in the Downloads folder to work on it (and make sure to back it up properly again afterwards).

If you want to open a .ipynb file that is located in the M: drive of a university computer, you need to open

Jupyter Notebook differently than explained above (because you cannot navigate to the M: drive if you open Jupyter Notebook via the Start menu). Instead, do the following:

1. Close all instances of Jupyter Notebook if any are running.
2. Look up the *Anaconda Prompt* application in the Start menu and Run it.
3. Type `jupyter notebook --notebook-dir=M:` and press Enter
4. Jupyter Notebook should now open in the web browser showing the files and folders of the M: drive
5. Open the desired .ipynb file to work on it.

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 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:

```
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
```

> ⚠ 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 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 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.

```
z[0]
```

```
3.1
```

```
z[1]
```

```
9
```

```
z[2]
```

```
1.9
```

```
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 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 $a = [1, 4, 2, 5, 6, 3, 7]$.
- 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 using `print(total_sum)`.
- The line `for i in [0,1,2,3,4,5,6]:` indicates that we want to carry out a piece of code multiple times with different values for the variable `i`.
  - The words `for` and `in` are *Python keywords*, meaning they have a very specific purpose in Python. They get a different color when you type them in a code block.
  - Note that the line should end with a colon `:`.
- The piece of code that has to be run multiple times for $i \in \{0, 1, 2, 3, 4, 5, 6\}$ is `total_sum = total_sum + a[i]`.
  - For Python to understand that `total_sum = total_sum + a[i]` has to be executed with different values for i, we indent this line (using Tab on the keyboard).
  - 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	$21 + 7 = 28$

`i`	`a[i]`	`total_sum` after this iteration

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, 99, 3]
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
117
120
```

One final note to make is that this approach might still require a lot of typing if the list a contains many values. For example, if we are given a list of a thousand values, we would have to type a list with values 0 through 999 in the for-loop above.

There is a way to do this quicker, by using the `range()` function in Python. If we instead use the line `for i in range(7):`, then Python executes the indented code below for the values $i = 0, 1, 2, 3, 4, 5, 6$. Note that $i = 7$ is not included!

In general using `for i in range(n):` executes the indented lines below for the (in total n) values $i = 0, 1, 2, \ldots, n - 1$.

```
a = [1, 4, 2, 5, 6, 99, 3]
total_sum = 0

for i in range(7):
    total_sum = total_sum + a[i]
    print(total_sum)
```

```
1
5
7
12
18
117
120
```

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. We first make a general remark about printing text in Python.

Printing text

If you want to print text in Python, you should put it in between quotation marks.

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

```
Hello world
```

If you want to print both text and variables, you can do that in the same `print()` command separating them with a comma.

```
x = 3
print("The value of x is", x)
```

```
The value of x is 3
```

> Note that in `"The value of x is"` part, the x-symbol is interpreted merely as a letter, not a variable.

Now let us look at an 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 code below, which is a print-statement in this case. Python then no longer checks the `else:` statement.
- If the condition is false (i.e., $x \leq 0$), then Python executes the indented code under `else:`, which is a different print-statement in this case.

We can also have multiple conditions using `elif`, which stands for "else if". Below we add the third statement that checks if x is precisely equal to zero. Also here, as soon as Python reaches a statement that is true, it does not check the remaining statements anymore.

```
x = 0

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

```
x is zero
```

In the code above, we use the syntax `x == 0` to define the statement that checks whether x is precisely equal to 0. You should not use `x = 0` (otherwise Python would confuse this with assigning a value of 0 to the variable x, which is not what we want).

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.

```python
x = 1

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 positive
1
```

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.

The solution(s) to this problam are given by the quadratic formula (Dutch: abc-formule):

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

Here $\pm$ means that one solution is given by choosing a plus symbol in the place of $\pm$, and the other solution by choosing the minus symbol.

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.

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(y,z)` *or use* `print(y)` *and* `print(z)` *on different indented lines.* For the chosen a, b and c, your output should show the roots -1 and 0.333 (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.

Exercise 3.7

Create the lists $a = [3, 7, 1, 4], b = [2, 7, 4, 4]$ and $c = [11, 3, 0, 1]$. Write a for-loop that executes your code of Exercise 3.6(ii) for every combination (a_i, b_i, c_i) where a_i, b_i, c_i are the numbers at position i in the lists a, b, c, respectively, for $i = 0, 1, 2, 3$.

Your output should look like this.

```
The formula has no real roots
The formula has no real roots
-4.0 0.0
-0.5
```

Exercise 3.8 (bonus)

Suppose we are given a list $g = [9.1, 1.3, 5.4, 5.6, 5.74, 6.74, 8.25, 9.2, 7.1, 6.9]$ of student grades.

i) Write a Python code that rounds every grade to the nearest half integer, i.e., to the value in the set $\{0, 0.5, 1, 1.5, 2, 2.5, \dots, 8, 8.5, 9, 9.5, 10\}$ it is closest to, and print this value. *Hint: You can round a number to its closest integer by using the* `round()` *function. For example* `round(5.3)` *gives* 5*, and* `round(5.9)` *gives* 6*. Think of a way to use the* `round()` *function to round to half integers.*

On g as given, the output should be as follows.

