Introduction to Python

dr. ir. Pieter-Jan Volders

February 17, 2020

Contents

1 Introduction to programming with Python

1.1 dr. ir. Pieter-Jan Volders

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

```
Hello world!
```

2 What is Python?

Do you speak computer? Python is a programming language, a language we can use to make computers do what we want. There's hundreds if not thousands of programming languages you can learn. So what's special about this one?

> Python is a widely used high-level programming language for general-purpose programming. - *Wikipedia*

So Python is popular, big deal? Yes! Because it is widely used, it is very easy to find example code and tutorials online. There's also plenty of books on Python and a large community of developers that work on improving the language. Programmers can contribute to the Python universe by writing packages that other programmers can use. A very active community means there's a lot of packages out there. In other words: if you have a *thing* you want to do in Python, it is very likely somebody else already did the *thing* and you can just use their code.

The Wikipedia quote also mentions that Python is a *high-level* programming level. Don't worry if you are afraid of heights, in computer terms the higher you are, the less you have to worry about what is happening with the hardware (the hardware level is the lowest level). In a *low-level* programming language you, for instance, have to define how and where you will be storing data in the memory of your computer. Programmers use *high-level* programming languages so they can focus on solving their problem quickly while *low-level* languages are used to optimise the speed and efficiency of a certain task.

Python can be used for a wide range of applications. From app and website development to manipulating robots and more. More important for us, it is very popular in scientific computing through packages such as NumPy, SciPy and Matplotlib. There's even a package called Biopython that makes it easy to work with DNA and RNA sequences from databases and sequencers.

The previous quote however, can also be applied to other programming languages such as Perl, Ruby or JavaScript. So what distinguishes Python from those languages? Here's a quote from Mark Lutz, an author who wrote several books on Python.

> Python is a powerful multiparadigm computer programming language, optimized for programmer **productivity**, code **readability** and software **quality**. - *Mark Lutz*

2.1 Code readability

Some people say that good code reads like a book. While this is mostly the task of the programmer him/herself, the language definitely helps! Have a look at the following bit of Python code. Can you guess what it does?

```python
for letter in "welcome":
    print( letter.upper() )
```

```
W
E
L
C
O
M
E
```

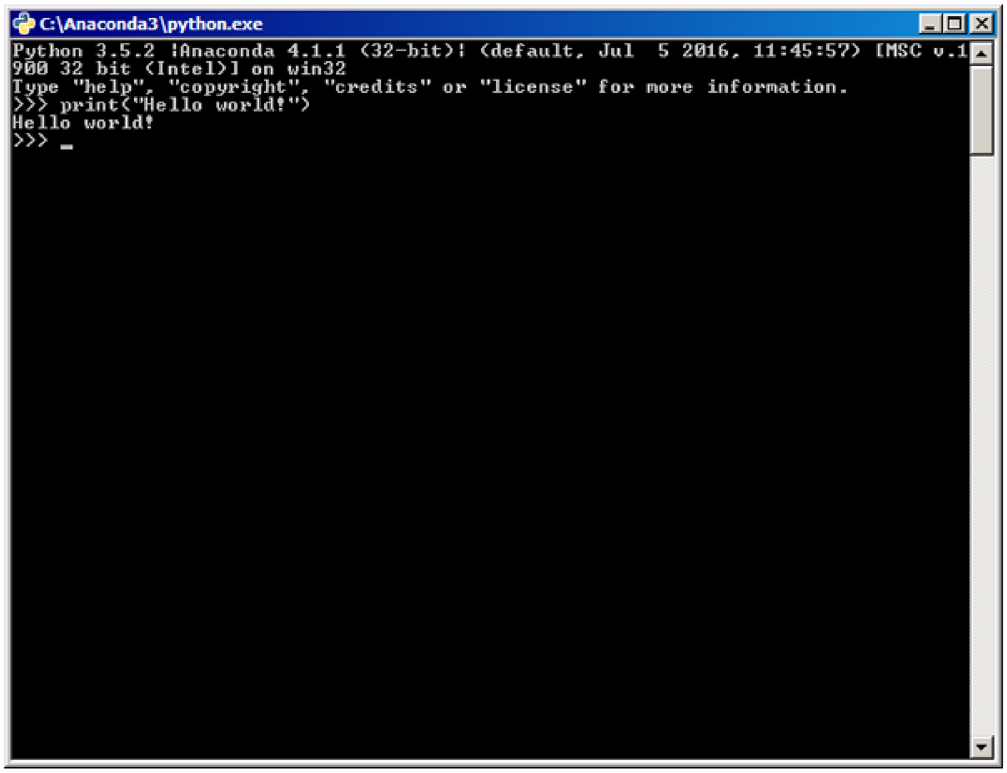
Eventhough this might be the first piece of Python you ever see and this small example already includes some advanced concepts such as object-oriented programming, it is pretty easy to grasp its function. Let's read it in human language:

> For every letter in the word welcome, print that letter uppercase.

