Python & ML - Módulo 01

Conceptos básicos 2

Resumen: El objetivo del módulo es familiarizarse con la programación orientada a objetos y mucho más.

Capítulo I

Instrucciones generales

- La versión de Python que se recomienda utilizar es la 3.7, puedes comprobar la versión de Python con el siguiente comando: `python -V`
- La norma: durante esta piscina, se recomienda seguir los estandares PEP 8, aunque no es obligatorio. Puedes instalar pycodestyle que es una herramienta para comprobar tu código Python.
- La función `eval` nunca está permitida.
- Los ejercicios están ordenados del más fácil al más difícil.
- Tus ejercicios van a ser evaluados por otras personas, así que asegúrate de que los nombres de tus variables y funciones sean apropiados y corteses.
- Tu manual es internet.
- Te animamos a crear programas de prueba para tu proyecto, aunque este trabajo **no tendrá que ser presentado y no será calificado**. Te dará la oportunidad de poner a prueba fácilmente tu trabajo y el de tus compañeros/as. Estos tests te serán especialmente útiles durante tu evaluación. De hecho, durante la evaluación, eres libre de utilizar tus pruebas y/o las pruebas del compañero/a al que estás evaluando.

Índice general

Capítulo II

Ejercicio 00

	Ejercicio : 00
	$PATH
Directorio de entrega : $ex00/$	
Archivos a entregar : `book.py, recipe.py, test.py`	
Funciones prohibidas : `Ninguna`	

Objetivo

El objetivo del ejercicio es que te familiarices con las nociones de clases y la manipulación de los objetos relacionados con esas clases.

Instrucciones

Tendrás que hacer una clase Book y una clase Recipe. Las clases Book y Recipe se escribirán en book.py y recipe.py respectivamente. Vamos a describir la clase Recipe. Tiene algunos atributos:

- name (str): nombre de la receta,
- cooking_lvl (int): rango de 1 a 5,
- cooking_time (int): en minutos (sin números negativos),
- ingredients (lista): lista de todos los ingredientes, cada uno representado por un string,
- description (str): descripción de la receta,
- recipe_type (str): puede ser “entrante”, “comida” o “postre”.

Tienes que inicializar el objeto Receta y comprobar todos sus valores, solo la descripción puede estar vacía. En caso de errores de entrada, debes imprimir lo que son y salir correctamente. Tendrás que implementar el método incorporado__ __str__ __. Es el método llamado cuando se ejecuta el siguiente código:

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

Se implementa de la siguiente forma:

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

La clase `Book` también tiene alguno atributos:

- `name` (str): nombre del libro,
- `last_update` (datetime): la fecha de la última actualización,
- `creation_date` (datetime): la fecha de creación,
- `recipes_list` (dict): un diccionario con 3 claves: "starter", "lunch", "dessert".

Tendrás que implementar algunos métodos en la clase `Book`:

```
def get_recipe_by_name(self, name):
    """Imprime la receta con el nombre \texttt{name} y devolver la instancia"""
    #... Aqui tu código ...

def get_recipes_by_types(self, recipe_type):
    """Devuelve todas las recetas dado un recipe_type """
    #... Aqui tu código ...

def add_recipe(self, recipe):
    """Añade una receta al libro y actualiza last_update"""
    #... Aqui tu código ...
```

Tienes que gestionar el error si el argumento pasado en `add_recipe` no es una `Recipe`.

Por último, proporcionarás un archivo `test.py` para probar tus clases y demostrar que funcionan correctamente. Puedes importar todas las clases en su archivo `test.py` añadiendo lo siguiente al principio del archivo `test.py`:

```
from book import Book
from recipe import Recipe

# ... Tus tests ...
```

Capítulo III

Ejercicio 01

	Ejercicio : 01
	Arbol genealógico
Directorio de entrega : $ex01/$	
Archivos a entregar : `game.py`	
Funciones prohibidas : `Ninguna`	

Objetivo

El objetivo del ejercicio es abordar la noción de herencia de clase.

Instrucciones

Crea una clase `GotCharacter` e inicialízala con los siguientes atributos:

- `first_name`,
- `is_alive` (por defecto es `True`).

Elige una Casa de GoT (por ejemplo, Stark, Lannister...) y crea una clase hija que herede de GotCharacter y define los siguientes atributos:

- `family_name` (por defecto debe ser el mismo que el de la Clase)
- `house_words` (por ejemplo, el lema de la Casa Stark es: "Winter is comming")

```
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"
```

Añade dos métodos a tu clase hija:

- `print_house_words`: imprime el lema de la casa,
- `die`: cambia el valor de `is_alive` a `False`.

