COMPUTER SCIENCE FOR KIDS

Learning by doing

Dimos Raptis

Table of Contents

Talking to a computer

Computers are our friends. They can help us complete many tasks that we would find very hard to complete otherwise. However, we need to find a way to tell them how to complete each task. In other words, we need to find a way to "talk" to a computer. In this chapter, you will learn how you can talk to a computer. If you didn't know how to do that already, you would probably imagine that would be a quite difficult thing to do. You will probably be surprised by how easy that can be in practice.

Of course, there are many different languages we can use to talk to computers. This is similar to how there are many different languages people can use to communicate with each other, such as English, French, Chinese etc. The languages that we can use to talk to a computer are typically called **programming languages**, because they help us program the computer to do what we want. Among all those languages that are available, in this book you will learn how to use a language called Python. Before you are able to talk to our computer, first you will have to make sure your computer knows how to speak the language you want to use. In other words, you will need to "install" a programming language.

In order to install Python, you can visit the following page, click on the "Download Python" button and then click on the downloaded file and follow the instructions: https://www.python.org/downloads.

Make sure you download version 3 of Python as this is used throughout this book - this is any version that starts with 3, such as 3.5, 3.6, 3.7 etc. After the installation is finished, you can make sure it has been completed successfully and your computer can now "speak" Python. In order to do this quickly, you need to open a terminal window first and type `python3` - in Windows, you need to type `py`, instead. If you don't know how to open a terminal, follow the instructions in Appendix A. You should now see something like the following in your terminal:

```
$ python3
Python 3.9.0 (default, Dec  3 2020, 16:09:02)
[Clang 12.0.0 (clang-1200.0.32.27)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This means your computer is waiting for your instructions. Let's try and give a simple instruction to the computer, such as adding up two numbers. You can just type a mathematical expression that's adding two numbers and press Enter. Your computer will perform the necessary calculations and show you the result:

```
>>> 40 + 2
42
```

That's cool, but we could also use a calculator to do that, right? Computers can do much more complicated things and this is what you will learn in this book. Let's go and write the simplest program we could think of, a program that says "Hello World!". In order to do that, we will need to write the instructions of the program in a text file. To make this easier, you could use a text editor, which is a program that helps you read the contents of a file and change them. One such program is Sublime, which you can download easily by visiting the following web page and clicking on the button that says "Download": https://www.sublimetext.com/

Ok, now that we have downloaded a text editor, we can create our first program. First, I want you to go in your Desktop and create a new folder, called "programs". If you don't know how to do this, see Appendix A. We will store here all the code for our programs. Open your text editor and create a new file. When your new file is open in the editor, you can type the following code inside the file:

```
print("Hello World!")
```

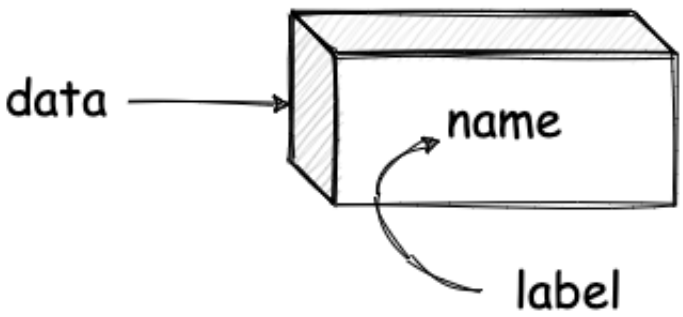
Before you can run this program, you will need to save the file. If you are using Sublime, you can do this by selecting the following options on the menu: File > Save and then select the folder where the file will be stored (the "programs" folder you previously created) and the name of the file (it should be "first_program.py"). Your first program is now ready! It is time to ask your computer to run it. In order to do this, you need to open a terminal again, but this time you should open it inside the folder you created. If you don't know how to do this, you can follow the steps in Appendix A. When the terminal is open, you can ask your computer to run your program. You can do this by typing `python3 first_program.py` in the terminal and then pressing Enter. You should see the following in the terminal:

