03 Getting Started with Python

February 7, 2022

1 Module 3: Getting Started with Python

February 16, 2022

Last time we learned about UML Activity Diagrams as a means to describe processes at different levels of abstraction. Such models can be used as guidelines for the implementation of a process in an actual programming language, for example in Python.

Today we will get started with Python. The lecture covers how to write, edit and run Python code, how to do simple and formatted printouts, how to read simple user inputs, the basic data types, arithmetic expressions, and variables. The exercises for this lecture will guide you in setting up a Python programming environment on your computer, and give you the first programming assignments to work on yourselves.

Next time we will proceed to boolean expressions and conditional branching.

1.1 The Python Programming Language

Python is one of the most popular programming languages today. It was released for the first time in 1990, but gained extreme popularity only in the last 10-20 years, hand in hand with the increasing importance of the world wide web, big data and data science.

Although older versions are still operational, we will use **Python 3** in the course to make full use of the features of the latest generation. As of January 2021, the latest stable release of Python has number 3.9.1.

There is a lot of free literature about Python available that you can use for the course in addition to the lecture notes provided. Especially if you have difficulties understanding a particular concept, it is often a good idea to look at alternative explanations. What works good for the one, might just not be the best way to put it for the other. Here are some links to useful Python online books, but please feel free to check out also other sources of information:

- https://python.swaroopch.com ("A Byte of Python", especially for beginners)
- http://greenteapress.com/wp/think-python-2e/ ("Think Python", also targeted at beginners)
- https://docs.python.org/3/tutorial/index.html (the official Python tutorial)
- https://runestone.academy/ns/books/published/pythonds/index.html ("Problem Solving with Algorithms and Data Structures using Python")

During the course we will work with the Anaconda Python Data Science Platform (official website: https://www.anaconda.com/), in particular we will use the Spyder IDE (Integrated Development Environment) and Jupyter notebooks. The exercises will guide you through the installation and first steps with these environments.

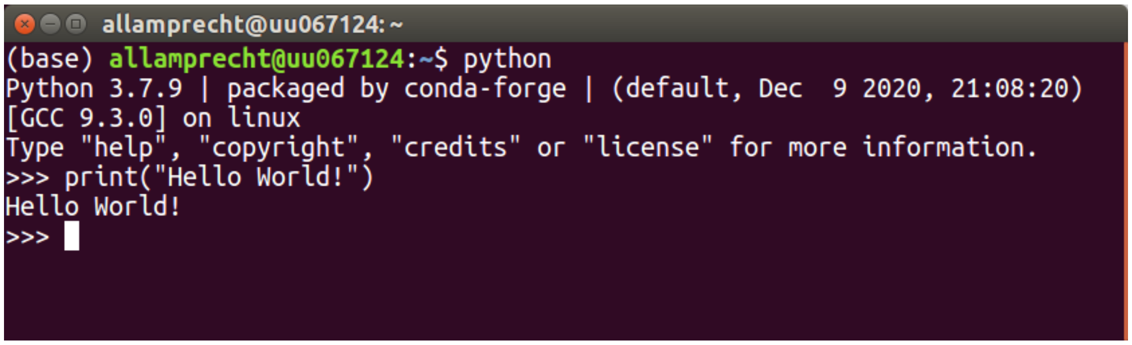
1.2 Hello World

The first program that one writes in any language is typically the “Hello world!” program, which simply prints “Hello world!” on the screen. In Python, this program is very simple and needs just one line:

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

The program just calls the `print` function with the character sequence (i.e., “string”) “Hello world!” as argument.

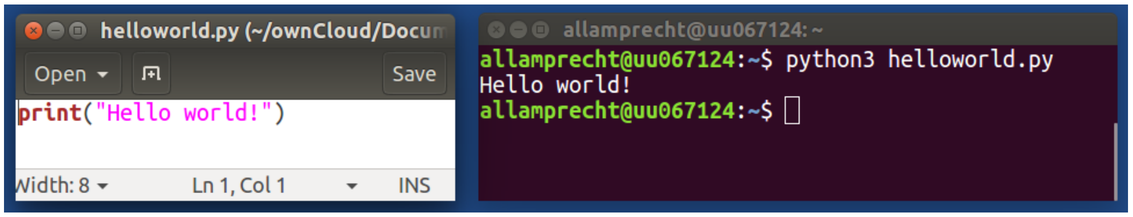
There are different ways to write and run this code. You can for example just type it into a Python console and hit “enter”:



```
(base) allamprecht@uu067124:~$ python
Python 3.7.9 | packaged by conda-forge | (default, Dec  9 2020, 21:08:20)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> 
```

Typing code directly into the console can be handy for testing commands or for quick calculations. Generally, however, we want to save our code so that we can easily execute it again later.