Ejemplos

Un ejemplo de lo que deberías obtener ejecutando comandos en la consola de Python:

```
      $> 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
```

Puedes añadir cualquier atributo o método que necesites a tu clase y formatear el docstring como quieras. Siéntete libre de crear otros hijos de la clase `GotCharacter`.

```
      $> 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.
```

Capítulo IV

Ejercicio 02

	Ejercicio : 02
El Vector	
Directorio de entrega : $ex02/$	
Archivos a entregar : `vector.py, test.py`	
Funciones prohibidas : `Librería Numpy`	

Objective

El objetivo del ejercicio es que te familiarices con los métodos incorporados, más concretamente con los que permiten realizar operaciones. Se espera que el estudiante codifique los métodos incorporados para las operaciones vector-vector y vector-escalar de la forma más rigurosa posible.

Instrucciones

En este ejercicio, tienes que crear una clase `Vector`. El objetivo es crear vectores y poder realizar operaciones matemáticas con ellos.

- Los vectores columna se representan como listas de listas de float simples (`[[1.], [2.], [3.]]`),
- Los vectores fila se representan como una lista de una lista de varios floats (`[[1., 2., 3.]]`).

`Un vector es una sola línea de floats o una sola columna de floats. Cuando se considera más de una linea/columna, se trata de una matriz, no de un vector`

La clase también debe tener 2 atributos:

- `values`: lista de lista de floats (para vector fila) o lista de listas de un solo float (para vector columna),
- `shape`: tupla de 2 enteros: `(1,n)` para un vector fila de dimensión n or `(n,1)` para un vector columna de dimension n.

Si no aprendiste en la colegion cuál es la dimensión de un vector, no te preocupes. Pero por ahora no pienses demasiado en lo que significa dimensión. Considera que la dimensión es el número de floats (elementos/coordenadas) de un vector, y la forma da la disposición: si $(1, n)$ el vector es una fila, si $(n, 1)$ el vector es una columna.

Finalmente tienes que implementar 2 métodos:

- `.dot()` produce un producto escalar entre dos vectores de la misma **shape**,
- `.T()` devuelve el vector transpuesto (es decir, un vector columna en un vector fila, o un vector fila en un vector columna).

También proporcionarás un archivo de prueba `test.py` para demostrar que tu clase funciona como se espera. En este archivo de prueba, demuestra que:

- la suma y la resta funcionan para 2 vectores de la misma forma,
- la multiplicación (`mul` and `rmul`) funcionan para un vector y un escalar,
- la división (`truediv`) funciona con un vector y un escalar,
- la división (`rtruediv`) produce un error aritmético (esta prueba puede comentarse para las demás pruebas y descomentarse para mostrar esta)

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]]
```

```
# 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)
```

```
# 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]]
```

Debes ser capaz de inicializar el objeto con:

- una lista de floats: `Vector([[0.0, 1.0, 2.0, 3.0]])`,
- una lista de listas de un solo float: `Vector([[0.0], [1.0], [2.0], [3.0]])`,
- un tamaño: `Vector(3)` ->el vector tendrá `valores = [[0.0], [1.0], [2.0]]`,
- un rango: `Vector((10,16))` ->el vector tendrá `valores = [[10.0], [11.0], [12.0], [13.0], [14.0], [15.0]]`. En el `Vector((a,b))`, si `a >b`, debes mostrar un mensaje de error preciso.

Por defecto, los vectores se generan como vectores columna clásicos si se inicializan con un tamaño o rango

Para realizar operaciones aritméticas para Vector-Vector o escalar-Vector, tienes que implementar todas las siguientes funciones integradas (llamadas `magic/special methods`) para tu claser `Vector`:

```
__add__
__radd__
# add & radd : solo vectores de la misma forma.
__sub__
__rsub__
# sub & rsub: solo vectores de la misma forma.
__truediv__
# truediv : solo con escaláres (para hacer la división entre un Vector por un escalar).
__rtruediv__
# rtruediv : genera un error tipo NotImplementedError con el
# mensaje "Division of a scalar by a Vector is not defined here."
__mul__
__rmul__
# mul & rmul: solo escalares (para hacer la multiplicación de un Vector por un escalar).
__str__
__repr__
# deben ser identicas, por ejemplo, esperamos que print(vector) y el vector dentro del
# interprete de python se comporten de la misma forma, ver sección de ejemplos
```

Podría ser una buena idea implementar values y shape antes de las funciones aritméticas integradas. Para el caso no especificado (por ejemplo, vector * vector), deberías generar NotImplementedError.

Nociones matemáticas

Las operaciones de vectores autorizadas son:

- Suma entre dos vectores de la misma dimensión m

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

- Resta entre dos vectores de la misma dimensión m

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

- Multiplicación y division entre un vector m y un escalar.

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

- Producto escalar entre dos vectores de la misma dimensíon m

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

¡No olvides gestionar correctamente todos los tipos de error!

Capítulo V

Ejercicio 03

	Ejercicio : 03
	¡Generador!
Directorio de entrega : $ex03/$	
Archivos a entregar : `generator.py`	
Funciones prohibidas : `random.shuffle, random.sample`	

Objetivo

El objetivo del ejercicio es descubrir el concepto de objeto generador en Python.

Instrucciones

Programa una función llamada `generator` que tome un texto como entrada (solo caracteres imprimibles), utilice el parámetro string `sep` como parametro separador, y que `produzca` las sub-strings resultantes.

La función puede recibir un argumento opcional. Estas opciones son:

- `shuffle`: mezcla la lista de palabras,
- `unique`: devuelve una lista de palabras donde cada palabra solo aparece una vez,
- `ordered`: ordena alfabéticamente las palabras.

