1 A Python Tutorial

1.1 Disclaimer

In this tutorial, we will try to touch the important basics and try to give an impression of how the language works. After reading this paper the interested reader should be able to start writing small scripts. However, it is not possible to compress everything that can be known about a programming language into a single paper. Therefore, it is recommended to continue learning by using some of the plenty freely available online material and refer to Python documentation for details.

1.2 Prerequisites

It is recommended to try out the examples while reading the tutorial and experiment with them to familiarize yourself with the syntax. The code examples in this tutorial require Python 3.6 or greater,[1] make sure that during the installation process the Python directory is added to the systems `PATH` variable. To follow the examples, an interactive Python console should be used. While Python itself provides such a console, it lacks comfort. A better version of this console is called IPython, it can be installed by typing:

```
pip install ipython
```

into the normal system terminal (i.e., Command Prompt if using Windows). To start IPython, type `ipython` into the terminal after a successful installation. The codes shown througout this tutorial work with IPhython version 7.0.1, using other versions might lead to error messages.

1.3 How Python Works

At the heart of Python is the interpreter. The interpreter is a program, exactly like a browser or MS Word. The interpreter itself gets commands as inputs and executes them one by one in an interactive session or in form of a text file, also called a script, which is the normal form of a Python program.

If a script is given as input, the interpreter will read it line by line from top to bottom and execute one line after another. The simplest Python script, and an iconic starting point to learn any programming language, is the hello-world program, which consists of just a single line in Python:

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

[1] The authors recommend to use the "miniconda" python distribution, which can be found at https://conda.io/miniconda.html

If this line is saved in a text file with the name *hello.py* and executed via terminal[2] by typing: *python hello.py*, it will output *Hello World* to the terminal. For Windows users, the terminal can be intimidating at first since they may not be used to it. Think of the terminal in the same way as the explorer, it has one folder opened, and you can navigate with it through the file system. You can delete and move files as well as execute programs, like using the Python interpreter. But opposed to the explorer, the terminal has a built-in possibility to communicate with the user of the program. To do the same outside of the terminal a program needs to have a GUI, which requires quite some effort to create. In the terminal you can simply use `print()` in Python to display something and `input()` to get an input from the user. However, in the Windows Explorer, if you installed Python with the default settings, you can also start a Python script simply by double-clicking it like every other program. This will then automatically open a terminal and execute the script within it. The downside of this approach is that the terminal closes instantly once the execution finished, and in case of a bug you will not be able to read the error message, which is printed to the terminal. Therefore, it is recommended to use the terminal to execute a script during development.

1.4 Variables

In Python, you can use variables like in every programming language. Here are a few example definitions:

```
a_number = 1
another_number = 1 + 2
a_floating_point_number = another_number / 2
a_boolean = True
a_string = "Hello World"
another_string = \
    input("Please type a word and then press return:")
```

As can be seen, variables are defined by simply assigning them a value via the equal sign. The type (text, number, etc.) is detected automatically and does not need to be specified. In the third line you can see an example of a division, where one variable is used and the result is stored in a second variable. If a variable is used in an expression (except for an assignment) before a value was assigned to it, the script will crash with an error message. A line can be broken into multiple lines by using a \ as demonstrated in the last two lines. If a line ends before all parentheses have been closed, no backslash is required. The last expression shows an example of how the user is queried for input, and the result is stored in a variable. Here is an overview of the most important basic data types of variables that are used in the definitions above:

[2] If using Windows, navigate via the File Explorer into the folder where you saved *hello.py*, do a shift + right mouse click somewhere in the background of this folder and select *Open command window here*.

- **int**: Short form for integer. It represents a signed number.
- **float**: A signed floating-point number.
- **str**: Short for string. Represents text or single letters as special case of text.
- **bool**: Short for Boolean. Represents a truth value, i.e. true or false.

Additionally, there are two other important types:

- **lists**: A list of objects, which can either be accessed by index or iterated (see below).
- **dictionaries**: Similar to a list, but the index is not 0 to length-1. Instead any type[3] can be used as keys.

