

## Chapter 2: Getting started

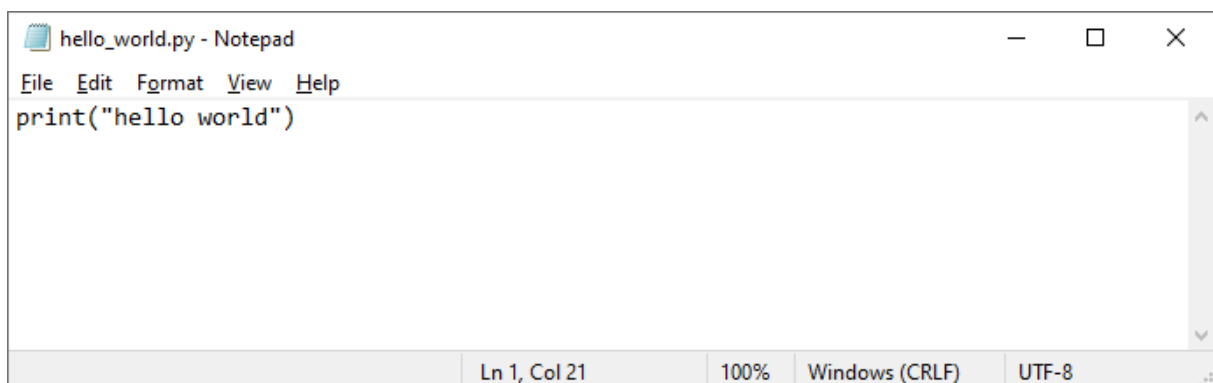
Before we get started with programming, it is important to consider the tools that we need. The first tool is a *text editor* for writing code computer. In principle any text editor, even something as simple as Notepad-type editor, could suffice. However, when writing code, it is useful if the editor comes with more features that help with and speed up the process of writing and debugging code. Therefore, we will use what is called more specifically a *code editor*, which is basically a text editor adapted for writing code.

The second tool is a code *interpreter*. Computer code can be written in many different languages. Some of the more popular ones are C++, C#, Java, and of course Python. In each case, the language is still human-readable and will still include English words as part of its syntax. Before being executed by a computer, this code needs to be translated in a format that the computer understands (i.e., it will not have English words anymore). This is the job of the interpreter. So, when we say that we are executing Python code, what we mean is that we will use the Python interpreter to read and translate our Python code into machine language, after which the program will be executed by the computer.

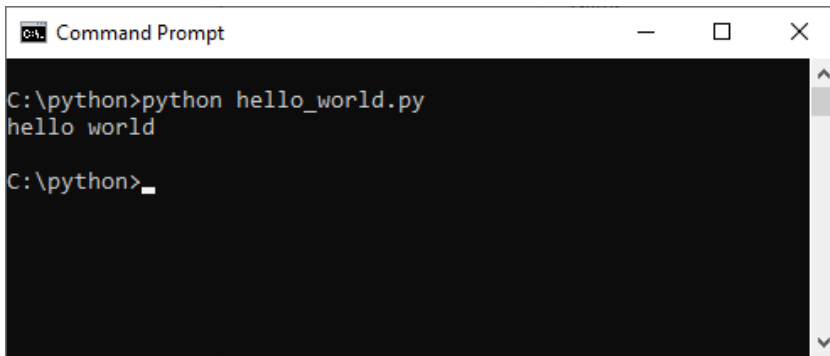
In the following sections we will show three different approaches for writing and executing Python code. The first two approaches are meant to illustrate what you miss when you are not using a full Integrated Development Environment (IDE), which will be introduced as the final approach. You can continue to that section, but having an overview of the different alternatives can be useful for gaining a perspective on how writing code is related to the tools that you use to write code.

### Back to the basics

The most basic setup to get started with coding is a combination of the Python interpreter (which can be downloaded from the website [python.org](https://python.org)), and a notepad-like editor to write your code. Figure 1 shows how we have used Notepad to write a Python program, to which we also refer as a Python script. The one line of code that is in it is the Python instruction for printing the text “hello world” on the screen. Python code files also use the .py file extension when they are saved.