One can, for example, write the code in an arbitrary text editor and save it to a file, e.g., helloworld.py. We can run this program in the command line and get the same output:

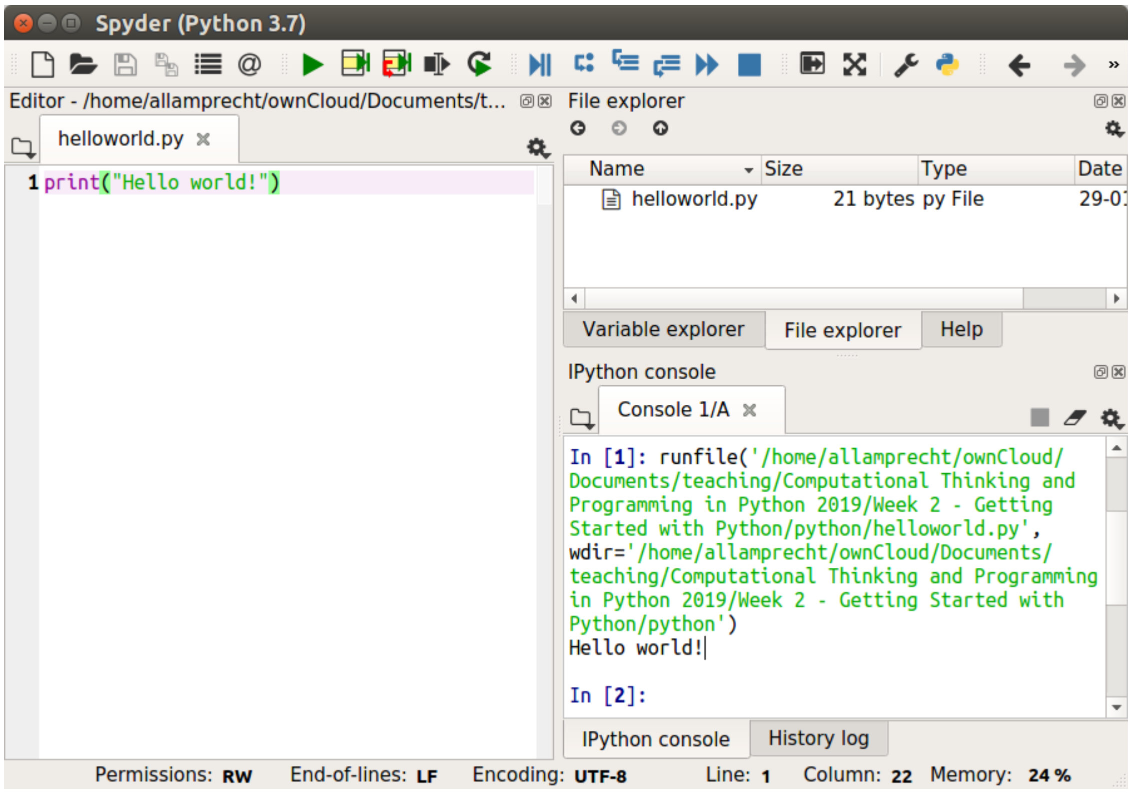


```
print("Hello world!")
```

```
allamprecht@uu067124:~$ python3 helloworld.py
Hello world!
allamprecht@uu067124:~$ 
```

Technically, .py files are text files and can be opened in any text editor. Some even recognize it as code and turn on syntax highlighthing, as shown in the example. When working on larger and more complex programs, however, it is better to use a real programming environment. In this course we use Spyder, the IDE that comes with Anaconda when you install it.

