Python & ML - Module 01

Basic 2

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

Chapter I

Common Instructions

- 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 piscine, it is recommended to follow the PEP 8 standards, though it is not mandatory. 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 42AI or 42born2code.
- If you find any issue or mistakes in the subject please create an issue on dedicated 42AI repository on Github.
- 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 run after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Contents

Chapter II

Exercise 00

	Exercise : 00
The Book	
Turn-in directory : $ex00/$	
Files to turn in : `book.py, recipe.py, test.py`	
Forbidden functions : `None`	

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`. The classes `Book` and `Recipe` will be written in `book.py` and `recipe.py` respectively.

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 is 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):
    """Prints a recipe with the name \texttt{name} and returns 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 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 properly. 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 ...
```

Chapter III

Exercise 01

	Exercise : 01
Family tree	
Turn-in directory : $ex01/$	
Files to turn in : `game.py`	
Forbidden functions : `None`	

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...) 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")

```
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.

```
      $> python
>>> from game import Stark

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

Chapter IV

Exercise 02

	Exercise : 02
Generator!	
Turn-in directory : $ex02/$	
Files to turn in : `generator.py`	
Forbidden functions : `random.shuffle, random.sample`	

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 (only printable characters), 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
```

Chapter V

Exercise 03

	Exercise : 03
Working with lists	
Turn-in directory : $ex03/$	
Files to turn in : `eval.py`	
Forbidden functions : `while`	

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 of coefficinents `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
```

Chapter VI

Exercise 04

	Exercise : 04
The Vector	
Turn-in directory : $ex04/$	
Files to turn in : `vector.py, test.py`	
Forbidden functions : `Numpy library`	

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.

Instructions

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 single float (`[[1.], [2.], [3.]]`),
- Row vectors are represented as a list of a list of several floats (`[[1., 2., 3.]]`).

`A vector is either a single line of floats or a single column of floats. When more than a line/column is consider, it is a matrix, not a vector.`

The class should also has 2 attributes:

- `values`: list of list of floats (for row vector) or list of lists of single float (for column vector),

- `shape`: tuple of 2 integers: `(1,n)` for a row vector of dimension n or `(n,1)` for a column vector of dimension n.

If you did not learn at school what is the dimension of a vector, don't worry. But for now do not think too hard about what dimension means. Just consider the dimension is the number of floats (elements/coordinates) of a vector, and shape gives the layout: if $(1, n)$ the vector is a row, if $(n, 1)$ the vector is a column.

Finally you have to implement 2 methods:

- `.dot()` produce a dot product between two vectors 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. In this testing file, demonstrate:

- the addition and substraction are working for 2 vectors of the same shape,
- the mutliplication (`mul` and `rmul`) are working for a vector and a scalar,
- the division (`truediv`) is working with a vector and a scalar,
- the division (`rtruediv`) raises an Arithmetic Error (this test can be commented for the other tests and uncommented to show this one),

Examples

```python
# Column vector of shape n * 1
v1 = Vector([[0.0], [1.0], [2.0], [3.0]])
v2 = v1 * 5
print(v2)
# Expected output:
# Vector([[0.0], [5.0], [10.0], [15.0]])

# Row vector of shape 1 * n
v1 = Vector([[0.0, 1.0, 2.0, 3.0]])
v2 = v1 * 5
print(v2)
# Expected output
# Vector([[0.0, 5.0, 10.0, 15.0]])

v2 = v1 / 2.0
print(v2)
# Expected output
# Vector([[0.0], [0.5], [1.0], [1.5]])

v1 / 0.0
# Expected ouput
# ZeroDivisionError: division by zero.

2.0 / v1
# Expected output:
# NotImplementedError: Division of a scalar by a Vector is not defined here.
```

```python
# Column vector of shape (n, 1)
print(Vector([[0.0], [1.0], [2.0], [3.0]]).shape)
# Expected output
# (4,1)

print(Vector([[0.0], [1.0], [2.0], [3.0]]).values)
# Expected output
# [[0.0], [1.0], [2.0], [3.0]]

# Row vector of shape (1, n)
print(Vector([[0.0, 1.0, 2.0, 3.0]]).shape)
# Expected output
# (1,4)

print(Vector([[0.0, 1.0, 2.0, 3.0]]).values)
# Expected output
# [[0.0, 1.0, 2.0, 3.0]]
```

```python
# Example 1:
v1 = Vector([[0.0], [1.0], [2.0], [3.0]])
print(v1.shape)
# Expected output:
(4,1)

print(v1.T())
# Expected output:
# Vector([[0.0, 1.0, 2.0, 3.0]])

print(v1.T().shape)
# Expected output:
# (1,4)

# Example 2:
v2 = Vector([[0.0, 1.0, 2.0, 3.0]])
print(v2.shape)
# Expected output:
# (1,4)

print(v2.T())
# Expected output:
# Vector([[0.0], [1.0], [2.0], [3.0]])

print(v2.T().shape)
# Expected output:
# (4,1)
```

```python
  # Example 1:
  v1 = Vector([[0.0], [1.0], [2.0], [3.0]])
  v2 = Vector([[2.0], [1.5], [2.25], [4.0]])
  print(v1.dot(v2))
  # Expected output:
  # 18.0

  v3 = Vector([[1.0, 3.0]])
  v4 = Vector([[2.0, 4.0]])
  print(v3.dot(v4))
  # Expected output:
  # 13.0

  v1
  # Expected output: to see what __repr__() should do
  # [[0.0, 1.0, 2.0, 3.0]]

  print(v1)
  # Expected output: to see what __str__() should do
  # [[0.0, 1.0, 2.0, 3.0]]
```

You should be able to initialize the object with:

- a list of a list of floats: `Vector([[0.0, 1.0, 2.0, 3.0]])`,

- a list of lists of single float: `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: `Vector((10,16))` -> the vector will have `values = [[10.0], [11.0], [12.0], [13.0], [14.0], [15.0]]`. in `Vector((a,b))`, if `a > b`, you must display accurate error message.

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

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

