Bootcamp Python

Bootcamp Python

Day01 - Basics 2

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

Notions of the day

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

General rules

- The version of Python 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 Pep8 standards https://www.python.org/dev/peps/pep-0008/
- 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 dedicated 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 dedicated repository on Github: https://github.com/42-AI/bootcamp_python/issues.

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
```

Exercice 00 - The book.

Exercice 01 - Family tree.

Exercice 02 - The vector.

Exercice 03 - Generator !

Exercice 04 - Working with list.

Exercice 05 - Bank account.

Exercise 00 - The book.

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

You will provide a test.py file to test your classes and prove that they are working the right way.

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
```

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

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

- name (str)
- cooking_lvl (int) : range 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 you execute this code:

```
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 goes here"""
    return txt
```

The Book class also has some attributes:

- name (str)
- last_update (datetime)
- creation_date (datetime)
- recipes_list (dict) : a dictionnary why 3 keys: "starter", "lunch", "dessert".

You will have to implement some methods in Book:

```
def get_recipe_by_name(self, name):
    """Print a recipe with the name `name` and return the instance"""
    pass

def get_recipes_by_types(self, recipe_type):
    """Get all recipe names for a given recipe_type """
    pass

def add_recipe(self, recipe):
    """Add a recipe to the book and update last_update"""
    pass
```

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

Exercise 01 - Family tree.

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

You will have to make a class and its children.

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")

Example:

```
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 to screen the House words
- die: changes the value of is_alive to False

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

```
> python
>>> from game import GotCharacter, 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`**.

```
>>> 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
Remarks :	n/a

You will provide a testing file to prove that your class works as expected.

You will have to create a helpful class, with more options and providing enhanced ease of use for the user.

The goal is to have vectors and be able to perform mathematical operations with them.

```
>> v1 = Vector([0.0, 1.0, 2.0, 3.0])
>> v2 = v1 * 5
>> print(v2)
(Vector [0.0, 5.0, 10.0, 15.0])
```

It has 2 attributes:

- values : list of float
- size : size of the vector -> Vector([0.0, 1.0, 2.0, 3.0]).size == 4

You should be able to initialize the object with:

- a list of floats: Vector([0.0, 1.0, 2.0, 3.0])
- a size Vector(3) -> the vector will have values = [0.0, 1.0, 2.0]
- a range or Vector((10,15)) -> the vector will have values = [10.0, 11.0, 12.0, 13.0, 14.0]

You will implement all the following built-in functions (called 'magic methods') for your Vector class:

```
    __add__
    __radd__
    # add : scalars and vectors, can have errors with vectors.
    __sub__
    __rsub__
    # sub : scalars and vectors, can have errors with vectors.
    __truediv__
    __rtruediv__
    # div : only scalars.
    __mul__
    __rmul__
    # mul : scalars and vectors, can have errors with vectors,
    # return a scalar if we perform Vector * Vector (dot product)
    __str__
    __repr__
```

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
Remarks :	n/a

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

The function can take an optional argument.

The options are:

- "shuffle": shuffle the list of words.
- "unique": return a list where each word appears only once.
- "ordered": alphabetically sort the words.

```
# function prototype
def generator(text, sep=" ", option=None):
    '''Option is an optional arg, sep is mandatory'''
```

You can only call one option at a time.

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

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

Exercice 04 - Working with list.

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

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`**.

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.

```
>> 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
```

Exercice 05 - Bank Account.

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

It's all about security.

There is a class called **Account**.

```
# 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 you have to code the class **Bank**.

It will have to handle the security part of each transfer attempt.

Security means checking if the Account is:

- the right object
- that it is not corrupted
- and that it has enough money

How do we define if a bank account is corrupted?

- It has an even number of attributes.
- It has an attribute starting with b.
- It has no attribute starting with zip or addr.
- It has no attribute name, id and value.

A transaction is invalid if `amount < 0` or if the amount is larger than the funds the first account has available for transfer.

```
# 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` function.

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