Python Introduction

Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby and Java.

- python (https://www.python.org/)
- Monty Python (https://en.wikipedia.org/wiki/Monty_Python)

MONTY PYTHON
and the Holy Grail

Python 2 or Python 3?

Short version: Python 2.x is legacy, Python 3.x is the present and future of the language final 2.x version 2.7 was released 2010

```
    - 3.0 was released in 2008
    - 3.4 was released 2014
    - recent standard library improvements are only available in 3.x
```

Software

A python interactive mode makes it easy to test short snippets of code, but there are also a bunch of editors like Sublime Text available as well as bundled development environments IDEs like PyCharm.

We will use:

```
    - Python (https://www.python.org/)
    - IPython (http://ipython.org/)
    - IPython Notebook (http://ipython.org/notebook.html)
```

You could use:

```
    - an editor of your choice like Sublime Text
    - an IDE (PyCharm)
    - Python debuggers (pdb)
    - Code checkers (pylint)
```

We will discuss:

- Variables
- Objects
- Strings
- Lists
- Dictionaries
- Functions
- Import of modules

Interactive mode

Start the python command line interpreter by typing `python`, or the Python interactive shell by typing `ipython` into your terminal.

You can use this a calculator:

```
In [2]: 2 + 3
Out[2]: 5
```

or print this:

```
In [3]: print ("Hello World")
        Hello World
```

Variables

you can assign number to variables like:

```
In [4]: height = 1
        width = 2
        print (height)
        print (width)

        1
        2
```

and you can reuse this variables

```
In [5]: add_height_width = height + width
        print (add_height_width)

        3
```

you can change variables

```
In [6]: width = 12
        add_height_width = height + width
        print (add_height_width)

        13
```

Naming Rules

- Variables can only contain letters, numbers, and underscores.
- Variable names cannot start with a number but should start with a letter or an underscore.
- Spaces are not allowed in variable names -> use underscores instead of spaces.
- It should be avoided to use Python keywords as variable names like int, float, list, input, if you absolutely need to add an underscore
- Variable names should be descriptive, without being too long. For example `n_dog_legs` is better than just `dog` or `number_of_legs_of_a_dog`.
- Never use singe letters like a, b, c as variable names
- Be careful about using the lowercase letter l and the uppercase letters O and I in places where they could be confused with the numbers 1 and 0.

Datatypes

int/floats

```
In [7]: width
Out[7]: 12
```

```
In [8]: width/5
Out[8]: 2
```

type conversion

```
In [13]: float(height)
Out[13]: 1.0
```

```
In [15]: 2.
Out[15]: 2.0
```

String

Strings are sets of characters and are contained either in single or double quotes.

```
In [2]: text = "this is a string"
        print (text)

        this is a string
```

Through this we are able to create strings which contains quotes.

```
In [1]: text = 'this is a string "containing a quote"'
        print (text)

        this is a string "containing a quote"
```

We can also use multiple line strings in triple quotes ''' or """

```
In [3]: text = """this is a string
        over
        more
        than one line
        """
        print (text)

        this is a string
        over
        more
        than one line
```

You can also concatenate strings with the + command

```
In [19]: first_part = "this is the first part ... "
         second_part = "and this is the second part of the text"
         concat = first_part + second_part
         print(concat)

         this is the first part ... and this is the second part of the text
```

If you want a specific number of repetitions of a text you can use the * command followed by a number.

```
In [4]: "example " * 3

Out[4]: 'example example example '
```

List

An other data type is the list or array. you can initialise it like:

```
In [22]: list_var = []
         print (list_var)

         []
```

but you can also fill it directly with entries.

```
In [23]: list_var = [1,2,3,4]
         print (list_var)

         [1, 2, 3, 4]
```

indexing and slicing

Lists can be indexed or sliced. Important for indexing is, that the index of a list starts with 0. Indexing and slicing works for build-in sequence types like lists or strings.

```
In [24]: list_var[0]

Out[24]: 1
```

```
In [25]: list_var[-1]

Out[25]: 4
```

```
In [28]: print (list_var[0:2])
         print (list_var[:2])

         [1, 2]
         [1, 2]
```

```
In [29]: list_var[-2:-1]

Out[29]: [3]
```

```
In [30]: list_var[0:-1:2]

Out[30]: [1, 3]
```