Below is the same thing in a different programming language called C. Do you see the point of code readability? Don't stare at it for too long, let's move on and appreciate the simplicity of Python!

```c
#include <stdio.h>

int main () {

  char my_string[] = "welcome";
  int i;
  char i_char;

  for ( i = 0; my_string[i] != '\0'; i++ ) {
    i_char = my_string[i] - 32;
    printf("%c\n", i_char);
  }

  return 0;
}
```

Having *readable* code is not only helpful for beginner programmers. It makes it easier to share your code and come back to work you have written ages ago.



```
Python 3.5.2 |Anaconda 4.1.1 (32-bit)| (default, Jul  5 2016, 11:45:57) [MSC v.1
900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world!")
Hello world!
>>> _
```

Python 3 prompt on Windows

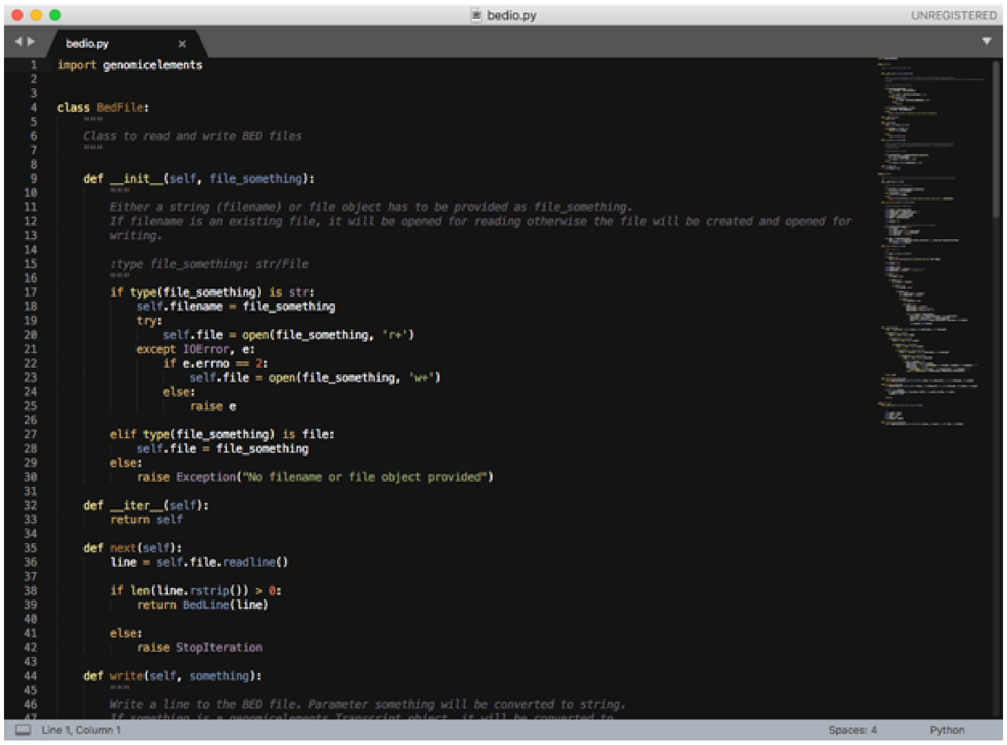
3 How to write and run Python programs

3.1 The Python prompt: running code interactively

The interactive Python prompt or *command-line interface* is the most basic way to run Python code and is available on all systems where Python is installed. You can start an interactive Python session by typing `python` in the Terminal (Mac & Linux) or Command Prompt (Windows) or click the appropriate icon if available. When a new session is started, it prompts for input with >>> and executes your code as soon as you press the Enter/Return key. The result (or *output*) of your code is displayed immediately below the input line and a new prompt (>>>) appears. Variables are stored until the session is closed by typing `exit()`, pressing `Ctrl-D` or just closing the window.

We will discus multiline statements and blocks such as `if` tests of `for` loops later, but it is worth noting here that in order to end such multiline statements in the Python prompt you must press the Enter key twice.

