Classes et Objets

Où en sommes-nous des types de données ?

• Quelques *types simples* : `int`, `float`, `bool`

```
>>> type(-45)
<class 'int'>
```

```
>>> type(23.7)
<class 'float'>
```

```
>>> type(False)
<class 'bool'>
```

• Quelques *types composés*, suites numérotées d'objets. Nous avons vu trois sortes de **séquences** : `str`, `list`, `tuple`

```
>>> type('foo')
<class 'str'>
```

```
>>> type([2,6,1])
<class 'list'>
```

```
>>> type((2,6,1))
<class 'tuple'>
```

• Et deux autres collections non numérotées : `set`, `dict`

```
>>> type({2,6,1})
<class 'set'>
```

```
>>> type({'prix':50, 'nom':'Eska'})
<class 'dict'>
```

•Uniquement des **classes** d'objets ! **Tous les types sont des classes en Python**. Dans la classe `str`, vous avez vu la méthode `append`...

Mais qu'est-ce qu'une classe ?

• Une classe modélise un **ensemble d'objets** et les **comportements** de ces objets.

· Histoire des classes : Simula (1962-1967), Smalltalk (1972), C++ (1979), Objective-C (1983), CLOS (1987), Java (1995), etc.

• Les langages permettant de travailler avec des classes sont dits **langages à objets**. Le style de programmation correspondant s'appelle **programmation orientée objet (POO)**.

• Pour définir une classe il faut se poser deux questions relatives aux attributs des objets et aux méthodes sur ces objets :

i) Quelles sont les caractéristiques des objets de la classe ?

ii) Quelles sont les opérations pertinentes sur ces objets ?

Vite un exemple !!

• On veut construire la **classe** des cercles. Les cercles seront les **objets** de cette classe. Tous les objets d'une classe ont les mêmes **attributs** [*propriétés*, *champs*]. L'attribut `attr` d'un objet `obj` sera noté `obj.attr` en Python.

`cercle.couleur` ⟶ *la couleur du cercle (circle's color)*

`p.x` ⟶ *l'abscisse du point p*

`papa.chapeau` ⟶ *le chapeau de papa*

• On définit une **classe** avec le mot `class`, suivi d'instructions :

```
class Cercle :                    # avec C majuscule !
    """La classe des cercles"""       # help(Cercle)
    <instruction1>
    <instruction2>
    ......
```

Le texte d'une classe

• La classe la plus simple ne contient rien. L'instruction pass ne fait rien, mais intervient lorsqu'il faut absolument une instruction !

```
class Bidon :
    """La classe vide"""
    pass
```

```
>>> Bidon
<class '__main__.Bidon'>
```

• La plupart des instructions d'une classe seront des définitions de fonctions qui vont préciser le comportement d'un objet de la classe.

```
class Cercle :                          # avec C majuscule !
    """La classe des cercles"""
    def ...
    def ...
    ......
```

• Les fonctions définies dans une classe se nomment aussi **méthodes**.

La méthode spéciale __init__

• Un objet de la classe Cercle aura 3 attributs : centre, rayon et couleur dans cet ordre.

• La première fonction du texte de la classe doit être nommée obligatoirement __init__ avec ici 4 paramètres (1 + 3), dont le premier doit se nommer self et représente l'objet en cours d'initialisation :

```
class Cercle :
    """La classe des cercles"""
    def __init__(self,centre,rayon,couleur) :
        self.centre = centre          # tuple
        self.rayon = rayon            # int
        self.couleur = couleur        # str
```

N.B. i) Le mot self représente l'objet courant, celui dont il s'agit.
ii) Nous utiliserons rarement nous-mêmes la méthode __init__, nous nous contentons de la programmer pour Python !

• Pour construire un objet particulier (une **instance**) de la classe Cercle, j'utilise le nom de la classe comme pseudo-fonction à laquelle je passe les valeurs des attributs, dans l'ordre des paramètres :

```
>>> cercle = Cercle((10,25),80,'red')
```

N.B. On ne passe pas self *en argument ! Il est implicite.*

• La variable cercle représente alors un objet, instance de Cercle :

```
>>> cercle
<__main__.Cercle object at 0x102516390>
```

• Je peux demander l'accès aux attributs de cercle :

```
>>> cercle.centre
(10,25)
```

```
>>> cercle.couleur
'red'
```

• Je peux aussi modifier les attributs de cercle (ils sont dits ***publics***) :

```
>>> cercle.rayon = 2 * cercle.rayon
>>> cercle.rayon
160
```


• On peut passer les valeurs initiales des attributs en les nommant, dans un ordre quelconque :

```
cercle = Cercle(rayon=80,couleur='red',centre=(10,25))
```

• On peut aussi prévoir dans les paramètres de __init__ des valeurs par défaut pour certains attributs [en queue des paramètres] :

```
def __init__(self,centre,rayon=1,couleur='red') :
    self.centre = centre
    self.rayon = rayon
    self.couleur = couleur
```

```
>>> cercle = Cercle((10,25))
>>> cercle.rayon
1
```

rayon *et* couleur *sont optionnels*

• Un attribut spécial caché __dict__ permet d'inspecter un objet :

```
>>> cercle.__dict__
{'centre': (10, 25), 'rayon': 1, 'couleur': 'red'}
```