1.4.1 Lists

An empty list can be created using brackets:

```
my_list = []
```

A list containing concrete items can be created in IPython like this:

```
In [1]: my_list = [1, 2.5, "Hello World"]
```

Accessing the list works by using the index of the value with the starting index 0:

```
In [2]: my_list[0]
Out[2]: 1

In [3]: my_list[1]
Out[3]: 2.5

In [4]: my_list[2]
Out[4]: 'Hello World'
```

1.4.2 Dictionaries

Dictionaries store key/value pairs. A dictionary can be defined like this:

```
phone_numbers = {"Max": "01234567", "Lisa": "01425364"}
```

The general syntax is {key0: value0, key1: value1, ..., keyN: valueN}. It is recommended to break the definition over multiple lines to increase readability:

```
In [9]: balances = {
   ...:     "Max": 1002.57,
   ...:     "Lisa": -100.4,
```

[3] Only types that are hashable are allowed as keys, but that applies to most types, and it is relatively easy to make unhashable types hashable.

```
...:     "Rachel": 50.3,
...:     "Bob": 247.83
...: }
```

A dictionary can then be accessed like this:

```
In [10]: phone_numbers["Lisa"]
Out[10]: '01425364'

In [11]: balances["Bob"]
Out[11]: 247.83
```

or alternatively:

```
In [12]: person_of_interest = "Lisa"

In [13]: phone_numbers[person_of_interest]
Out[13]: '01425364'

In [14]: balances[person_of_interest]
Out[14]: -100.4
```

1.4.3 Variable Conversion

A string can contain a number, e.g. `"100"`, but it is still a string, i.e. a text object, and it will be treated as such by the interpreter, independent of whether it contains a number or not. Adding two strings will concatenate the strings, `"Hello " + "World"` will produce `"Hello World"`, and `"100" + "100"` {.python} will produce `"100100"`. For strings this is reasonable, for numbers it is not. To get the desired result of $100 + 100 = 200$, it is necessary to convert the string into an integer:

```
In [6]: my_str = "100"

In [7]: my_int = int(my_str)

In [8]: my_str + my_str
Out[8]: '100100'

In [9]: my_int + my_int
Out[9]: 200
```

This process is called casting.

1.4.4 The None Type

It is also useful that a variable can be declared as invalid or empty. As every number could be valid, it is not possible to reserve one number, e.g. 0, for the invalid variable value. In Python, there is a built-in value called `None` for this purpose. Since the type of a variable can change, it is no problem to assign `None` to any variable to mark it as invalid or empty. sec. 1.6 contains an example.

1.5 Flow Control

If every line of a program would always be executed, writing useful programs would be hard. Therefore, we need flow-control constructs to specify whether or how often a block of code is executed. In Python the two most important flow-control constructs are the if statement and the for loop. The if statement allows for conditional code execution, while the for loop allows us to execute a block of code multiple times. But first, we need to know a little more about Boolean expressions.

1.5.1 Boolean Expressions

Boolean expressions are no flow-control topic, but need to be understood to understand the next section. A Boolean expression is an expression that evaluates to True or False, i.e. a Boolean value. A few examples:

```
# Check whether two variables are equal:
In [4]: a = 1

In [5]: b = 2

In [6]: a == b
Out[6]: False

# Check whether a variable is in a certain range:
In [7]: 1 < b < 100
Out[7]: True

# Check whether a list is empty:
# [] creates an empty list, which is then passed to the len() function,
# which returns the number of elements in a list.
In [8]: len([]) == 0
Out[8]: True
```

Lines starting with a `#` are called comments and are ignored by the interpreter.

1.5.2 The If Statement

The if statement can be used to execute a block of code depending on a Boolean expression. Note thatyou can use ‘`control + o` to be able to enter multiple lines. To execute multiple lines, type `Enter` twice.

```
inp = input("Enter 1 or 2: ")
if inp == "1":
    print("1 was entered")
else:
    print("2 was entered")
print("A statement that will be executed anyway")
```

In this example, if the user entered 1, the expression `inp == "1"` will yield `True`. In this case, the code block that follows directly will be executed. If the user entered something different than 1, `inp == "1"` will be `False`, and the code in the optional else block will be executed.