In Spyder the same program in development and during execution looks like this:



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

```
In [1]: runfile('/home/allamprecht/ownCloud/
Documents/teaching/Computational Thinking and
Programming in Python 2019/Week 2 - Getting
Started with Python/python/helloworld.py',
wdir='/home/allamprecht/ownCloud/Documents/
teaching/Computational Thinking and Programming
in Python 2019/Week 2 - Getting Started with
Python/python')
Hello world!

In [2]:
```

Another popular environment for writing and running Python code are Jupyter Notebooks. The lecture notes are an example. Notebooks are handy when one wants to combine code with text and other resources, as it allows for their easy interleaving in one document. Individual notebook cells can be run with just one click, and easily be changed to play around with the code in it. That makes them suitable as interactive textbooks, but also for sharing data, analysis code and results of a research project.

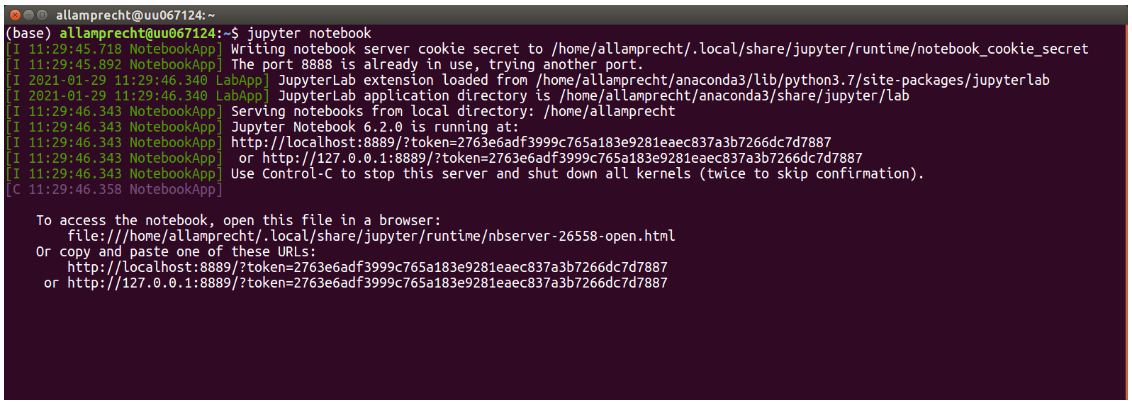
Here is the “Hello world!” program in a Jupyter code cell:

```
[1]: print("Hello World!")
```

```
Hello World!
```

Because there is more in them than just Python code, Jupyter notebooks are not .py files, but use the file ending .ipynb. Opening them in a plain text editor is again possible, but not very useful. Jupyter notebooks are made for being viewed in a web browser. If you have local .ipynb files, you need to start a notebook server first.

