Programmation objet

Avant de commencer

On pourra relire le notebook `chiens_et_autres_objets.ipynb`.

I : Classes et Attributs

Une *classe* définit et nomme une structure de données qui peut regrouper plusieurs attributs (aussi appelés *champs* ou *propriétés*). Une fois qu'une classe est définie, on peut facilement obtenir plusieurs *instances* différentes de cette classe (on dit parfois *objet* au lieu d'instance). La classe peut être vue comme un modèle, les instances comme des exemplaires différents du modèle.

I.1 : Définition d'une classe et création des instances ou objets

En Python la définition d'une classe :
- est introduite par le mot-clef `class` suivi du nom choisi pour la classe commençant par une majuscule,
- comporte (sauf exception) une méthode *constructeur* __init__ qui permet d'initialiser les attributs des différentes instances,
- utilise par convention le nom de variable `self` qui désigne l'instance de la classe,
- `self` *possédant l'extraordinaire particularité d'être obligatoire dans les en-têtes des méthodes mais ne devant pas être mentionné lors des appels (il est passé en argument de façon transparente !).*

L'exemple ci-contre est archétypique. Lorsqu'on crée les deux instances (ou objets) `anniv_alice` et `anniv_bob`, la méthode constructeur `__init__` est appelée deux fois avec :

- l'argument `self` (égal à `anniv_alice` puis `anniv_bob`) qui est transmis de façon invisible,

- les arguments `j, m, a` (égaux à `7, 12, 2003` puis `29, 2, 2004`) qui sont transmis de façon explicite pour être affectés aux trois attributs `jour, mois, annee`.

```
class Date:
    '''Une classe pour représenter une date'''

    def __init__(self, j, m, a):
        self.jour = j
        self.mois = m
        self.annee = a


anniv_alice = Date(7, 12, 2003)
anniv_bob = Date(29, 2, 2004)
```

I.2 : lecture et modification des attributs des instances

On accède en lecture ou en écriture aux attributs des instances grâce à une *notation pointée*. Les attributs sont en effet *mutables*.

Ainsi pour accéder à l'attribut `att` d'un objet (ou instance) `obj` on utilise la notation `obj.att` qui permet aussi bien de lire la valeur de l'attribut que de la modifier.

```
>>> date_mensualite = Date(7, 10, 2020)
>>> date_paie = Date(30, 9, 2020)
>>> date_paie.mois
9
>>> date_mensualite.jour
7
>>> date_paie.mois = date_paie.mois + 1
>>> date_paie.mois
10
```

II . Méthodes d'instances

Une fois les attributs des objets d'une classe définis, on a souvent besoin de fonctions pour manipuler les attributs des objets. Ces fonctions, définies à l'intérieur des classes, sont appelées des *méthodes* : elles peuvent par exemple modifier certains attributs, renvoyer des valeurs calculées à partir des attributs, créer de nouveaux objets, modifier d'autres objets etc.

II . 1 Définition des méthodes et appels aux méthodes

Les méthodes sont définies à l'intérieur de la définition des classes comme les fonctions habituelles à l'aide du mot-clef `def`. La seule différence, déjà évoquée dans le I, est que leur en-tête comporte obligatoirement comme premier paramètre `self` qui ne sera pas mentionné lors des appels puisqu'il sera remplacé automatiquement de façon invisible lors de l'appel par l'instance elle-même.