The Python prompt is useful for experimenting with Python or testing small chunks of code. It is not so useful for writing actual programs or scripts since there is no way to store your code or even modify a previous statement. Imagine you discover you've made a mistake on your first line when you are working on your 100th... Nevertheless, it is a great tool for learning or quickly trying out complex statements. Even some advanced Python developers like to keep a prompt at hand while they are programming!



```python
import genomicelements


class BedFile:
    """
    Class to read and write BED files
    """

    def __init__(self, file_something):
        """
        Either a string (filename) or file object has to be provided as file_something.
        If filename is an existing file, it will be opened for reading otherwise the file will be created and opened for
        writing.

        :type file_something: str/File
        """
        if type(file_something) is str:
            self.filename = file_something
            try:
                self.file = open(file_something, 'r+')
            except IOError, e:
                if e.errno == 2:
                    self.file = open(file_something, 'w+')
                else:
                    raise e

        elif type(file_something) is file:
            self.file = file_something
        else:
            raise Exception("No filename or file object provided")

    def __iter__(self):
        return self

    def next(self):
        line = self.file.readline()

        if len(line.rstrip()) > 0:
            return BedLine(line)

        else:
            raise StopIteration

    def write(self, something):
        """
        Write a line to the BED file. Parameter something will be converted to string.
        If something is a genomicelements Transcript object, it will be converted to
```

Sublime Text 3 on MacOS

3.2 Writing Python with a code editor or IDE

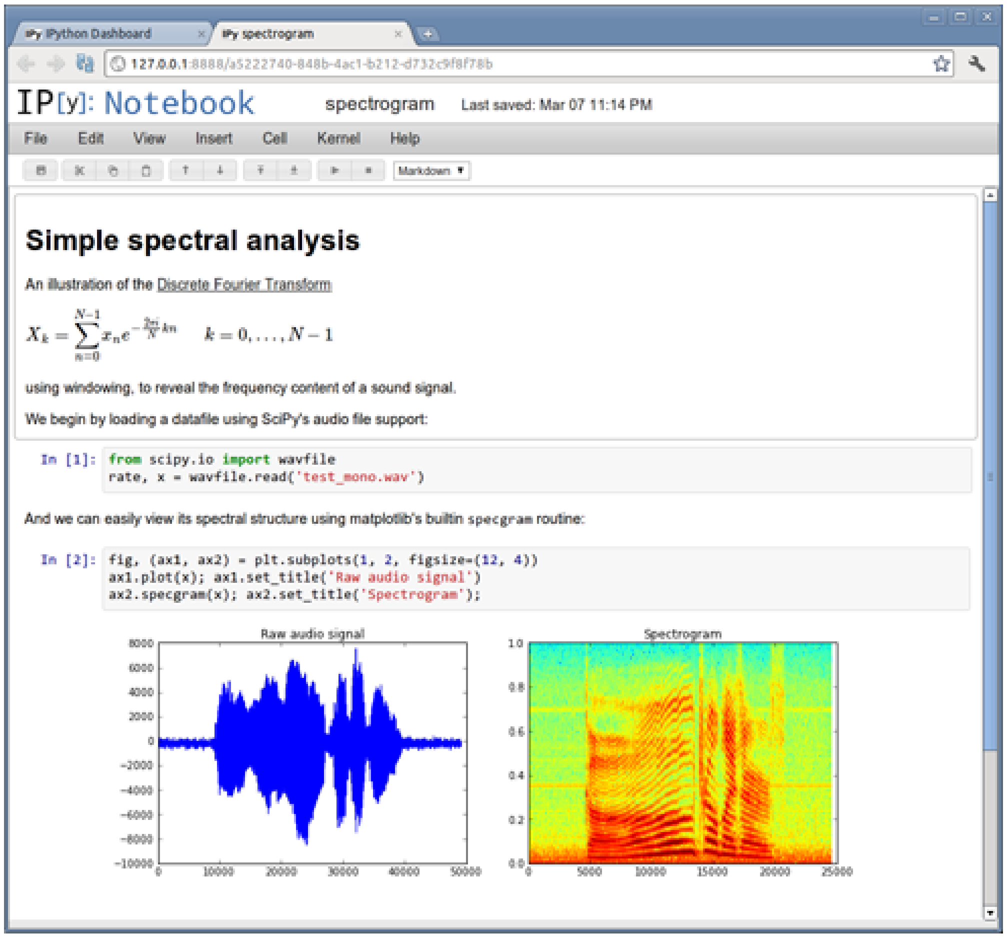
Since Python code is stored in a plain text format (think of it as a `.txt` file) you can write it with any editor, even notepad or good old MS Word. It is however much more convenient to use editors specifically designed for programming, or *code editors*. These programs assist you in writing and structuring your code, make programming faster by auto-completing and give visual cues that help you spot mistakes.

An important feature of code editors is syntax highlighting. Take a look at the code examples in the Section 2.1. Note how parts of the code have distinct colors and typography. As a result, control flow statements such as `for` appear very different from strings such as `"welcome"`. When you learn how to program, you will automatically learn how to use these visual cues and spot typo's in your code.

Python code is saved in `.py` files. While some code editors have the option to run the code directly in the editor, most programmers will run their code directly using a command-line interface. This however requires some knowledge of using terminals and command prompts and is out of the scope of this introduction. The Python installation manual is a good place to start when researching how to install Python and run Python programs on your own computer.