You can use indexing to change entries, but this works only for mutable object types like lists not for immutable like strings

```
In [31]: list_var[0] = [1,2,3]
         print (list_var)

         [[1, 2, 3], 2, 3, 4]
```

```
In [32]: list_var[::-1]

Out[32]: [4, 3, 2, [1, 2, 3]]
```

append/extend

You can also append or extend your list

```
In [33]: list_var.append("wuff")
         print (list_var)

         [[1, 2, 3], 2, 3, 4, 'wuff']
```

```
In [34]: list_extention = ["a","b","c"]
         list_var.extend(list_extention)
         print (list_var)

         [[1, 2, 3], 2, 3, 4, 'wuff', 'a', 'b', 'c']
```

Occurrence testing

```
In [35]: 'wuff' in list_var

Out[35]: True
```

```
In [36]: 'wiff' in list_var

Out[36]: False
```

Dictionaries

Dictionaries are an other data type object structure. A dictionary contains key / value pairs. Instead of the position you use the key to access the respective value.

```
In [37]: dictionary = {'one_key': 1, 'second_key': 2}
         print (dictionary)

         {'one_key': 1, 'second_key': 2}
```

```
In [39]: dictionary['one_key']

Out[39]: 1
```

```
In [42]: dictionary['one_key'] = 21
         print (dictionary)

         {'one_key': 21, 'second_key': 2}
```

```
In [43]: dictionary['third_key'] = 3
         print (dictionary)

         {'one_key': 21, 'third_key': 3, 'second_key': 2}
```

```
In [44]: new_dict = {'one_key':12, 'third_key':14, 'fourth_key': 4}
         dictionary.update(new_dict)
         print (dictionary)

         {'one_key': 12, 'fourth_key': 4, 'third_key': 14, 'second_key': 2}
```

Occurrence testing

```
In [45]: 'one_key' in dictionary

Out[45]: True
```

Control structure if/else/elif

```
In [ ]: statement = True
        if statement is True:
            do something
        else:
            do something else
```

```
In [7]: amount = 12
        if amount > 10:
            print ('many')
        else:
            print('few')

        many
```

Bool

Values: True, False

```
- []         False
- [a, b]     True
- 0          False
- all other  True
```

None

None is used to present the absence of a value

Intendation

In [8]:
```
test_list = [1,2,3,4,5]
```

In [17]:
```
if test_list:
    print ('test_list != []') # list is not empty
print ('always true')
```

```
test_list != []
always true
```

In [19]:
```
if len(test_list) == 0:
    print ("length is 0")
elif len(test_list) >0 and len(test_list) <= 4:
    print ('length test_list is between 1 and 4')
else:
    print ('length test_List is larger then 4')
```

```
length test_List is larger then 4
```

For loop and while loop

In [29]:
```
iter_list = ['a', 'b', 'c']
for i in iter_list:
    print (i)
```

```
a
b
c
```

In [32]:
```
iter_list = [1,2,3,4]
```

In [37]:
```
step = 0
while step != len(iter_list):
    print ("value: %i" % step )
    step += 1
```

```
value: 0
value: 1
value: 2
value: 3
```

In [34]:
```
step = 0
while step != len(iter_list):
    print (step, iter_list[step])
    step += 1
```

```
0 1
1 2
2 3
3 4
```

range and enumerate

In [35]:
```
for i in range(3):
    print (i)
```

```
0
1
2
```

In [40]:
```
for i in range(len(iter_list)):
    print ("%i value:%i" % (i, iter_list[i]))
```

```
0 value:1
1 value:2
2 value:3
3 value:4
```

In [41]:
```
for i, value in enumerate(iter_list):
    print (i, value)
```

```
0 1
1 2
2 3
3 4
```

