Formation Python

PAR ALAIN CARIOU, NOVEMBRE 2023

I – La bibliothèque standard

La bibliothèque standard

- La bibliothèque standard du Python propose de nombreux modules utiles :
  - **os** qui permet d'interagir avec le système d'exploitation
  - **json** qui permet de manipuler le json
  - **datetime** qui permet de manipuler les dates
  - **sys** qui permet à l'utilisateur d'accéder aux variables systèmes de l'interpréteur
  - **math** qui donne accès à des fonctions mathématiques
  - **random** qui gère les affichages aléatoires
  - et de nombreux autres !

Passer des arguments à un programme Python avec le module sys

- En utilisant le module **sys**, il est possible de récupérer des arguments passer à un script Python.
- Ils sont stockés dans la variable **sys.argv**. Le premier argument est toujours le nom du script.
- Cela permet de modifier le comportement d'un programme en fonction de la manière dont il est lancé.

```
import sys

nb_args = len(sys.argv)
list_args = str(sys.argv)

print(f"Nombre d'arguments du programme : {nb_args}")
print(f"Liste des arguments : {list_args}")
```

II – Les requêtes

Effectuer une requête GET - 1

- Il existe différents modules Python permettant d'effectuer des requêtes HTTP (urllib2, urllib3, httplib). Aujourd'hui, le module **requests** permet de faire ces choses là très simplement.

- Pour l'installer :
  - **> pip install requests**

- Il faudra ensuite faire un **import** de **requests** dans les fichiers adéquats

Effectuer une requête GET - 2

- Une méthode **get** permet de faire des requêtes **get**.
- L'attribut **status_code** permet de récupérer le code de **status** de la requête.
- Les attributs **text** et **json** permettent de manipuler les données sous format **texte brute** ou en **JSON**.
- En cas de headers, on peut toujours remplir le **tableau de headers[« head1 », « head2 », …]**.

Exemple

- Exemple simple d'utilisation d'une requête **GET** :

```
import requests

r = requests.get("https://randomuser.me/api/")

print(r.status_code)

result = r.json()

print(r.text)

print("Bonjour " + result['results'][0]['name']['first'])
```

III – Les environnements virtuels

Présentation des virtualenv

- Un des points de Python concerne la possibilité d'utiliser des **environnements virtuels**.
- Il s'agit d'environnements composés de leur propre binaire Python et de ses dépendances.
- Cela permet d'avoir plusieurs projets chacun fonctionnant avec une version différente de Python ou de ses modules.

Installer un virtualenv

- Si vous avez besoin d'installer un environnement virtuel, exécuter la commande suivante :
  - **pip install virtualenv**

- Une fois cela effectué, vous pouvez créer un environnement virtuel comme suit :
  - **python -m venv <*NOM_DU_VENV*>**

- Cela crée un répertoire contenant un dossier **Scripts** (contient les exécutables), un dossier **Include** (contient les entêtes de l'environnement) et un dossier **Lib** (contient les librairies).

Activer et désactiver un virtualenv

- Une fois votre environnement virtuel créé, il faut l'activer à partir d'un script :
  - ***<PATH_TO_ENV>*/Scripts/activate**

- Cela permet d'installer des modules dans cet environnement virtuel et d'exécuter une application Python dans un environnement différent que celui générique.

- Pour désactiver l'environnement, cela va aussi se faire via un script :
  - ***<PATH_TO_ENV>*/Scripts/deactivate** ou tout simplement : ***deactivate***

Exercice API :

- A partir de l'API Harry Potter, vous devez développer une application qui va récupérer les personnage de la série et les lister chacun dans un fichier en fonction de leur maison.

- Il y aura donc un fichier Gryffondor, un fichier Serdaigle, un fichier Poufsouffle et un fichier Serpentard.

- Chaque fichier contiendra le nom et prénom des personnages, leur date de naissance ainsi que la composition de leur baguette magique.

