cours_complement_POO

December 17, 2020

[ ]:
```
# Pour pythontutor
%load_ext nbtutor
```

1 Encapsulation : interface et implémentation

1.1 Rappel

Nous avons introduit en première la notion de fonction
* Une fonction est **définie** par son **"concepteur"**. Celui-ci écrit aussi sa documentation afin d'indiquer à l'**"utilisateur"** comment utiliser la fonction lors d'un **appel** * Lors de l'appel de la fonction, l'utilisateur ne s'intéresse pas au code de la fonction. Il n'a pas besoin de le comprendre ou même de le regarder pour utiliser la fonction

On dit que la fonction **encapsule** du code. Il faut bien distinguer : * le **"concepteur"** dont le rôle est d'écrire la **définition de la fonction** * l'**"utilisateur"** qui appelle la fonction sans se préoccuper du code écrit par le **"concepteur"**

$\Longrightarrow$ la même notion est généralisée en terminale aux classes et aux structures de données abstraites.

[1]:
```
class MaClasse :
    def __init__(self, valeur):
        self.attribut = valeur

objet = MaClasse(1)
```

[2]:
```
type(objet)
```

[2]:
```
__main__.MaClasse
```

Tout comme il existe des types natifs, vous pouvez en créant de nouvelles classes définir des objets d'un nouveau type. Une fois ce nouveau type créé (par le concepteur de classe), n'importe quel utilisateur peut importer cette nouvelle classe et manipuler des objets de ce nouveau type sans se préoccuper de la façon dont ils ont été codés par le concepteur. C'est ce que vous avez fait depuis le début. Par exemple, avec un objet de type `lst` :

[3]:
```
# Création d'un objet ma_liste de type lst
ma_liste = list("abcd")
print(ma_liste)
```

```
['a', 'b', 'c', 'd']
```

[4]:
```
# ma_liste est un objet instance de la classe lst

isinstance(ma_liste, list) #la fonction isinstance permet de tester si un objet␣
↪est l'instance d'une classe
```

[4]:
```
True
```

[5]:
```
ma_liste.append("e")
print(ma_liste)
```

```
['a', 'b', 'c', 'd', 'e']
```

Dans les quelques lignes de code ci-dessus, vous **utilisez** la classe `lst` pour créer un objet et vous le manipulez (par exemple en utilisant la méthode `append`). Mais en aucun cas vous n'avez regardé le code écrit par le concepteur pour effectivement créer cet objet, ni le code de la méthode `append`

1.2 Implémentation

L'**implémentation** relève du travail du **"concepteur"** de la classe. L'**implémentation** permet de définir le nouveau type abstrait : * construire les objets instance de cette classe (méthode `__init__`) * définir toutes les **méthodes** qui permettent de manipuler les objets de cette classe

1.3 Interface

L'**interface** est l'ensemble des moyens mis à disposition de l'**"utilisateur"** pour manipuler la structure de donnée abstraite.

Même si l'interface a bien été créée par le **"concepteur"** lors de l'**implémentation**, l'**"utilisateur"** ne fait qu'utiliser cette interface **sans jamais se soucier de la construction interne de l'objet**

Pour créer une interface, le **"concepteur"** crée des méthodes, chacune d'entre elles permet de faire une manipulation spécifique sur l'objet

1.4 Analogie

- Lorsque vous **utilisez** une voiture (objet), vous manipulez celle-ci via son **interface** (volant, pédales, leviers de vitesses, commandes du tableau de bord).
- Ce qui se passe "sous le capot" (embrayage, injecteurs, pistons, soupapes etc…) ne vous intéresse absolument pas. Vous n'avez pas besoin d'y mettre les mains ni de de comprendre comment tout ça fonctionne pour "manipuler" votre voiture

1.5 Modularité

- Comme nous avons fait en première sur les fonctions, il est possible de regrouper différentes implémentations dans une **bibliothèque**. Cette bibliothèque est une simple fichier python `.py` contenant de nouvelles classes
- L'utilisateur n'a plus qu'à importer cette bibliothèque grâce au module `import`

2 Programmer en orienté objet lors de projets

2.1 Démarche à suivre

Encore plus qu'en programmation procédurale, il est important de bien réfléchir ("brainstorming" sur une feuille) sur comment aborder le problème **avant** de commencer à coder. Il existe d'ailleurs toute une méthodologie permettant de bien développer en POO comme la modélisation UML (hors programme)

Basiquement, voici quelques questions à se poser avant de coder : * Quels sont les différents objets dont j'ai besoin ? * Quels attributs doivent avoir ces objets. * Quels sont les interactions des objets entre eux et leurs comportements

