Python basics training

11.12.2018

intive

Marcin Bąkowski
Principal Software Engineer

Table of Contents

Introduction....4
- What is Python?....4
- Why Python?....5
- History of Python....6
- Python implementations....7
- Zen of Python....8
- Python Enhancement Proposals (PEPs)....9

Python on the surface....10
- Installation....11
- Let’s start: Hello world....12
- Python interpreter interactive mode....13
- Python documentation and help functions....14

Builtin types....15
- Boolean type....15
- Strings....16
- Numerical types....20
- Lists....22
- Tuples....27
- Sets....31
- Dictionaries (mappings)....34

Control statements....38
- The If Statement....38
- for loop statement....40
- while loop statement....43
- Functions....44
- Scopes and Namespaces....48
- Classes....51
- Modules....60
- Packages....63
- Errors....64
- Exceptions....65

I/O operations....68
Useful modules and packages....71
Regular expression operations....71
Python requests....75
Datetime....80
Advanced language features....81
Decorators....81
Context managers....84
List comprehension....87
Iterators....89
Generators....91
Python tools....92
virtualenv....92
PIP....93
PEP8....94
unittest....95
Coverage.py....98
Python useful links....100

Introduction

What is Python?

Python: Dynamic programming language which supports multiple different programming paradigms:

- Procedural programming
- Object oriented programming
- Functional programming
- Has a large and comprehensive standard library

Why Python?

Website development, data analysis, server maintenance, numerical analysis, ...

- Syntax is clear, easy to read and learn (almost pseudo code)
- Intuitive object oriented programming
- Full modularity, hierarchical packages
- Comprehensive standard library for many tasks
- Big community
- Most parts of Python are written in C
- Simply extendable via C/C++, wrapping of C/C++ libraries
- Programming speed
- Very good for I/O bound problems

intive

History of Python

- First created in December 1989 by Guido van Rossum
- Python 2.0 released in October 2000 (unicode support, garbage collector, ..)
- Python 3.0 released in December 2008 (no full backward compatibility)
- Current versions: 2.7.15 (end of life: 2020.01.01), 3.7.1

Python implementations

- **CPython** - is the reference implementation of Python, written in C. It compiles Python code to intermediate bytecode which is then interpreted by a virtual machine. CPython provides the highest level of compatibility with Python packages and C extension modules.
- **PyPy** is a Python interpreter implemented in a restricted statically-typed subset of the Python language called RPython. PyPy aims for maximum compatibility with the reference CPython implementation while improving performance.
- **Jython** is a Python implementation that compiles Python code to Java bytecode which is then executed by the JVM (Java Virtual Machine). Additionally, it is able to import and use any Java class like a Python module.
- **IronPython** is an implementation of Python for the .NET framework. It can use both Python and .NET framework libraries, and can also expose Python code to other languages in the .NET framework.
- **PythonNet** for .NET is a package which provides near seamless integration of a natively installed Python installation with the .NET Common Language Runtime (CLR).

Zen of Python

```
In [1]: 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!
```

Marcin Bąkowski
Principal Software Engineer

Python Enhancement Proposals (PEPs)

What is a PEP?

- PEP stands for Python Enhancement Proposal.
- A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment.
- The PEP should provide a concise technical specification of the feature and a rationale for the feature.
- PEP index: https://www.python.org/dev/peps/

Python on the surface

Code used in this training is available on gitbub:
https://github.com/marcin-bakowski-intive/python3-training/

https://github.com/marcin-bakowski-intive/python3-training-exercises

```
$ git clone git@github.com:marcin-bakowski-intive/python3-training.git
$ git clone git@github.com:marcin-bakowski-intive/python3-training-exercises.git
```

intive

Installation

- MS Windows: Go to https://www.python.org/downloads/ and download Python package/installer for your OS
- Linux distributions: most of distributions provides package with python3. If you need different version you can get it from Python downloads page.

Ex:

```
$ sudo apt install python3
```

- Mac OS X: python3 is available to install using homebrew. You can also install Python using installers from Python downloads page.

```
$ brew install python3
```

Let’s start: Hello world

```
#!/usr/bin/env python3
print("Hello world")
```

Let’s execute the *hello_world.py* script:

```
$ python3 hello_world.py
Hello World!!
```

Python interpreter interactive mode

```
$ python3
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("this is simple print")
this is simple print
>>> 1+1
2
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Python interpreter extended alternatives (with auto complete feature and many others):

- Ipython: https://ipython.org/install.html

```
pip install ipython
```

- bpython: https://bpython-interpreter.org/downloads.html

```
pip install bpython
```

Python documentation and help functions

Python online documentation: https://docs.python.org/3/

Python builtin help functions:

- help(), help(module)
- dir(object)
- globals()
- locals()

Builtin types

Boolean type

https://docs.python.org/3/library/stdtypes.html#truth-value-testing

```
a = True
b = False

print("%s AND %s == %s" % (a, b, a and b))
print("%s & %s == %s" % (a, b, a & b))

print("%s OR %s == %s" % (a, b, a or b))
print("%s | %s == %s" % (a, b, a or b))

print("NOT %s == %s" % (a, not a))
```

Strings

https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

- strings are immutable
- sequence of characters, can be accessed like lists or tuples
- there are multiple ways to create a sting instance

Accessing string content:

idx	0	1	2	3	4	5	6	7	8	9	10
-idx	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
char	t	e	s	t		s	t	r	i	n	g

```
>>> test_string = "test string"
>>> test_string2 = 'test string'
>>> test_string_multi_line = """
    test multi lines
    """
>>> test_string3 = str(1.0)
>>> print(test_string[0])  # prints first character: t
t
>>> print(test_string[-1]) # prints last character: g
g
>>> print(test_string[:4]) # print slice of string (first 4 characters) -> test
test
>>> print(test_string[-6:]) # print slice of string (last 6 characters) -> string
string
>>> print(test_string[-11:]) # print "test string" using negative indexing
test string
```

Useful string's methods:

- s.count(sub [, start[, end]]) - count appearance of substrings
- s.strip([chars]) – removes white characters from beginning and end
- s.split([sub [,maxsplit]]) – splits string into sequence of sub-string parts
- s.upper(), s.lower() - upper/lowercase all the characters
- s.replace(old, new[, count]) – replace sub-string with new one

All string methods are available:

- online: https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
- help(str)

String formatting

- using s.format() method