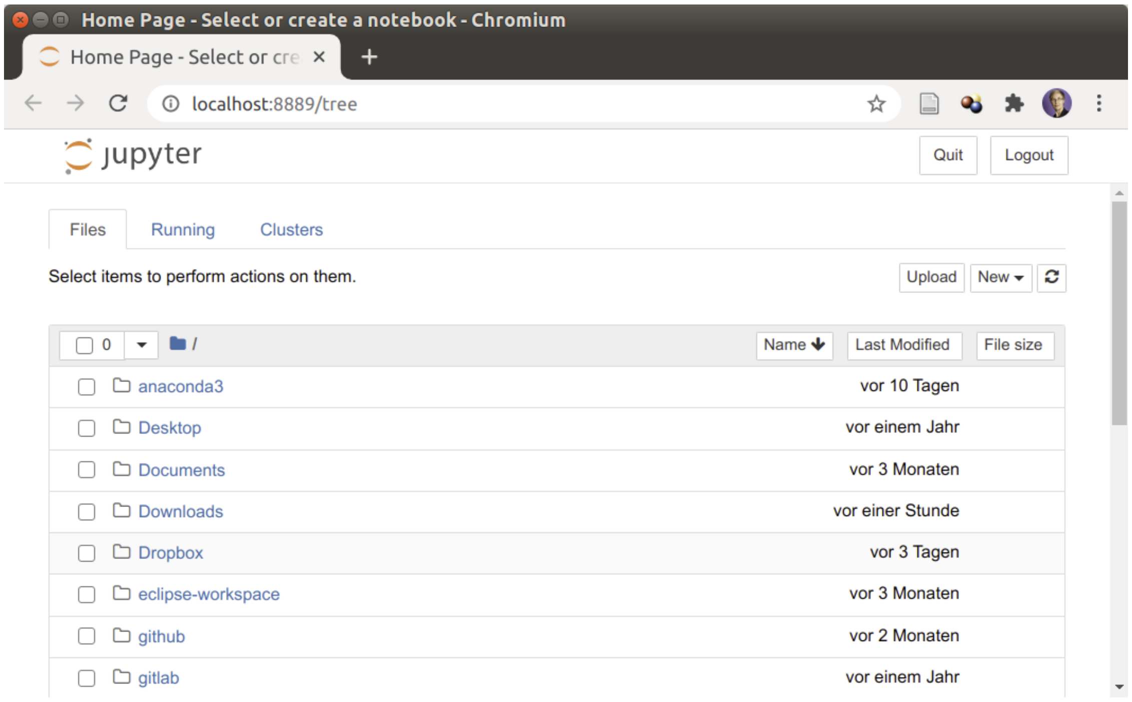
The Jupyter notebook server also comes with Anaconda as you install it. For starting up the server, you only need to type `jupyter notebook` in the command line:



```
(base) allamprecht@uu067124:~$ jupyter notebook
[I 11:29:45.718 NotebookApp] Writing notebook server cookie secret to /home/allamprecht/.local/share/jupyter/runtime/notebook_cookie_secret
[I 11:29:45.892 NotebookApp] The port 8888 is already in use, trying another port.
[I 2021-01-29 11:29:46.340 LabApp] JupyterLab extension loaded from /home/allamprecht/anaconda3/lib/python3.7/site-packages/jupyterlab
[I 2021-01-29 11:29:46.340 LabApp] JupyterLab application directory is /home/allamprecht/anaconda3/share/jupyter/lab
[I 11:29:46.343 NotebookApp] Serving notebooks from local directory: /home/allamprecht
[I 11:29:46.343 NotebookApp] Jupyter Notebook 6.2.0 is running at:
[I 11:29:46.343 NotebookApp] http://localhost:8889/?token=2763e6adf3999c765a183e9281eaec837a3b7266dc7d7887
[I 11:29:46.343 NotebookApp]  or http://127.0.0.1:8889/?token=2763e6adf3999c765a183e9281eaec837a3b7266dc7d7887
[I 11:29:46.343 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 11:29:46.358 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/allamprecht/.local/share/jupyter/runtime/nbserver-26558-open.html
    Or copy and paste one of these URLs:
        http://localhost:8889/?token=2763e6adf3999c765a183e9281eaec837a3b7266dc7d7887
     or http://127.0.0.1:8889/?token=2763e6adf3999c765a183e9281eaec837a3b7266dc7d7887
```

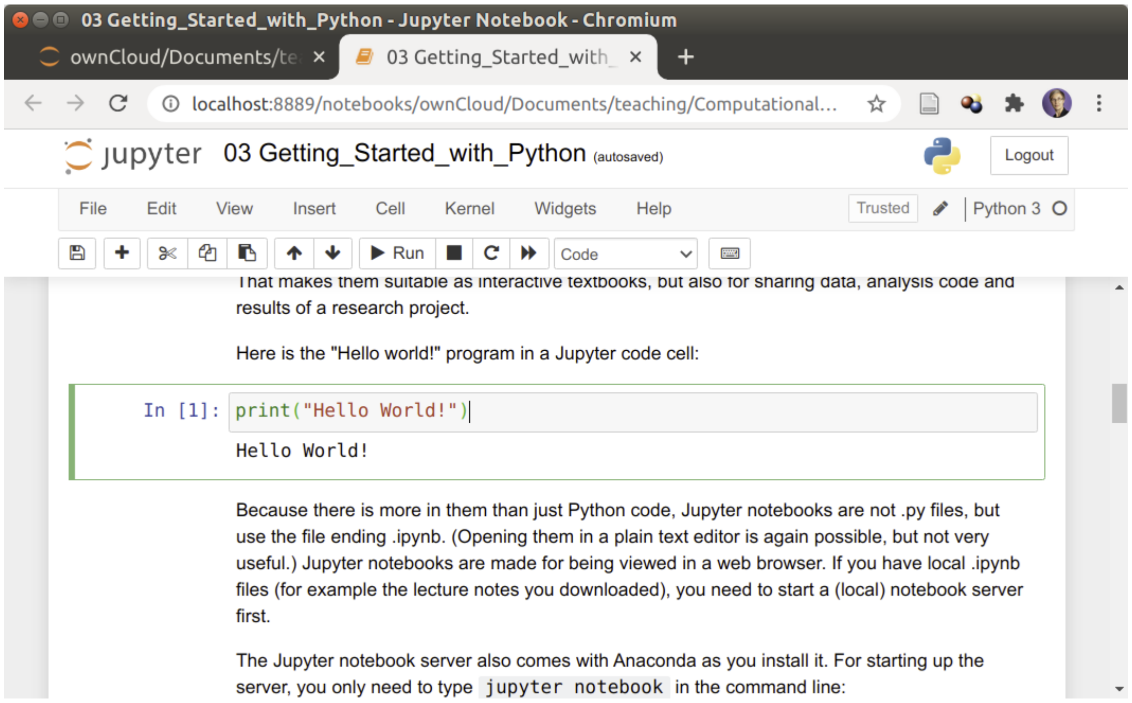
A browser window will open with a local address like `http://localhost:8889/tree` and show a view on your local file system:



Select items to perform actions on them.

Name	Last Modified	File size
anaconda3	vor 10 Tagen	
Desktop	vor einem Jahr	
Documents	vor 3 Monaten	
Downloads	vor einer Stunde	
Dropbox	vor 3 Tagen	
eclipse-workspace	vor 3 Monaten	
github	vor 2 Monaten	
gitlab	vor einem Jahr	

From there you can navigate to any .ipynb on your computer and interact with it in the browser:


03 Getting_Started_with_Python - Jupyter Notebook - Chromium
ownCloud/Documents/te
03 Getting_Started_with_
localhost:8889/notebooks/ownCloud/Documents/teaching/Computational...
jupyter 03 Getting_Started_with_Python (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

That makes them suitable as interactive textbooks, but also for sharing data, analysis code and results of a research project.

Here is the "Hello world!" program in a Jupyter code cell:

```
In [1]: print("Hello World!")

        Hello World!
```

Because there is more in them than just Python code, Jupyter notebooks are not .py files, but use the file ending .ipynb. (Opening them in a plain text editor is again possible, but not very useful.) Jupyter notebooks are made for being viewed in a web browser. If you have local .ipynb files (for example the lecture notes you downloaded), you need to start a (local) notebook server first.

The Jupyter notebook server also comes with Anaconda as you install it. For starting up the server, you only need to type `jupyter notebook` in the command line:

1.3 Sequential Execution

If we want our program to greet not only the world, but for example also the Netherlands and especially Utrecht, we can add more statements to the program, like this:

[2]:
```python
print("Hello World!")
print("Hello Netherlands!")
print("Hello Utrecht!")
```

```
Hello World!
Hello Netherlands!
Hello Utrecht!
```

Note here that Python is an interpreted language. That means that Python programs are executed directly by an interpreter, which runs the program line by line. This is in contrast to compiled languages, which need to be translated into another representation before being executable.

1.4 Comments

It is good practice to include comments in your program that explain what is happening. *"Code tells you how, comments should tell you why."* Comment lines in Python start wish a hash (#). They are ignored by the interpreter during execution, but can be very helpful for you or another person trying to understand the code, especially when it does more complicated things. Make commenting your code a habit directly from the beginning, even if it feels unnecessary for the simpler first examples.

Example:

[3]:
```
# greet the world
print("Hello World!")

# greet the Dutch
print("Hello Netherlands!")

# greet the people of Utrecht
print("Hello Utrecht!")
```

```
Hello World!
Hello Netherlands!
Hello Utrecht!
```

The output of this program is still the same as before.

1.5 Literal Constants

The string `"Hello world!"` from the above example is a so-called literal constant. Literal because its value is used exactly as it is written in the program code, and constant because it just represents itself and cannot be changed during runtime.

In the same way also numbers can be literal constants in a program, for example `42` or `3.14`, but more on numbers later.

1.6 Strings

Strings, i.e., sequences of characters, including white spaces and other sorts of special characters, are one of the basic data types in Python. Strings can be denoted by using single quotes (”) as well as double quotes (””), which work exactly the same way.