$\Longrightarrow$ Créer les classes, attributs et méthodes uniquement nécessaires au programme. Par exemple, si votre projet concerne la gestion d'un parking. Vous allez certainement créer des classes `Voiture`, `place`, `parking`. L'attribut `hauteur` de la voiture sera sans doute nécessaire (par exemple pour savoir si la voiture peut accéder au parking) mais l'attribut `couleur` beaucoup moins. **Ne pas vouloir décrire complètement vos objets : juste le nécessaire à votre projet**

3 Mutabilité des objets

Attention : Comme on l'a vu en première sur les listes et les dictionnaires, les objets instances d'une classe sont **mutables**. (Il est possible de les rendre immuables mais c'est hors programme)

Exemple :

[7]:
```
%%nbtutor -r -f

class MaClasse :
    def __init__(self, valeur):
        self.attribut = valeur

objet = MaClasse(1)
autre_objet = objet
```

[8]:
```
%%nbtutor -r -f

objet.attribut = 2
print(autre_objet.attribut)
```

```
2
```

`objet` et `autre_objet` référencent en fait le même objet en mémoire. Toute modification sur l'un entraîne donc *implicitement* une modification sur l'autre.

4 Méthodes spéciales

- En python, une méthode spéciale a un nom entouré de part et d'autre par deux *underscore*. Le nom d'une méthode spéciale prend donc la forme : `__methodespeciale__`.

- les méthodes spéciales permet de donner du sens à certaines opérations. Il en existe beaucoup comme par exemple :
  - la méthode `__add__` permet de donner du sens à l'écriture `objet1 + objet2`
  - la méthode `__mul__` permet de donner du sens à l'écriture `objet1 * objet2`
  - la méthode `__eq__` permet de donner du sens à l'écriture `objet1 == objet2`
  - la méthode `__len__` permet de donner du sens à l'écriture `len(objet1)`
  - etc…
- En l'absence de définition de ces méthodes (ce qui sera le plus souvent le cas), un sens à l'opération en question pourrait être donné par la classe parente (notion d'héritage hors programme). A votre niveau, retenez juste que **rien ne garantit le résultat obtenu si la méthode spéciale correspondante n'a pas été définie**

4.1 Exemple : la méthode `__repr__`

La méthode `__repr__` est appelée lorsqu'on souhaite afficher un objet (en tappant directement son nom dans l'interpréteur ou via la fonction `print`) comme ci-dessous :

[9]:
```
class Personne :
    def __init__(self, nom, profession):
        self.nom = nom
        self.profession = profession

personnage = Personne("Alice", "informaticienne")
```

[10]:
```
personnage
```

[10]:
```
<__main__.Personne at 0x5058ba8>
```

[11]:
```
print(personnage)
```

```
<__main__.Personne object at 0x0000000005058BA8>
```

Ici comme `__repr__` n'a pas été définie, le résultat provient de la classe parente (et qui renvoie ici un équivalent de l'adresse mémoire où a été stockée l'objet `personne`, ce qui ne nous intéresse pas beaucoup…)

Or, il est souvent utile d'afficher un objet pour connaître ses caractéristiques, surtout quand on souhaite debugger un code. Il suffit donc de définir la méthode `__repr__`. On est alors libre de choisir la représentation qui nous intéresse…

[12]:
```
class Personne :
    def __init__(self, nom, profession):
        self.nom = nom
        self.profession = profession

    def __repr__(self):
        return "Personne : "+self.nom+". Profession "+self.profession
```

```
personnage = Personne("Alice", "informaticienne")
```

[13]:
```
personnage
```

[13]:
```
Personne : Alice. Profession informaticienne
```

[14]:
```
print(personnage)
```

```
Personne : Alice. Profession informaticienne
```

5 Le cas particulier du langage python en POO

Reprenons l'exemple du compte bancaire du cours d'introduction à la POO

5.1 Implémentation

Il est important de bien comprendre que le code ci-dessous correspond à une implémentation d'un nouveau type `CompteBancaire`

[15]:
```
class CompteBancaire :
    def __init__(self, numero, titulaire):
        self.numero = numero
        self.titulaire = titulaire
        self.solde = 0

    def deposer_argent(self, montant) :
        self.solde = self.solde + montant

    def retirer_argent(self, montant) :
        self.solde = self.solde - montant
```

5.2 Interface

Il est important de bien comprendre que le code ci-dessous correspond au travail d'un individu qui va simplement utiliser des objets de type `CompteBancaire` et de les manipuler par son interface. Cet individu est A PRIORI DIFFERENT de celui qui a écrit limplémentation de `CompteBancaire`

5.2.1 Utilisation "normale"

[16]:
```
compte_Alice = CompteBancaire (12345, "Alice") # Création du compte

compte_Alice.deposer_argent(200)
compte_Alice.retirer_argent(50)
print(compte_Alice.solde)
```

```
150
```

5.2.2 Utilisation “anormale”

Le code ci-desssous fonctionne. **Question :** Pourquoi peut-on qualifier le code ci-dessous comme utilisation anormale du nouveau type `CompteBancaire` ?