```
# using positional arguments
print(“This is test no. {}”.format(1))

# using kwargs
print(“This is test no. {test_number}”.format(test_number=2))

# print float with 2 digits precision
test_number = 1.0
print(“This is test no: {:.2f}”.format(test_number))
```

https://docs.python.org/3/library/string.html#format-examples

- printf-style formatting

```
test_number = 1.0
# print float as string
print(“This is test no: %s” % test_number)

# print float with 2 digits precision
print(“This is test no: %.2f” % test_number)
```

https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

Numerical types

https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

- int – integer numbers
- float – floating point numbers (corresponds double in C)
- complex – complex numbers

Code:

```
a = 10
b = 5.5

print("Sum: %.2f " % sum([a, b]))
print("Integer sum: %d " % sum([a, b]))
print("Divide: %.3f" % (a / 3.0))
print("Decimal {value}, binary {value:b}".format(value=a))
```

Operators on numbers:

- Basic arithmetic: +, - , * , /
- Div and modulo: //, %, divmod(x, y)
- Power: pow(x, y), x**y
- Absolute value: abs(value)
- Round value: round(value)
- Conversion: int(value), float(value), complex(value)

Lists

https://docs.python.org/3/library/stdtypes.html#lists

- list is a finite sequence of items
- lists are not required to be homogeneous, i.e., the items could be of different types.
- lists are mutable, elements can be add/removed at any time

Useful list methods:

- Append a single element x to the “s” list: s.append(x)
- Extend “s” list with a s2 list: s.extend(s2)
- Count appearance of a “x” element: s.count(x)
- Position of an element: s.index(x[, min[, max]])
- Check if element is in the list: x in s
- Insert element “x” at position “i”: s.insert(i, x)
- Remove and return element at position: s.pop([i])
- Delete element: s.remove(x)
- Reverse list: s.reverse()
- Sort: s.sort([cmp[, key[, reverse]]])
- Sum of the elements: sum(s)

Code:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> print("is plum in fruits: %s" % ("plum" in fruits))
is plum in fruits: False
```

Iterating over the list’s elements:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> for fruit in fruits:
...     print("This is a %s" % fruit)
...
This is a orange
This is a apple
This is a pear
This is a banana
This is a kiwi
This is a apple
This is a banana

>>> while fruits:
...     print("Let's eat %s" % fruits.pop())
...
Let's eat banana
Let's eat apple
Let's eat kiwi
Let's eat banana
Let's eat pear
Let's eat apple
Let's eat orange

>>> print(fruits)
[]
```

Accessing list elements:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']

>>> print("First fruit: %s" % fruits[0])
First fruit: orange

>>> print("Last fruit: %s" % fruits[-1])
Last fruit: banana

>>> print("There are %d fruits" % len(fruits))
There are 7 fruits

>>> print("Let's take first 2 fruits: %s" % fruits[:2])
Let's take first 2 fruits: ['orange', 'apple']

>>> print("Let's take last 3 fruits: %s" % fruits[-3:])
Let's take last 3 fruits: ['kiwi', 'apple', 'banana']

>>> print("Let's take every second fruit: %s" % fruits[::2])
Let's take every second fruit: ['orange', 'pear', 'kiwi', 'banana']
```

Tuples

https://docs.python.org/3/library/stdtypes.html#tuple

- tuples is a finite sequence of items
- tuples are not required to be homogeneous, i.e., the items could be of different types.
- tuples are immutable, once created tuples cannot be modified

Useful tuples methods:

- Count appearance of an element: s.count(x)
- Position of an element: s.index(x[, min[, max]])
- Sum of the elements: sum(s)
- Check if element is in the tuple: x in s

Code examples:

```
>>> fruits = ('orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana')

>>> print("apple count: %s" % fruits.count('apple'))
apple count: 2

>>> print("banana index: %s" % fruits.index('banana'))
banana index: 3

>>> for fruit in fruits:
...     print("This is a %s" % fruit)
...
This is a orange
This is a apple
This is a pear
This is a banana
This is a kiwi
This is a apple
This is a banana
>>> print("There are %d fruits" % len(fruits))
There are 7 fruits
```

```
>>> fruits = ('orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana')

>>> print("First fruit: %s" % fruits[0])
First fruit: orange

>>> print("Last fruit: %s" % fruits[-1])
Last fruit: banana

>>> print("There are %d fruits" % len(fruits))
There are 7 fruits

>>> print("Let's take first 2 fruits: %s" % (fruits[:2],))
Let's take first 2 fruits: ('orange', 'apple')

>>> print("Let's take last 3 fruits: %s" % (fruits[-3:],))
Let's take last 3 fruits: ('kiwi', 'apple', 'banana')

>>> print("Let's take every second fruit: %s" % (fruits[::2],))
Let's take every second fruit: ('orange', 'pear', 'kiwi', 'banana')

>>> print("is plum in fruits: %s" % ("plum" in fruits))
is plum in fruits: False

>>> fruit[0] = "plum"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Sets

https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

- A set object is an unordered collection of distinct hashable objects
- Being an unordered collection, sets do not record element position or order of insertion
- There are currently two built-in set types, `set` and `frozenset`.
- The `set` type is mutable
- The `frozenset` type is immutable and hashable — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Useful sets methods:

- `issubset`(*other*) - test whether every element in the set is in *other*.
- `union`(**others*) - return a new set with elements from the set and all others.
- `intersection`(**others*) - return a new set with elements common to the set and all others.
- `difference`(**others*) - return a new set with elements in the set that are not in the others

Code example:

```
>>> basket_1 = {'orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana'}
>>> print("There are %d unique fruits." % len(basket_1))
There are 5 unique fruits.