Triple single or double quotes can be used to specify multi-line strings, which can be convenient when dealing with longer pieces of text. For example, the following code again produces the same output as above:

[4]:
```
print("""Hello World!
Hello Netherlands!
Hello Utrecht!""")
```

```
Hello World!
Hello Netherlands!
Hello Utrecht!
```

A single backslash at the end of a line is used to indicate that the string is continued in the next line, but without adding a newline. For example:

[5]:
```
print("Hello World! \
Hello Netherlands! \
Hello Utrecht!")
```

```
Hello World! Hello Netherlands! Hello Utrecht!
```

Clearly, if you use a quotation mark to indicate the beginning and the end of a string, you cannot just use the same character within the string itself, as it would be interpreted as the end of the string. For these cases, there are so-called escape sequences, beginning with the backslash character \, which change the standard interpretation of the character(s) in the escape sequence. For example, to print the sentence It's called "Brexit" correctly, the following code can be used:

```
[6]: print("It's called \"Brexit\".")
```

```
It's called "Brexit".
```

or

```
[7]: print('It\'s called "Brexit".')
```

```
It's called "Brexit".
```

Some other frequently useful escape sequences are \\ for including a backslash in a string, \n for a newline and \t for a tab.

For a somewhat different purpose, Python allows to handle strings as "raw" strings, for which it does no processing of escape sequences and the like. Strings can be declared raw by prefixing them with r or R. For example:

```
[8]: print(r"It's called \"Brexit\".")
```

```
It's called \"Brexit\".
```

1.7 Numbers

Python knows basically two types of numbers: integers and floating point numbers, or floats for short, which are numbers with a decimal point. The E notation can be used to indicate powers of ten. For example:

```
[9]: print(42)
     print(3.14)
     print(2E-3)
```

```
42
3.14
0.002
```

Note that in this example, the numbers are again literal constants.

1.8 Arithmetic Expressions

Python supports seven basic operators for working with numbers:

```
+  (addition)
-  (subtraction)
*  (multiplication)
/  (division)
** (power, exponentiation)
```

```
// (integer division)
%  (remainder, modulo)
```

The order of expressions is the same as you know it from mathematics, and like there you can also use parentheses to structure more complex expressions.

For example:

[10]:
```
# do some random arithmetics
print("2+2 is", 2+2)
print(10*2, "is the result of 10*2.")
print("10/6 is", 10/6, "and 10//6 is", 10//6)
```

```
2+2 is 4
20 is the result of 10*2.
10/6 is 1.6666666666666667 and 10//6 is 1
```

Interestingly, the + (addition) and * (multiplication) operators are also defined for strings, where they function as concatenation and repetition operators, respectively:

For example:

[11]:
```
# string concatenation
print("Hello " + "world!")

# string repetition
print("Hello! " * 3)
```

```
Hello world!
Hello! Hello! Hello!
```

1.9 Variables

With only literal constants, as in the examples so far, programming would be quite limited and boring, but luckily there are variables. Variables store data, and their content can be altered during runtime. A variable has a name and we can assign a value to it. An assignment statement has the name on the left and the value to be assigned on the right. For example:

[12]:
```
# variables storing numbers
number1 = 5
number2 = 2.3

# variable storing a string
string1 = "Hello!"
```

Python is a dynamically typed language. That means that data objects get assigned their type only at runtime, and that it is not necessary to declare the type of the variable in the program, in contrast to programming languages such as C or Java.

Note that there are some rules and conventions for identifiers: Variable names in Python may consist of letters, numbers and the underscore _, but they must not start with a number. If they do, or if any other characters are used, this will lead to errors. Variable names are case sensitive, that is, `Name` is something different than `name`. Many Python developers use only lowercase letters in variable names, if necessary separating words with underscores, to improve readability. For example, `my_first_name` instead of `myfirstname`. Some prefer the so-called camelCase instead, that would mean `myFirstName` for the example above. All variants are fine, but for good readability it is strongly advisable to choose one and follow it consistently.

If a program is to do different things with the same numbers or strings, it handy to use variables to assign the values to them at one point in the program and read the values from the variables again where they are needed. This way, if we want to run the program with other values, we need to change them only once.

