THE INTERPRETER, AN INTERACTIVE SHELL

THE TERMS 'INTERACTIVE' AND 'SHELL'

The term "interactive" traces back to the Latin expression "inter agere". The verb "agere" means amongst other things "to do something" and "to act", while "inter" denotes the spatial and temporal position to things and events, i.e. "Between" or "among" objects, persons, and events. So "inter agere" means "to act between" or "to act among" these.

With this in mind, we can say that the interactive shell is between the user and the operating system (e.g. Linux, Unix, Windows or others). Instead of an operating system an interpreter can be used for a programming language like Python as well. The Python interpreter can be used from an interactive shell.

The interactive shell is also interactive in the way that it stands between the commands or actions and their execution. This means the Shell waits for commands from the user, which it executes and returns the result of the execution. After this the shell waits for the next input.

A shell in biology is a calcium carbonate "wall" which protects snails or mussels from its environment or its enemies. Similarly, a shell in operating systems lies between the kernel of the operating system and the user. It's a "protection" in both direction. The user doesn't have to use the complicated basic functions of the OS but is capable of using simple and easier to understand shell commands. The kernel is protected from unintended incorrect usages of system function.

Python offers a comfortable command line interface with the Python shell, which is also known as the "Python interactive shell".
It looks like the term "interactive Shell" is a tautology, because "Shell" is interactive on its own, at least the kind of shells we have described in the previous paragraphs.

USING THE PYTHON INTERACTIVE SHELL

With the Python interactive interpreter it is easy to check Python commands. The Python interpreter can be invoked by typing the command "python" without any parameter followed by the "return" key at the shell prompt:

```
python
```

Python comes back with the following information:

```
$ python
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

A closer look at the output above reveals that we have the wrong Python version. We wanted to use Python 3.x, but what we got is the installed standard of the operating system, i.e. version 2.7.11+.

The easiest way to check if a Python 3.x version is installed: Open a Terminal. Type in python but no return. Instead type the "Tab" key. You will see possible extensions and other installed versions, if there are some:

```
bernd@venus:~$ $ python
python      python2.7   python3.5   python3m
python2     python3     python3.5m
bernd@venus:~$ python
```

If no other Python version shows up, python3.x has to be installed. Afterwards, we can start the newly installed version by typing python3:

```
$ python3
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] on linux
Type "help", "copyright", "credits" or "license" for more
```

```
information.
>>>
```

Once the Python interpreter is started, you can issue any command at the command prompt ">>>". Let's see, what happens, if we type in the word "hello":

```
>>> hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hello' is not defined
>>>
```

Of course, "hello" is not a proper Python command, so the interactive shell returns ("raises") an error.

The first real command we will use is the print command. We will create the mandatory "Hello World" statement:

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

It couldn't have been easier, could it? Oh yes, it can be written in a even simpler way. In the interactive Python interpreter - but not in a program - the print is not necessary. We can just type in a string or a number and it will be "echoed"

```
>>> "Hello World"
'Hello World'
>>> 3
3
>>>
```

HOW TO QUIT THE PYTHON SHELL

So, we have just started, and we already talk about quitting the shell. We do this, because we know, how annoying it can be, if you don't know how to properly quit a program.

It's easy to end the interactive session: You can either use exit() or Ctrl-D (i.e. EOF) to exit. The brackets behind the exit function are crucial. (Warning: exit without brackets works in Python2.x but doesn't work anymore in Python3.x)

THE SHELL AS A SIMPLE CALCULATOR

In the following example we use the interpreter as a simple calculator by typing an arithmetic expression:

```
>>> 4.567 * 8.323 * 17
646.18939699999999
>>>
```

Python follows the usual order of operations in expressions. The standard order of operations is expressed in the following enumeration:

1. exponents and roots
2. multiplication and division
3. addition and subtraction

This means that we don't need parenthesis in the expression "3 + (2 * 4):

```
>>> 3 + 2 * 4
11
>>>
```

The most recent output value is automatically stored by the interpreter in a special variable with the name "_". So we can print the output from the recent example again by typing an underscore after the prompt:

```
>>> _
11
>>>
```

The underscore can be used in other expressions like any other variable:

```
>>> _ * 3
33
>>>
```

The underscore variable is only available in the Python shell. It's NOT available in Python scripts or programs.

USING VARIABLES

It's simple to use variables in the Python shell. If you are an absolute beginner and if you don't know anything about variable, please confer our chapter about variables and data types.
Values can be saved in variables. Variable names don't require any special tagging, like they do in Perl, where you have to use dollar signs, percentage signs and at signs to tag variables:

```
>>> maximal = 124
>>> width = 94
>>> print(maximal - width)
30
>>>
```

MULTILINE STATEMENTS

We haven't introduced multine statements so far. So beginners can skip the rest of this chapter and can continue with the following chapters.

We will show, how the interactive prompt deals with multiline statements like for loops.

```
>>> l = ["A",42,78,"Just a String"]
>>> for character in l:
...     print(character)
...
A
42
78
Just a String
>>>
```

After having input "for character in l:" the interpretor expects the input of the next line to be indented. In other words: The interpretor expects an indented block, which is the body of the for loop. This indented block will be iterated. The interpretor shows this "expectation" by showing three dots "..." instead of the standard Python interactive prompt ">>>". Another special feature of the interactive shell: When we have finished with the indented lines, i.e. the block, we have to enter an empty line to indicate that the block is finished.
Attention: The additional empty line is only necessary in the interactive shell! In a Python program, it is enough to return to the indentation level of the "for" line, the one with the colon ":" at the end.

STRINGS

Strings are created by putting a sequence of characters in quotes. Strings can be surrounded by single quotes, double quotes or triple quotes, which are made up of three single or three double quotes. Strings are immutable. This means that once defined, they cannot be changed. We will cover this topic in detail in another chapter.

```
  >>> "Hello" + " " + "World"
'Hello World'
```

String in triple quotes can span several lines without using the escape character:

```
>>> city = """
... Toronto is the largest city in Canada
... and the provincial capital of Ontario.
... It is located in Southern Ontario on the
... northwestern shore of Lake Ontario.
... """
>>> print(city)

Toronto is the largest city in Canada
and the provincial capital of Ontario.
It is located in Southern Ontario on the
northwestern shore of Lake Ontario.

>>>
```

There is a multiplication on strings defined, which is essentially a multiple concatenation:

```
  >>> ".-." * 4
'.-..-..-..-.'
>>>
```