```
9.0
1.5
5.5
5.5
5.5
6.5
8.0
9.0
7.0
```

```
7.0
```

ii) Adjust your code so that grades that lie in the interval $(5, 6)$ are rounded either up to 6 or down to 5 depending which of the two a number is closest to (in other words, rounding to 5.5 is no longer allowed). *Hint: You can use the* `and` *keyword to check multiple conditions in an if-statement.*

On g as given, the output should be as follows.

```
9.0
1.5
5
6
6
6.5
8.0
9.0
7.0
7.0
```

The latter procedure is in fact how your grades in the EOR bachelor program are rounded.

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!

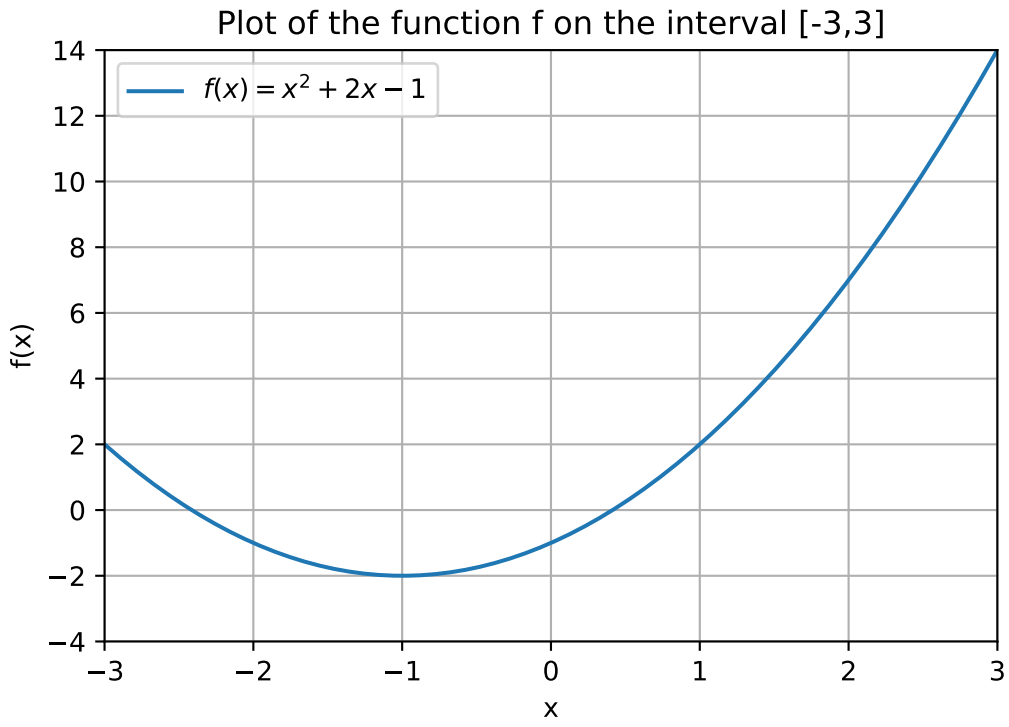
4.2 Plotting

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)
```

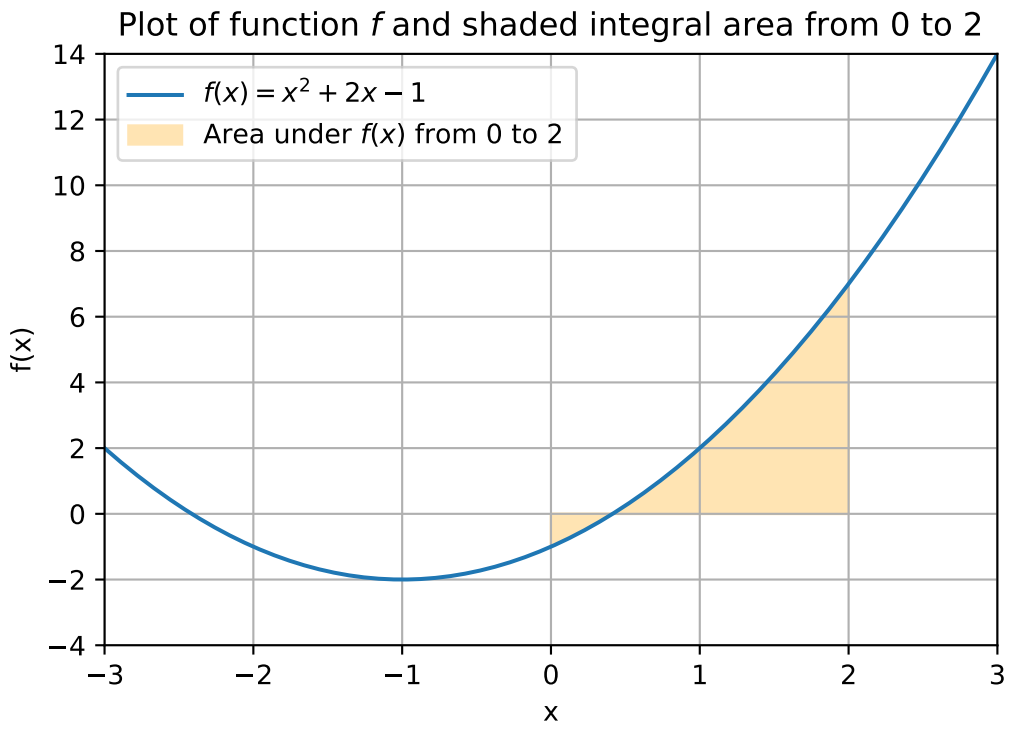
```
A root of the function f is given by [-2.41421356]
```

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.