Voyons cela sur une classe `Cercle_magique` disposant de deux attributs : `rayon` et `couleur` et pour laquelle on souhaite :
- une méthode permettant de "gonfler" le cercle d'une valeur `dr` donnée (ce qui augmente le rayon de `dr` unités),
- une méthode permettant d'obtenir l'aire du cercle,
- une méthode permettant à un cercle de "pondre" un nouveau cercle de rayon 1 et de même couleur que lui (ce qui fait diminuer de 1 le rayon du cercle pondeur et nécessite donc que le cercle pondeur soit de rayon supérieur ou égal à 1).

```
from math import pi

class Cercle_magique:
    '''une classe pour représenter des cercles magiques'''

    def __init__(self, r, c):
        self.rayon = r
        self.couleur = c

    def gonfler(self, dr):
        self.rayon = self.rayon + dr

    def aire(self):
        return pi*pow(self.rayon, 2)

    def pondre(self):
        assert self.rayon >= 1
        self.rayon = self.rayon - 1
        return Cercle_magique(1, self.couleur)
```

```
>>> c_v = Cercle_magique(3, 'vert')
>>> c_r = Cercle_magique(7, 'rouge')

>>> c_v.rayon
3
>>> c_v.gonfler(2)
>>> c_v.rayon
5

>>> c_r.aire()
153.93804002589985

>>> c_x = c_v.pondre()
>>> c_x.rayon
1
>>> c_x.couleur
'vert'
>>> c_x.aire()
3.141592653589793
```

Ici il faut comprendre :
- que l'appel aux méthodes des objets se fait avec la notation pointée (ce que nous avons déjà utilisé, par exemple pour trier un tableau en le mutant grâce à l'instruction `mon_tableau.sorted()`),
- que lors des appels aux méthodes, Python remplace de façon invisible le paramètre `self` par l'instance (ou objet) appelant.

Par exemple lors de l'appel `c_v.gonfler(2)`, dans le code de la fonction `gonfler` :
- `self` va correspondre à `c_v`,
- `dr` va correspondre à `2`.

II . 2 : Quelques noms de méthodes spéciales …

On a vu que la méthode constructeur `__init__` aura toujours le même nom dans toutes les classes en langage Python : c'est ce qui permet à Python de savoir que c'est cette méthode là le constructeur. Le fait que ce soit un nom réservé est mis en évidence par les double underscore (tiret bas en français) de part et d'autre. Il existe d'autres méthodes réservées qui permettent d'utiliser des opérations standard de Python (`obj` désigne l'objet). En voici quelques-unes :

- `__str__(self)` doit renvoyer une chaîne de caractères qui sera renvoyée/affichée par `str(obj)` ou `print(obj)`,
- `__lt__(self, u)` doit renvoyer `True` si `obj` est strictement plus petit que `u` et permettra d'utiliser `obj < u`,
- `__len__(self)` doit renvoyer un nombre entier ou flottant qui sera renvoyé par `len(obj)`,
- `__contains__(self, x)` doit renvoyer `True` si `obj` contient x et permettra d'utiliser `x in obj`.

Ainsi en rajoutant la méthode ci-contre à notre classe Cercle_magique, nous serions en mesure d'utiliser ensuite l'instruction print :

```
>>> print(c_v)
rayon 5, couleur vert
```

```
class Cercle_magique:
    ...
    def __str__(self):
        s = "rayon " + self.rayon + ", couleur " + self.couleur
        return s
    ...
```

III : Mise à disposition dans un module

En première approche, pour mettre à disposition la classe précédente dans un module, il suffit :

1. de créer un fichier `.py` (par exemple `cercle_m.py`) dont le nom (sauf l'extension `.py`) sera le nom du module,
2. de copier le code donnant la définition de la classe dans ce fichier,
3. de sauvegarder ce fichier,
4. de copier ce fichier dans le même dossier que les fichiers python ou les notebooks qui doivent l'utiliser,
5. d'utiliser les instructions suivantes (au choix) dans les fichiers python ou les notebooks :
    - `import cercle_m` puis, lorsqu'on en a besoin, `c_v = cercle_m.Cercle_magique`
    - `from cercle_m import Cercle_magique` puis, lorsqu'on en a besoin, `c_v = Cercle_magique`

Nous verrons plus tard qu'il convient aussi de spécifier l'*interface* du module, c'est-à-dire les fonctions et/ou classes et/ou méthodes de classes qui sont destinées à être utilisées par les *clients*. Certaines méthodes peuvent ne pas figurer dans l'*interface* si elles ne servent que pour des calculs internes au module et n'ont pas à être connues des clients (au même titre que le mode d'emploi d'un ordinateur n'indique pas comment changer un port USB puisque officiellement ce doit être fait au SAV).