The code above has a flaw. If the user entered anything but 1, the else block will be executed, and it will say that 2 was entered, which is not necessarily true. The code above could be corrected like this:

```
inp = input("Enter 1 or 2")
if inp == "1":
    print("1 was entered")
elif inp == "2":
    print("2 was entered")
else:
    print("That was neither 1 nor 2")
print("A statement that will be executed anyway")
```

`elif` is short for “else if”, which means that a condition must be met, but will only be applied if the previous checks failed. It is possible to chain any number of `elif` blocks one after another, and at the end there *can* be an `else`, which will be executed if no previous condition was met.

1.5.3 The For Loop

A for loop is used to iterate over an iterable, meaning that an expression is executed for each element within the iterable. More or less every container type in Python, like lists and dictionaries, are iterables. An example would be:

```
In [5]: for item in my_list:
   ...:     print(item)
1
2.5
Hello World
```

Here, the body of the for loop (the print function) is executed once for every element in `my_list`. The `item` in the first line is the name of the running variable and can be freely chosen.

To mark which code belongs to the for loop, it needs to be intended conventionally using 4 spaces. Python does not force a certain intention on the user as long as every block of code is consistent with itself. Four spaces is the standard, though, and every Python editor will insert 4 spaces when the user presses the tab key. Also note the colon at the end of the for statement. Every statement that is followed by additional code ends with a colon.

It is also possible to call a code block n times, e.g.:

```
In [1]: for i in range(5):
   ...:     print("Hello")
   ...:     print("World")
   ...:
Hello
World
Hello
World
Hello
World
Hello
World
Hello
World
```

Here, `range(5)` returns a generator (which is an iterable too) that yields the numbers from 0 to 4, therefore the body of the for loop is called 5 times. Note that there still is a variable, this time called `i`, even if it is not used in the body.

1.5.4 The While Loop

The while loop repeats a code block as long as a condition is met. While loops are barely used. There will be an example in sec. 1.6.

1.5.5 The Try-Except Statement

Certain things cannot work, e.g. it is not possible to cast a string containing words to an integer.

```
In [10]: int("Hello")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<iPython-input-10-5cdea6865089> in <module>()
----> 1 int("Hello")
```

```
ValueError: invalid literal for int() with base 10: 'Hello'
```

This provokes an error, or more precisely, raises an exception, which can be handled with a try-except statement. Make sure that `except` is not indented by 4 spaces (just delete them), otherwise an error will occur as except is not supposed to be part of the try statement.

```
try:
    my_var = int(input("Enter a number: "))
except ValueError:
    my_var = 0
```

In this scenario, the return value of `input()` is converted to an integer. In case it works, the except block is never executed. If, however, something else than a number is entered, the cast will fail and raise an exception. When this happens within a try-code block, the execution of the block will stop immediately, and the except block will be executed (if the raised exception has the type of the block, which, in this example, would be `ValueError`). It is possible to define multiple except blocks, each with a different exception type. Moreover, you can define an except block without an exception type (by just putting the colon directly behind the `except`), which will then be used for all exceptions that were not caught by previous except blocks. It is also possible to add a `finally` block at the very bottom of the statement. The code block with `finally` will be executed when the `try` block is left, independent of whether it was executed successfully or an exception was raised.

