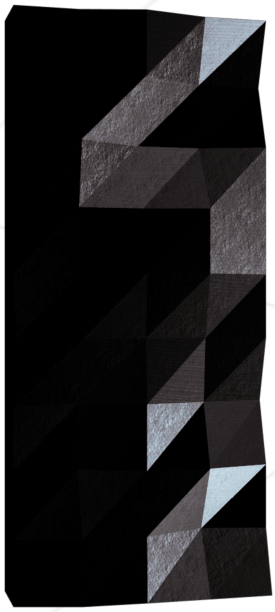

Training Piscine Python for datascience - 3

Oriented Object Programming

Summary: Today, you will see the classes and the heritage.

Version: 1.00

Contents

Chapter I

General rules

- You have to render your modules from a computer in the cluster either using a virtual machine:
    - You can choose the operating system to use for your virtual machine
    - Your virtual machine must have all the necessary software to realize your project. This software must be configured and installed.
- Or you can use the computer directly in case the tools are available.
    - Make sure you have the space on your session to install what you need for all the modules (use the goinfre if your campus has it)
    - You must have everything installed before the evaluations
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.
- You must use the Python 3.10 version
- Your lib imports must be explicit, for example you must "import numpy as np". Importing "from pandas import *" is not allowed, and you will get 0 on the exercise.
- There is no global variable.
- By Odin, by Thor ! Use your brain !!!

Chapter II

Specific instructions of the day

A common complaint to data scientists is that they write shitcode (by the way, only for educational purposes you may find a lot of examples of Python shitcode here, provided strictly for educational purposes). Why? Because the average data scientist uses a lot of inefficient techniques and hard coded variables and neglects object-oriented programming. Do not be like them.

- No code in the global scope. Use functions!
- Each program must have its main and not be a simple script:

```
def main():
  # your tests and your error handling

if __name__ == "__main__":
  main()
```

- Any exception not caught will invalidate the exercices, even in the event of an error that you were asked to test.
- You can use any built-in function if it is not prohibited in the exercise.
- All your functions, class and method must have a documentation (___doc___)
- Your code must be at the norm
  - pip install flake8
  - alias norminette=flake8

Chapter III

Exercise 00

	Exercise 00
Exercice 00: GOT S1E9	
Turn-in directory : $ex00/$	
Files to turn in : `S1E9.py`	
Allowed functions : `None`	

Create an abstract class "character" which can take a first_name as first parameter, is_alive as second non mandatory parameter set to True by default and can change the health state of the character with a method that passes is_alive from True to False.

And a "stark" class which inherits from Character

The prototype of Class is:

```
from abc import ABC, abstractmethod

class Character(ABC):
    """Your docstring for Class"""
    abstractmethod
        #your code here

class Stark(Character):
    """Your docstring for Class"""
        #your code here
```

Your tester.py:

```
from S1E9 import Character, Stark

Ned = Stark("Ned")
print(Ned.__dict__)
print(Ned.is_alive)
Ned.die()
print(Ned.is_alive)
print(Ned.__doc__)
print(Ned.__init__.__doc__)
print(Ned.die.__doc__)
print("---")
Lyanna = Stark("Lyanna", False)
print(Lyanna.__dict__)
```

Expected output: (docstrings can be different)

```
$> python tester.py
{'first_name': 'Ned', 'is_alive': True}
True
False
Your docstring for Class
Your docstring for Constructor
Your docstring for Method
---
{'first_name': 'Lyanna', 'is_alive': False}
$>
```

Make sure you have used an abstract class, the code below should make an error.

```
from S1E9 import Character

hodor = Character("hodor")
```

```
TypeError: Can't instantiate abstract class Character with abstract method
$>
```

Chapter IV

Exercise 01

	Exercise 01
Exercice 01: GOT S1E7	
Turn-in directory : $ex01/$	
Files to turn in : `Files from previous exercises + S1E7.py`	
Allowed functions : `None`	

Create two families that inherit from the Character class, that we can instantiate without going through the Character class. Find a solution so that "___str___" and "___repr___" return strings and not objects. Write a Class method to create characters in a chain.

The prototype of Class is:

```
from S1E9 import Character

class Baratheon(Character):
 #your code here

class Lannister(Character):
 #your code here

        # decorator
    def create_lannister(your code here):
     #your code here
```

Your tester.py:

```
from S1E7 import Baratheon, Lannister

Robert = Baratheon("Robert")
print(Robert.__dict__)
print(Robert.__str__)
print(Robert.__repr__)
print(Robert.is_alive)
Robert.die()
print(Robert.is_alive)
print(Robert.__doc__)
print("---")
Cersei = Lannister("Cersei")
print(Cersei.__dict__)
print(Cersei.__str__)
print(Cersei.is_alive)
print("---")
Jaine = Lannister.create_lannister("Jaine", True)
print(f"Name : {Jaine.first_name, type(Jaine).__name__}, Alive : {Jaine.is_alive}")
```