Popular code editors are Sublime Text, Atom and Notepad++. All these support Python and many, many more languages.

For large Python projects, the really hard-core Python developers will use advanced software



Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

using windowing, to reveal the frequency content of a sound signal.

We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile
        rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

```
In [2]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```

Jupyter notebook with Python code

called integrated development environment or IDE. Compared to code editors, these programs help manage software that consists of multiple Python scripts and are generally have even smarter autocomplete and code annotation features. Notable examples are Eclipse and PyCharm.

3.3 Jupyter notebooks

Jupyter notebooks (previously called iPython notebooks) are a great tool for learning Python. You can write, test, edit and run Python code without using any external software. In addition to code, notebooks can contain text with markup, mathematical formulas and images. They are widely popular in data science since you can store the code, output and written report in the same file! You can even export your notebook as a presentation or PDF. In fact, you are reading an exported notebook right now!

The most convenient way to install the software required to open and create Jupyter notebooks is through the Anaconda platform.

A bowl of delicious ramen

4 Elements of Python

All this talk about Python! Enough already! Let's get programming!

What follows is by no means a complete guide or manual to Python, merely an (incomplete) overview of the different elements that make up the language. Think of programming as cooking. Today we will go over all the ingredients we need and in the practical sessions, you'll learn how to combine them into delicious dishes. Ready to make some awesome Python soup? Here's your shopping list:

- Variables
- Lists and dictionaries
- Functions
- Methods
- Tests
- Control flow statements
- Errors
- Libraries

4.1 Variables

4.1.1 First steps

Let's write our first Python program. Something simple like

```
1+1
```

```
2
```

That was simple right? So simple it is hard to believe it is an actual program. But it is! The code I just wrote was processed by the Python interpreter and the output is returned and displayed on the screen. Python is great calculator, `+`, `-`, `/`, `*`, ... it is all there! Math is important in programming, but programs are much more than a list of calculations. The first thing we'll need to go from a simple calculation to a multiline program is a way to store *values*, so we can use them later in our program. That's where variables come in.

```
result = 1+1
```

We have now stored the result of our calculation in a variable. This process is refered to as *variable assignment* and happens in the form of `x = expr` where `x` is the name of the variable and everything right of the equation mark an *expression*. The expression will be evaluated and the result will be assigned to the variable name. We can now use and reuse this variable:

```
result = result + 4
result
```

```
6
```

Let's go over what happened here. First, we have used our variable `result` in an expression (`result + 4`) and assigned the result of the expression to the same variable name. This means our first result (2) is lost, it's overwritten so to speak. On the next line we evaluate the variable to see it's current value.

4.1.2 Variable names

A variable name is a combination of letters, numbers and underscores. Other characters, such as @, $, %, - and spaces are not allowed.

```
this_is_ok = 10
this is not = 10 # ERROR!
```

Variable names cannot start with numbers

```
variable1 = 10.5
1variable = 10.5 # ERROR!
```

Case matters, these are two different variables

```
spam = 2
Spam = 2
```

Reserved words such as `if`, `not`, `and`, ... are not allowed as variable names.

```
if = 456  # ERROR!
```

Reserved words are words that serve important functions in the Python language and cannot be used as variable names to avoid confusion. Later in this introduction, we will encouter some of these words. There is no need to learn them by heart, syntax highlighting is here te help! Can you see that the `if` in the previous example has a different color then the other variable names? This is syntax highlighting telling you "`If` is an important word in Python, you better not try to do anything funny with it".

There's no error when you use function names as variable names but this will overwrite the function!

```
print = 11001  # no error
print("help")  # ERROR!
```

When a function is overwritten, it can no longer be used. We will talk about functions later.

Meaningful variable names are encouraged.

```
a = 10
my_favorite_number = 10
```

4.1.3 Basic variable types

So far we have only used numbers as values for our variables. Off course, there are other options as well. Variables thus have a *name*, a *value* and a *type*. Let's create a few variables and check their types with the `type()` function.

```
variable_a = 1
variable_b = 1.1
```

```
type(variable_a)
```

```
int
```

```
type(variable_b)
```

```
float
```

`variable_a` and `variable_b` are both numbers, but their types differ! `variable_a` is an `int`, wich is short for *integer*. 1, 10 and -5 are examples of integers. `variable_b` on the other hand is a `float`, wich is short for *floating point number*. 1.1, 3.1415 and -5.004 are examples of floating point numbers. Note how we use a dot as the decimal separator and not a comma! In Python, we do not have to worry when two variables have different numeric types and we can use them together in calculations.

```
variable_a + variable_b
```

```
2.1
```

Strings In addition to numbers, we can also store text in variables.