La méthode spéciale __repr__

• Une fois l'objet `cercle` construit, son affichage laisse à désirer :

```
>>> cercle
<__main__.Cercle object at 0x102516390>
```

• En fait, pour afficher un objet, la fonction `print` de Python fait appel à la fonction `str` qui elle-même invoque la méthode cachée `__repr__` de la classe de l'objet à afficher, chargée de convertir l'objet en *string* :

```
>>> str(cercle)     #  ⇔ cercle.__repr__()
'<__main__.Cercle object at 0x102516390>'
```

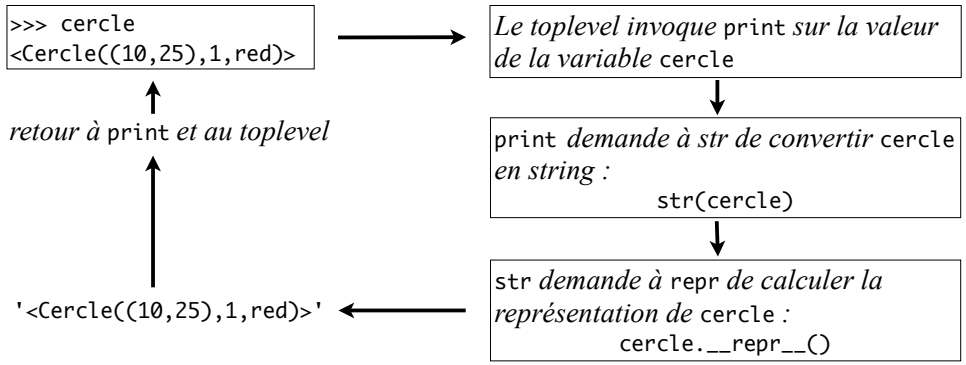
• Il suffit donc de redéfinir `__repr__` dans la classe `Cercle` :

```
def __repr__(self) :
    return '<Cercle(' + str(self.centre) + ','
           + str(self.rayon) + ',' + self.couleur + ')>'
```

```
>>> cercle
<Cercle((10, 25),1,red)>
```


__repr__ est une méthode cachée !

• On n'utilise PAS soi-même la méthode `__repr__` mais plutôt la fonction `str` qui va automatiquement invoquer `__repr__` :



NB. Python utilise **beaucoup de méthodes cachées**. Par exemple la fonction `len` sur les collections (listes, chaînes, ensembles…) cache la méthode `__len__` de la classe appropriée (`list`, `str`, `set`…) : `len([1,4,5])` ⇔ `[1,4,5].__len__()`

• Donc si dans une classe A je définis une méthode `__len__` retournant un entier, alors je pourrai écrire `len(obj)` lorsque `obj` ∈ A.

Les autres méthodes d'instance

• Il y a beaucoup d'autres **méthodes et attributs cachés** en Python.

• Mais les autres méthodes que vous allez rédiger dans le texte de la classe sont bien plus importantes !

• A quoi vont-elles servir ? A envoyer un **message** à un objet quelconque de la classe. Par exemple :

- demander à un `cercle` de bien vouloir calculer et retourner son aire : `cercle.aire()` `# avec résultat`

- lui demander de grossir en multipliant son rayon par k : `cercle.zoom(k)` `# sans résultat`

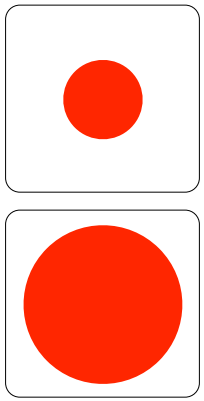
• Etc. En rédigeant le texte d'une classe, il faut imaginer quels sont les messages que nous aurons à **envoyer** aux objets, et comment ceux-ci **répondront**. Bref, quelles sont les *fonctionnalités* de la classe ?


```
import math
from turtle import *

class Cercle :
    """La classe des Cercles en L1"""

    def __init__(self,centre,rayon=1,couleur='red') :
        self.centre = centre
        self.rayon = rayon
        self.couleur = couleur

    def __repr__(self) :
        return '<Cercle(' + str(self.centre) + ',' + str(self.rayon)
               + ',' + self.couleur + ')>'

    def aire(self) :                    # méthode d'instance avec résultat
        return math.pi * self.rayon ** 2

    def zoom(self,k) :                  # méthode d'instance avec résultat
        self.rayon = k * self.rayon

    def dessine_toi(self) :             # méthode d'instance sans résultat
        p = self.centre ; pensize(self.rayon) ; pencolor(self.couleur)
        ht() ; up(); goto(p) ; down() ; dot()
```

Utilisation de la classe : parlons aux objets !