Expected output: (docstrings can be different)

```
$> python tester.py
{'first_name': 'Robert', 'is_alive': True, 'family_name': 'Baratheon', 'eyes': 'brown', 'hairs': 'dark'}
<bound method Baratheon.__str__ of Vector: ('Baratheon', 'brown', 'dark')>
<bound method Baratheon.__repr__ of Vector: ('Baratheon', 'brown', 'dark')>
True
False
Representing the Baratheon family.
---
{'first_name': 'Cersei', 'is_alive': True, 'family_name': 'Lannister', 'eyes': 'blue', 'hairs': 'light'}
<bound method Lannister.__str__ of Vector: ('Lannister', 'blue', 'light')>
True
---
Name : ('Jaine', 'Lannister'), Alive : True
$>
```

Chapter V

Exercise 02

	Exercise 02
Exercice 02: Now it's weird!	
Turn-in directory : $ex02/$	
Files to turn in : `Files from previous exercises + DiamondTrap.py`	
Allowed functions : `None`	

In this exercise, you will create a monster: Joffrey Baratheon.
This is so risky!
There is something inconsistent with this new "false" king.
You must use the Properties to change the physical characteristics of our new king.
The prototype of Class is:

```
from S1E7 import Baratheon, Lannister

class King(Baratheon, Lannister):
 #your code here
```

Your tester.py:

```
from DiamondTrap import King

Joffrey = King("Joffrey")
print(Joffrey.__dict__)
Joffrey.set_eyes("blue")
Joffrey.set_hairs("light")
print(Joffrey.get_eyes())
print(Joffrey.get_hairs())
print(Joffrey.__dict__)
```

Expected output: (docstrings can be different)

```
$> python tester.py
{'first_name': 'Joffrey', 'is_alive': True, 'family_name': 'Baratheon', 'eyes': 'brown', 'hair': 'dark'}
blue
light
{'first_name': 'Joffrey', 'is_alive': True, 'family_name': 'Baratheon', 'eyes': 'blue', 'hairs': 'light'}
$>
```

Since python 2.3 the language uses C3 linearization to counter the problem of inheritance in diamand.

Chapter VI

Exercise 03

	Exercise 03
Exercice 03: Calculate my vector	
Turn-in directory : $ex03/$	
Files to turn in : `ft_calculator.py`	
Allowed functions : `None`	

Write a calculator class that is able to do calculations (addition, multiplication, subtraction, division) of vector with a scalar.
The prototype of Class is:

```
class calculator:
 #your code here

        def __add__(self, object) -> None:
                #your code here
        def __mul__(self, object) -> None:
                #your code here
        def __sub__(self, object) -> None:
                #your code here
        def __truediv__(self, object) -> None:
                #your code here
```

Your tester.py:

```
from ft_calculator import calculator

v1 = calculator([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
v1 + 5
Print("---")
v2 = calculator([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
v2 * 5
Print("---")
v3 = calculator([10.0, 15.0, 20.0])
v3 - 5
v3 / 5
```

Expected output: (docstrings can be different)

```
$> python tester.py
[5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
---
[0.0, 5.0, 10.0, 15.0, 20.0, 25.0]
---
[5.0, 10.0, 15.0]
[1.0, 2.0, 3.0]
$>
```

You don't have to do any error handling, except for the division by 0

Chapter VII

Exercise 04

	Exercise 04
Exercice 04: Calculate my dot product	
Turn-in directory : $ex04/$	
Files to turn in : `ft_calculator.py`	
Allowed functions : `None`	

Write a calculator class that is able to do calculations (dot product, addition, subtraction) of 2 vectors.
Vector will always have identical sizes, no error handling.
It's up to you to find a decorator that can help you to use the Methods of the calculator class without instantiating this class.
The prototype of Class is:

```
class calculator:
 #your code here

        # decorator
        def dotproduct(V1: list[float], V2: list[float]) -> None:
                #your code here

        # decorator
        def add_vec(V1: list[float], V2: list[float]) -> None:
                #your code here

        # decorator
        def sous_vec(V1: list[float], V2: list[float]) -> None:
                #your code here
```

Your tester.py:

```
from ft_calculator import calculator

a = [5, 10, 2]
b = [2, 4, 3]
calculator.dotproduct(a,b)
calculator.add_vec(a,b)
calculator.sous_vec(a,b)
```

Expected output: (docstrings can be different)

```
$> python tester.py
Dot product is: 56
Add Vector is : [7.0, 14.0, 5.0]
Sous Vector is: [3.0, 6.0, -1.0]
$>
```

You don’t have to do any error handling

If you want to go further in vector or matrix calculations go to the matrix project after this "piscine".

Chapter VIII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.

`The evaluation process will happen on the computer of the evaluated group.`