[17]:
```
print(compte_Alice.solde)

compte_Alice.solde = 10000
print(compte_Alice.solde)
```

```
150
10000
```

Réponse : Personnellement j’aimerais bien que mon compte bancaire passe de 150€ à 10000€ **sans qu’il n’y ait eu aucun dépot d’argent !!**. Malheureusement, force est de constater que cela n’arrive jamais … :-(

5.2.3 Utilisation “anormale” 2

On peut même faire pire : voir le code ci-dessous toujours fonctionnel mais franchement horrible !

[18]:
```
del compte_Alice.titulaire
del compte_Alice.numero
compte_Alice.taux = 0.02
```

Qu’est-ce qui a été fait ?

Il supprime les attrributs `titulaire` et `numero` de l’objet `compte_Alice` et il en crée un nouveau `taux`. La preuve ci desssous :

[19]:
```
print(compte_Alice.taux)
print(compte_Alice.solde)
print(compte_Alice.titulaire)
print(compte_Alice.numero)
```

```
0.02
10000


      ␣
↪---------------------------------------------------------------------------

      AttributeError                            Traceback (most recent call␣
↪last)

      <ipython-input-19-f7ef21309889> in <module>
        1 print(compte_Alice.taux)
        2 print(compte_Alice.solde)
   ----> 3 print(compte_Alice.titulaire)
        4 print(compte_Alice.numero)
```

```
AttributeError: 'CompteBancaire' object has no attribute 'titulaire'
```

Pourquoi ce code est il horrible ?

- Vous avez un compte bancaire sans titulaire et sans numéro de compte. Bref un compte bancaire qui n'appartient plus à personne
- Pire, `compte_Alice` est toujours un objet instance de `CompteBancaire` alors que l'objet n'a plus grand chose à voir avec la classe dont il est l'instance

$\Longrightarrow$ En bref, en python il est possible de manipuler un objet d'une manière totalement incohérente avec l'idée initiale de la classe dont il est l'instance

[20]:
```
isinstance(compte_Alice, CompteBancaire)
```

[20]:
```
True
```

5.2.4 Le cas particulier de python en POO

Cet exemple montre l'intérêt de restreindre les manipulations afin d'empêcher les utilisateurs d'une classe de "faire n'importe quoi" avec les objets. Dans beaucoup de langages orientés objets (comme java), le concepteur qui écrit l'implémentation de la classe peut rendre certains attributs (et certaines méthodes) privés tandis que d'autres sont publics. Le langage se charge d'interdire toute manipulation sur les attributs et méthodes privés en dehors de la classe. Seules les méthodes publiques composent l'interface de l'objet.

En python, rien de tout cela n'existe : il est impossible de protéger les objets d'une utilisation inappropriée en rendant certains attributs privés.

Comment faire en python ? Il est d'usage de nommer tout ce qui est "privé" par un nom commençant par un double underscore comme ceci `__attributPrive`.

cette syntaxe agit juste comme un panneau "attention danger". Mais rien n'interdit de passer outre le panneau…

Inutile de lancer la polémique : *"java c'est mieux ou moins bien que python"*. Ces 2 langages ont juste des "philosophies" différentes : En python, on considère l'utilisateur de la classe comme un "grand garçon" qui sait ce qu'il fait et donc on ne met aucune interdiction, juste une mise en garde.