```
variable_c = 'Hello there!'
type(variable_c)
```

```
str
```

The variable type to store text is `str` or *string*. Strings always start and end with quotation marks to avoid confusion with actual code. You can use both single quotation marks `'` and double quotation marks `"`. Some examples of strings:

```
string_1 = "None shall pass"
string_2 = 'None shall pass'
string_3 = "It's just a flesh wound"
```

> Note: you can use single quotation marks in double quoted strings and vice versa.

Variable types are important! Different types of variables will behave differently:

```
"1" + "1"
```

```
'11'
```

Using an addition operator (+) on strings will *concatenate* them: combine the two strings into one. It is thus important that you keep track of the types of your variables, or unexpected things might happen!

Boolean Boolean is the most simple variable type, it has only two possible values: `True` and `False`.

```
variable_d = True
type(variable_d)
```

```
bool
```

If you are wondering why this variable could ever be useful: we will later see that booleans play an important role in Python (and any other programming language).

`True` and `False` are always capitalized and writting without quotation marks. If we were to use quotes, we would create a string instead!

```
variable_e = "True"
type(variable_e)
```

```
str
```

4.1.4 Compound variable types

Lists Python knows a number of compound data types, used to group together other values. The most versatile is the `list`, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
list_of_things = ["a shrubbery", "another shrubbery", "a herring"]
type(list_of_things)
```

```
list
```

We can access a specific element in the list using `listname[index]`. The *index* is a number, starting at 0. So position 2 is actually the third element in the list. Indexes can be used to read or write elements of a list.

```
list_of_things[1]
```

```
'another shrubbery'
```

```
list_of_things[2] = "the mightiest tree in the forest"
list_of_things
```

```
['a shrubbery', 'another shrubbery', 'the mightiest tree in the forest']
```

We can use *slices* to obtain multiple elements in a new list. Slices are formatted as `listname[start:end]`.

```
my_numbers = [1, 2, 3, 4, 5]
my_numbers[2:4]
```

```
[3, 4]
```

> Note: the end is non inclusive! `2:4` thus gives you the third and fourth element in the list.

If the end is omited, the slice will continue to the end of the list.

```
my_numbers[3:]
```

```
[4, 5]
```

Slices and indexes also work on strings!

```
my_quest = "The holy grail"
my_quest[9:]
```

```
'grail'
```

Dictionaries Dictionaries (`dict`) are similar to lists in that they hold multiple elements. But every element is identified by a unique *key* and the order of the elements is not important. The elements of a dictionary are typically referred to as *key-value pairs*. Dictionaries are created using braces (`{` and `}`).

```
my_friend = {"name": "Jeff", "age": 23}
type(my_friend)
```

```
dict
```

In this example, `name` and `age` are the keys and `Jeff` and `23` the values. Note how the variable type of the values can be different but keys are always strings!

Keys can be used in the same way as indexes on lists.

```
my_friend['name']
```

```
'Jeff'
```

```
my_friend['hometown'] = 'Ghent'
my_friend
```

```
{'name': 'Jeff', 'age': 23, 'hometown': 'Ghent'}
```

So far we have introduced the following variable types: `int`, `float`, `str`, `list` and `dict`. There are a few other types available in Python and you can even define your own, but let's stick to those five for now!

If you are interested to learn more about Python's built-in types or want a more formal introduction, you can check out the Python manual.

4.2 Functions

If variables are the ingredients of the Python soup, functions are the instructions of your recipe. **Dice** the carrots, **sauté** the onion and **cook** your broth!

A basic function call in Python looks like this: `some_function()`. Most functions have one ore more arguments, that we can pass between the parentheses: `some_function(arg_1, arg_2)`. Some functions will return output that we can store in a variable: `my_variable = some_function(arg_1, arg_2)`.

Let's go over some of the functions you will need:

4.2.1 `print()`

Probably the most frequently used function by beginning programmers is `print()`. It has nothing to do with paper and printers but displays text as output on the screen.

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

```
Hello world!
```

print() accepts any variable type as argument. If more than one argument is given, they will be glued together by spaces. This behaviour comes in handy when you want to display a sentence containing a variable:

```
nr_of_swallows = 2
print("It takes", nr_of_swallows, "swallows to carry a coconut")
```

```
It takes 2 swallows to carry a coconut
```

> Note: You may have noticed that some of the code in the previous sections also had output without using the print() function. This is different from printing however! Jupyter notebooks by default evaluate the last line of the code block and display the results.

4.2.2 int(), float(), str(), ...