```
# function prototype
def generator(text, sep=" ", option=None):
  '''Divide el texto de acuerdo al valor de sep y producirá las sub-strings.
     option especifica si una acción se realizará sobre las sub-strings antes de ser producidas.
  '''
```

Solo puedes llamar a una opción cada vez.

Ejemplos

```
>> 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
```

La función debe devolver .ERRORüna vez si el argumento `text` no es un string, o si el argumento `option` no es válido.

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

Capítulo VI

Ejercicio 04

	Ejercicio : 04
Trabajando con listas	
Directorio de entrega : $ex04/$	
Archivos a entregar : `eval.py`	
Funciones prohibidas : `while`	

Objetivos

El objetivo del ejercicio es descubrir 2 métodos útiles para listas, tuplas, diccionarios (objetos de clase iterables en general) llamados `zip` and `enumerate`.

Instrucciones

Crea una clase `Evaluator`, que tiene dos funciones estáticas llamadas `zip_evaluate` and `enumerate_evaluate`.

El objetivo de estas 2 funciones es calcular la suma de las longitudes de cada palabra de una lista `words` ponderadas por una lista de coeficientes `coefs` (Si, las dos funciones deben hacer lo mismo).

Las listas `coefs` y `words` tienen que tener la misma longitud. Si este no es el caso, la función debe devolver -1.

Debes obtener el resultado deseado utilizando `zip` en la función `zip_evaluate`, y `enumerate` en la función `enumerate_value`.

Ejemplos

```
>> 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
```

Capítulo VII

Ejercicio 05

	Ejercicio : 05
Cuenta Bancaria	
Directorio de entrega : $ex05/$	
Archivos a entregar : `the_bank.py`	
Funciones prohibidas : `None`	

Objetivo

Los objetivos de este ejercicio son descubrir nuevas funciones incorporadas y profundizar en la manipulación de clases, así como conocer la posibilidad de modificar objetos instanciados. En este ejercicio aprenderás a modificar o añadir atributos a un objeto.

Instrucciones

Todo es cuestión de seguridad. Echa un vistazo a la clase denominada `Account` en el siguiente fragmento de código.

```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
```

Ahroa te toca programar una clase llamada `Bank`! Su objetivo será gestionar la parte de seguridad de cada intento de transferencia.

Seguridad significa comprobar que `Account`:

- es el onjeto adecuado,
- no está corrupto,
- y almacena dinero suficiente para completar la transferencia.

¿Cómo definimos si una cuenta bancaria está corrompida? Una cuenta bancaria corrupta:

- tiene un número par de atributos,
- tiene un atributo que empieza por `b`,
- no tiene ningún atributo que empieza por `zip` o `addr`,
- no tiene atributos `name`, `id` o `value`,
- `name` no es un `string`,
- `id` no es un `int`,
- `value` no es un `int` o un `float`.

Para el resto de atributos (`addr`, `zip`, etc ... no se espera ninguna comprobación específica. Esto significa que no se espera que evalúes la validez de la cuenta basándote en el tipo de los otros atributos (las condiciones enumeradas anteriormente son suficientes).

Además, la verificación debe realizarse cuando se añaden objetos `Account` a la instancia de `Bank` (`bank.add(Account(...))`). La verificación en `add` solo comprueba el tipo de la new_account y si no hay ninguna cuenta entre las ya existentes en la instancia de `Bank` con el mismo nombre.

Una transacción no es válida si `amount <0` si el importe es superior al saldo de la cuenta. Antes de la transferencia, se comprueba la validez de las 2 cuentas (`origin`

y `dest`) (según la lista de criterios anterior). Una transferencia entre la misma cuenta (`bank.transfer('Wiliam John', 'William John')`) es válida, pero no hay ningún movimiento de fondos.

`fix_account` recupera una cuenta dañada si el parámetro `name` parámetro corresponde al nombre de atributo de una de las cuentas en `accounts` (atributo de Bank). Si el nombre no es un string o no corresponde a un nombre de cuenta, el método devuelve `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  ...
```

Echa un vistazo a la función incorporada `dir`.

TENDRÁS QUE MODIFICAR LOS ATRIBUTOS DE LAS INSTANCIAS PARA ARREGLARLAS.

Ejemplos

El script `banking_test1.py` es un test que debe imprimir `Failed`. El segundo script `banking_test2.py` es un test que debe imprimir `Failed` y luego `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.
```

Reconocimientos

Los módulos Python & ML son el resultado de un trabajo colectivo, al que queremos dar las gracias:

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

que supervisó la creación, la mejora y esta transcripción.

- 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)

por su inversión en la creación y desarrollo de estos módulos.

- 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)

que realizaron las pruebas beta de la primera versión de los módulos de aprendizaje automático.