Dictionaries and For loops

In [83]:
```python
for key in dictionary:
    print (key, dictionary[key])
```
```
one_key 12
fourth_key 4
third_key 14
second_key 2
```

In [84]:
```python
for key, value in dictionary.items():
    print (key, value)
```
```
one_key 12
fourth_key 4
third_key 14
second_key 2
```

In [85]:
```python
for i, key in enumerate(dictionary):
    print (i, key)
```
```
0 one_key
1 fourth_key
2 third_key
3 second_key
```

break, continue, pass

In [88]:
```python
for i in iter_list:
    print (i)
    if i == 3:
        print ('i equal 3 - break')
        break
```
```
1
2
3
i equal 3 - break
```

In [42]:
```python
for i in iter_list:
    print (i)
    if i == 3:
        print ("i equal 3 - continue")
        continue
        print ("behind continue")
```
```
1
2
3
i equal 3 - continue
4
```

In [90]:
```
for i in iter_list:
    print (i)
    if i == 3:
        pass
        print ("behind pass")
```

```
1
2
3
behind pass
4
```

Functions

In [91]:
```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n.""" # docstring
    a, b = 0, 1  # multiple assignement
    while a < n:
        print (a),  # prevents the newline
        a, b = b, a+b  # another multiple assignement

fib(500)
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

The keyword def initialized a function followed by the name and a list of parameters

Documentation strings

You should put a triple quoted string into the first line after the function definition, containing a description of the function. This is called doc string, and can be used to automatically produce documentation.

return statement

```
In [93]: def fib(n):    # write Fibonacci series up to n
             """Print a Fibonacci series up to n.""" # docstring
             result = []
             a, b = 0, 1  # multiple assignement
             while a < n:
                 result.append(a)
                 a, b = b, a+b  # another multiple assignement
             return result
         fib_result = fib(500)
         print (fib_result)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

The `return` statement returns a value from a function. `return` without an expression argument returns `None` as well as falling off the end of a function.

Default Arguments and Keyword Arguments

```
In [107]: def extended_fib(number, default=True, first_number=0, second_numbe
          r=1):
              """Print a Fibonacci series up to n.""" # docstring
              if default:
                  a, b = 0, 1
              else:
                  a, b = first_number, second_number  # multiple assignement

              while a < number:
                  print (a, end=' ')  # space instead of newline
                  a, b = b, a+b  # another multiple assignement
          extended_fib(500)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
In [101]: extended_fib(500, first_number=55, second_number=89)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

```
In [102]: extended_fib(500, default=False, first_number=55, second_number=89)
```

```
55 89 144 233 377
```

Import of Modules

You can import other python packages/modules quite easy with the import function.

```
In [103]: import math
```

Afterwards you can use specific functions of the modules.

```
In [104]: math.cos(1)

Out[104]: 0.5403023058681398
```

```
In [105]: math.exp(1)

Out[105]: 2.718281828459045
```

Coding style

“The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly." - Donald E. Knuth, Selected Papers on Computer Science

PEP8

Style guide for Python code

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

A few rules

- never use tabs, always 4 spaces
- try to limit lines to 79 characters
- use whitespace to make your code more readable

In [ ]:
```
spam(ham[1], {eggs: 2})        # YES!
spam( ham[ 1 ], { eggs: 2 } )  # NO!!

x, y = y, x     # YES!
x , y = y , x   # NO!!

counter = counter + 1  # YES!
counter=counter+1      # NO!!

result = add(x+1, 3)   # YES!
result = add(x + 1, 3) # YES!


def complex(real, imag=0.0):          # YES!
    return magic(r=real, i=imag)

def complex(real, imag = 0.0):        # NO!!
    return magic(r = real, i = imag)
```

- Follow these naming conventions:
  - lower_case_under for variables and functions and methods
  - WordCap for classes
  - ALL_CAPS for constants

And of course, there is more: https://www.python.org/dev/peps/pep-0008/ (https://www.python.org/dev/peps/pep-0008/)

The Zen of Python

In [109]:
```python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do
it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Exercise

- codeacademy (https://www.codecademy.com/learn/python/)

In [ ]:
```python

```