- Lien vers l'API : *https://hp-api.onrender.com/*

IV – La programmation orientée objet

La programmation orientée objet

- Le Python est un langage qui supporte la programmation orientée objet.
- Pour déclarer une **classe** on utilisera le mot-clé **class**.
- Les méthodes sont définies comme des fonctions grâce au mot-clé **def**.
- Une classe peut avoir un **constructeur** à travers la méthode **__init__()** afin d'instancier automatiquement la classe lorsqu'un nouvel objet est créé.

Exemple de classe :

- Voici deux exemples de classe :

```
class MyClass:
    num = 12345

    def hello(self):
        return 'hello world'


x = MyClass()
print(x.num)
print(x.hello())
```

```
class Wolf:

    kind = 'wolf'

    def __init__(self, name):
        self.name = name


g = Wolf('Ghost')
n = Wolf('Nymeria')
print(g.kind)
print(n.kind)
print(g.name)
print(n.name)
```

- A noter que toutes les méthodes et attributs sont **publics** !

Méthodes statiques et de classes

- Méthode **statique** :
  - Décorator ***@staticmethod***
  - Ne peut pas accéder à l'état de la classe
  - Méthode statique classique
- Méthode de **classe** :
  - Décorator ***@classmethod***
  - Peut accéder à l'état de la classe à travers ***cls***
  - Est utilisé pour les factory

```python
class Item:
    nbr_instance = 0

    def __init__(self, name, desc) -> None:
        self.name = name
        self.description = desc
        Item.nbr_instance += 1

    @classmethod
    def get_nbr_instance(cls):
        print("Description :", cls.nbr_instance)

    @staticmethod
    def display():
        print("C'est un objet !")
```

Attributs publics et encapsulation

- Ainsi lorsque des méthodes ou des attributs sont spécifiés normalement au sein de la classe, ils sont **publics**.

- C'est à dire qu'ils partagent une valeur commune au sein de toutes les instances de la classe.

- Un des problèmes de Python, c'est **qu'il n'assure pas l'encapsulation**.

- En effet **tout est modifiable** !

Membres privés et protégés

- Si un attribut ou une méthode est instanciée avec deux « _ » devant leur nom alors ce membre est **privé**. sa valeur est spécifique à chaque instance de la classe :
    - Exemple : ***self.__firstname = firstname*** signifie que self.__firstname est un attribut privé.

- Si par contre il est instanciée avec un seul « _ » devant son nom alors ce membre est **protégé**.
    - Exemple : ***self._lastname = lastname*** signifie que self._lastname est un attribut protégé.

Le décorateur @property

- Une manière de gérer l'encapsulation consiste à utiliser le **décorateur *@property***.
-
- La méthode ayant le même nom que l'attribut et avec le décorateur « ***@property*** » sert **d'accesseur**.
- Celle ayant le même nom que l'attribut et avec le décorateur « ***@name.setter*** » sert de **mutateur**.

```python
class Person:
    def __init__(self, name):
        self.name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

p = Person("Titi")
print(p.name)
```

Méthodes « magiques »

- Les classes en Python disposent d'un certain nombre de **méthodes « magiques »** déjà existantes mais pouvant être surchargées :
  - **__init__** : le constructeur.
  - **__str__** : la méthode *toString*.
  - **__eq__** : permet de comparer l'instance courante à un autre objet passé en paramètre.
  - **__del__** : le destructeur appelé par le ramasse-miette.
  - **__hash__** : retourne un hash de l'objet sous forme d'entier.

L’héritage

- Les classes en Python peuvent aussi supporter un héritage simple ou multiple.

- En cas d’héritage on déclare la classe de cette manière :
  - **class Child(ParentClass):**

- Ensuite il est possible d’appeler les méthodes et attributs de la classe parent dans la classe enfant comme si c’était ceux de l’enfant.