To run, or execute, this program with the Python interpreter, we can open a command prompt window (on Windows: press the Windows key and type Command Prompt followed by the enter key). In this window, navigate to the location where the Python program is stored and give the Python interpreter the instruction to execute the script by writing `python` followed by the name of the script. This process is shown in figure X, and you can see that after instructing Python to execute the script, we get the expected output.



```
C:\python>python hello_world.py
hello world

C:\python>
```

[Historical note]

Traditionally, writing a hello world application is usually the first thing that you do when learning a new computer language. In some languages, even this simple program might already require many lines of code! Luckily, Python is a very accessible language, and it is probably not too difficult to understand how the code works.



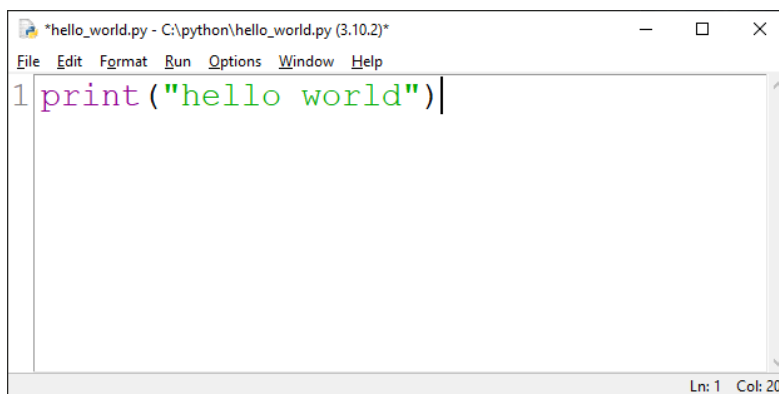
There are several drawbacks to using Notepad as our code editor. First, it is not very practical if you write an explicit instruction each time you want to run your code. It would be much easier if we simply had a button that we could click.

Notepad also has none of the handy features that make writing code more pleasant. For example, when your code has an error, Python will let you know by telling the line on which the error occurs. To find that line in your code, it would be useful to have the line numbers right in front of your code. Reading code is also easier if parts of the code receive colors, a feature called syntax highlighting.

Finally, if we can combine the functionality to run the Python interpreter into the code editor, we finally arrive at what is referred to as an integrated development environment.

### [IDLE: a basic integrated development environment](#)

If you have downloaded the Python interpreter from their website, you will also have access to IDLE, Python's own integrated development environment. The environment has two parts. The Python Shell is the command line tool in which you can enter single lines of code which will at once be executed by the interpreter. You can also open a text editor to write a whole script. Figure X shows the program that displays the hello world message again.



The left margin now has line numbers as an easy reference. Words that are part of the Python language (i.e., Python syntax), are colored purple, and a green color has been used for the text value within the print statement. In addition, we can simply run this code navigating to the Run menu and selecting the Run Module option (or even using the keyboard shortcut F5).

Working with the default Python installation still has some drawbacks. Perhaps the most important one is that in most code you make use of Python modules, but most modules are not included with the default Python installation. Therefore, we will switch to using a Python distribution.

### [The Anaconda Python distribution](#)

The main advantage of working with a distribution such as Anaconda is that it comes preconfigured with a large set of modules (or Python packages). Modules can be thought of as extensions of the core Python language that help with solving specific tasks. For example, if you want to create a graphical plot of data that you have, a plotting module can take care of most of the work. Or if you want to perform numerical operations on large sets of data you can use the numpy module which implements algorithms that are designed to perform these calculations in the most efficient way. In both cases, the big advantage of using a module is that it abstracts away most of the technical details for the tasks you want to perform. For example, with a plotting module you only need to supply a set of x and y-

coordinates, and the module will take care of creating a figure and make sure that everything looks nice.

One of the main challenges when installing a module is *dependency management*. Most modules depend on other modules themselves. Most modules also improve and change over time. It could therefore happen that you install a module A which depends on module B. Module B could already be present on your computer because you needed it in the past when you installed module C. But module A needs a more recent version of module B. You can see how this can easily get out of hand. In a distribution this dependency management will already have been taken care of.