>>> for fruit in basket_1:
...     print("This is a %s" % fruit)
...
This is a banana
This is a apple
This is a kiwi
This is a orange
This is a pear

>>> basket_2 = {'orange', 'apple'}
>>> print("Fruits in both buckets: %s" % basket_1.intersection(basket_2))
Fruits in both buckets: {'orange', 'apple'}
>>> print("Fruits only in 1rst bucket: %s" % basket_1.difference(basket_2))
Fruits only in 1rst bucket: {'pear', 'banana', 'kiwi'}

>>> print("Are all fruits from 2nd basket in the 1rst bucket?: %s" %
basket_2.issubset(basket_1))
Are all fruits from 2nd basket in the 1rst bucket?: True

>>> print("is plum in fruits: %s" % ("plum" in basket_1))
is plum in fruits: False
```

Dictionaries (mappings)

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

- A mapping object maps hashable values to arbitrary objects.
- Mappings are mutable objects.
- A dictionary's keys are *almost* arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys.

Useful dictionary methods:

- `get`(*key*[, *default*]) - return the value for *key* if *key* is in the dictionary, else *default*.
- `items`() - return a new view of the dictionary's items (`(key, value)` pairs)
- `keys`() - return a new view of the dictionary's keys.
- `update`([*other*]) - update the dictionary with the key/value pairs from *other*, overwriting existing keys.
- `Values`() - return a new view of the dictionary's values.

Code example:

```
>>> basket_1 = {'orange': 1,
...             'apple': 2,
...             'pear': 1,
...             'banana': 3,
...             'kiwi': 2}

>>> print(basket_1)
{'orange': 1, 'apple': 2, 'pear': 1, 'banana': 3, 'kiwi': 2}

>>> for fruit, quantity in basket_1.items():
...     print("There are %s %s" % (quantity, fruit))
...
There are 1 orange
There are 2 apple
There are 1 pear
There are 3 banana
There are 2 kiwi
>>> basket_2 = {'orange': 0, 'apple': 10}
>>> basket_1.update(basket_2)
```

```
>>> for fruit in basket_1:
...     print("There are %s %s" % (basket_1[fruit], fruit))
...
There are 0 orange
There are 10 apple
There are 1 pear
There are 3 banana
There are 2 kiwi

>>> print(basket_1)
{'orange': 0, 'apple': 10, 'pear': 1, 'banana': 3, 'kiwi': 2}

>>> print("There are %s fruits in basket 1" % sum(basket_1.values()))
There are 16 fruits in basket 1

>>> print("is plum in fruits: %s" % ("plum" in basket_1))
is plum in fruits: False

>>> print("Get plum: %s" % basket_1.get('plum', "sorry, no plum found"))
Get plum: sorry, no plum found

>>> print("Get plum: %s" % basket_1['plum'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'plum'
```

Control statements

The If Statement

```
if …:
   pass

if …:
   pass
else:
   pass

if …:
   pass:
elif …:
   pass
else:
   pass

result = True if condition else False
```

Marcin Bąkowski
Principal Software Engineer

Code example:

```
>>> fruit = "apple"
>>> if fruit == "orange":
...     print("This is an orange")
... elif fruit == "apple":
...     print("This is an apple")
... else:
...     print("Unknown fruit")
...
This is an apple

>>> print("Is this fruit an apple? %s" % (fruit == "apple"))
Is this fruit an apple? True

>>> print("Is this fruit an apple? %s" % ("yes" if fruit == "apple" else "no"))
Is this fruit an apple? yes
```

`for` loop statement

- `for .. in iterable`
- `break` – breaks the loop
- `continue` – go to next item in the loop
- `else` – optional block, it is executed only in case loop was not stopped using break statement

```
for ... in iterable:
    instruction
else:
    instruction
```

Code example:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi']
>>> print("Available fruits: %s" % ", ".join(fruits))
Available fruits: orange, apple, pear, banana, kiwi

>>> stop_fruit = input("Choose fruit, which breaks the loop? ")
Choose fruit, which breaks the loop? Banana

>>> skip_fruit = input("Choose fruit, which skip the loop? ")
Choose fruit, which skip the loop? Apple
```

```
>>> for fruit in fruits:
...     if fruit == skip_fruit:
...         print("Found %s, go to next item" % skip_fruit)
...         continue
...     elif fruit == stop_fruit:
...         print("Found %s, breaking the look" % stop_fruit)
...         break
...     print("Fruit: %s" % fruit)
... else:
...     print("Loop was not stopped with break")
...
Fruit: orange
Found apple, go to next item
Fruit: pear
Fruit: banana
Fruit: kiwi
Loop was not stopped with break

>>> fruits_names_contain_letter_a = [fruit for fruit in fruits if "a" in fruit]
>>> print(fruits_names_contain_letter_a)
['orange', 'apple', 'pear', 'banana']
```

`while` loop statement

- `while condition: pass`
- `break` – breaks the loop
- `continue` – go to next item in the loop
- while loop is most often used for infinitive loops (while True: pass)

Code example:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi']
>>> while fruits:
...     print("Current fruit: %s, there are still %d fruits left" % (fruits.pop(),
len(fruits)))
...
Current fruit: kiwi, there are still 4 fruits left
Current fruit: banana, there are still 3 fruits left
Current fruit: pear, there are still 2 fruits left
Current fruit: apple, there are still 1 fruits left
Current fruit: orange, there are still 0 fruits left
```

Functions

- Functions accept arbitrary objects as parameters and return values
- Types of parameters and return values are unspecified
- Functions without explicit return value return None

Code example:

```
>>> def hello_world():
...     print("Hello World !!")
...

>>> result = hello_world()
Hello World !!
>>> print(result)
None

>>> def get_sum(x, y):
...     return x + y
...

>>> s = get_sum(1, 2)
>>> print(s)
3

>>> def noop():
...     pass
...
>>> noop()
```

Functions arguments:

- positional arguments

```
>>> def get_sum(x, y):
...     return x + y
...

>>> s = get_sum(1, 2)
```

- keyword arguments

```
>>> def get_sum(x, y):
...     return x + y
...