Chapter 5

Linear algebra

In this chapter we will learn how to use Python for linear algebra.

5.1 Packages

In the first session of this crash course you have learned the basics of Python: how to use Python as a calculator, use lists, for-loops, and if-else-statements.

Python can be used for much more advanced operations as well. As you can imagine, code quickly grows more complicated. Fortunately, we can reuse code written by other through so-called **packages** (or libraries). In the optional chapter Chapter 4 you might have seen several packages already.

Using packages has several advantages. If you use a standard package, then it makes your code more readable. It also reduces the risk of errors: Python packages are typically developed by expert programmers and thoroughly tested before they are released to the public. For the mainstream packages that we will be using, you can thus be reasonably confident that they deliver what they promise.

Another major advantage of packages is that they are often heavily optimized in terms of speed and memory efficiency.[1]

Today we will use several well-known packages:

- NumPy,
- Sympy,
- and Matplotlib.

In a nutshell: packages are functions written by other people to make our life easy, i.e., so that we do not have to write every code file from scratch in Python.

The NumPy, SymPy and Matplotlib packages should be installed in a standard Anaconda installation. If you have another Python installation, typically using the PIP package manager should allow you to install packages. In this case, commands like `pip install numpy` *and* `pip install sympy` *should do the job.*

[1] Under the hood, numpy for instance relies on BLAS and LAPACK for most of its linear algebraic subroutines. BLAS and LAPACK are written in a programming language called Fortran, see for example the LAPACK documentation. LAPACK is used by many other programming languages, including Matlab.

5.2 Why `numpy` and `sympy`?

The package `numpy` is designed for fast numerical calculations, e.g., with matrices. We can compute matrix products, solve linear systems, and compute inverses with `numpy` very quicky. The package `sympy`, on the other hand, is a symbolic mathematics library. It can do exact algebraic manipulations (also with variables x and y), but it is less quick than numerical computations with `numpy`. `sympy` also contains a method to do exact row-reduction, to bring a matrix into reduced echelon form. This may also be useful if you did row-reduction by hand and want to verify your result with the computer.

5.3 Basic matrix and vector operations with Numpy

Suppose we have the following vectors **x** and **y**, and matrix A:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 4 \\ 5 \\ 7 \end{bmatrix}, A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

We can define vectors in Python as so-called NumPy arrays. Think of these as lists (that we saw earlier) on which we can perform numerical computations. To create such arrays, we have to import the NumPy package `numpy`. We do this under the *alias* `np` (so that we can use the short-hand notation `np` for `numpy` everywhere).

```
import numpy as np

# Define the vectors and matrix
x = np.array([1, 2, 3])
y = np.array([4, 5, 7])
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

In Python, we can add vectors, compute the matrix-vector product and multiply matrices. Note: To compute a matrix-vector product you use the command `A@x`. What happens if you use the command `A*x`? Now, try to compute $A\mathbf{x}$, $A\mathbf{y}$, and $A(\mathbf{x}+\mathbf{y})$ with Python. What do you notice?

```
# your code here
```

5.4 Application: input-output models

(From Python Linear Algebra Notebook by Herbert Hamers)

A chemical plant receives oil from three different regions: the Middle East, South America, and the NorthSea. The quality of the oil is different for each region. These oils will be used to produce gasoline, diesel fuel, and bike chain oil. The oil from the Middle east will be only used for the production of gasoline. Of the oil of South America, 20 % will be used for the production of gasoline, and 80 % for the production of diesel fuel. Finally, 25 % of the North Sea oil will be used to produce gasoline, 25 % to produce diesel fuel, and 50 % for the production of bike chain oil.

Suppose today a shipment arrives of 5000 barrels of oil from the Middle East, 9000 barrels of oil from South America, and 1000 barrels from the North Sea. The plant wants to know how much gasoline, diesel fuel, and bike chain oil it can produce from this shipment.

Obviously, the 5000 barrels of oil from the Middle East will completely be used for the production of gasoline, i.e. $1 \cdot 5000 = 5000$ barrels of gasoline. Of the 9000 barrels of South America oil, 20 % will be used for the production of gasoline. Hence, this leads to $0.2 \cdot 9000 = 1800$ barrels of gasoline. Finally, of the 1000 barrels of North sea oil 25 % will be used for the production of gasoline. Hence, we obtain $0.25 \cdot 1000 = 250$ barrels of gasoline. Hence, the total number of barrels of gasoline that will be produced is

$$1 \cdot 5000 + 0.2 \cdot 9000 + 0.25 \cdot 1000 = 5000 + 1800 + 250 = 7050.$$

By a similar computation, one can compute the total number of barrels of diesel fuel:

$$0 \cdot 5000 + 0.8 \cdot 9000 + 0.25 \cdot 1000 = 7450.$$

Finally, one can show that the total number of barrels of bike chain oil is 500, using a similar computation.

The production process can be summarized using the matrix-vector product. The production process is represented by the following matrix

$$\begin{bmatrix} 1 & 0.2 & 0.25 \\ 0 & 0.8 & 0.25 \\ 0 & 0 & 0.5 \end{bmatrix},$$

where the first column represents the division in the three end products of the Middle East oil, the second column represents the division in the three end products of the South America oil, and the third column represents the division in the three end products of the North Sea oil.

