Bootcamp Python

Module01 Basics 2

Module01 - Basics 2

The goal of the module is to get familiar with object-oriented programming and much more.

Notions of the module

Objects, cast, class, inheritance, built-in functions, magic methods, generator, constructor, iterator, ...

General rules

- The version of Python recommended to use is 3.7, you can check the version of Python with the following command: `python -V`
- The norm: during this bootcamp you will follow the PEP 8 standards. You can install pycodestyle which is a tool to check your Python code.
- The function eval is never allowed.
- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else, so make sure that your variable names and function names are appropriate and civil.
- Your manual is the internet.
- You can also ask questions in the `#bootcamps` channel in the 42 AI Slack: 42-ai.slack.com.
- If you find any issue or mistakes in the subject please create an issue on our bootcamp python repository on Github.

Helper

Ensure that you have the right Python version.

```
> which python
/goinfre/miniconda/bin/python
> python -V
Python 3.7.*
> which pip
/goinfre/miniconda/bin/pip
```


Exercise 00 - The Book

Turn-in directory:	ex00/
Files to turn in:	book.py, recipe.py, test.py
Forbidden functions:	None
Remarks:	n/a

Objective:

The goal of the exercise is to get you familiar with the notions of classes and the manipulation of the objects related to those classes.

Instructions:

You will have to make a class `Book` and a class `Recipe`

Let's describe the `Recipe` class. It has some attributes:

- `name` (str): name of the recipe,
- `cooking_lvl` (int): range from 1 to 5,
- `cooking_time` (int): in minutes (no negative numbers),
- `ingredients` (list): list of all ingredients each represented by a string,
- `description` (str): description of the recipe,
- `recipe_type` (str): can be "starter", "lunch" or "dessert".

You have to **initialize** the object `Recipe` and check all its values, only the description can be empty. In case of input errors, you should print what they are and exit properly.

You will have to implement the built-in method `__str__`. It's the method called when the following code is executed:

```
tourte = Recipe(...)
to_print = str(tourte)
print(to_print)
```

It's implemented this way:

```
def __str__(self):
    """Return the string to print with the recipe info"""
    txt = ""
    #... Your code here ...
    return txt
```

The `Book` class also has some attributes:

- `name` (str): name of the book,
- `last_update` (datetime): the date of the last update,
- `creation_date` (datetime): the creation date,
- `recipes_list` (dict): a dictionnary with 3 keys: "starter", "lunch", "dessert".

You will have to implement some methods in `Book` class:

```
def get_recipe_by_name(self, name):
    """Print a recipe with the name `name` and return the instance"""
    #... Your code here ...

def get_recipes_by_types(self, recipe_type):
    """Get all recipe names for a given recipe_type """
    #... Your code here ...

def add_recipe(self, recipe):
    """Add a recipe to the book and update last_update"""
    #... Your code here ...
```

You will have to handle the error if the argument passed in `add_recipe` is not a `Recipe`.

Finally, you will provide a `test.py` file to test your classes and prove that they are working well. You can import all the classes into your test.py file by adding these lines at the top of the `test.py` file:

```
from book import Book
from recipe import Recipe

# ... Your tests ...
```

Exercise 01 - Family tree

Turn-in directory:	ex01/
Files to turn in:	game.py
Forbidden functions:	None
Remarks:	n/a

Objective:

The goal of the exercise is to tackle the notion inheritance of class.

Instructions:

Create a `GotCharacter` class and initialize it with the following attributes:

- `first_name`,
- `is_alive` (by default is `True`).

Pick up a GoT House (e.g., Stark, Lannister...). Create a child class that inherits from `GotCharacter` and define the following attributes:

- `family_name` (by default should be the same as the Class),
- `house_words` (e.g., the House words for the Stark House is: "Winter is Coming")

```
class Stark(GotCharacter):
    def __init__(self, first_name=None, is_alive=True):
        super().__init__(first_name=first_name, is_alive=is_alive)
        self.family_name = "Stark"
        self.house_words = "Winter is Coming"
```

Add two methods to your child class:

- `print_house_words`: prints the House words,
- `die`: changes the value of `is_alive` to `False`.

Examples:

Running commands in the Python console, an example of what you should get:

```
> python
>>> from game import Stark
>>> arya = Stark("Arya")
>>> print(arya.__dict__)
{'first_name': 'Arya', 'is_alive': True, 'family_name': 'Stark', 'house_words': 'Winter is
 ↪  Coming'}

>>> arya.print_house_words()
Winter is Coming

>>> print(arya.is_alive)
```

```
True

>>> arya.die()
>>> print(arya.is_alive)
False
```

You can add any attribute or method you need to your class and format the docstring the way you want to. Feel free to create other children of `GotCharacter` class.

```
>>> print(arya.__doc__)
A class representing the Stark family. Or when bad things happen to good people.
```

Exercise 02 - The Vector

Turn-in directory:	ex02/
Files to turn in:	vector.py, test.py
Forbidden functions:	None
Forbidden libraries:	Numpy
Remarks:	n/a

Objective:

The goal of the exercise is to get you used with built-in methods, more particularly with those allowing to perform operations. Student is expected to code built-in methods for vector-vector and vector-scalar operations as rigorously as possible.