The Anaconda Distribution comes with an integrated development environment called Spyder. It has all the features that we already mentioned (code completion, syntax coloring, debugging support, ...). It also has features that are useful in a workflow focused on data-analysis, such as integrated plots.

### Overview of the Spyder IDE

After launching the Spyder IDE for the first time, you will see a window with a layout like the one presented in figure X. There are three key areas in this screen: The IPython console window (1), the code editor (2) and the help window (3). Let's discuss each of these parts in turn.

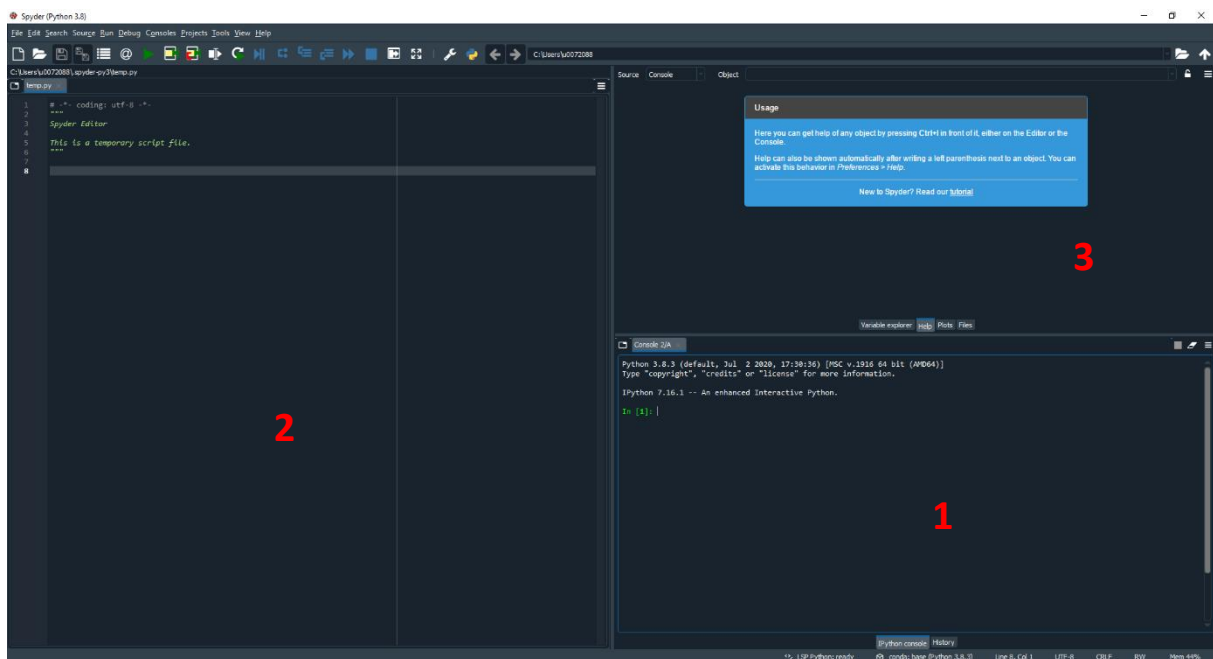


Figure 1 The Spyder IDE. The numbers correspond to the IPython console (1), the code editor (2), and the help window

### IPython console window

The IPython (which stands for Interactive Python) console window is a live Python interpreter, like the Python Shell window discussed previously. At the prompt you can enter a single line of code and pressing return will execute that line. Try writing the instruction: `print("hello world")`, and observe how this produces the text `hello world` (just like the program we wrote earlier). Every time

you use a print statement (whether directly at the prompt or in a script as we will do later), this is where its output will appear. Python syntax can also be appended here by a question mark, and running this line will usually print a Docstring in the console window. A docstring, or documentation string, is a brief description that explains how a specific piece of Python syntax can be used.

#### *Code editor*

The code editor is where we will Python programs or scripts. A newly created script will already have some text formatted as a comment on top. This is where you would have a small description of the code's purpose.

#### *Tools window*

This window has several auxiliary tools that can be accessed across for tabs. The help tab can be used to look up information on specific Python keywords (a sort of dictionary for the Python language). The variable explorer tab gives an overview of all the active variables that are present in a script, as well as their type and corresponding values. You can check that if you create a new variable in the console, it will pop up in this list. There is one more tab for showing plots (but there is an option that can be configured so that plots are always displayed as a separate window). The final tab is an integrated file explorer.