```
A=np.array([[1,0.2,0.25],[0,0.8,0.25],[0,0,0.5]])
print(A)
```

```
[[1.   0.2  0.25]
 [0.   0.8  0.25]
 [0.   0.   0.5 ]]
```

The shipment of the oil can be summarized by the following vector:

$$q = \begin{bmatrix} 5000 \\ 9000 \\ 1000 \end{bmatrix}$$

Using the matrix-vector product, it can easily be calculated how much gasoline, diesel fuel, and bike chain oil can be produced from this oil-delivery:

$$Aq = \begin{bmatrix} 1 & 0.2 & 0.25 \\ 0 & 0.8 & 0.25 \\ 0 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} 5000 \\ 9000 \\ 1000 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5000 + 0.2 \cdot 9000 + 0.25 \cdot 1000 \\ 0 \cdot 5000 + 0.8 \cdot 9000 + 0.25 \cdot 1000 \\ 0 \cdot 5000 + 0 \cdot 9000 + 0.5 \cdot 1000 \end{bmatrix} = \begin{bmatrix} 7050 \\ 7450 \\ 500 \end{bmatrix}$$

Hence, using the matrix-vector product we come to the same conclusion: the oil-delivery will result in 7050 barrels of gasoline, 7450 barrels of diesel fuel, and 500 barrels of bike chain oil.

The next day a shipment of 10000 barrels of oil from the Middle East, 1000 barrels of South America, and 200 barrels of the North Sea arrives. Determine the output of gasoline, diesel fuel, and key chain oil of this oil-delivery.

```
# your code here
```

Let us continue with a variation on the above problem.

Now, a chemical plant receives oil from five different regions: the Middle East, South America, the North Sea,Africa and Asia. Again, the quality of the oil is different for each region. These oils will be used to produce gasoline, diesel fuel, bike chain oil, fuel 95 and fuel 98. The oil from the Middle east will be only used for the production of gasoline. Of the oil of South America, 20% will be used for the production of gasoline, and 80% for the production of diesel fuel. 25% of the North Sea oil will be used to produce gasoline, 25% to produce diesel fuel, and 50% for the production of bike chain oil. Of the oil of Africa, 10% will be used for the production of gasoline, 10% for the production of bike chain oil, 30% for the production of fuel 95 and 50% for the production of fuel 98. Finally, 30% of the Asia oil will be used to produce gasoline, 5 % to produce diesel fuel, 50 % for the production of fuel 95, and 15% for the production of fuel 98.

Suppose today a shipment arrives of 5000 barrels of oil from the Middle East, 9000 barrels of oil from South America, 1000 barrels from the North Sea, 4000 barrels of oil from Africa, and 4000 barrels of oil from Asia. The plant wants to know how much gasoline, diesel fuel, bike chain oil, fuel 95, and fuel 98 it can produce from this shipment.

With the new information A will be a 5x5 matrix and q will be a 5x1 vector. Determine the output of gasoline, diesel fuel, bike chain oil, fuel 95, and fuel 98 of this oil-delivery.

```
# your code here
```

5.5 Matrix operations: inverse, determinant, solving linear systems

Assuming we imported NumPy under the alias `np`, the module `np.linalg` from NumPy contains several standard linear algebra methods that we will encounter today.

- `det(A)` : Determinant of matrix A
- `inv(A)` : Inverse of matrix A
- `solve(A,b)` : Solution to linear system $Ax = b$

To use such a method, you should use the syntax `np.linalg.method_name` with `method_name` replaced by one of the three options above.

```
import numpy as np

A = np.array([[0,1],[1,0]]) #Define A
print(A)

print("The determinant of A is", np.linalg.det(A))
```

```
[[0 1]
 [1 0]]
The determinant of A is -1.0
```

Exercise 5.1:

i) Compute the determinant of the following matrices

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 1 & 1 & 1 \end{pmatrix}, \qquad C = \begin{pmatrix} 1 & 3 & 2 \\ 2 & 3 & 7 \\ 1 & 3 & 1 \end{pmatrix}$$

ii) Which of the matrices A, B, C are invertible? For each of these matrices, compute the inverse.

```
# your code here
```

Exercise 5.2:

In the Linear Algebra course we have seen that if we multiply a single row by a constant k, then the determinant gets multiplied by k as well.

i) Verify this (using Python) for the matrix obtained from C by multiplying the first row by 10:

$$D = \begin{pmatrix} 10 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} C$$

ii) What is $\det(10C)$?