>>> s = get_sum(x=1, y=2)
```

- default arguments

```
>>> def increment(value, step=1):
...     return value + step
...
>>> val = increment(10)
>>> print(val)
```

- arbitrary arguments

```
>>> def sum_elements(*args, **kwargs):
...     odd_only = kwargs.get("odd_only", None)
...     items = [v for v in args if v % 2 == 0] if odd_only else args
...     return sum(v for v in items)
...
>>> val = sum_elements(1, 2, 3, 4, odd_only=True)
>>> print(val)
6

>>> val = sum_elements(odd_only=True, 1, 2, 3, 4)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Scopes and Namespaces

- *namespace* is a mapping from names to objects
- Namespaces are created at different moments and have different lifetimes.
- The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted.
- The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits.
- The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)
- The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. Recursive invocations each have their own local namespace.

- A *scope* is a textual region of a Python program where a namespace is directly accessible.
- `import` statements and function definitions bind the module or function name in the local scope.
- `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there
- the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

There are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

Code example:

```
In [1]: def scope_test():
   ...:     def do_local():
   ...:         spam = "local spam"
   ...:
   ...:     def do_nonlocal():
   ...:         nonlocal spam
   ...:         spam = "nonlocal spam"
   ...:
   ...:     def do_global():
   ...:         global spam
   ...:         spam = "global spam"
   ...:
   ...:     spam = "test spam"
   ...:     do_local()
   ...:     print("After local assignment:", spam)
   ...:     do_nonlocal()
   ...:     print("After nonlocal assignment:", spam)
   ...:     do_global()
   ...:     print("After global assignment:", spam)
   ...:
   ...: scope_test()
   ...: print("In global scope:", spam)
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Classes

- Classes provide a means of bundling data and functionality together.
- Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made.
- Each class instance can have attributes attached to it for maintaining its state.
- Class instances can also have methods (defined by its class) for modifying its state.
- Overloading of methods not possible
- Attribute, which starts with leading underscore _ are considered as non-public
- Attribute, which name has at least two leading underscores is textually replaced with `__name → _classname__name` (name mangling)

Code example:

```
>>> class Empty:    # empty class
...     pass
...
>>>
>>> class Point:
...     def __init__(self, x, y):   # constructor
...         self.x = x
...         self.y = y
...
>>>
>>> p = Point(2.0, 3.0)
>>> print(p.x, p.y)
2.0 3.0
```

Instance attributes and methods

```
In [3]: import math

In [4]: class Point:
   ...:     def __init__(self, x, y):
   ...:         self.x = x
   ...:         self.y = y
   ...:
   ...:     def distance(self, point):
   ...:         return math.sqrt((self.x - point.x) ** 2 + (self.y - point.y) **
2)
   ...:
   ...:     def __str__(self):
   ...:         return "Point({point.x:.2f}, {point.y:.2f})".format(point=self)
   ...:
   ...:
   ...: p1, p2 = Point(2.0, 3.0), Point(1.0, 1.0)
   ...: print("Distance between %s and %s is %.2f" % (p1, p2, p1.distance(p2)))
   ...:
   ...:
   ...:
Distance between Point(2.00, 3.00) and Point(1.00, 1.00) is 2.24
```

Class variables and methods

- class variables and methods are available from both class and its object instances
- be careful while using mutable type as class variable, because they are shared across all class object instances.

```
In [5]: class Point:
   ...:     display_name = ""
   ...:
   ...:     # it's not safe to use mutable types like list or dict as class
   ...:     # attributes (shared across the all class instances)
   ...:     coordinates = []
   ...:
   ...:     # private attribute starts with double underscore
   ...:     __private_name = “just a private name”
   ...:
   ...:     def __init__(self, x, y):
   ...:         self.x = x
   ...:         self.y = y
   ...:         self.coordinates.append((x, y))
   ...:
   ...:     def distance(self, point):
   ...:         return math.sqrt((self.x - point.x) ** 2 + (self.y - point.y) **
2)
   ...:
```

```
   ...:     @classmethod
   ...:     def help(cls):
   ...:         print("This is a class method example")
   ...:
   ...:     def __str__(self):
   ...:         return "Point[{point.display_name}]({point.x:.2f},
{point.y:.2f})".format(point=self)
   ...:
   ...:
   ...: p1, p2 = Point(2.0, 3.0), Point(1.0, 1.0)
   ...: p1.display_name = "p1"
   ...: p2.display_name = "p2"
   ...: print("Distance between %s and %s is %.2f" % (p1, p2, p1.distance(p2)))
   ...:
   ...: print("P1 coordinates: %s" % p1.coordinates)
   ...: print("P2 coordinates: %s" % p2.coordinates)
   ...:
Distance between Point[p1](2.00, 3.00) and Point[p2](1.00, 1.00) is 2.24
P1 coordinates: [(2.0, 3.0), (1.0, 1.0)]
P2 coordinates: [(2.0, 3.0), (1.0, 1.0)]

In [6]: p1.help()
This is a class method example

In [7]: Point.help()
This is a class method example
```

Class inheritance

```
In [8]: class Point3D(Point):
   ...:     def __init__(self, x, y, z):
   ...:         super().__init__(x, y)
   ...:         self.z = z
   ...:
   ...:     def distance(self, point):
   ...:         return math.sqrt((self.x - point.x) ** 2 +
   ...:                          (self.y - point.y) ** 2 +
   ...:                          (self.z - point.z) ** 2)
   ...:
   ...:     def __str__(self):
   ...:         return "Point[{point.display_name}]({point.x:.2f}, {point.y:.2f},
{point.z:.2f})".format(point=self)
   ...:
In [11]: p_3d = Point3D(1.0, 2.0, 3.0)