Every variable type has its own constructor: a function that creates a variable of that type from other types of variables. Constructors are useful for converting from one variable type to another:

```
var_a = "123"
print(var_a + var_a)
var_b = int(var_a)
print(var_b + var_b)
```

```
123123
246
```

4.2.3 input()

Another useful function is input(). It stops the program and waits for input. Depending on the way the Python script is executed, this input can come from the user or from a file. In Jupyter notebooks, a small box appears that users can type in. input() has an optional argument, a string that is displayed before the program waits for input. This is useful for indicating to the user what he/she is supposed to type.

```
some_name = input("Hello! What's your name?")
print("Welcome", some_name, "!")
```

```
Welcome Arthur !
```

4.2.4 Reading function documentation

There's an incredible number of functions available in Python. After a while you will know some of them by heart, but it is impossible to memorize them all. In fact, programming is often more about reading documentation and being able to find the stuff you need quickly than it is about typing code from your head. Learning to program in a new language is thus also learning where to find documentation on the language and how to read it.

Let's have a look at the documentation for the function `round()`. The first lines of the documentation look something like:

```
round(number[, ndigits])
    Return number rounded to ndigits precision after the decimal point. If
    ndigits is omitted or is None, it returns the nearest integer to its input
```

The first line defines the function and its parameters. We can read it as follows: `round()` is a function with two parameters, one required parameter `number` and one optional parameter `ndigits`. The square brackets indicate an optional part, in this case the `ndigits` parameter. The rest of the documentation explains what the meaning of the parameters is and what happens in the presence/absence of the `ndigits` parameter.

```
my_fraction = 1/3
round(my_fraction, 4)
```

```
0.3333
```

```
round(my_fraction)
```

```
0
```

Sometimes, functions have parameters with a *default* value. For instance the `print()` function:

```
print(*objects, sep= , end=\n, file=sys.stdout, flush=False)
    Print objects to the text stream file, separated by sep and followed by
    end. sep, end, file and flush, if present, must be given as keyword
    arguments.
```

The first argument `*objects` is required and has a little star next to it. This star indicates that more than one `objects` argument can be supplied. The second argument `sep` and all the subsequent arguments are optional, but they have a default value. For `sep`, the default value is `' '` (a space). This means that if you provide more than one `objects` argument to `print()` by default they will be separated by a space in the output.

In the documentation, we can also read that these arguments "must be given as keyword arguments". We have to specify the name of the parameter when we want to set it. In the `round()` function, the order of the arguments idicates what parameter you set: the first parameter is always `number` and the second `ndigits`. Since we can supply any number of `objects` arguments to `print()`, we have to make it clear that an argument is for instance the `sep` argument instead. For instance:

```
print("Hello", "world", sep = "<--->")
```

```
Hello<--->world
```

4.3 Methods

Methods are a special kind of functions that belong to specific variable types. They are part of a programming paradigm called *object oriented programming* that provides a way to structure code in large programs or libraries. *Object oriented programming* is beyond the scope of this course so I will not go into detail. But think of it in this way: if a bunch of functions only work on a specific variable type, it would be convenient to group these functions together in one way or another. For instance, the `str` variable type has a group of functions we refer to as *string methods*. Here's an example of one of them:

```
my_string = "hahahaha!"
my_string.upper()
```

```
'HAHAHAHA!'
```

The `str.upper()` method returns an uppercase version of the string. It would not make sense to have an `upper()` function since we can only make an uppercase of a string. Therefore, `upper()` is a method of the `str` variable type class. As you would have guessed, there's also a `str.lower()` method. But many more methods are available, all available string methods can be found in the Python documentation.

Note how using a method differs from using a function: `variable_name` dot `method_name()`.

4.4 Tests

In the previous section, we have learned how to create variables and set their values. In our examples, we always knew the value of our variables. In practice however, this is often not the case!

Imagine you are programming a game. You might have a variable called `highscore` to keep track of the highest score of the player. As programmer, you have no idea what the value might be and yet you have to keep track of it. For instance: when the player beats his/her highscore, balloons have to appear on the screen and their friends have be publically shamed on social media.

Let's start by creating a variable:

```
bird = "swallow"
```

With tests, we can check if the value of our variables meets certain criteria. For instance:

```
bird == "duck"
```

```
False
```

```
bird == "swallow"
```

```
True
```

Here, we have used the == operator to test whether the value of the `bird` variable equals a specific string. `True` and `False` are Pythons way of saying yes and no and tests always return either `True` or `False`.

'==' is an example of a *relational operator*, it tests how two entities relate. In this example we've compared two strings but the same operators work on numerical and other types. Several more relational operators are available in Python:

Symbol	Test
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

4.4.1 Tests for compound variable types

For lists and dictionaries, Python provides a very useful `in` operator that tests whether a specific element is present in the list or dictionary.

```
my_list = ["one", "two", "three"]
"two" in my_list
```

```
True
```

The opposite of `in` is as simple as this:

```
"four" not in my_list
```

```
True
```

We've talked about code readability earlier. This is one of thoses examples where Python helps you to write beautiful code!