```
# your code here
```

We can use Python to solve linear systems of equations using `numpy`. Consider for example the following system of equations:

$$\begin{cases} x_1 - 3x_2 = 5 \\ -x_1 + x_2 + 5x_3 = 2 \\ x_2 + x_3 = 0 \end{cases}$$

We can solve the system very efficiently numerically with the command `np.linalg.solve(A,b)`:

```
import numpy as np

# Coefficient matrix A
A = np.array([[1, -3, 0],
              [-1, 1, 5],
              [0, 1, 1]])

# Right-hand side vector b
b = np.array([5, 2, 0])

# Solve the system
x = np.linalg.solve(A, b)
print("Solution:", x)
```

```
Solution: [ 2. -1.  1.]
```

As you can see, the solution is $x_1 = 2.0, x_2 = -1.0, x_3 = 1.0$. Note that the output is *numerical* i.e., it is an approximation computed with finite precision arithmetic inside the computer. `numpy` is faster for large systems than `sympy`, but it may introduce tiny rounding errors (e.g., 2.00000000001 instead of 2).

Exercise 5.3: Use `numpy` to solve the following system:

$$\begin{cases} x_1 - 3x_2 + 4x_3 &= -4 \\ 3x_1 - 7x_2 + 7x_3 &= -8 \\ -4x_1 + 6x_2 + 2x_3 &= 4 \end{cases}$$

```
# your code here
```

5.6 Modifying matrix entries

Consider the following large matrix A, and a zero matrix B of the same size as A:

```
A = np.array([
    [5, 12, 7, 3, 14, 6, 9, 2, 11, 8],
    [1, 13, 4, 10, 7, 5, 12, 6, 8, 3],
    [9, 2, 11, 5, 13, 7, 4, 10, 6, 12],
    [8, 1, 14, 6, 9, 3, 11, 2, 5, 13],
    [7, 10, 3, 12, 6, 9, 2, 8, 4, 11],
    [2, 6, 9, 5, 11, 7, 3, 12, 10, 1],
    [4, 8, 2, 10, 5, 13, 6, 9, 1, 7],
    [12, 3, 6, 11, 2, 8, 5, 14, 7, 10],
    [10, 5, 1, 7, 12, 4, 8, 3, 6, 9],
    [3, 9, 5, 8, 1, 10, 7, 11, 2, 12]
])
B=np.zeros((10,10))
```

We can modify entries of B separately. E.g., we can modify the top-left entry of B to be 100:

```
B[0,0]=100
print(B)
```

```
[[100.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]
```

Exercise 5.4: Use for loops over the row indices i and column indices j to modify the entries of B as follows, for each entry of A:

- If $A_{ij} < 6$, set $B_{ij} = 100$.
- If $A_{ij} \geq 6$, set $B_{ij} = A_{ij}$.

Then, compute the sum of all entries in B with `np.sum(B)`. What is the result?

```
# your code here
```

5.7 Linear transformations of images

In this section we will illustrate the concept of applying linear transformations to real-life images. We use the `cv2` library, the `numpy` library, and the `matplotlib` packages for this.

The above packages should be installed in a standard Anaconda installation. If you have another Python installation, typically using the PIP package manager should allow you to install packages. In this case, commands like "pip install opencv-python" "pip install numpy" "pip install matplotlib" in a (Windows) powershell should do the job.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

A linear transformation for a vector in $\mathbb{R}^2$ can be represented by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

For images, (x, y) are the pixel coordinates of the original image and (x', y') are the pixel coordinates of the transformed image. First we load in an image (this can be any image with the name 'image.jpg', but it must be in the same folder as the Python notebook):

```
img2 = cv2.imread('image.jpg')