Intructions:

In this exercise, you have to create a `Vector` class. The goal is to create vectors and be able to perform mathematical operations with them.

- Column vectors are represented as list of lists of one float,
- Row vectors are represented as list of floats.

The class should also have 2 attributes:

- `values`: a list (or a list of lists) of floats,
- `shape`: dimension of the vector.

Finally you have to implement 2 methods:

- `dot()`: produce a dot product with the vector given as parameter (should have the of same shape),
- `T()`: returns the transpose vector (i.e. a column vector into a row vector or a row vector into a column vector).

You will also provide a testing file `test.py` to demonstrate your class works as expected.

Examples:

```
# Column vector of dimensions n * 1
>> v1 = Vector([[0.0], [1.0], [2.0], [3.0]])
>> v2 = v1 * 5
>> print(v2)
(Vector [[0.0], [5.0], [10.0], [15.0]])
```

```
# Row vector of dimensions 1 * n
v1 = Vector([0.0, 1.0, 2.0, 3.0])
v2 = v1 * 5
# Output
Vector([0.0, 5.0, 10.0, 15.0])
```

```
# Column vector of dimensions n * 1
Vector([[0.0], [1.0], [2.0], [3.0]]).shape
# Output
(4,1)

Vector([[0.0], [1.0], [2.0], [3.0]]).values
# Output
[[0.0], [1.0], [2.0], [3.0]]

# Row vector of dimensions 1 * n
Vector([0.0, 1.0, 2.0, 3.0]).shape
# Output
(1, 4)

Vector([0.0, 1.0, 2.0, 3.0]).values
# Output
[0.0, 1.0, 2.0, 3.0]
```

```
# Column vector of dimensions n * 1
v1 = Vector([[0.0], [1.0], [2.0], [3.0]])
v2 = Vector([[2.0], [1.5], [2.25], [4.0]])
v1.dot(v2)
# Output
18

v1
# Output
[[0.0], [1.0], [2.0], [3.0]]

v1.T()
# Output
[0.0, 1.0, 2.0, 3.0]
```

You should be able to initialize the object with:

- a list of floats: `Vector([0.0, 1.0, 2.0, 3.0])`,
- a list of lists of float: `Vector([[0.0], [1.0], [2.0], [3.0]])`,
- a size: `Vector(3)` -> the vector will be: `[[0.0], [1.0], [2.0]]`,
- a range (min, max): `Vector((10,16))` -> the vector will be: `[[10.0], [11.0], [12.0], [13.0], [14.0], [15.0]]`.

By default, the vectors are generated as classical column vectors if initialized with a size of a range.

To perform arithmetic operations for Vector-Vector or scalar-Vector, you have to implement all the following built-in functions (also called 'magic methods') for your `Vector` class:

```
    __add__
    __radd__
    # add : handle vector-vector and scalar-vector addition, can have errors with
    ↪  vector-vector.
    __sub__
    __rsub__
    # sub : handle vector-vector and scalar-vector substraction, can have errors with
    ↪  vector-vector.
```

```
__truediv__
__rtruediv__
# div : scalars only.
__mul__
__rmul__
# mul : handle vector-vector and scalar-vector multiplication, can have errors with
↪  vector-vector.
# two vectors can be multiplied using the Dot product, return a scalar.
__str__
__repr__
```

Mathematic notions:

Scalar-Vector authorized operations are:

- Multiplication and division between one vector (m * 1) and one scalar:

$$x \cdot a = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \cdot a = \begin{bmatrix} x_1 \cdot a \\ \vdots \\ x_m \cdot a \end{bmatrix}$$

Vector-Vector authorized operations are:

- Addition between two vectors of same dimension (m * 1):

$$x + y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_m + y_m \end{bmatrix}$$

- Subtraction between two vectors of same dimension (m * 1):

$$x - y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} - \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} x_1 - y_1 \\ \vdots \\ x_m - y_m \end{bmatrix}$$

- Compute the dot product between two vectors of same dimenson (m * 1):

$$x \cdot y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \sum_{i=1}^{m} x_i \cdot y_i = x_1 \cdot y_1 + \cdots + x_m \cdot y_m$$

Don't forget to handle all kind of errors properly!

Exercise 03 - Generator!

Turn-in directory:	ex03/
Files to turn in:	generator.py

Forbidden functions:	random.shuffle
Authorized functions:	random.randint
Remarks:	n/a

Objective:

The goal of the exercise is to discover the concept of generator object in Python.