In [12]: print(p_3d)
Point[](1.00, 2.00, 3.00)
```

```
In [13]: class Base1:
    ...:     def base_1(self):
    ...:         print("Base 1 method")
    ...:
    ...:
    ...: class Base2:
    ...:     def base_2(self):
    ...:         print("Base 2 method")
    ...:
    ...:
    ...: class Extended(Base1, Base2):
    ...:     pass
    ...:
    ...:
    ...: obj = Extended()
    ...: obj.base_1()
    ...: obj.base_2()
Base 1 method
Base 2 method
```

Object dynamic attributes

Attributes can be added/removed to python objects at runtime.

Object’s attributes methods:

- hasattr – checks if objects has given attribute
- getattr – gets given object’s attribute
- setattr – set object’s attribute
- delattr – remove object’s attribute

Code example:

```
In [1]: class Empty:
   ...:     pass
   ...:

In [2]: empty_obj = Empty()
   ...: print("Does empty object have 'name' attribute? %s" % hasattr(empty_obj,
"name"))
   ...: print("empty_obj.name: '%s'" % getattr(empty_obj, "name", ""))
Does empty object have 'name' attribute? False
empty_obj.name: ''


In [3]: setattr(empty_obj, 'name', 'EMPTY')
   ...: print("Does empty object have 'name' attribute? %s" % hasattr(empty_obj,
"name"))
   ...: print("empty_obj.name: '%s'" % getattr(empty_obj, "name", ""))
Does empty object have 'name' attribute? True
empty_obj.name: 'EMPTY'


In [4]: delattr(empty_obj, "name")
   ...: print("Does empty object have 'name' attribute? %s" % hasattr(empty_obj,
"name"))
   ...: print("empty_obj.name: '%s'" % getattr(empty_obj, "name", ""))
Does empty object have 'name' attribute? False
empty_obj.name: ''
```

Modules

- Every Python script can be imported as a module.
- Top level instructions are executed during import
- module.__file__ contains real path of the module

```
In [1]: from code_examples.basic_operators import functions
Hello World !!
get_sum(): 2

In [2]: functions.hello_world()
Hello World !!


In [8]: functions.__file__
Out[8]:
'/home/marcin/projects/intive/python3-training/code_examples/basic_operators/
functions.py'
```

Modules are searched for in (see sys.path):

- The directory of the running script
- Directories in the environment variable PYTHONPATH
- Installation-dependent directories

```
In [3]: import sys

In [4]: sys.path
Out[4]:
['',
 '/usr/lib/python36.zip',
 '/usr/lib/python3.6',
 '/usr/lib/python3.6/lib-dynload',
 '/home/marcin/.local/lib/python3.6/site-packages',
 '/usr/local/lib/python3.6/dist-packages',
 '/usr/lib/python3/dist-packages',
 '/usr/lib/python3/dist-packages/IPython/extensions',
 '/home/marcin/.ipython']
```

Avoiding namespace conflicts during the import when importing multiple modules with the same name:

```
# import and use it as an alias

In [1]: from code_examples.basic_operators import functions as fn
Hello World !!
get_sum(): 2

In [2]: fn.hello_world()
Hello World !!
```

Packages

Packages are sub-directories, which in each package directory there is an __init__.py file. __init__.py file my be empty.

code_examples package directory structure:

```
code_examples/
├── basic_operators
│   ├── classes.py
│   ├── conditionals.py
│   ├── functions.py
│   ├── __init__.py
│   ├── loops.py
│   └── scopes.py
├── hello_world
│   ├── hello_world.py
│   └── __init__.py
└── __init__.py
```

Errors

- Parsing errors: Program will not be executed.
- Mismatched or missing parenthesis
- Missing or misplaced semicolons, colons, commas
- Indentation errors

```
In [1]: print(")
  File "<ipython-input-1-93263f49d639>", line 1
    print(")
          ^
SyntaxError: EOL while scanning string literal
```

Exceptions

Exceptions occur at runtime.

```
In [2]:  1 / 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-bc757c3fda29> in <module>()
----> 1 1 / 0

ZeroDivisionError: division by zero
```

Handling exceptions:

```
In [3]: try:
   ...:     1 / 0
   ...: except (ZeroDivisionError, TypeError, ValueError) as e:
   ...:     print(e)
   ...: else:
   ...:     print ("no error")
   ...: finally:
   ...:     print ("End of try.")
   ...:
division by zero
```

Raising exception:

```
In [4]: class MyCustomError(Exception):
   ...:     pass
   ...:
   ...:
   ...: raise MyCustomError("ERROR!!!")
   ...:
---------------------------------------------------------------------------
MyCustomError                             Traceback (most recent call last)
<ipython-input-4-384ff0a2202a> in <module>()
      3
      4
----> 5 raise MyCustomError("ERROR!!!")

MyCustomError: ERROR!!!
```

Builtin exceptions: https://docs.python.org/3/library/exceptions.html

I/O operations

- File handling (open/close) can be done by the context manager *with*.
- After finishing the with block the file object is closed, even if an exception occurred inside the block.

```
In [5]: with open("/var/log/syslog") as src_file:
   ...:     for line in src_file:
   ...:         if "systemd" in line:
   ...:             print(line.strip())
   ...:
Nov 30 09:04:33 marcin-lapek systemd[3181]: Starting Virtual filesystem metadata
service...
Nov 30 09:04:33 marcin-lapek systemd[3181]: Started Virtual filesystem metadata
service.
```

Alternative method:

```
In [6]: src_file = open("/var/log/syslog")
   ...: for line in src_file:
   ...:     if "systemd" in line:
   ...:         print(line.strip())
   ...: src_file.close()
   ...:
Nov 30 09:04:33 marcin-lapek systemd[3181]: Starting Virtual filesystem metadata
service...
Nov 30 09:04:33 marcin-lapek systemd[3181]: Started Virtual filesystem metadata
service.
```

Operations on Files:

- open
  - Read mode: r
  - Write mode (new file): w
  - Write mode, appending to the end: a
  - Handling binary files: e.g. rb
  - Read and write (update): r+
- Read: f.read([size])
- Read a line: f.readline()
- Read multiple lines: f.readlines([sizehint])
- Write: f.write(str)
- Write multiple lines: f.writelines(sequence)
- Close file: f.close()

Useful modules and packages

Regular expression operations

Formal language for pattern matching in strings: https://docs.python.org/3/library/re.html

Code examples:

```
In [3]: import re

In [4]:
   ...: def print_match_status(re_compiled_pattern, string):
   ...:     match = re_compiled_pattern.match(string)
   ...:     print("Does '%s' match pattern %s? %s" % (string, re_compiled_pattern,
bool(match)))
   ...:     return match

In [3]: print(re.findall("a.c", " abc aac aa abb a c "))
['abc', 'aac', 'a c']
```

```
In [4]: pattern = re.compile("^c.*t$")
   ...: print_match_status(pattern, "cat")
   ...: print_match_status(pattern, "caaat")
   ...: print_match_status(pattern, "at")
   ...:
Does 'cat' match pattern re.compile('^c.*t$')? True
Does 'caaat' match pattern re.compile('^c.*t$')? True
Does 'at' match pattern re.compile('^c.*t$')? False

In [5]: pattern = re.compile("^[a-z]+$")
   ...: print_match_status(pattern, "123")
   ...: print_match_status(pattern, "cat")
   ...: print_match_status(pattern, "Cat")
   ...:
Does '123' match pattern re.compile('^[a-z]+$')? False
Does 'cat' match pattern re.compile('^[a-z]+$')? True
Does 'Cat' match pattern re.compile('^[a-z]+$')? False

In [6]: pattern = re.compile("^[^a-z]+$")
   ...: print_match_status(pattern, "123")
   ...: print_match_status(pattern, "cat")
   ...: print_match_status(pattern, "Cat")
   ...:
Does '123' match pattern re.compile('^[^a-z]+$')? True
Does 'cat' match pattern re.compile('^[^a-z]+$')? False
Does 'Cat' match pattern re.compile('^[^a-z]+$')? False
```

```
In [7]: pattern = re.compile("^\w+$")
   ...: print_match_status(pattern, "123")
   ...: print_match_status(pattern, "cat")
   ...: print_match_status(pattern, "cat!")
   ...:
Does '123' match pattern re.compile('^\\w+$')? True
Does 'cat' match pattern re.compile('^\\w+$')? True
Does 'cat!' match pattern re.compile('^\\w+$')? False

In [8]: pattern = re.compile("^\d+$")
   ...: print_match_status(pattern, "123")
   ...: print_match_status(pattern, "cat")
   ...:
Does '123' match pattern re.compile('^\\d+$')? True
Does 'cat' match pattern re.compile('^\\d+$')? False

In [9]: pattern = re.compile("^(\w+) (\w+)$")
   ...: print_match_status(pattern, "123")
   ...: match = print_match_status(pattern, "funny cat")
   ...:
Does '123' match pattern re.compile('^(\\w+) (\\w+)$')? False
Does 'funny cat' match pattern re.compile('^(\\w+) (\\w+)$')? True
```

```
In [10]: # print whole match
    ...: print(match.group(0))
    ...: # print matching 1rst group
    ...: print(match.group(1))
    ...: # print matching 2nd group
    ...: print(match.group(2))
    ...:
funny cat
funny
cat

In [11]: pattern = re.compile("^(\w{1,3}) (\w{3})$")
    ...: print_match_status(pattern, "funny cat")
    ...: match = print_match_status(pattern, "a cat")
    ...: print(match.groups())
    ...:
Does 'funny cat' match pattern re.compile('^(\\w{1,3}) (\\w{3})$')? False
Does 'a cat' match pattern re.compile('^(\\w{1,3}) (\\w{3})$')? True
('a', 'cat')

In [12]: match = print_match_status(pattern, "bad dog")
    ...: print(match.groups())
    ...:
Does 'bad dog' match pattern re.compile('^(\\w{1,3}) (\\w{3})$')? True
('bad', 'dog')
```

Python requests

Requests: HTTP for Humans™

http://docs.python-requests.org/en/master/

Code example:

```
In [1]: import requests

In [2]: payload = {'key1': 'value1', 'key2': 'value2'}
   ...: headers = {"user-agent": "dummy-test"}
   ...: response = requests.get('https://httpbin.org/get', params=payload,
headers=headers)
   ...: print(response)

<Response [200]>
```

```
In [3]: print("Content-type: %s" % response.headers['content-type'])
   ...: print("Raw binary content: ")
   ...: print(response.content)
Content-type: application/json
Raw binary content:
b'{\n  "args": {\n    "key1": "value1", \n    "key2": "value2"\n  }, \n
"headers": {\n    "Accept": "*/*", \n    "Accept-Encoding": "gzip, deflate", \n
"Connection": "close", \n    "Host": "httpbin.org", \n    "User-Agent": "dummy-
test"\n  }, \n  "origin": "195.111.111.111", \n  "url": "https://httpbin.org/get?
key1=value1&key2=value2"\n}\n'

In [4]: print("Content as json: ")
   ...: print(response.json())
   ...: print("Response time %s" % response.elapsed)
Content as json:
{'args': {'key1': 'value1', 'key2': 'value2'}, 'headers': {'Accept': '*/*',
'Accept-Encoding': 'gzip, deflate', 'Connection': 'close', 'Host': 'httpbin.org',
'User-Agent': 'dummy-test'}, 'origin': '195.111.111.111', 'url':
'https://httpbin.org/get?key1=value1&key2=value2'}
Response time 0:00:00.751960
```

```
In [5]: payload = {'key1': 'value1', 'key2': 'value2'}
   ...: response = requests.post('https://httpbin.org/post', json=payload)
   ...: print(response)
   ...: print(response.json())
   ...: print("Response time %s" % response.elapsed)
   ...: # if response is ok then raise_for_status won't raise an error
   ...: response.raise_for_status()
<Response [200]>
{'args': {}, 'data': '{"key1": "value1", "key2": "value2"}', 'files': {}, 'form':
{}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate', 'Connection':
'close', 'Content-Length': '36', 'Content-Type': 'application/json', 'Host':
'httpbin.org', 'User-Agent': 'python-requests/2.20.1'}, 'json': {'key1': 'value1',
'key2': 'value2'}, 'origin': '195.111.111.111', 'url': 'https://httpbin.org/post'}
Response time 0:00:00.610978
```