1.6 Functions

Using the knowledge from this tutorial up to this point would already enable a reader to write different programs, but they would probably involve a lot of code duplication. Functions are code blocks that can be executed with a single statement. Functions are preceeded by the `def` keyword.

```
def get_number_input_from_user(lower, upper):
    # In the while loop header, inp will be read out, so it has to
    # exist prior to the while call.
    inp = None
    # Everything between the while and the colon is a Boolean expression.
    # The expression is a combination of multiple subexpressions, the
    # interpreter will look at it like this:
    # ((inp is None) or (not(lower <= inp <= upper)))
    # Connecting two Boolean expressions with "or" means the expression
    # will be true if at least one of the subexpressions is true.
    # Therefore, the second subexpression, to the right of "or" will not be
    # evaluated if "inp is None" yields True, which is required here since
    # checking whether something is smaller than "None" would raise an
```

```
    # exception. The "not" will invert the result of its subexpression.
    # Subsequently, the loop will run, "while inp is not None, and inp is
    # not in the valid range between lower and upper".
    while inp is None or not lower <= inp <= upper:
        try:
            inp = int(input("Please input a number: "))
        except:
            inp = None
    return inp
```

Executing the function in a console could look like this:

```
In [17]: get_number_input_from_user(50, 100)
please input a number: g
please input a number: hello world
please input a number: 20
please input a number: 70
Out[17]: 70
```

A function definition is composed of the following things:

1. The `def` keyword at the beginning.
2. The name of the function "get_number_input_from_user".
3. The parenthesis after the function name, which contains the function arguments (e.g. lower and upper).
4. The trailing colon that implies a following code block.
5. The body of the function, which contains the code that will be executed when the function is called.

1.6.1 Variadic Arguments

If a function accepts any number of arguments, variadic arguments can be used:

```
def my_func(arg1, *args):
    print("Arg1:", arg1)
    # The type() function returns the type of its argument.
    print("type(args):", type(args))
    print("args:", args)
```

In this example, `my_func()` can be called with $n \geq 1$ arguments, here it is called with one argument:

```
In [20]: my_func(1)
Arg1: 1
type(args): <class 'tuple'>
args: ()
```

As can be seen, `arg1` contains the argument, and `args` is empty. If `my_func()` is called with multiple arguments, it produces the following output:

```
In [21]: my_func(1, 2, 3, 4, 5)
Arg1: 1
type(args): <class 'tuple'>
args: (2, 3, 4, 5)
```

In this scenario, `args` contains all arguments but the first one. The type of `args` is `tuple`, which is a read-only list. It could, for example, be used like this:

```
def my_func(arg1, *args):
    print("Arg1:", arg1)
    print("type(args):", type(args))
    # "enumerate()" returns a generator, which will yield
    # a tuple with 2 elements for each element in its
    # argument (here it is args). The first entry in
    # its tuple is the index of the element. The index
    # will end up in the variable i, and the second
    # one is the corresponding element itself, which
    # will end up in arg.
    for i, arg in enumerate(args):
        # If a string is prefixed with an f, its called a
        # "format string". Within a format string, {} can
        # be used to evaluate Python expressions. The
        # result of the expressions will be cast to str
        # and inserted at the corresponding place.
        print(f"Arg[{i}]: {arg}")
```

Calling this function would produce something like this:

```
In [23]: my_func(1, 2, 3, 4, 5)
Arg1: 1
type(args): <class 'tuple'>
Arg[0]: 2
Arg[1]: 3
Arg[2]: 4
Arg[3]: 5
```

1.6.2 Named and Keyword Arguments

In Python, arguments can be named on call, e.g. the `get_number_input_from_user()` function could be called like this:

```
get_number_input_from_user(lower=50, upper=100)
```

If this is done, the order does not matter anymore, e.g.:

```
get_number_input_from_user(upper=100, lower=50)
```

will produce the exact same results. It is also possible to allow a function to take named arguments that were not explicitly defined, so called keyword arguments.

```
def my_kwarg_func(**kwargs):
    print("type:", type(kwargs))
    print(kwargs)
```

Using two stars defines an argument to be a keyword argument dict, e.g.:

```
In [28]: my_kwarg_func(arg1="Hello", foo="bar", arg2="World")
type: <class 'dict'>
{'arg1': 'Hello', 'foo': 'bar', 'arg2': 'World'}
```

All of these methods can be combined:

```
def my_combined_func(arg1, arg2, *args, **kwargs):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("args:", args)
    print("kwargs:", kwargs)
```