### Diving into Python

After covering the tools, we can finally dive into some real programming! In this section we will start with Python code that can be entered in the IPython console window so that you get an immediate result. Although this is probably not too exciting, let us start by using Python as a calculator by entering the following instruction: `1 + 1`

This will produce the number 2 as a result in the console window. More typically we would like to store the results of any kind of calculations in a variable. That way we can easily refer to the result of the calculation later in our script, without having to perform the actual calculation again. The line of code below has the same instruction, but now the outcome will be assigned to a variable named `result`.

```
result = 1 + 1
```

Because we are assigning the result of this calculation to a variable, this line of code will not produce any output. Instead, a variable will have been created. You can verify this by now simply entering the name of the variable as an instruction, which will show the content of that variable in the console window. Alternatively, you can go to the Variable Explorer tab in the tools window. This will show the name of all created variables and their respective values.

```
division_result = 5/2
```

In addition, it has a 'type' column, which shows what sort of data is stored in the variable. Natural numbers are represented in Python using the integer or *int* type. If you do a division, the result will be a real number which in Python is called a *float* type. Textual data is stored in a *str* (*string*) type variable. To assign textual data, you place the actual text between either single quotes or double quotes. Which one you use does not matter, but the opening quote must be the same as the closing quote.

```
my_name = "Christophe"
```

### Type conversions

Data can occasionally be stored in a type not suited for the operations you want to perform on it. For example, we can store the number two both as an integer type as well as the string "2":

```
my_number = 2
```

```
my_string_number = "2"
```

Although you might initially think that both variables have the same value, the way they are represented in Python makes them fundamentally different. You can confirm this by asking Python to compare both variables with each other. To make this comparison, we use two equality symbols (==):

```
my_number == my_string_number
```

Python will return the value `True` if the content of both variables is the same, and the value `False` if that is not the case. After creating the two variables above, run the line of code and see what result you get. Is it what you expect?

The variable types determine what type of operations can be performed with them. When `a` and `b` are both integer type variables, we can perform mathematical operations on them by writing `a + b`, or `a - b`. But what if `a` and `b` are both string type variables that have numbers formatted as text? Would the same mathematical operations still have the same effect? The answer is no, and it is a good exercise to play around in the console window to see which operations also apply for text.

If you have numerical values stored as string types, but you need to work with them as actual numerical values, you can perform a type conversion. In the line of code below, the `int()` function has been used to convert the value stored in the string variable into an integer type that is now stored in the variable `a`. For this operation to be possible, the string variable does need to have an integer. If not,

the conversion will fail, and an error will be raised (we will see how to handle this situation in later chapters). The second line shows that the conversion can also be done in the other direction.

```
a = int(my_string_number)
```

```
b = str(my_number)
```

### **A note on naming variables**

Choosing proper variable names is important. Studies have shown that on average, a programmer spends ten times as much time reading code than writing code (because we need to fix errors or try to find better ways to solve a problem). Therefore, we should do everything we can to make our code as readable as possible.

Variable names should be chosen so that they reflect what that variable means within your program. For example, the number 18 can be assigned to a variable labeled `age` if that is what it means in that program. But in another program, the number 18 could reflect the number of pages in a book, and `n_pages` would be a better name. In both cases, using single letters such as `x` or a general name such as `number` is less meaningful.

There are a few cases where convention deviates from this rule. When dealing with spatial coordinates one can use the usual mathematical names such as `x`, `y` and `z`. Likewise, when dealing with indices in data structures such as `list` (which will be covered in later chapters), the use of the letters `i`, `j`, and `k` is common.

### **A note on writing variable names**

Once you have decided on a name, there are still several options for writing out that name. For example, if you have a variable to hold the age of a user, will you write `userAge`, `userage`, or `user_age`? The first way is called *camel case* labeling. It uses lower case characters for the first word. Later words are added by capitalizing the first letter of each word.