```
In [6]: response = requests.post('https://httpbin.org/get')
   ...: print(response)
   ...: print("Response headers %s" % response.headers)
   ...: if not response.ok:
   ...:     print("response status is %s, error will be raised" %
response.status_code)
   ...: response.raise_for_status()
   ...:
<Response [405]>
Response headers {'Connection': 'keep-alive', 'Server': 'gunicorn/19.9.0', 'Date':
'Wed, 05 Dec 2018 11:45:25 GMT', 'Content-Type': 'text/html', 'Allow': 'OPTIONS,
GET, HEAD', 'Content-Length': '178', 'Access-Control-Allow-Origin': '*', 'Access-
Control-Allow-Credentials': 'true', 'Via': '1.1 vegur'}
response status is 405, error will be raised
---------------------------------------------------------------------------
HTTPError                                 Traceback (most recent call last)
<ipython-input-6-42b0a79d2703> in <module>
      4 if not response.ok:
      5     print("response status is %s, error will be raised" %
response.status_code)
----> 6 response.raise_for_status()

~/.virtualenvs/python-basics-training/lib/python3.6/site-packages/requests/
models.py in raise_for_status(self)
    938
    939         if http_error_msg:
--> 940             raise HTTPError(http_error_msg, response=self)
    941
```

```
    942     def close(self):

HTTPError: 405 Client Error: METHOD NOT ALLOWED for url: https://httpbin.org/get
```

Datetime

Basic date and time types. The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways.

Code examples:

```
In [7]: from datetime import datetime, timedelta
   ...:
   ...: utc_now = datetime.utcnow()
   ...: local_now = datetime.now()

In [8]: print("UTC now: %s" % utc_now)
   ...: print("Now: %s" % local_now)
UTC now: 2018-12-05 20:03:07.098138
Now: 2018-12-05 21:03:07.098240

In [9]: datetime_str = local_now.strftime("%A, %d. %B %Y %I:%M%p")
   ...: print("Datetime string in custom format: %s" % datetime_str)
Datetime string in custom format: Wednesday, 05. December 2018 09:03PM

In [10]: dt_parsed = datetime.strptime(datetime_str, "%A, %d. %B %Y %I:%M%p")
    ...: print("Parsed datetime: %s" % dt_parsed)
Parsed datetime: 2018-12-05 13:03:00
```

Advanced language features

Decorators

Decorators are functions which modify the functionality of other functions. They help to make our code shorter and more Pythonic.

Code example:

```
In [5]: def shout(fn):
   ...:     """ Function appends '!!!' to the original result"""
   ...:     def decorated(name):
   ...:         ret = fn(name)
   ...:         print("Function '%s' returned '%s'" % (fn.__name__, ret))
   ...:         return "%s!!!" % ret
   ...:     return decorated
   ...:

In [6]: @shout
   ...: def hello(name):
   ...:     return "Hello %s" % name.capitalize()
   ...:

In [7]: print("Call results: '%s'" % hello("marcin"))
```

```
   ...: print(hello)
Function 'hello' returned 'Hello Marcin'
Call results: 'Hello Marcin!!!'
<function shout.<locals>.decorated at 0x7f0ca3ed4ea0>
```

`functools` — Higher-order functions and operations on callable objects

https://docs.python.org/3/library/functools.html

```
In [10]: from functools import wraps

In [11]: def shout2(fn):
    ...:     """ Function appends '!!!' to the original result"""
    ...:     @wraps(fn)
    ...:     def decorated(name):
    ...:         ret = fn(name)
    ...:         print("Function '%s' returned '%s'" % (fn.__name__, ret))
    ...:         return "%s!!!" % ret
    ...:     return decorated
    ...:

In [12]: @shout2
    ...: def hello2(name):
    ...:     return "Hello %s" % name.capitalize()
    ...:

In [13]: print("Call results: '%s'" % hello2("marcin"))
    ...: print(hello2)
Function 'hello2' returned 'Hello Marcin'
Call results: 'Hello Marcin!!!'
<function hello2 at 0x7f0ca3e5ba60>
```

Context managers

Context managers allow you to allocate and release resources precisely when you want to. The most widely used example of context managers is the `with` statement. Suppose you have two related operations which you'd like to execute as a pair, with a block of code in between. Context managers allow you to do specifically that.

Example:

```
file = open('some_file', 'w')
try:
    file.write('Hello!!')
finally:
    file.close()
```

Code above can be replaced with open() used as context manager:

```
with open('some_file', 'w') as opened_file:
    opened_file.write('Hello!')
```

A context manager has to have an `__enter__` and `__exit__` method defined.

Code example:

```
In [14]: class File(object):
    ...:     def __init__(self, file_name, method):
    ...:         self.file_obj = open(file_name, method)
    ...:
    ...:     def __enter__(self):
    ...:         return self.file_obj
    ...:
    ...:     def __exit__(self, type, value, traceback):
    ...:         self.file_obj.close()
    ...:

In [15]:
    ...: with File('demo.txt', 'w') as opened_file:
    ...:     opened_file.write('Hello!!!')
```

Python standard library provides `contextlib` - utilities for `with`-statement contexts.

Code example:

```
In [16]: from contextlib import contextmanager

In [17]: @contextmanager
    ...: def open_file(file_name, method):
    ...:     f = open(file_name, method)
    ...:     yield f
    ...:     f.close()
    ...:

In [18]: with open_file('demo2.txt', 'w') as opened_file:
    ...:     opened_file.write('Hello!!!')
```

List comprehension

List comprehension allows sequences to be build by sequences. Instead of using **for** loop.