And here is a calling example:

```
In [32]: my_combined_func(1, 2, 3, 4, 5, 6,
    ...:     foo=8, bar="Hello World")
    ...:
arg1: 1
arg2: 2
args: (3, 4, 5, 6)
kwargs: {'foo': 8, 'bar': 'Hello World'}
```

1.7 Object-Oriented Programming

If a variable is defined inside a function, e.g:

```
def foo():
    a = 1
    print(a)
```

it is only defined within the scope of the function and cannot be used outside of it:

```
In [34]: foo()
1

In [35]: a
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<iPython-input-35-3f786850e387> in <module>()
----> 1 a

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

Classes are a complex type of variable that allow the programmer to bundle a set of variables and functions together to represent objects (as opposed to functions, which usually represent processes). Functions that belong to a class are called *methods*. A common example is to use a class to describe a car. A car can have a speed, at which it is traveling, and a certain mileage. Before we can create a Car objects, we need define the class Car. This can be done in the following way:

```
class Car:
    def __init__(self, brand, mileage):
        self.brand = brand
        self.mileage = mileage
        self.fuel = 40
        self.velocity = 0

    def refill(self):
        self.fuel = 40

    def move(self, duration):
        dist = self.velocity * duration
        consumed_fuel = dist / 100 * self.mileage
        if consumed_fuel > self.fuel:
            self.fuel = 0
            print("Well, we have to walk now.")
        else:
            self.fuel -= consumed_fuel
            print(f"I drove {dist} km.")
```

A class appears to be a collection of functions, but there are a few things to note. All functions have a parameter called `self` as first argument. This variable represents the state of the object. The `__init__()` method is called by the interpreter when an object is created. The self argument (or more exactly the first argument, "self" is just a naming convention) will be passed by the interpreter, while the remaining arguments will be passed by the programmer. It is possible to create a new variable belonging to the object by calling `self.var_name = "something"`, i.e. it is possible to access the object members with a dot. Variables can be added to an object any time, but it is common practice do this when an object is created use the `__init__()` method.

An object can be created like this:

```
In [37]: mycar = Car("PyCar", 5.6)
```

In this example the "empty"[4] object will be passed to the `__init__()` method of `Car` as `self`, `brand` will be *"PyCar"* and `mileage` will be 5.6. Then, both fields will be added to the object, and additionally the fields `fuel` and `velocity`.

[4]The passed object is absolutely not empty. It already contains all methods and a set of so-called "magic methods", which will not be covered in this tutorial. However, so far it contains no user-defined non-method fields.

Now it would be possible to set a velocity like this:

```
In [38]: mycar.velocity = 100
```

And let mycar "move" like this:

```
In [40]: mycar.move(3)
I drove 300 km
```

If the fuel level is queried like this:

```
In [41]: mycar.fuel
Out[41]: 23.200000000000003
```

Noticed, that the fuel level changed after moving the car. Multiple objects of class `Car` could be created, and changing a value in one of them would not affect the value of another object[5].

1.8 Using Code From Other Files

There are multiple good reasons to not write a whole program into a single file. Big files slow down editors, they make it hard to find certain code segments, and last but not least it is difficult to reuse code. Python files that are not meant to be executed by the user directly are called *modules*. Code from modules can be accessed by importing them. Consider a file named my_module.py with the following contents:

```
def _my_hidden_func():
    print("_my_hidden_func")

def my_func():
    print("my_func")

def my_2nd_func():
    print("my_2nd_func")
```

The code from this file can be used in the 3 following ways:

1.

```
In [3]: import my_module

In [4]: my_module._my_hidden_func()
_my_hidden_func
```

2. Here, an alias is used to save you from typing long module names over and over:

[5] It is possible to create fields that are common to all objects of one class. Those are called "static fields" and are not covered here.