The last method is referred to as *snake case* labeling. It uses only lower-case characters, and spaces are represented by using underscores. This is in fact the recommended style according to the Python style guide and is therefore the style that we will use in this book. Using a consistent style throughout your code is also another important way to improve readability. And although we do not want to force you to use a specific style, there is in fact some research that suggests that snake case labeling has the most benefits when it comes to readability.

Sometimes you will see that variables are typed using all uppercases. This is another Python convention that is used for variables for which we assume that their value will not change once they have been created. For example, the background color of an application could be indicated with the variable `BACKGROUND_COLOR = "green."` In Python, this does not prevent you from assigning a new value to this variable (in some languages, declaring one as constant will raise an error when you try to change it). It only acts as a visual reminder for the person working on the code that the content of that variable is not supposed to be changed.

## Diving deeper into Python

In the following section, we will revisit the code from the earlier section and introduce technical jargon for the operations we have performed. The reason for treating this in two separate parts is that the technical terminology can make things sound more complicated than they are. Even without knowing the right terminology, you had no difficulties understanding the code in the earlier section. But knowing the right terminology can be helpful in situations where your code produces an error (which it usually will). That error message will use technical jargon, so being familiar with it will help you fix errors more easily.

### Operators, operands, and expressions

Let us retake the first line of code we wrote:

```
1 + 1
```

In Python and other programming languages, the addition symbol is called an *operator*. Python has six categories of operators: *arithmetic operators*, *comparison operators*, *logical operators*, *assignment operators*, *membership operators* and *bitwise operators*. In this chapter we will cover arithmetic, comparison, logical and assignment operators. The membership operator will be discussed in the chapter on data structures. The bitwise operator is beyond the scope of this book.

Operators need to work on something. In Python, we refer to the value on the left and the value on the right side of the operator as the *left operand* and the *right operand*. The operators and the operands together form an *expression* (here, an arithmetic expression). Loosely speaking, an expression can be thought of as code that produces a value when it is executed by the Python interpreter. Python even has the concept of a *literal expressions*, which means to any value by itself (e.g., 5 is a literal expression).

### Arithmetic operators

Arithmetic operators are used to write arithmetic expressions. The different operators are listed in table 1. If your mathematics is a little bit rusty, the modulus operator might need some refreshment. It goes hand in hand with the floor division operator, and both deal with the realm of integer numbers.



If you have the integer number 5 and perform a floor division with the number 2, you will get 2 as a result. This is because using integer division, the number 2 fits exactly 2 times in the number 5. But this leaves us with a rest of 1, which can be obtained by performing the modulus operation  $5 \% 2$ . The mathematical background is that for a given number  $y$  and a divisor  $x$ , the number  $y$  can be expressed as

$$y = ax + b$$

In this equation, the value  $a$  is produced by performing the floor division  $y // x$ , and the value  $b$  is produced by the modulus operation  $y \% x$ . Other than being a mathematical technicality, there are also some useful applications of this operator. For example, you can use it to check if a number is divisible by another number (think what the result of the modulo operator will be here). This means that in an experiment with many trials, you could use this operation to pause every  $N$ th trial (by checking if the current trial number is divisible by  $N$ ).

OPERATION	OPERATOR
Addition	+
Subtraction	-
Multiplication	*
Division	/
Floor division	//
Exponentiation	**
Modulus	%

Table 1 An overview of the arithmetic operators

### Comparison operators

Once a variable has been assigned a value, we typically want to perform checks on its value. For example, before letting people take part in an experiment, we might ask for their age and allow only participants within a certain age range. Another example is a login system. After providing a username and a password, you will want to check if the password matches a stored password that corresponds with the username. These checks can be performed using the comparison or relational operators (table 2). They form expressions that result in a truth value or a Boolean value. A Boolean value is either `true` or `false` and reflects the relation between the left and right operand established by the operator holds.