```
$ python3 first_program.py
Hello World!
```

You can see your program talking! Before we move on, let's try and understand the meaning of what you typed. The first part of the instruction (`python3`) meant that your computer should run a Python program and the second part (`first_program.py`) is the name of the file, where the program is written. This is needed so that the computer knows where to go and find the program.

Ok, you have now managed to write your first program and ask your computer to run it. However, your program still doesn't seem very cool. In order to build cool programs, you will have to learn how to speak Python a bit better. There are many many great things we can do in Python, so we will focus on learning the most basic things now.

As you saw previously, Python can print things in a terminal. Instead of using a single text inside quotes, we can also combine multiple texts by adding them to each other with the `+` operator. For example, you could replace `print("Hello World!")` with `print("Hello" + "World!")` in your program and it would still work fine. We can also ask the user to give us some information in the terminal, if we want. This can be done by using the command `input` followed by the text that will be shown to the user enclosed in parentheses. We can also store the information provided by the user in a variable, so that we can use it later. You can think of a variable as a box with a label that can hold some data inside. Whenever we want to use the data contained in this box (variable), we can refer to them using the label of the box (the variable name).



Let's try and use this to make our program a bit cooler. We can ask the user to give us their name and then say hello to them using their name. You can open the file that contains our first program and change the contents to the following (remember to save the file again!):

```
name = input("Give me your name: ")

print("Hello " + name + "!")
```

If you run the program again, you will see the terminal printing "Give me your name: " and then waiting. After you type your name and press Enter, the program continues and greets you.

```
$ python3 first_program.py
Give me your name: Dimos
Hello Dimos!
```

Hopefully, what is going on here is clear. The data the user provides in the terminal (e.g. "Dimos") is stored in the variable `name` and the message printed to the console is reading the value of this variable to generate the message to be printed (`"Hello" + name + "!"` → `"Hello" + "Dimos" + "!"` → `"Hello Dimos!"`).

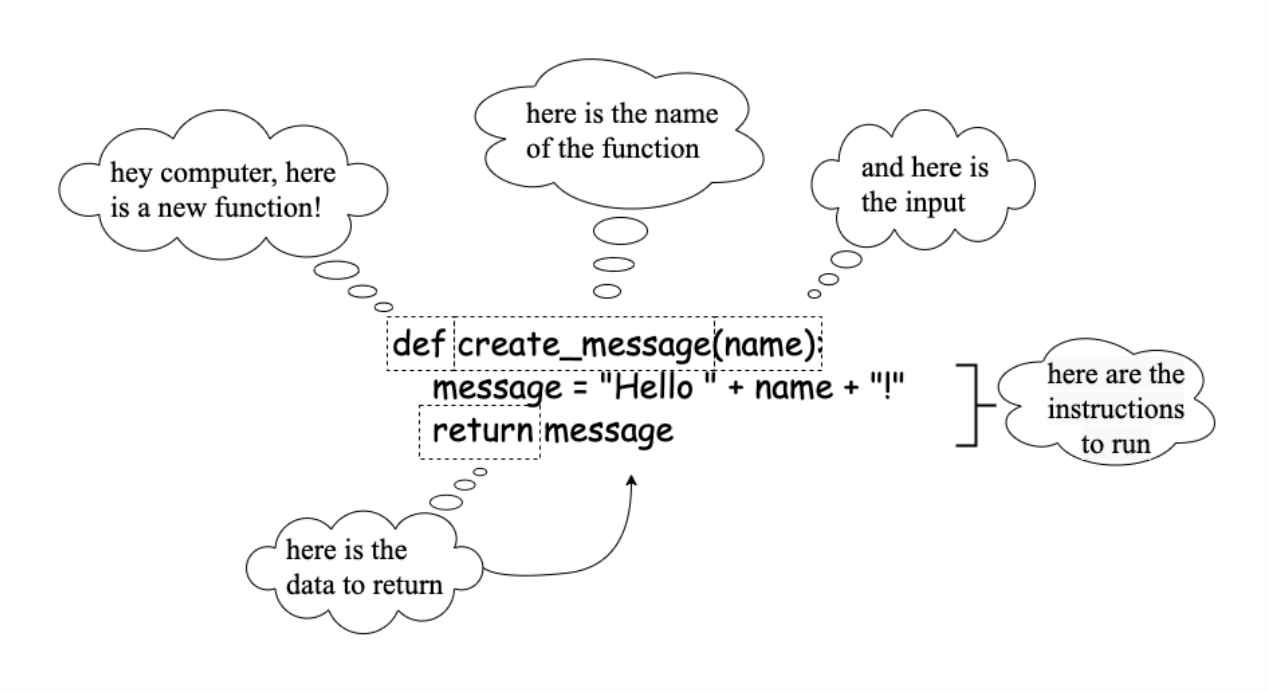
The `input` and `print` instructions we used are called **functions**. A function is a block of code that can receive some data as input, perform a specific task and it can also return some data as a result, if needed. Everytime, you need to execute this specific task, you can just call this function using its name, instead of rewriting all the code again. As a result, functions can help you organise your code into small blocks you can reuse in different places. For example, we could create a small function that takes as input the name of the user and returns the full message text. Our program would look like the following:

```
def create_message(name):
  message = "Hello " + name + "!"
  return message

name = input("Give me your name: ")
message_text = create_message(name)
print(message_text)
```

If you try and run this program, you will notice the result is exactly the same. What is different is we defined a function (`create_message`) and used it to create the message that was printed.

In order to use a function, you first need to define what the function will do. This is what is done at the top of the program. The definition of a function starts with the `def` keyword, followed by the name of the function (`create_message` here) and the input to the function which can be one or more variables (here it is just one: `name`) enclosed in parentheses. This is then followed by a list of instructions that will be run by the computer every time you use the function. The `return` instruction is a special instruction that terminates the function and returns the data contained in a variable. You might have noticed we used some functions (`print` and `input`) before without defining them. This is because the language contains a group of basic functions that are already defined and can be used immediately by programmers and these are some of them. There are two ways you can use a function. You can just type the name of the function and the input variables - e.g. `print("Hello World!")` - or you can create a variable that will contain the data returned from the function - e.g. `name = input("Give me your name: ")`. That is a lot of information, so I tried to summarise it for you in the following picture.



Of course, there are many different types of data a program can handle. So far, we have only seen a single type, text that consists of a sequence of characters. This type of data in programming is typically called a **string**. Some other basic data types are the following:

- **integers**, e.g. -10, 0, 25 etc..
- real numbers, also known as **floating numbers** or **floats**, e.g. 1.4, 0.3 etc.
- **booleans**, which can take two values `True` and `False`.

Sometimes, we might have to convert data from one type to another. Sometimes, this is done automatically, but Python also provides specific functions to do this, when needed. For example, if we have a variable `age` that contains the age of a person as an integer and we want to convert it into a string in order to print it in the console, we can use the `str` function, e.g. `str(age)`. If we want to do the opposite and convert a variable `weight` that contains a user's weight as a string into an integer to make calculations, we can do so using the `int` function, e.g. `int(weight)`.

There are cases where a program has to do different things depending on the value of some data. One way to achieve this is by using the `if` statement. This comes in many different flavors:

- A simple `if` statement will execute some piece of code only if some condition is true. Its structure is the following:

```
if <condition>:
  <code to execute>
```

- An `if-else` statement will execute some piece of code if some condition is true. Otherwise, it will execute a different piece of code. Its structure is the following:

```
if <condition A>:
  <code to execute>
else:
  <other code to execute>
```

- An `if-elif` statement will execute different pieces of code depending on which of the specified conditions is true and will execute a different piece of code if none of them is true. Its structure is the following:

```
if <condition A>:
  <code to execute>
elif <condition B>:
  <other code to execute>
else:
  <other code to execute>
```

Warning: In Python, indentation matters! So, be careful and insert the a tab (or the same amount of space) as shown above. Otherwise, the computer will not be able to understand what the program says and will complain. In this case, the error might look something like the following:

```
        ^
IndentationError: expected an indented block
```

We have also seen how to assign a value to a variable already by using the = operator, e.g. `name = "Dimos"`. We can also check if a variable has a specific value in order to decide what action to take in a program. We can do this by using the == operator, e.g. `name == "George"`. We can either store the result of this in a separate variable (e.g. `is_george = name == "George"`) or use it directly inside an `if` statement as a condition (e.g. `if name == "George"`). We can also compare different variables with each other using other operators, such as < (less than) and > (greater than). For example, if we have two variables with the age of two students `maria_age` and `george_age`, we can check if Maria is older than George in the following way: `maria_age > george_age`.

Let's go and use all of the cool stuff you learned to make your program a bit more complex. I want you to change the program, so that it asks the user to also provide the number of programs they have written (*top tip*: you will need to convert the user's data from string to int for this). Depending on the number of programs, they have written, the program will display a different message.

- If the user specified a negative number, the program should print "Come on! This is impossible."
- If the user gave the number zero, the program should print "It's not that hard! Give it a try."
- If the user gave a positive number, the program should print "Well done! Keep practicing."

When you execute the program, it should look like the following:

```
$ python3 first_program.py
Give me your name: Dimos
How many programs have you written so far: 3
Hello Dimos!
Well done! Keep practicing.
```

Let's pause now. I want you to try and write the program on your own. If the program contains syntactic errors, the computer might complain when you try to run it. Don't give up immediately though, try and see if you can understand what's wrong by reading the error message of the computer and fix your program.

Done? Ok, here's the solution:

```
name = input("Give me your name: ")
programs_as_string = input("How many programs have you written so far: ")
programs = int(programs_as_string)

print("Hello " + name + "!")

if programs < 0:
  print("Come on! This is impossible.")
elif programs == 0:
  print("It's not that hard! Give it a try.")
else:
  print("Well done! Keep practicing.")
```

Your program starts becoming a bit more useful now, right? But, it can still only handle data for a single user. What if we wanted to receive data for many students, say for the students of a whole class? If that was possible, we would be able to do cool things, like finding out the best and the worst programmer in the class!

You will now learn two additional concepts that will help you achieve that:

- **data structures**
- **loops**

Data structures are collections of values in a specific form that allows you to access the data included in them in different ways. If we go back to our previous example, imagine if we had to handle data for many students, instead of just one. We would probably need to save the data for them and we could do that using different variables. For example, if we wanted to process data for 3 students we could do it in the following way:

```
first_name = input("Give me the name of the first student: ")
second_name = input("Give me the name of the second student: ")
third_name = input("Give me the name of the third student: ")
```

If we wanted to greet all of them, we could do the following:

```
print("Hello " + first_name + ", " + second_name ", and " + third_name)
```

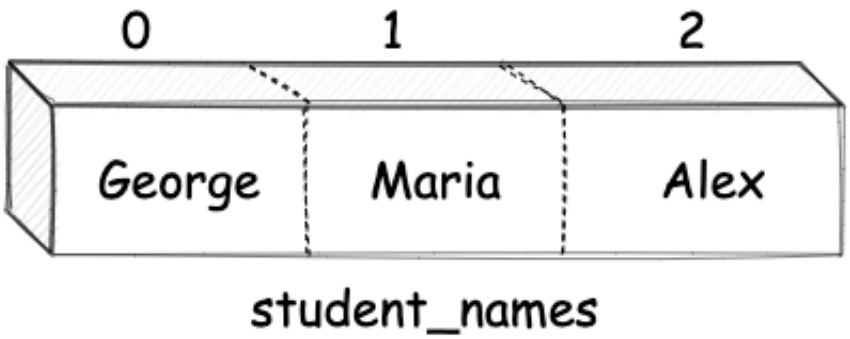
However, this would become uglier and harder to code the more students the program needed to handle. For example, imagine how the program would look like if it needed to handle 100 students in this way. Another problem is the program might not know the number of students in advance. For example, the program might have to ask the user for the number of the students in the class, before asking for their names. This is where data structures and loops can help:

- Data structures allow a program to store many values in a single variable.
- Loops allow a program to perform the same task again and again.

Let's look at our first data structure, the **list**. A list is an ordered sequence of values. You can create a list by writing all its values enclosed in square brackets and separated by commas. Here is an example showing how you could create a list containing some student names and assign it to a variable:

```
student_names = ["George", "Maria", "Alex"]
```

Visually, you could think of this variable in the following way.



You can do many interesting things with the list now:

- You can easily calculate the size of the list using the `len` function, e.g. `len(student_names)` would have the value 3, since there are 3 values in the list.
- You can retrieve an element at a specific location in the list. You can do this using the variable name followed by square brackets and the position you are looking for. As you can see in the picture above, the position of the first value is zero and then it increases by one for every value. So, `student_names[1]` would return the second value in the list, which is `Maria`.
- You can also add new values in the list by typing the name of the variable, followed by `.append()` and the value you want to add inside the parentheses. For example, if you want to add the value `Helen` at the end of the list you can do `student_names.append("Helen")`.

Now that we have a list, we might want to repeat a specific task for each value in the list. For example, we might want to greet each student separately. A `for` loop statement can help us achieve this. Its structure is the following:

```
for <value> in <list>:
  <do something>
```

You can specify a list you want to iterate over (`<list>`) and the name of the variable that will contain one value at a time (`<value>`) and the instructions included in the following block will be executed one time for each value in the list. An example will probably help you understand this better. If you want to greet all the students with their names included in the `student_names` list, you could do that in the following way:

```
for name in student_names:
  print("Hello " + name + "!")
```

In some cases, you might not have a list of items and you might just need to perform the same task for a number of times. You can also do that using a `for` loop and the `range`() function in the following way:

```
times = 10
for number in range(times):
  print("This is number: " + str(number))
```

What is happening here is the program is going through the numbers in the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`.

You have now learned how to use data structures (lists, specifically) and for loops. We can use this to make our program a bit fancier. I want you to change the program so that it does the following:

- It asks the user to provide the number of students in the class.
- Then, the program repeats the following for each student in the class. First, it asks for the name of the student and then it asks for the number of programs they have written.
- The program stores the names of the students and the number of programs they have written in two lists. It iterates over the values in these lists, it finds the student with the largest number of programs and prints it in the terminal.

All in all, the output of the program should look like the following:

```
$ python3 first_program.py
Give me the number of students in the class: 3
Give me the name of the next student: Dimos
Give me the number of programs written by this student: 3
Give me the name of the next student: Maria
Give me the number of programs written by this student: 2
Give me the name of the next student: Helen
Give me the number of programs written by this student: 6
The best student is: Helen
```

I'll give you some time to try and write this program yourself. While doing this, you might realise you need to do something that was not mentioned here yet. Don't worry, because this will happen to you very frequently when programming. Before giving up, try to find out the information you need on your own. Internet is your friend.

Done? Ok, here's the solution:

```
number_of_students_as_string = input("Give me the number of students in the
class: ")
number_of_students = int(number_of_students_as_string)

all_names = []
all_programs = []

for student_number in range(number_of_students):
  student_name = input("Give me the name of the next student: ")
  all_names.append(student_name)
  student_programs_as_string = input("Give me the number of programs written
by this student: ")
  student_programs = int(student_programs_as_string)
  all_programs.append(student_programs)
```

```
best_student = all_names[0]
best_effort = all_programs[0]

for student_number in range(number_of_students):
  if all_programs[student_number] > best_effort:
    best_student = all_names[student_number]
    best_effort = all_programs[student_number]