```
In [19]: # fill list in for loop
    ...: examples = []
    ...: for i in range(10):
    ...:     examples.append(i)
    ...: print(examples)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Code above can be replaced with:

```
In [20]: # fill list in-line
    ...: examples2 = [i for i in range(10)]
    ...: print(examples2)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehension with conditional statement:

```
In [21]: # conditional list comprehension
    ...: examples3 = [i for i in range(10) if i < 3]
    ...: print(examples3)
[0, 1, 2]
```

Iterators

Iterators are “dynamic” lists, which allows to access its content items looping then sequentially.

- The __iter__ method for obj is called, return an iterator.
- On each loop cycle the iterator.__next__() method will be called.
- The exception StopIteration is raised when there are no more elements.
- Advantage: Memory efficient (access time)

Code example:

```
In [22]: class Reverse:
    ...:     def __init__(self, data):
    ...:         self.data = data
    ...:         self.index = len(data)
    ...:
    ...:     def __iter__(self):
    ...:         return self
    ...:
    ...:     def __next__(self):
    ...:         if self.index == 0:
    ...:             self.index = len(self.data)
    ...:             raise StopIteration
    ...:         self.index = self.index - 1
    ...:         return self.data[self.index]
    ...:

In [23]: for char in Reverse("SPAM"):
    ...:     print(char, end=" ")
    ...:
M A P S
```

Generators

- Simpler way to create iterators
- Methods uses the *yield* statement
- breaks at this point, returns element and continues there on the next iterator.__next__() call.
- Generators are iterators, but you can only iterate over them once. It's because they do not store all the values in memory, they generate the values on the fly.

Code example:

```
In [24]: def reverse(data):
    ...:     for element in data[:: -1]:
    ...:         yield element
    ...:

In [25]: for char in reverse("SPAM"):
    ...:     print(char, end=" ")
    ...:
    ...:
M A P S
```

Python tools

virtualenv

`virtualenv` creates isolated environments for your python application and allows you to install Python libraries in that isolated environment instead of installing them globally.

Install virtualenv:

```
$ pip install virtualenv
```

The most important commands are:

```
$ virtualenv myproject
$ source myproject/bin/activate
$ deactivate
```

intive

PIP

PIP is a package manager for Python packages and modules. PIP can be used together with *virtualenv* to manage your application's dependencies.

Check if pip is already installed:

```
$ python -m ensurepip --default-pip
```

Create isolated virtualenv and install *requests* package:

```
$ git clone git@github.com:marcin-bakowski-intive/python3-training.git
$ cd python3-traning
$ virtualenv python3_training
$ source python3_training/bin/activate
$ pip install requests
# freeze your dependencies in requirements.txt
$ pip freeze >requirements.txt
# to install dependencies from requirements.txt
$ pip install -r requirements.txt
```

PEP8

PEP8 is a style guide for Python and gives coding conventions for code layout, string quotes, comments, etc… pycodestyle is a tool to check your Python code against some of the style conventions in PEP8.

Installation:

```
$ pip3 install pycodestyle
$ pycodestyle code_examples/modules/datetime_examples.py
code_examples/modules/datetime_examples.py:8:66: E225 missing whitespace around
operator
```

unittest

The `unittest` unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

Example:

```
$ cd code_examples/
$ python -m unittest
FFFF
======================================================================
FAIL: test_round_2_points_precision
(excercises.test.test_builtin_types.BuiltinTypeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/marcin/projects/intive/python3-training/code_examples/excercises/
test/test_builtin_types.py", line 8, in test_round_2_points_precision
    self.assertEqual(round_2_points_precision("123.111111"), 123.11)
AssertionError: None != 123.11
```

```
======================================================================
FAIL: test_fibonacci_sequence (excercises.test.test_fibonacci.FibonacciTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/marcin/projects/intive/python3-training/code_examples/excercises/
test/test_fibonacci.py", line 10, in test_fibonacci_sequence
    self.assertEqual(get_fibonacci_nth_value(2), 1)
AssertionError: None != 1

======================================================================
FAIL: test_get_odd_squares_list (excercises.test.test_word_count.OperatorsTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/marcin/projects/intive/python3-training/code_examples/excercises/
test/test_word_count.py", line 11, in test_get_odd_squares_list
    (2, 4)
AssertionError: First sequence is not a list: None

======================================================================
FAIL: test_get_word_count_dict (excercises.test.test_word_count.OperatorsTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/marcin/projects/intive/python3-training/code_examples/excercises/
test/test_word_count.py", line 20, in test_get_word_count_dict
    "chicken": 1
AssertionError: None is not an instance of <class 'dict'> : First argument is not
a dictionary
```

```
----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=4)
```

Run single test:

```
$ python -m unittest excercises.test.test_builtin_types.BuiltinTypeTest
F
======================================================================
FAIL: test_round_2_points_precision
(excercises.test.test_builtin_types.BuiltinTypeTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/marcin/projects/intive/python3-training/code_examples/excercises/
test/test_builtin_types.py", line 8, in test_round_2_points_precision
    self.assertEqual(round_2_points_precision("123.111111"), 123.11)
AssertionError: None != 123.11

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Coverage.py

Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.

https://coverage.readthedocs.io/en/v4.5.x/

Installation:

```
$ pip install coverage
```

Usage:

```
$ cd code_examples/
$ PYTHONPATH=$(pwd) coverage run --source='excercises'
excercises/test/test_word_count.py
```

```
$ coverage report
Name                                          Stmts   Miss  Cover
-----------------------------------------------------------------
excercises/__init__.py                            0      0   100%
excercises/builtin_types.py                       2      2     0%
excercises/fibonacci.py                           4      4     0%
excercises/hello_world.py                         2      2     0%
excercises/test/__init__.py                       0      0   100%
excercises/test/test_builtin_types.py             5      5     0%
excercises/test/test_fibonacci.py                11     11     0%
excercises/test/test_word_count.py               11      0   100%
excercises/word_count.py                          4      0   100%
-----------------------------------------------------------------
TOTAL                                            39     24    38%
(python-basics-training)
marcin@marcin-lapek:~/projects/intive/python3-training/code_examples$
```

Python useful links

https://www.python.org/

https://learnpython.org/

https://wiki.python.org/moin/BeginnersGuide/Programmers

https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie

http://book.pythontips.com/en/latest/index.html

https://www.cheatography.com/davechild/cheat-sheets/python/

https://www.djangoproject.com/

https://www.django-rest-framework.org/

https://www.jetbrains.com/pycharm/download/