```
In [7]: import my_module as mm

In [8]: mm.my_func()
my_func
```

3. Particular elements from the module can also be imported into the local namespace like this:

```
In [9]: from my_module import my_func

In [10]: my_func()
my_func
```

It is also possible to import multiple elements from the module in one line the following way: `from my_module import my_func, my_2nd_func`

Alternatively, you can import all public functions from a module:

```
In [11]: from my_module import *

In [12]: my_2nd_func()
my_2nd_func

In [13]: _my_hidden_func()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<iPython-input-13-17d9be3116cf> in <module>()
----> 1 _my_hidden_func()

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

The last example shows that `_my_hidden_func()` was not imported because it is a private member of the module. Module or class members that start with `_` are considered "private", i.e. only to be used inside the class and not by a module user. Therefore, the star will import everything that has a name that does not start with `_`.

It is recommended to use direct imports (`from x import y`) unless that causes name collisions, e.g. by using two different modules containing a symbol[6] with the same name. In this case it is recommended to use aliased module imports.

1.8.1 Packages

A package is a collection of modules. Bigger libraries are nearly always packages. A folder that contains multiple modules and should be recognized as a package must contain a file named ***init***.*py*, which can be empty.

[6] A symbol is everything that can be a top-level member of a module: a variable, a function, or a class.

If a folder structure like this is given:

```
working_directory
|- my_package
    |- __init__.py
    |- my_module.py
    |- my_module2.py
```

the modules can be imported as follows:

```
import my_package.my_module
# and
import my_package.my_module2
```

Note, importing `my_package` will not automatically import the modules too.

1.8.2 The Search Path

Python modules are found if they are on the search path. The search path is the union of the following places:

- the installation directory of the interpreter
- the current working directory
- the environment variable "PYTHONPATH" of the operating system (OS)

1.8.3 Installing External Modules

The Python community generally publishes libraries (i.e. packages) on the *Python package index (PyPi)*. These packages can be downloaded and installed with a tool named *pip*, which comes with any modern Python installation. If there is a library named *PyParadigm* on PyPi (which happens to be the case), it can be installed using pip by typing

```
pip install PyParadigm
```

into the terminal (of the OS, not the interactive Python console).

Warning: Anyone can upload code to PyPi in a matter of seconds and without any checks. Therefore, PyPi cannot be regarded a trusted source. It is save to install libraries that you already trust from there (names cannot be stolen). So pip can be used safely to install commonly known libraries, but it would be a bad idea to search PyPi for a library and simply install it before checking it.

1.9 The Python Standard Library

So far the tutorial dealt with syntax and basic code organisation, but at this point the interested reader might wonder how to write scripts that do something

useful. Python comes with many pre-installed modules, but demonstrating them in detail is not within the scope of this introduction. However, they are very well explained in the official Python documentation. Here, a list of the most important modules (in the authors' opinion) and a short summary of their functionality is given:

- **argparse**: Access to script parameters that were given via the terminal.
- **math** and **statistics**: Mathematical functions.
- **json**: Read and write json files.
- **pathlib**: File system access.
- **pickle**: Saving and loading of arbitrary Python data structures.
- **sys**: Access interpreter settings and information.
- **time**: Time access and conversions.
- **subprocess**: Control execution of other programs.

Note that this covers not even 10% of all available modules. There are modules for sending mails, accessing websites, creating GUIs, using databases, and much more. Therefore, it is recommended to read through the Python module index at least once.

1.10 Important Libraries For Social Science

Besides the standard library, there are many high-quality packages available for specific user communities. Here is a list of the most important ones (again in the authors' opinion) for the social sciences:

- **numpy**: Provides an n-dimensional matrix class and powerful numerical methods.
- **matplotlib**: Provides plotting functionality, similar to Matlab.
- **pandas**: Provides a table data class much like R's data frame.
- **seaborn**: Plotting library that extends matplotlib and works with Pandas DataFrame.
- **statsmodels**: Provides many statistical models.
- **scikit-learn**: Provides machine learning algorithms.
- **qt5**: A library to build GUIs.
- **jupyter**: Provides tools to work with notebooks in a browser, contains also the IPython console.
- **requests**: Provides simple functions to send HTTP requests.