The `in` operator also works on dictionaries, but it only tests whether a *key* is present! In this case, `'pj'` is a key in the `my_ages` dictionary, hence the test returns `True`.

```
my_ages = {"pj": 30, "niels": 32}
'pj' in my_ages
```

```
True
```

```
30 in my_ages
```

```
False
```

4.4.2 Combining tests with and/or

You can combine two tests by using *bitwise operators*. The most common bitwise operators are `and` and `or`. The logical *and* will evaluate as `True` when both tests evaluate as `True`, while the logical *or* will evaluate if one of the tests evaluates as `True`.

Both tests (or expressions) we are combining have to be set apart by parentheses!

```
day = 'friday'
dinner = 'steak'
```

```
(day == 'friday') & (dinner == 'steak')
```

```
True
```

Must be my lucky day!

These are the most common bitwise operators available:

Symbol	Task Performed
`and`	Both are `True`
`or`	One or the other is `True`
`not`	Inverts the result: `True` becomes `False` and vice versa

Note: `and` and `or` can also be writting as `&` and `|` respectively. Even though theres a small difference, you will often find this notation in example code online.

4.5 Control flow statements

Now there's a fancy word! The *flow of control* in a program is the order in which individual statements are executed. Normally, every line in a Python program is excecuted one by one, from top to bottom. But with *control flow statements*, we can alter this order and skip a bunch of lines under certain conditions or execute some lines more than once. We can do this based on the tests we've seen in the previous section.

4.5.1 If, else & elif

The most basic control flow statement is `if`. We can use it to start a block of code that will only be executed under certain conditions. For instance:

```
user_name = input("Hello! What's your name?")
if user_name == "Brian":
    print("He is the messiah!")
```

```
He is the messiah!
```

Let's review this script line by line. We've already met the `input()` function. We use it here to ask for the user's name and store it in a variable. The test `user_name == "Brian"` tests whether this variable is equal to a specific string. If this is the case, "He is the messiah!" is displayed on the screen! `if` statements are always written in the following form: "`if` *expression*:". Most of the time, *expression* is a test, but it can be anything that evaluates as `True` or `False`. After the colon (`:`) follows a *block* of code that will only be executed if the expression evaluates as `True`.

If you look carefully at the code above, you'll notice that the `print()` statement is indented to the right by spaces. Indentation is the way to tell Python what code belongs with the `if` statement. The colon (:) marks the start of the *indentation block*, all code in the block will be run only if the if statement is `True`.

```
if user_name == "Brian":
    print("He is the messiah!")
    print("All hail Brian!")
print("Welcome "+user_name)
```

```
He is the messiah!
All hail Brian!
Welcome Brian
```

An indentation block can consist of multiple lines of code. Every line has to be preceeded by the same number of spaces. As soon the Python interpreter ecounters a line without indentation, the block ends. In the previous example, the "Welcome *user_name*" message will be displayed for all users while the other two messages only if the user's name is Brian.

Most programming languages use a different syntax than indentation to delineate blocks. Curly brackets, for instance, are used in many languages. Here's an example of what the previous code would look like in R, another popular language in science and statistics. Note that the four spaces are still there, but now they're optional. Programmers use the spaces to structure their code, you can clearly see where the blocks begin and end when they're indented. So basically Python forces programmers to always structure their code.

```
if (user_name == "Brian"){
    print("He is the messiah!")
    print("All hail Brian!")
}
print(paste0("Welcome ",user_name))
```

The `if` statement can be extended with an `else` statement with its own block. If the expression evaluates as `True`, the block after the `if` statement is executed, but if it evaluates as `False`, the block after the `else` statement is executed.

```
if user_name == "Brian":
    print("He is the messiah!")
else:
    print("It is but a simple peasant...")
```

```
He is the messiah!
```

The `elif` statement is available when you need to test for more than two options. `elif` is short for "else if" and follows an `if` statement just as `else`. For instance:

```
if user_name == "Brian":
    print("He is the messiah!")
elif user_name == "Biggus":
    print("What's so funny about him?")
else:
    print("You are but a simple peasant")
```

```
He is the messiah!
```

4.5.2 While

A `while` statement is very similar to `if` but the code in the indentation block will be executed an indefinite number of times, as long as the expression is `True`. While thus creates a loop.

```
a_nr = 10
while a_nr > 5:
    print(a_nr)
    a_nr = a_nr-1
print("we're done!")
```

```
10
9
8
7
6
we're done!
```

An error beginning programmers (but experienced ones too) sometimes make is creating infinite loops.

Say we were programming the `while` loop from the previous example, but forgot the part were `a_nr` is changed:

```
a_nr = 10
while a_nr > 5:
    print(a_nr)

print("we're done!")
```

Now, the condition will be `True` every time the loop is repeated! The program never prints the final statement and keeps on running until it's killed by the user or the operating system.