OPERATION	OPERATOR
Equality	==
Inequality	!=

Smaller than	<
Larger than	>
Smaller or equal to	<=
Larger or equal to	>=

Table 2 An overview of the comparison operators

### Logical operators

Logical operators are used to evaluate the relationship between truth values (this means that the left and right operand are always Boolean values). Suppose we have created an `age` variable. We want to verify that the value of this variable lies in the interval [5, 8]. This is true if the value is both larger than 4 and smaller than 9. We write the expressions for each individual comparison, and then we use the logical and-operator to verify that both relations hold simultaneously:

```
age > 4 and age < 9
```

Let us decompose the expression above in small steps to see how Python interprets it. Suppose that the value of `age` is 6, then the first thing that happens is that the name of the variable will be replaced by the actual number:

```
6 > 4 and 6 < 9
```

Evaluating the comparisons will lead to the following

```
True and True
```

At this point, you can use the truth tables to check what the result will be. The `and` operator will return True only if both the left and the right operand are true at the same time. The `or` operator will return true if either the left or the right operand (or both) are true.

LEFT OPERAND	RIGHT OPERAND	RESULT
True	True	True
True	False	False
False	True	False
False	False	False

Table 3 Truth table for the and operator

LEFT OPERAND	RIGHT OPERAND	RESULT
True	True	True
True	False	True
False	True	True
False	False	False

Table 4 Truth table for the or operator

A third operator, the `not` operator, can be used to 'switch' a Boolean value. That is: `not True` will return `False` and vice versa, `not False` will return the value `True`. For example, if you do not want participants in an experiment to be older than 12, you could check the condition

```
age <= 12
```

Alternatively, you could also evaluate the condition

```
not age > 12
```

Both expressions result in the same value and are in that sense equivalent.

### Assignment operator

The assignment operator is used to assign values to variables. It uses a single equality symbol (as opposed to the double equality symbol when making comparisons). The assignment only works in one direction. That is, the value is always taken by evaluating the right operand and assigning it to the left operand:

```
result = 1 + 1
```

### Writing your first script

The earlier section could be followed by writing out each line of code in the IPython console. We will now switch to using the code editor and start writing more elaborate programs. To execute this code, we now need to give explicit instructions to run it. If we want any output to appear, we now also need to give an explicit instruction to print this output in the IPython window. That is, writing `1 + 1` in the console window will produce the result 2. But writing `1 + 1` in the code editor will not show this result. Here we need to give the instruction `print(1 + 1)`.

A program can also take input from a user. This can be achieved by using the `input()` function. Between the parentheses you can supply a string that will be shown as a prompt in the console window. The user can now type something, and by pressing enter the control is handed back to the code in the script, and anything that the user has typed is assigned to a variable. This is shown in the code below:

```
1. print("Welcome to WelcomeApplication v1.0")
2. user_name = input("Please enter your name: ")
3. greeting = "hello, " + user_name
4. print(user_name)
```

Listing 1: A program that provides a welcome message to a user

Before running this code, make sure to read it first and that you understand what each line of code does. As already mentioned, people typically spend significantly more time reading code than writing code. It is therefore important that in addition to learning specific Python syntax and terminology, you also become fluent at reading and understanding code examples. It can therefore be informative to first try to predict what a piece of code will do so that you can test your interpretation against the actual output. It can also be fun to just play around with the code by making some slight changes and seeing how they affect the output of the program, and if this change in output is what you predicted.

### A final comment

A typical script will also have comments in addition to the Python code. Single line comments are started with the # symbol. Multiline comments start with three double quotes and are closed with three double quotes.

```
1. # Print a welcome message
2. print("Welcome to WelcomeApplication v1.0")
3.
4. # Request the name
5. user_name = input("Please enter your name: ") # This is an inline comment
6.
7. # Construct the greeting text
8. greeting = "hello, " + user_name # Produce a welcome string
9.
10. # Display the greeting text
11. print(greeting) # Display the greeting string
```

*Listing 2 Our Welcome program extended with comments*

Comments can be used to add documentation to your code, and there are [official guidelines](#) for how to construct and format comments in Python. They can also be a useful tool when you are starting out to learn about programming. In that case, you could use them to add annotations, or you could use them to set up the structure of your code in comments first and then fill in the actual code in a later stage based on the structure that you have provided.

In professional environments, the comments that are present in the example below would be considered redundant as the meaning is quite clear from the code itself. If you produce code that needs to be shared with multiple people, it is indeed a good practice to clean this up and make sure that your comments add something to the clarity of the overall code. But if you are not in that situation, consider the editor your canvas and add any comments that you find useful for yourself.