plt.imshow(img2[:,:,::-1])
plt.axis('off')
```

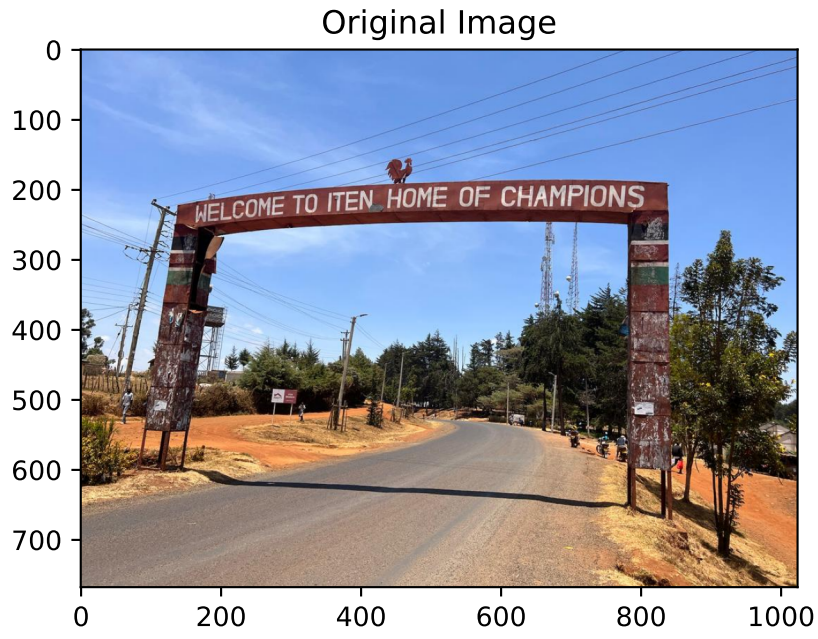
The picture can be seen as a set of points in $\mathbb{R}^2$, corresponding to the pixels, where each point is assigned a color. We are now going to move the points according to a linear transformation $\mathbb{R}^2 \to \mathbb{R}^2$ which maps $\mathbf{x}$ to $A\mathbf{x}$.

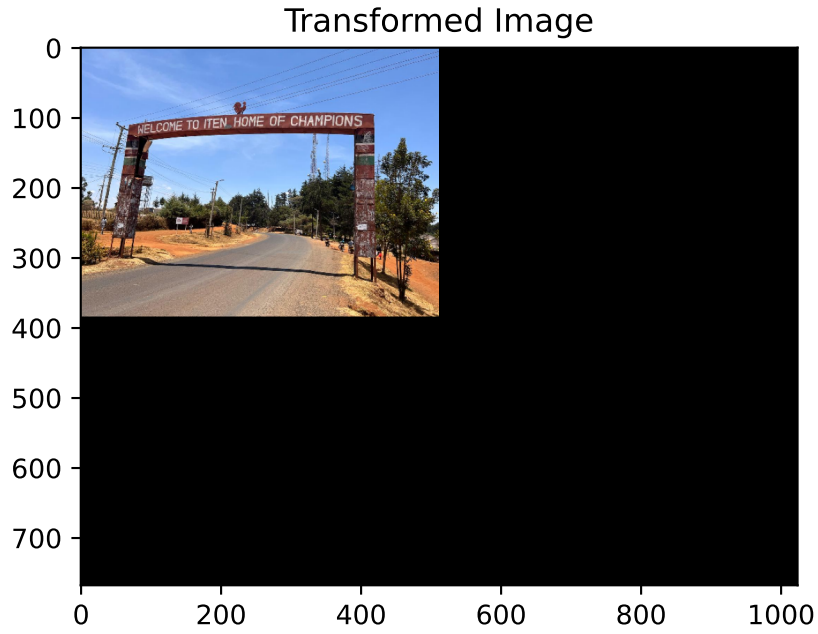
We first predefine a Python function called `perform_Transformation()` to apply a transformation to the image. You don't have to understand this code, but only remember that its purpose is to apply transformations to images. The function takes as input an image and a transformation matrix.

```
# Function to apply transformation and visualize the result (source: kaggle.com)
def perform_Transformation(image, A):
    rows,cols,ch = image.shape

    M = np.array([[A[0,0], A[0,1], 0],
                  [A[1,0], A[1,1], 0]])

    dst = cv2.warpAffine(image,M,(cols,rows))


    plt.figure(figsize = (24,8))
    plt.subplot(211); plt.imshow(image[:,:,::-1]); plt.title('Original Image')

    plt.subplot(212); plt.imshow(dst[:,:,::-1]); plt.title("Transformed Image")
```

Next, we create a 2x2 (transformation) matrix A and use the image that we stored in `img2` variable and apply the transformation function to these inputs. The function prints both the original and transformed image.

```
# The matrix A scales the picture down 50% in both the x direction and y direction
A = np.array([[0.5,0],
                [0,0.5]])