Example:

[13]:
```
# do some random arithmetics with the same two numbers
a = 10
b = 6

print(a, "+", b, "is", a+b)
print(a*b, "is the result of", a, "*", b)
print(a, "/", b, "is", a/b, "and", a, "//", b, "is", a//b)
```

```
10 + 6 is 16
60 is the result of 10 * 6
10 / 6 is 1.6666666666666667 and 10 // 6 is 1
```

Variables can also be used to store the result of operations, for example:

[14]:
```
# variables storing the results of operations
number3 = 5 * 2
number4 = number1 * 2.3
number5 = number1 * number2
string2 = "Hello " + string1
```

The type function can be used to check of which type a variable is, for example:

[15]:
```
# check types of variables
print(type(number5))
print(type(string2))
```

```
<class 'float'>
<class 'str'>
```

1.10 Shortcut Assignment

For assignments where a variable is updated using an arithmetic expression, that is, where the variable name at the left side of an assignment statement is also used in the expression on the right, there is a shorter way to write it. For example:

```
a = a + 1       does the same as       a += 1
```

None of the two is generally better than the other, and using the shortcut or not is a matter of personal taste and style, but every programmers should understand both.

1.11 Formatted Output

We have used the `print()` function so far to print out individual strings or sequences of strings by simply passing them to the function as a comma-separated list of arguments. We could also construct the string by using the + operator as follows:

```
[16]: a = 10
      b = 6
      print(str(a) + " + " + str(b) + " is " + str(a+b))
```

```
10 + 6 is 16
```

With the `format()` method for strings you can also construct strings from other information and format it properly:

```
[17]: print("{0} + {1} is {2}".format(a, b, a+b))
      print("{0} is the result of {1} * {2}".format(a*b, a, b))
      print("{0} / {1} is {2} and {0} // {1} is {3}".format(a, b, a/b, a//b))
```

```
10 + 6 is 16
60 is the result of 10 * 6
10 / 6 is 1.6666666666666667 and 10 // 6 is 1
```

That is, placeholders {0}, {1}, {2}, … are placed in the string at the points where we want to include particular values, which are passed as parameters to the `format()` method. These can be literals, variables or expressions, and they will be converted to string format immediately.

Note that it is also possible to omit the numbers and use empty pairs of parentheses. If the numbers of parenthesis pairs and arguments do not match, there is an error:

```
[18]: print("{} + {} is {}".format(a, b, a+b))
      print("{} is the result of {} * {}".format(a, b, a*b))
      print("{} / {} is {} and {} // {} is {}".format(a, b, a/b, a//b))
```

```
10 + 6 is 16
10 is the result of 6 * 60
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-18-bd96512b3755> in <module>
      1 print("{} + {} is {}".format(a, b, a+b))
      2 print("{} is the result of {} * {}".format(a, b, a*b))
----> 3 print("{} / {} is {} and {} // {} is {}".format(a, b, a/b, a//b))

IndexError: Replacement index 4 out of range for positional args tuple
```

Instead of using indices, one can also use named parameters:

[ ]:
```
print("{a} + {b} is {result}".format(a=a, b=b, result=a+b))
print("{result} is the result of {a} * {b}".format(a=a, b=b, \
            result=a*b))
print("{a} / {b} is {result1} and {a} // {b} is \
{result2}".format(a=a, b=b, result1=a/b, result2=a//b))
```

Another frequent use of the `format` method is to limit the number of decimal places of a number that are printed out, for example to 3 decimal places by adding `:.3f` to the corresponding placeholder:

[ ]:
```
print("{a} / {b} is {result1:.3f} and {a} // {b} is \
{result2}".format(a=a, b=b, result1=a/b, result2=a//b))
```

Finally, note that Python has only quite recently introduced the so-called f-strings, which allow for an even shorter way of accessing previously defined variables in a string:

[ ]:
```
print(f"{a} + {b} is {a+b}")
print(f"{a*b} is the result of {a} * {b}")
print(f"{a} / {b} is {a/b:.3f} and {a} // {b} is {a//b}")
```

Due to their good readability, we will mostly use f-strings for formatted outputs.

1.12 Interactive Input

Finally, it is also not very useful if the program can only print or calculate with fixed values that have been "hard-coded" during the development of the program. Rather, it should get inputs at runtime and do something with them. We can for instance ask the user to enter some information into the console at runtime with the input function and store it in a variable for later use.