Exemple d'héritage simple :

- Voici un exemple d'héritage simple dans une classe :

- A noter que le premier argument d'une méthode est, par convention, ***self***. Ce mot fait référence à l'instance courante de la classe actuellement appelée. Cela permet de manipuler ses méthodes et ses attributs.

```
class Parent:
    parentAttr = 100

    def __init__(self):
        print("Calling parent constructor")

    def parent_method(self):
        print('Calling parent method')

    def set_attr(self, attr):
        Parent.parentAttr = attr

    def get_attr(self):
        print("Parent attribute :", Parent.parentAttr)

    def my_method(self):
        print('Calling parent method')


class Child(Parent):
    def __init__(self):
        print("Calling child constructor")

    def child_method(self):
        print('Calling child method')

    def my_method(self):
        print('Calling child method')


c = Child()
c.child_method()
c.parent_method()
c.set_attr(200)
c.get_attr()
c.my_method()
```

Fonction __init__() et héritage

- Il est à noter que lorsqu'on ajoute une fonction **__init__()** à une classe enfant, elle n'hérite plus de la fonction **__init__()** de sa classe parent.

- On peut donc appeler directement la fonction **__init__()** du parent :

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

L’héritage en diamant

- Seulement les problèmes surviennent dans le cas d’un **héritage en diamant**.

- Dans l’exemple ci-contre, la classe ***Animal*** est appelée deux fois :
  - Une fois par la classe ***Marcheur***.
  - Et une autre fois par la classe ***Volant***.

```
class Animal:
    def __init__(self):
        print('Animal.__init__ ()')

class Marcheur(Animal):
    def __init__(self):
        print('Marcheur.__init__ ()')
        Animal.__init__(self)

class Volant(Animal):
    def __init__(self):
        print('Volant.__init__ ()')
        Animal.__init__(self)

class Oiseau (Marcheur, Volant):
    def __init__(self):
        Marcheur.__init__(self)
        Volant.__init__(self)

o = Oiseau ()
```

L’héritage en diamant c’est super !

- Pour pallier à ce problème, on peut utiliser la fonction **super()**.
- Cette fonction récupère l’instance et la classe courante et déroule la hiérarchie des classes afin d’appeler ses fonctions parentes une seule fois.
- Ceci est une explication simpliste de l’utilisateur de **super()**. Cette fonction étant beaucoup plus complexe.

```
class Animal:
    def __init__(self):
        print('Animal.__init__ ()')

class Marcheur(Animal):
    def __init__(self):
        print('Marcheur.__init__ ()')
        super().__init__()

class Volant(Animal):
    def __init__(self):
        print('Volant.__init__ ()')
        super().__init__()

class Oiseau (Marcheur, Volant):
    def __init__(self):
        super().__init__()

o = Oiseau ()
```

Exercice : le JDR

- Créez les classes suivantes : une classe « personnage », une classe « guerrier », une classe « clerc », une classe « paladin », une classe « mage » et une classe « archer ».

- Chaque classe aura un *nom*, des *pointsdevie*, des *pointsdeviemax*, une *attaque*, une *défense* et une *catchphrase*.

- De plus elles auront toutes une méthode *attaquer(personnage)* et *sePresenter()*. La méthode attaquer() fait perdre des points de vie égale à l'attaque de l'attaquant – la défense de l'attaqué. La méthode sePresenter() permet au personnage de donner son nom et sa catchphrase.

Exercice : le JDR - suite

- En sachant que :
  - Le clerc a une méthode *soigner()* qui prend un personnage en paramètre et lui redonne des points vie.
  - Le guerrier a une méthode *criDeGuerre()* qui lui permet de crier sa catchphrase.
  - Le paladin est un guerrier et un clerc.
  - L'archer est un guerrier qui peut *tirer(personnage)* ce qui inflige a ses cibles des dégâts équivalent à l'attaque de l'archer.
  - Le mage a un attribut *mana* et il peut *jeterUnSort(personnage)* qui inflige des dégâts à la cible équivalent à son attaque et redonne au mage le même nombre de points de vie. Cette action coûte du mana.