Piscine 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, inheritance, built-in functions, generator, construtors, iterator, …

General rules

- The norm : during this pool you will follow the Pep8 standards
  https://www.python.org/dev/peps/pep-0008/
- Forbidden functions: eval, …

Helper

```
No helper there
```

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.

Turnin directory :	ex00
Files to turn in :	book.py recipy.py test.py

You will provide a test.py file to test your classes and prove that it is working the right way.
You can import all the classes in the test.py by adding these lines at the beggining of test.py file :

```
from book import *
```

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
- `type` (str) : can be "starter", "main_course" or "dessert".

You have to initialize the object Recipe and check all the values, only the description can be empty.
In case of input errors, you should print it 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 now has also some attributes :

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

You will have to implement some methods in book.

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

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

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

Exercise 01 - Family tree.

Turnin directory :	ex01
Files to turn in :	game.py

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., Star, Lannister…) and 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': 'Wi
>>> 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 as you want to.
Feel free to create other children of `GotCharacter`.

```
>>> print(arya.__doc__)
A class representing the Stark family. Not bad people but a lot of bad luck.
```

Exercise 02 - The vector.

Turnin directory :	ex02
Files to turn in :	vector.py test.py

You will provide a testing file to prove that your class is working the right way.
You will have to create a better class, with more options 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
- len : size of the vector

You have to initialize the object with a list of float. `Vector([0.0, 1.0, 2.0, 3.0])`
Or a range `Vector(3)` or `Vector(0,3)` -> the vector will have `values = [0.0, 1.0, 2.0]`

You will implement all those built-in functions:

```
    __add__
    # add : scalars and vectors, can't have errors with vectors.
    __sub__
    # sub : scalars and vectors, can't have errors with vectors.
    __div__
    # div : only scalars.
    __mul__
    # mul : scalars and vectors, can't have errors with vectors, return a scalar is
    __str__
    __repr__
```

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

Exercise 03 - Generator !

Turnin directory :	ex03
Files to turn in :	generator.py

Code a function `generator` that takes a text as input and split it on `sep` parameter, and `yield` the words.

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

- "shuffle": shuffle the list of words.
- "unique": return only the unique words.
- "ordered": alphabetically sort the words.

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
```

Exercice 04 - Working with list.

Turnin directory :	ex04
Files to turn in :	zip_eval.py enumerate_eval.py
Forbidden function :	while

Code a function `evaluate`, that takes 2 lists as parameters.

```
def evaluate(coefs, words):
    ...
    return score
```

The goal of the function is to calculate the sum of the len of every `words` of a given list weighted by a list a `coefs`.

`coefs` and `words` has to be the same len, if it's not the case the function should return -1.

You have to do it using `zip` in the zip_eval.py file, and with `enumerate` in the enumerate_eval.py.

```
>> from zip_eval import evaluate as zip_evaluate
>> from enumerate_eval import evaluate as enumerate_evaluate
>> words = ["Le", "Lorem", "Ipsum", "est", "simple"]
>> coefs = [1.0, 2.0, 1.0, 4.0, 0.5]
>> zip_evaluate(coefs, words)
34.0
>> words = ["Le", "Lorem", "Ipsum", "n'", "est", "pas", "simple"]
>> coefs = [0.0, -1.0, 1.0, -12.0, 0.0, 42.42]
>> enumerate_evaluate(coefs, words)
-1
```

Exercice 05 - Bank Account.

Turnin directory :	ex05
Files to turn in :	the_bank.py

It's all about security.
There is a class Account.

```
class Account(object):

    ID_COUNT = 0

    def __init__(self, name, **kwargs):
        self.id = self.ID_COUNT
        self.name = name
        self.__dict__.update(kwargs)
        if hasattr(self, 'value'):
            self.value = 0
        self.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 at 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 to define if a bank account is corrupted ?

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

```
class Bank(object):
    """The bank"""
```

```
    def __init__(self):
        self.account = []

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

    def transfer(self, origine, dest, amount):
        """
            @origine: 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 is success, False is an error occured
        """

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

Check out the `dir` function.