For example:

[ ]:
```
# ask the user for name and greet him/her
user_name = input("What is your name? ")
print(f"Hello {user_name}!")

# ask the user for age (in years) and print age in months
user_age = int(input("What is your age (in years)? "))
print(f"Then you are at least {user_age*12} months old.")
```

The plain `input` function reads the user input as a string. If you want to read the input as an integer or floating-point number, you have to add a type cast to `int` or `float`, respectively:

[ ]:
```
# read string
input_string = input("Enter string: ")

# read integer
input_int = int(input("Enter integer: "))

# read float
input_float = float(input("Enter float: "))
```

It is best to do the type cast directly with the input, as it is easily forgotten to do it at a later stage in the program. Furthermore, directly when reading means that it only needs to be done once, and not (up to) every time the entered value is used.

1.13 Exercises

Please use Quarterfall to submit and check your answers.

1.13.1 1. Understanding Python Code ( )

Consider the following piece of code, which is very similar to, but not exactly the same as the interactive input example from the lecture.

```
user_name = input("What is your name? ")
print(f"Hello {user_name}!")
user_age = input("What is your age (in years)? ")
print(f"Then you are at least {user_age*12} months old.")
```

What is its output? What is the difference in the code, and can you explain why the output is different?

Important: Do not immediately paste the code into an editor or code cell and run it to see what it does. First try to read the code and figure it out from that, then check by executing it. Same for any other piece of Python code that you come across. That will greatly improve your understanding of Python programs.

1.13.2 2. Understanding more Python Code ( )

Here is another piece of code:

```
a = 2.3
b = 42
print(f"{a} + {b} is {a+b}")
print(f"{a} + {b} is {str(a+b)}")
print(f"{a} + {b} is {str(a)+str(b)}")
print(f"{a} + {b} is {int(a+b)}")
print(f"{a} + {b} is {int(a)+int(b)}")
print(f"{a} + {b} is {float(a+b)}")
print(f"{a} + {b} is {float(a)+float(b)}")
```

What is the output? Explain what causes the differences between the lines.

1.13.3 3. Printing Source Code ( )

Write a program that prints the Python source code from the previous exercise to the screen. The output should look as the piece of code above.

Important: If you get any error messages when you try to execute your programs, don't panic. Read the message and try to understand where and what the problem is. You can also use Google to find out more, common errors are usually discussed in different development forums. If that does not get you any further, ask your teaching assistant.

1.13.4 4. List Your Lectures ( )

Write a program that prints out for all of the weekdays the lectures that you have there. The output of the program should be something like:

```
Monday:     Evolutionary Computing
Tuesday:    (nothing)
Wednesday:  Evolutionary Computing, Programming with Python
Thursday:   (nothing)
Friday:     Programming with Python
```

Make sure that the list of lectures begins at the same position in every line.

1.13.5 5. Arithmetic Operations ( )

Write a program that asks the user to enter two integer numbers and then executes all the seven arithmetic operations with it for which Python has standard operators. The output of the program should be something like:

```
Please enter an integer number: 7
Please enter another integer number: 4
7 ** 4 is 2401
7 * 4 is 28
7 / 4 is 1.75
7 // 4 is 1
7 % 4 is 3
7 + 4 is 11
7 - 4 is 3
```

You can assume that the user enters two positive integer numbers (>0). Nevertheless, try what happens when you enter a negative number or 0.

1.13.6 6. Celsius-to-Fahrenheit Converter ( )

Write a Python program that asks the user to enter a temperature (as float) in degrees Celsius and computes what the temperature is in degrees Fahrenheit. The formula to compute Fahrenheit from Celsius is: 32.0 + degrees Celsius * (9.0 / 5.0) The output of the program should be something like:

```
Please enter the temperature in degrees Celsius: 12.5
12.5 degrees Celsius is 54.5 degrees Fahrenheit.
```

1.13.7 7. BMI Calculation ( )

Write a Python program that welcomes the user, asks for his name (string), weight in kg (integer) and height in m (float), computes the body mass index (BMI) from the information (weight/height2) and finally displays a message to the user, saying something like `"Hello Jim, your BMI is 23.4."`. You can assume that the user enters correct values.

1.14 Extras for the Weekend

We have seen in the lecture that the “Hello World!” program in Python is really easy. The “Hello World Collection” at http://helloworldcollection.de/ has the program in 585 different programming languages. Look at some examples to see that it is not that easy in a lot of other languages. Can you find the program for some other programming languages that you have heard of? The website https://www.scriptol.com/programming/history.php gives a nice overview of the history and evolution of programming languages since the 1950s.