print("The best student is: " + best_student)
```

Of course, this is just one of the possible solutions. If your program has the same behaviour when you run it, it's OK if the code is slightly different.

Looking back at the code, you might notice something slightly disappointing. Each student is associated with a name and the number of programs they have written. However, each piece of information lives separately in two different places. The names of the students live in the `all_names` list and the number of programs written by them live in the `all_programs` list. This means if we want to find the number of programs written by a specific student, we have to find the location in the first list and search for the value of the same location in the second list. Wouldn't it be nice if we could have all the information for each student in the same place? You are now going to learn one way to do this, **classes**.

A class is a specific category of objects that share some common characteristics. In our previous example, all students have a name and a number of programs they have written. The concept of the class allows you to define these common characteristics that all the objects of this class have. Then, everytime you need such an object you can create an **instance** of this class. I will now show you how you can define a class for students:

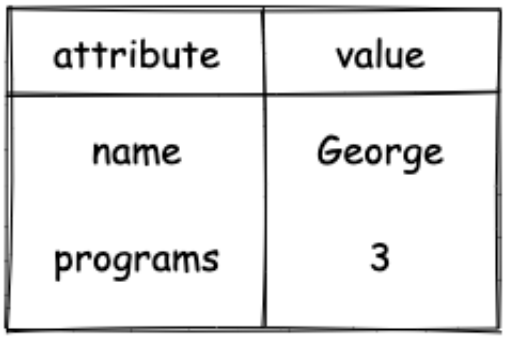
```
class Student:
  def __init__(self, student_name, number_of_programs):
    self.name = student_name
    self.programs = number_of_programs
```

This creates a class called `Student`. Every object of this class has two attributes, called `name` and `programs`. `__init__` is a special type of function, called a **method** . A method is a function that is associated with a specific class. This method can only be used on a variable that has an object of this class and this is done by typing the name of the variable, followed by a dot (`.`), the method name and the input data. You might haven't noticed previously, but you have already used a method, `append()`. `append()` is a method that is associated with lists. For example, if you want to add the value `George` to a list variable called `student_names`, you need to write `student_names.append("George")`. This `__init__` is a very special method, called **constructor**, that is used to create new instances of a class. It receives as input the instance of the class that will be created (`self`) and a set of variables we specify (e.g. `name`, `number_of_programs`). Inside the constructor, we can assign values to the attributes of a class using the data passed into the `__init__` method. In order to do this, we need to specify the instance followed by a dot (`.`), the name of the attribute, the assignment operator (=) and the value we need to assign. This is what `self.name = student_name` does - it creates a new attribute called `name` and it assigns the data that is inside the input variable `student_name`. When you want to create an instance of the class, you can just write the name of the class and the data passed into the constructor enclosed in parentheses. **Attention**: you only need to specify values for the variables after `self` in the `__init__` method, since `self` is calculated automatically by the computer. Back in our example, if you

wanted to create an instance of a student with the name `George` that has written 3 programs, you could do this with the following piece of code:

```
some_student = Student("George", 3)
```

Visually, you could think of this object in the following way.



You can also access the values of a specific attribute of an object. For instance, if you wanted to access the field `name` of the student object we just created, you could do this by writing `some_student.name`, which would return the value `George`.

Let's go back to our last program now and see if we can improve it. If we defined a student class as I just showed you, we could have a single list that stores all the student objects, instead of two separate lists. In order to find the best student, we could just iterate over all the objects in this list. If you follow this advice and change the code, the program will look something like the following:

```
class Student:
  def __init__(self, student_name, number_of_programs):
    self.name = student_name
    self.programs = number_of_programs

number_of_students_as_string = input("Give me the number of students in the
class: ")
number_of_students = int(number_of_students_as_string)

all_students = []

for student_number in range(number_of_students):
  student_name = input("Give me the name of the next student: ")
  student_programs_as_string = input("Give me the number of programs written
by this student: ")
  student_programs = int(student_programs_as_string)
  student = Student(student_name, student_programs)
  all_students.append(student)

best_student = all_students[0]
```

```
for student in all_students:
  if student.programs > best_student.programs:
    best_student = all_students[student_number]