```
__add__
__radd__
# add & radd : only vectors of same shape.
__sub__
__rsub__
# sub & rsub: only vectors of same shape.
__truediv__
# truediv : only with scalars (to perform division of Vector by a scalar).
__rtruediv__
# rtruediv : raises an NotImplementedError with the message "Division of a scalar by a Vector is not defined here."
__mul__
__rmul__
# mul & rmul: only scalars (to perform multiplication of Vector by a scalar).
__str__
__repr__
# must be identical, i.e we expect that print(vector) and vector within python interpretor behave the same, see correspond
```

So it might be a good idea to implement values and shape before built-in arithmetic functions. For the case not specify (e.g vector * vector) you should raise NotImplementedError.

Mathematic notions

The authorized vector operations are:

- Addition between two vectors of same dimension m

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

- Subtraction between two vectors of same dimension m

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

- Multiplication and division between one vector m and one scalar.

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

- Dot product between two vectors of same dimension m

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

Do not forget to handle all types of error properly!

Chapter VII

Exercise 05

	Exercise : 05
Bank Account	
Turn-in directory : $ex05/$	
Files to turn in : `the_bank.py`	
Forbidden functions : `None`	

Objective

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

Instructions

It is 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.__dict__.update(kwargs)

        self.id = self.ID_COUNT
        Account.ID_COUNT += 1
        self.name = name
        if not hasattr(self, 'value'):
            self.value = 0

        if self.value < 0:
          raise AttributeError("Attribute value cannot be negative.")
        if not isinstance(self.name, str)
          raise AttributeError("Attribute name must be a str object.")

    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`,
- `name` not being a string,
- `id` not being an `int`,
- `value` not being an `int` or a `float`.

For the rest of the attributes (`addr`, `zip`, etc ... there is no specific check expected. Meaning you are not expected to evaluate the validity of the account based on the type of the other attributes (the conditions listed above are sufficient).

Moreover, verification has to be performed when account objects are added to to Bank instance (`bank.add(Account(...))`). The verification in `add` only check the type of the new_account and if there is no account among the one already in Bank instance with the same name.

A transaction is invalid if `amount < 0` or if the amount is larger than the balance of the account. Prior to the transfer, the validity of the 2 accounts (`origin` and `dest`) are checked (according to the list of criteria above). A transfer between the same account (`bank.transfer('Wiliam John', 'William John')`) is valid but there is no fund movement.

`fix_account` recovers a corrupted account if it parameter `name` correspond to the attribute name of one of the account in `accounts` (attribute of Bank). If name is not a string or does not corresponded to an account name, the method return `False`.

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

    def add(self, new_account):
        """ Add new_account in the Bank
            @new_account:  Account() new account to append
            @return   True if success, False if an error occured
        """
        # test if new_account is an Account() instance and if
        # it can be appended to the attribute accounts

        # ... Your code  ...

        self.accounts.append(new_account)

    def transfer(self, origin, dest, amount):
        """" Perform the fund transfer
            @origin:  str(name) of the first account
            @dest:    str(name) of the destination account
            @amount:  float(amount) amount to transfer
            @return   True if success, False if an error occured
        """
        # ... Your code  ...

    def fix_account(self, name):
    """ fix account associated to name if corrupted
            @name:   str(name) of the account
            @return  True if success, False if an error occured
        """
        # ... Your code  ...
```

Check out the `dir` built-in function.

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

Examples

The script `banking_test1.py` is a test which must print `Failed`. The second script `banking_test2.py` is a test which must print `Failed` and then `Success`.

```
>> python banking_test1.py
Failed
# The transaction is not performed has the account of Smith Jane is corrupted (due to the attribute 'bref').

>> python banking_test2.py
Failed
Success
# the account are false due to the abscence of addr attribute, fix_account recover the account,
# thus they become valid.
```

Contact

You can contact 42AI association by email: contact@42ai.fr
You can join the association on 42AI slack and/or posutale to one of the association teams.

Acknowledgements

The modules Python & ML is the result of a collective work, we would like to thanks:

- Maxime Choulika (cmaxime),
- Pierre Peigné (ppeigne, pierre@42ai.fr),
- Matthieu David (mdavid, matthieu@42ai.fr),
- Quentin Feuillade–Montixi (qfeuilla, quentin@42ai.fr)

who supervised the creation, the enhancement and this present transcription.

- Louis Develle (ldevelle, louis@42ai.fr)
- Augustin Lopez (aulopez)
- Luc Lenotre (llenotre)
- Owen Roberts (oroberts)
- Thomas Flahault (thflahau)
- Amric Trudel (amric@42ai.fr)
- Baptiste Lefeuvre (blefeuvr@student.42.fr)
- Mathilde Boivin (mboivin@student.42.fr)
- Tristan Duquesne (tduquesn@student.42.fr)

for your investment for the creation and development of these modules.

- Barthélémy Leveque (bleveque@student.42.fr)
- Remy Oster (roster@student.42.fr)
- Quentin Bragard (qbragard@student.42.fr)
- Marie Dufourq (madufour@student.42.fr)
- Adrien Vardon (advardon@student.42.fr)

who betatest the first version of the modules of Machine Learning.