4.5.3 For

The `for` statement also starts a loop but now, the code in the block will be executed a *specific* number of times.

```
my_list = ['one', 'two', 'three', 'four']
for element in my_list:
    print("And a", element)
```

```
And a one
And a two
And a three
And a four
```

If you are confused by the previous example, try to read it as follows: "for every element in my list, . . . ".

The basic syntax for the for statement is `for` *variable_name* in *iterable*. An iterable is a variable type that holds or returns multiple elements. `list`, `dict` and `str` are examples of iterables. The loop runs once for every element in the iterable and the value of each element will be assigned to variable_name.

If you do need to iterate over a sequence of numbers or want the loop to execute a specific number of times, the built-in function range() comes in handy. It generates arithmetic progressions:

```python
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Break Sometimes, we need to end a `for` loop before all elements in the iterable have been assessed. This is where the `break` statement comes in. It breaks the loop and continues after the block. In the following example we loop over every element in a list until we find what we are looking for: the holy grail!

```python
my_list = ["duck", "knight", "the holy grail", "rabbit", "camelot"]
for something in my_list:
    print("Now checking", something)
    if something == "the holy grail":
        print("We have found the holy grail!!!")
        break
print("Our quest has ended")
```

```
Now checking duck
Now checking knight
Now checking the holy grail
We have found the holy grail!!!
Our quest has ended
```

4.6 Errors

Don't you just hate error messages? Unfortunately you will probably see them often when you are learning a new programming language. Don't feel judged, exprerienced programmers get them too. It's so easy to make a typo of forget something. Although errors look scary, they contain a lot of helpful information so it's crucial we know how to interpret them. Let's raise an error by using a variable that doesn't exist yet:

```python
print(some_var)
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-49-9c21af92efc9> in <module>()
    ----> 1 print(some_var)


        NameError: name 'some_var' is not defined
```

There are many different kinds of mistakes a programmer can make and hence there are many different kinds of errors. The first thing you'll see when you get an error is the type of error. In this case, we've encoutered a `NameError`. If we check the Python manual on errors, we find this for `NameErrors`:

> Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

Indeed, the message on the final line of the error clearly states what *name* it could not find: `some_var`. The line above this message (the final line of the Traceback) shows you where in your program the error was raised. This program only has a single line, but when you start to write larger programs, this indication can be very helpful.

4.7 Comments

Comments are useful for annotating code. If you write readable code, the code often speaks for itself and doesn't require any additional explanation. But sometimes it helps if you write down a few words explaining what it is you are trying to do. This comes in handy when you have to use code you have written some time ago or share your code with somebody else. In Python, everything that starts with a pound (#) is a comment and will be ignored by the Python interpreter.

```python
# Hello, this is a comment
# Even if a comment contains code, it will be ignored:
# print("Something")
# Comments can start at the beginning of a line or after a statement
print("I'm not a comment") # I am!
# Pound signs inside string are not considered a comment
print("There's a # in this string")
```

```
I'm not a comment
There's a # in this string
```

As always, syntax highlighting is there to help. Comments are clearly indicated by a different shade. Here's a personal comment for you:

```python
# You are almost at the end of the chapter. You are doing great!
```

4.8 Packages and modules

There's only so much you can do with the built-in functions and variable types in Python. Luckily, there are pleny of packages and modules to extend the functionality of Python. Some of those will come with your Python installation, but many more are contributed by programmers in the Python community and have to be installed separately before you can use them.

For instance, the sys module has a variable that stores the version of Python we are currently using. But if we try to use it without actually loading the module, we get an error:

```python
print(sys.version)
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-52-c4f95efce128> in <module>()
    ----> 1 print(sys.version)


        NameError: name 'sys' is not defined
```

If we load the module first with the `import` statement the code works just fine:

```python
import sys
print(sys.version)
```

```
3.6.6 |Anaconda custom (64-bit)| (default, Jun 28 2018, 11:07:29)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)]
```

Did you notice the dot? Just like we access methods of a variable with a dot, we can access variables and functions in a module. The logic is similar: we are grouping stuff together to keep everything nice and tidy.

Modules can be grouped into packages, for instance:

```python
import datetime
print(datetime.date.today())
```

```
2020-02-15
```

In this case, `datetime` is a package that groups modules together that have something to do with timekeeping. `date` is a module in the `datetime` package and `today()` is a function in that module that returns todays date. That's a lot of dots right? We can also import specific modules from packages directly and reduce the dot-typing:

```python
from datetime import date
print(date.today())
```

```
2020-02-15
```

Still too much typing? You can rename modules on import as well.

```python
import datetime as dt
print(dt.date.today())
```

```
2020-02-15
```

Here, we renamed the `datetime` package to `dt` on importing. We can now type `dt` whenever we need it. You will often see this in notebooks or example code online. For instance, the popular `numpy` and `pandas` modules are typically imported like this:

```python
import numpy as np
import pandas as pd
```