print("The best student is: " + best_student.name)
```

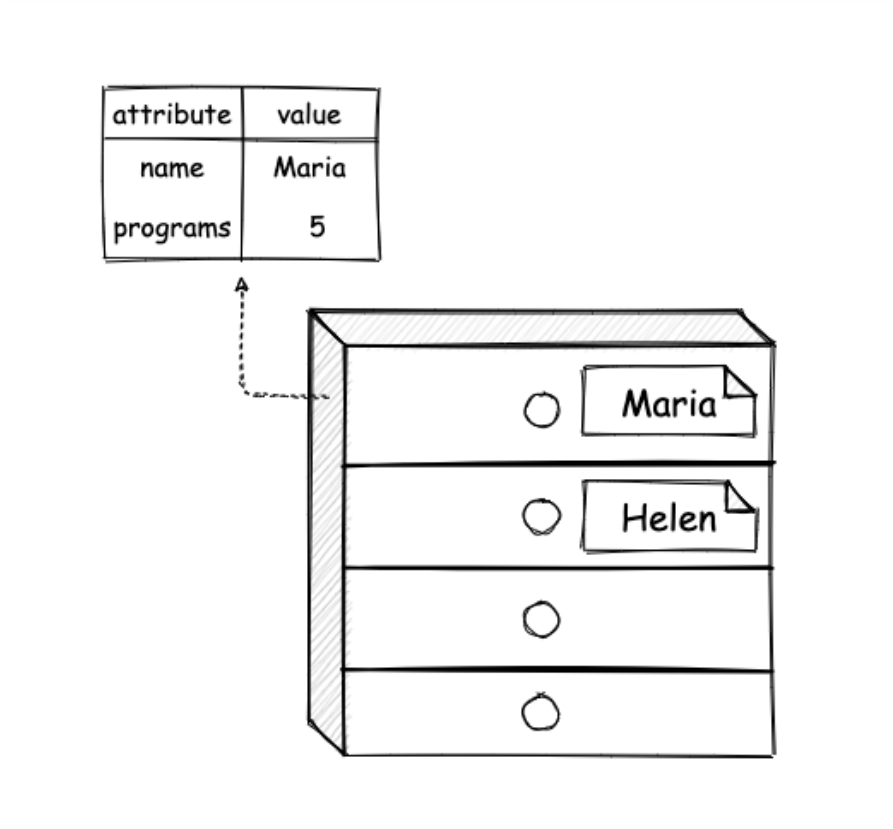
Feel free to study the code to understand exactly what it does. You can also compare with the previous program and see if you like this one better. They both achieve the same purpose, but in a different way. How the code looks like might seem irrelevant to you now as long as it performs the task you need. However, it is very important as you start working on the same code with other people, since they also need to understand your code and also change it in order to improve the program.

Before completing this chapter, I want to show you one more data structure that can be very useful, **dictionaries**. A dictionary contains pairs of data, known as "key - value" pairs. The first item of the pair is the key and the second is the value for this key. Dictionaries make it easy to store data in an organised way and search easily for the value of a specific key. For example, we could use a dictionary to store the students of a class organised by their names. This would make it easier to find a student without having to search a full list. Here is some code showing how to create an empty dictionary, add students by their names and then search a student by their name:

```
students_dictionary = {}
students_dictionary["Maria"] = Student("Maria", 5)
students_dictionary["Helen"] = Student("Helen", 6)

helen = students_dictionary["Helen"]
helen_programs = helen.programs
print("Helen has written " + str(helen_programs) + " programs." )
```

If that helps, you can think of dictionaries as a chest of drawers, where each drawer contains a label with the key outside and the value inside it.



Let's recap

First of all, congratulations for making it this far! Let's review what you have learned so far:

- We can use programming languages in order to give instructions to a computer, so that it can perform the tasks we need.
- In a program, we can have simple data types, such as numbers or strings, but we can also have more complicated data structures, such as lists and dictionaries. We can also create our own types, by defining classes.
- A programming language provides us with structures we can use to do complicated tasks. For example, we can use `if` statements to perform different tasks depending on whether something is true or not. We can also use `for` loops to execute the same task many times.

Challenge

Here is an extra challenge, in case you want to try something harder. Build a program that will do the following:

- It will receive the number of students in a class. If the size of the class is smaller than 1, the program will do nothing. Otherwise, the program will follow the steps below.
- It will ask the user to provide some data for every student of the class. This data will contain the name of the student and their grade in three different classes: English, Geography and Mathematics. The grade will be a number from 0 to 10.

- The program will calculate the sum of all grades for each student and will find the students with the highest and lowest sum. It will print these students on the terminal.

To give you an idea, when you run the program it will look like the following:

```
$ python3 first_program.py
Give me the number of students in the class: 3
Give me the name of the next student: Alex
Give me the grade this student got in English: 6
Give me the grade this student got in Geography: 7
Give me the grade this student got in Mathematics: 5
Give me the name of the next student: Maria
Give me the grade this student got in English: 6
Give me the grade this student got in Geography: 4
Give me the grade this student got in Mathematics: 8
Give me the name of the next student: Helen
Give me the grade this student got in English: 9
Give me the grade this student got in Geography: 8
Give me the grade this student got in Mathematics: 9
The student with the highest grade is: Helen
The student with the lowest grade is: Alex
```