Instructions:

Code a function called `generator` that takes a text as input, uses the string parameter `sep` as a splitting parameter, and `yield`s the resulting substrings.

The function can take an optional argument.
The options are:

- `shuffle`: shuffles the list of words.
- `unique`: returns a list where each word appears only once.
- `ordered`: alphabetically sorts the words.

```
# function prototype
def generator(text, sep=" ", option=None):
    '''Splits the text according to sep value and yield the substrings.
       option precise if a action is performed to the substrings before it is yielded.
    '''
```

You can only call one option at a time.

Examples:

```
>> text = "Le Lorem Ipsum est simplement du faux texte."
>> for word in generator(text, sep=" "):
...     print(word)
...
Le
Lorem
Ipsum
est
simplement
du
faux
texte.

>> for word in generator(text, sep=" ", option="shuffle"):
...     print(word)
...
simplement
texte.
est
faux
Le
Lorem
Ipsum
du

>> for word in generator(text, sep=" ", option="ordered"):
...     print(word)
...
```

```
Ipsum
Le
Lorem
du
est
faux
simplement
texte.
```

```
>> text = "Lorem Ipsum Lorem Ipsum"
>> for word in generator(text, sep=" ", option="unique"):
...     print(word)
...
Lorem
Ipsum
```

The function should return "ERROR" one time if the `text` argument is not a string, or if the `option` argument is not valid.

```
>> text = 1.0
>> for word in generator(text, sep="."):
...     print(word)
...
ERROR
```

Exercise 04 - Working with lists

Turn-in directory:	ex04/
Files to turn in:	eval.py
Forbidden functions	while
Remarks:	use zip & enumerate

Objective:

The goal of the exercise is to discover 2 useful methods for lists, tuples, dictionnaries (iterable class objects more generally) named `zip` and `enumerate`.

Instructions:

Code a class `Evaluator`, that has two static functions named `zip_evaluate` and `enumerate_evaluate`.

The goal of these 2 functions is to compute the sum of the lengths of every `words` of a given list weighted by a list a `coefs` (yes, the 2 functions should do the same thing).

The lists `coefs` and `words` have to be the same length. If this is not the case, the function should return -1.

You have to obtain the desired result using `zip` in the `zip_evaluate` function, and with `enumerate` in the `enumerate_evaluate` function.

Examples:

```
>> from eval import Evaluator
>>
>> words = ["Le", "Lorem", "Ipsum", "est", "simple"]
>> coefs = [1.0, 2.0, 1.0, 4.0, 0.5]
```

```
>> Evaluator.zip_evaluate(coefs, words)
32.0
>> words = ["Le", "Lorem", "Ipsum", "n'", "est", "pas", "simple"]
>> coefs = [0.0, -1.0, 1.0, -12.0, 0.0, 42.42]
>> Evaluator.enumerate_evaluate(coefs, words)
-1
```

Exercise 05 - Bank Account

Turn-in directory:	ex05/
Files to turn in:	the_bank.py
Forbidden functions:	None
Remarks:	n/a

Objective:

The goals of this excercise is to discover new built-in functions, deepen the class manipulation and to be aware of the possibility to modify instanced objects. In this excercise you learn how to modify or add attributes to an object.

Instructions:

It's all about security. Have a look at the class named `Account` in the snippet of code below.

```python
# in the_bank.py
class Account(object):

    ID_COUNT = 1

    def __init__(self, name, **kwargs):
        self.id = self.ID_COUNT
        self.name = name
        self.__dict__.update(kwargs)
        if hasattr(self, 'value'):
            self.value = 0
        Account.ID_COUNT += 1

    def transfer(self, amount):
        self.value += amount
```

Now, it is your turn to code a class named `Bank`!
Its purpose will be to handle the security part of each transfer attempt.

Security means checking if the `Account` is:

- the right object,
- not corrupted,
- and stores enough money to complete the transfer.

How do we define if a bank account is corrupted? A corrupted bank account has:

- an even number of attributes,

- an attribute starting with `b`,
- no attribute starting with `zip` or `addr`,
- no attribute `name`, `id` and `value`.

A transaction is invalid if `amount < 0` or if the amount is larger than the balance of the sending account.

```
# in the_bank.py
class Bank(object):
    """The bank"""
    def __init__(self):
        self.account = []

    def add(self, account):
        self.account.append(account)

    def transfer(self, origin, dest, amount):
        """
            @origin:  int(id) or str(name) of the first account
            @dest:    int(id) or str(name) of the destination account
            @amount:  float(amount) amount to transfer
            @return         True if success, False if an error occured
        """

    def fix_account(self, account):
        """
            fix the corrupted account
            @account: int(id) or str(name) of the account
            @return         True if success, False if an error occured
        """
```

Check out the `dir` built-in function.

WARNING: YOU WILL HAVE TO MODIFY THE INSTANCES' ATTRIBUTES IN ORDER TO FIX THEM.