perform_Transformation(img2, A)
```





Exercise 5.5:

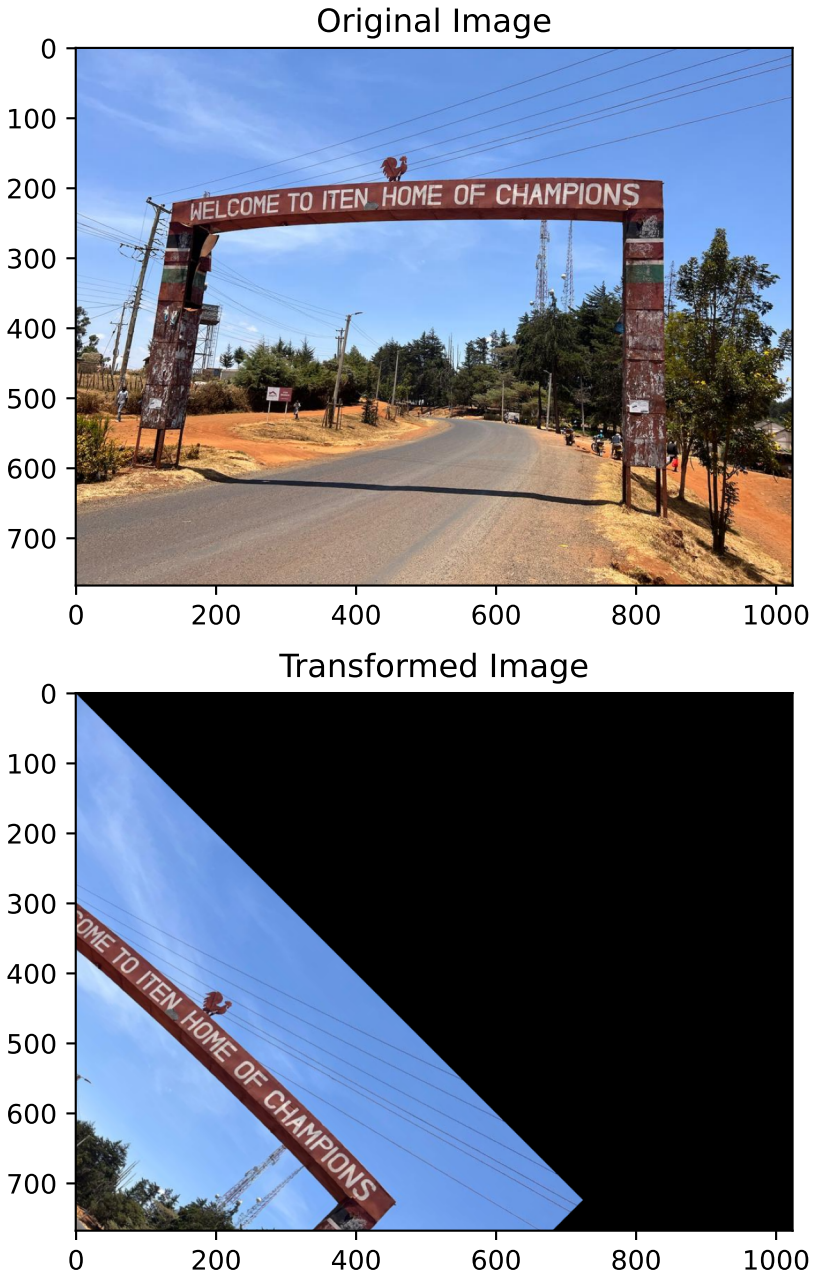
i. Can you scale the picture 50% down in the x-direction only?

ii. Investigate what the matrix $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ does to the picture. Can you explain why?

```
# your code here
```

The following matrix rotates the picture approximately 45 degrees (note that $\sin(\pi/4) = \cos(\pi/4) \approx 0.707$). Sadly, the image is only partly visible then.

```
# The matrix A rotates the picture 45 degrees
A = np.array([[0.707,-0.707],
              [0.707,0.707]])

perform_Transformation(img2, A)
```



Exercise 5.6:

i. Let $A = \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Compute the matrix products AB and BA.

ii. Investigate what the matrix $BA = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \end{bmatrix}$ does to the picture. Can you explain why?

iii. What does the matrix $AB = \begin{bmatrix} 0.5 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ to the picture and why?

iv. Can you scale the picture 50% down in the x-direction only, and then rotate the picture (approximately) 45 degrees? What matrix corresponds to this?

```
# your code here
```

5.8 Symbolic computations: sympy

The `sympy` library is for symbolic computations. We will now demonstrate how this library works. `sympy` also contains a function to bring a matrix into reduced echelon form. This is very useful if you want to verify manual calculations.

```
import sympy as sp
```

We now show how to solve the following system of equations:

$$\begin{cases} x_1 - 3x_2 + 4x_3 = -4 \\ 3x_1 - 7x_2 + 7x_3 = -8 \\ -4x_1 + 6x_2 + 2x_3 = 4 \end{cases}$$

The augmented matrix of the system of linear equations is

$$\left[\begin{array}{ccc|c} 1 & -3 & 4 & -4 \\ 3 & -7 & 7 & -8 \\ -4 & 6 & 2 & 4 \end{array}\right]$$

In Python you can do this with `sympy` as follows.

```
Ab=sp.Matrix([[1,-3,4,-4],[3,-7,7,-8],[-4,6,2,4]])
```

Now we can find the reduced form of the system using `rref()` from the `sympy` library. The method returns two elements. The first is the row-reduced echelon form of the matrix, and the second is a list of the pivot columns. By typing `Ab_rref[0]`, only the row-reduced echelon form is printed.

```
Ab_rref = Ab.rref()
Ab_rref[0]
```

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The system is consistent and we can immediately see the general solution: $x_1 = 2$, $x_2 = 2$ and $x_3 = 0$. The solution is exact, e.g., Python did not round the number 2 to 2.00000 as it does when using the `numpy`

module. `sympy` allows us to do exact computation with matrices, as opposed to approximate computations with `numpy`. Computations with `sympy` are generally slower than with `numpy`, so for very large systems, `numpy` is preferred.

Exercise 5.7: Use Python to solve the following system of equations:

$$\begin{cases} x_1 - 3x_2 = 5 \\ -x_1 + x_2 + 5x_3 = 2 \\ x_2 + x_3 = 0 \end{cases}$$