Voici le code respectant cette convention : * les attributs "sensibles" sont "protégés" par la notation double underscore. * On y a ajouté une nouvelle méthode `get_solde` car on veut garder la possibilité d'obtenir le solde du compte : c'est la moindre des choses pour un compte bancaire… * On s'est bien gardé d'écrire la méthode `set_solde` permettant d'affecter une nouvelle valeur au solde du compte * Remarque : on devrait faire de même pour les attributs `__numero` et `__solde`

[21]:
```
class CompteBancaire :
    def __init__(self, numero, titulaire):
        self.__numero = numero
        self.__titulaire = titulaire
        self.__solde = 0
```

```
    def deposer_argent(self, montant) :
        self.__solde = self.__solde + montant

    def retirer_argent(self, montant) :
        self.__solde = self.__solde - montant

    def get_solde(self):
        return self.__solde
```

Remarque : Attention, on lit parfois (sur certains sites web) que les doubles underscore `__` protègent les attributs et qu'ils les rendent privés. En fait non…

[22]:
```
compte_Alice = CompteBancaire(12345,"Alice")
compte_Alice.deposer_argent(100)
```

[23]:
```
# L'attribut solde semble vraiment privé...
compte_Alice.__solde

#... mais en fait non, il existe une syntaxe particulière (non donnée dans ce␣
 ↪cours)
# permettant d'accéder directement à __solde (et sans passer par la méthode␣
 ↪get_solde)
```

```
      ␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-23-1026f9c8b732> in <module>
         1 # L'attribut solde semble vraiment privé
   ----> 2 compte_Alice.__solde
         3
         4 #… mais en fait non, il existe une syntaxe particulière (non donnée␣
↪dans ce cours)
         5 # permettanbt d'accéder directement à __solde (et sans passer par la␣
↪méthode get_solde)…


       AttributeError: 'CompteBancaire' object has no attribute '__solde'
```

[24]:
```
# Utilisation de la méthode get_solde pour accéder au solde de compte_Alice
compte_Alice.get_solde()
```

[24]: 100

5.2.5 Le débutant en POO sur python

Comme vous débutez en POO, vous n'êtes pas des "grands garçons" et cette liberté offerte par python est à mon avis néfaste pour l'apprentissage

- Avoir conscience que vous, utilisateur de la classe, ne devriez pas toucher à tout ce qui commence par des doubles underscore `__` c'est-à-dire tout ce qui est "privé" (hors cas particulier)
- **Surtout ne pas créer ou supprimer des attributs "à la volée"** (comme ci dessus `taux`, `titulaire`, `numero`)
- Lorsque vous implémentez une classe, nommer les attributs et méthodes sensibles par un nom commençcant par des doubles underscore `__` pour "mettre un panneau danger"
- **l'INTERFACE de l'objet est constituée des attributs et des méthode ne commençant pas par des doubles underscore `__`**

Remarque : Ne pas confondre ce qui est "privé" et qui commence par des doubles underscore `__` avec les méthodes spéciales qui commencent **et** finissent par des doubles underscore `__` : * `__methodePrivee` * `__methodeSpeciale__`

6 Quelques outils supplémentaires

- la fonction `isinstance` permet de contrôler si un objet est instance d'une classe
- le mot clé `is` permet de comparer l'identité de 2 objets (`is` est "plus fort" que `==`)
- la fonction `help` permet d'obtenir la documentation sur une classe donnée
- la fonction `dir` permet de lister les méthodes définies dans une classe

Exemple :

[25]:
```
compte_Alice = CompteBancaire(12345,"Alice")
compte_Bob = CompteBancaire(6789,"Bob")
```

[26]:
```
isinstance(compte_Bob,CompteBancaire)
```

[26]: True

[27]:
```
isinstance(compte_Bob,list)
```

[27]: False

[28]:
```
isinstance([2,6,4],list)
```

[28]: True

[29]:
```
compte_Alice is compte_Bob
```

[29]: False

[30]:
```
liste1 = [2,6,4]
liste2 = list(liste1)
liste3 = liste1

print(liste1, liste2, liste3)
```

```
[2, 6, 4] [2, 6, 4] [2, 6, 4]
```

[31]:
```
liste1 is liste2
```

[31]:
```
False
```

[32]:
```
liste1 is liste3
```

[32]:
```
True
```

[33]:
```
help(CompteBancaire)
```

```
Help on class CompteBancaire in module __main__:

class CompteBancaire(builtins.object)
 |  CompteBancaire(numero, titulaire)
 |
 |  Methods defined here:
 |
 |  __init__(self, numero, titulaire)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  deposer_argent(self, montant)
 |
 |  get_solde(self)
 |
 |  retirer_argent(self, montant)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

[34]:
```
dir(CompteBancaire)
```

[34]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'deposer_argent',
 'get_solde',
 'retirer_argent']
```

Remarque : vous pouvez remarquer qu'il y a beaucoup de méthodes spéciales qui existent alors qu'on ne les a pas écrites dans la classe `CompteBancaire`. Python les crée automatiquement : certaines sont héritées de la classe mère `object` (hors programme), d'autres ne sont pas implémentées…