```
# your code here
```

5.9 Application: supply and demand model

(From Linear Algebra notebook by Herbert Hamers)

The demand q_d and supply q_s of an item depend on the price p and income Y. Suppose that the relation between demand, price and income can be described by the equation

$$q_d = 8 - 0.2p + 0.1Y,$$

and the relation between supply and price by the equation

$$q_s = 6 + 0.3p.$$

The market is in equilibrium if $q_s = q_d$. Replacing both q_s and q_d by the new variable q, the market equilibrium can be found by solving the following system of two linear equations for the unknowns p, q and Y:

$$\begin{cases} q = 8 - 0.2p + 0.1Y \\ q = 6 + 0.3p. \end{cases}$$

First, we will rewrite this system of linear equations. We place all terms with a variable to the left of the equal sign and moreover we put terms with the same variable in the different equations right below each other:

$$\begin{cases} q + 0.2p - 0.1Y = 8 \\ q - 0.3p = 6. \end{cases}$$

```
Ab = sp.Matrix([[1, 0.2, -0.1, 8], [1, -0.3, 0, 6]])
Ab
```

$$\begin{bmatrix} 1 & 0.2 & -0.1 & 8 \\ 1 & -0.3 & 0 & 6 \end{bmatrix}$$

Exercise 5.8:

How many solutions does this system of linear equations have? You can use the function `rref()` again to find the row reduced echelon form of the above matrix. Determine the solution if $Y = 50$.

```
# your code here
```

5.10 Symbolic computation of the determinant in Python with `sympy`

The `sympy` package can also be used to calculate determinants of symbolic matrices.

Exercise 5.9:

We now investigate how to compute determinants of *symbolic* matrices using sympy. Let

$$A = \begin{pmatrix} 1 & 0 & x \\ 1 & -x & 0 \\ x & 0 & -x \end{pmatrix}.$$

Compute $\det(A)$ using the `sympy` module. First we define the variable x:

```
import sympy as sp
x = sp.symbols('x')
```

Next, define the matrix A, and compute the determinant of A using the `sympy` function A.det(). For which values of x does the determinant equal zero? To be able to see more easily for which values of x the determinant equals 0, you can use `sympy`'s factor function.

```
# your code here
```

For which values of x is A invertible?

5.11 (Optional) Row reduction with intermediate steps

This optional section describes a more advanced code snippet that implements a row-reduction algorithm from scratch, and allows you to output the intermediate operations. You should be able to read most of the code and recognize the algorithm, but some details have not been explained in this notebook. More specifically, you will see again for-loops and if-statements, but also "new" functions such as .copy(), .as_mutable(), .append(). If you are interested, we invite you to look up their documentation. You can also use this algorithm to revisit some examples that you have seen in the linear algebra course.

As a disclaimer: this code has been generated using ChatGPT.

```
from sympy import Matrix

def rref_with_steps(mat):
    """
    Perform row-reduction to RREF while recording intermediate steps and row operations.
    Returns:
        rref_matrix (Matrix),
```

```
        pivot_columns (tuple),
        steps (list of (Matrix, str))  # Each step is (matrix_snapshot, operation_descriptio
    """
    A = mat.as_mutable().copy()
    rows, cols = A.shape
    pivots = []
    steps = [(A.copy(), "Initial matrix")]

    row = 0
    for col in range(cols):
        if row >= rows:
            break

        # Find pivot row
        pivot_row = None
        for r in range(row, rows):
            if A[r, col] != 0:
                pivot_row = r
                break

        if pivot_row is None:
            continue

        # Swap rows if needed
        if pivot_row != row:
            A.row_swap(pivot_row, row)
            steps.append((A.copy(), f"Swap R{pivot_row+1}  R{row+1}"))

        # Scale pivot row
        pivot_val = A[row, col]
        if pivot_val != 1:
            A.row_op(row, lambda x, _: x / pivot_val)
            steps.append((A.copy(), f"R{row+1} → (1/{pivot_val})·R{row+1}"))

        # Eliminate other rows
        for r in range(rows):
            if r != row and A[r, col] != 0:
                factor = A[r, col]
                A.row_op(r, lambda x, j: x - factor * A[row, j])
                steps.append((A.copy(), f"R{r+1} → R{r+1} - ({factor})·R{row+1}"))

        pivots.append(col)
        row += 1

    return A, tuple(pivots), steps
```

```
A = Matrix([[1, 2, 1],
            [2, 4, 3],
            [3, 6, 5]])

rref_matrix, pivots, steps = rref_with_steps(A)

print("RREF:")
print(rref_matrix)
print("Pivot columns:", pivots)

print("\nSteps:")
for i, (s, op) in enumerate(steps):
    print(f"Step {i}: {op}")
    display(s)
```

```
RREF:
Matrix([[1, 2, 0], [0, 0, 1], [0, 0, 0]])
Pivot columns: (0, 2)

Steps:
Step 0: Initial matrix
```

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 3 \\ 3 & 6 & 5 \end{bmatrix}$$

```
Step 1: R2 → R2 - (2)·R1
```

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 1 \\ 3 & 6 & 5 \end{bmatrix}$$

```
Step 2: R3 → R3 - (3)·R1
```

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

```
Step 3: R1 → R1 - (1)·R2
```

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

```
Step 4: R3 → R3 - (2)·R2
```

$$\begin{bmatrix} 1 & 2 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$