• Une fois la classe rédigée, il faudra créer des objets de cette classe et leur envoyer des messages. Notez au passage que les objets peuvent très bien s'envoyer des messages entre eux !...

```
>>> cercle = Cercle((10,25),rayon=100)
>>> cercle.__dict__           # attribut caché !
{'couleur':'red', 'centre':(10,25), 'rayon':100}
>>> cercle.aire()
31415.926535897932
>>> cercle.dessine_toi()
>>> cercle.zoom(2)
>>> cercle.__dict__
{'couleur':'red', 'centre':(10,25), 'rayon':200}
>>> cercle.dessine_toi()
>>> cercle
<Cercle((10,25),200,red)>
```



La liste des cercles : une variable de classe

• Dans un logiciel, il est parfois intéressant de maintenir une **liste** des objets de la classe qui ont été créés.

• Cette liste `cercles` ne fait pas partie de l'état d'un objet, ce ne sera donc pas une *variable d'instance* (instance == objet de la classe).

• Elle dépend plutôt de la classe qui est l'usine fabriquant les objets. Ce sera une **variable de classe**, à laquelle on fera référence (à l'intérieur ou à l'extérieur de la classe) sous le nom `Cercle.cercles` :

```
class Cercle :
    """La classe des Cercles en L1"""
    cercles = []            # une variable de classe

    def __init__(self,centre,rayon=1,couleur='red') :
        self.centre = centre
        self.rayon = rayon
        self.couleur = couleur
        Cercle.cercles.append(self)
    ...
```


Exemple : la classe des rationnels en Python

• Python possède un module `fractions` dont les objets sont des *nombres rationnels*.

```
>>> r = Fraction(6,15)
>>> r
Fraction(2, 5)   # 2/5
>>> r + 3
Fraction(17, 5)
>>> r ** 2
Fraction(4, 25)
>>> r ** 0.5
0.6324555320336759
>>> type(r) == Fraction
True
```

$$r = \frac{6}{15} = \frac{2}{5}$$

$$\frac{2}{5} + 3 = \frac{17}{5}$$

$$\left(\frac{2}{5}\right)^2 = \frac{4}{25}$$

$$\sqrt{\frac{2}{5}} = 0.63\ldots$$

• Comment l'opérateur + peut-il reconnaître les rationnels ? Il a suffi au programmeur du module de prévoir une méthode `__add__` dans la classe `Fraction`. Encore les méthodes spéciales !...

Et si Fraction n'existait pas ?...

```
class Rationnel :
    """Une mini-classe de rationnels en L1"""

    def __init__(self, num, den = 1) :
        if type(num) == int and type(den) == int :
            self.num = num
            self.den = den
        else : raise TypeError('Rationnel(int,int) !')
```

```
>>> r1 = Rationnel(2,-4)
>>> (r1.num, r1.den)
(2, -4)
>>> r2 = Rationnel(5)
>>> (r2.num, r2.den)
(5, 1)
>>> r3 = Rationnel('Daft','Punk')
TypeError: Rationnel(int, int) !
```

• La fraction 2/-4 n'a pas été réduite en -1/2. Il fallait faire monter le signe moins au numérateur, et simplifier par le PGCD.

```
def pgcd(a,b) :
    if b == 0 : return a
    return pgcd(b,a % b)
```

Le théorème d'Euclide !

• Modifions donc le constructeur __init__ :

```
def __init__(self, num, den = 1) :
    if type(num) == int and type(den) == int :
        if den < 0 : (num,den) = (-num,-den)
        g = pgcd(num,den)
        self.num = num // g
        self.den = den // g
    else :
        raise TypeError('Rationnel(int,int) !')
```

```
>>> r1 = Rationnel(2,-4)
```

```
>>> (r1.num, r1.den)
(-1, 2)
```


• Afin de pouvoir afficher un rationnel, j'ajoute la méthode __repr__ :

```
def __repr__(self) :
    if self.den == 1 : return str(self.num)
    if self.num == 0 : return '0'
    return '{}/{}'.format(self.num,self.den)
```

```
>>> r1
-1/2
```

```
>>> r2
5
```

• Il me reste à implémenter les **opérations arithmétiques** et les **comparaisons** sur les rationnels. Je peux le faire de manière classique en inventant des noms de méthodes :

```
def somme(self,r) :        # retourne self + r
```

• Ou bien nommer ces méthodes avec les *noms spéciaux* de Python :

```
def __add__(self,r) :      # retourne self + r
```


• Le langage Python étant **dynamiquement typé**, je ne précise pas le type du paramètre r. Ce peut être un Rationnel, un entier int, etc. Je peux le savoir avec type. On parle alors d'**arithmétique générique** [même opérateur pour divers types numériques].

```
def __add__(self,r) :                    # r est un "nombre"
    if type(r) == Rationnel : return Rationnel(...)
    if type(r) == int :  return Rationnel(...)
    etc.
```

• Le monde n'est pas aussi simple, mais l'idée est là...
Par exemple, on aimerait que 1/2 + 1/2 donne un int et non un Rationnel. Or déjà en Python :

```
>>> Fraction(1,2) + Fraction(1,2)
Fraction(1, 1)
```

• Ces problèmes profonds relèvent de la **conception par objets** qui sera étudiée dans une année ultérieure !...

Complément historique

Wikipedia : Programmation orientée objet

Le langage Simula-67, en implantant les Record Class de Hoare, pose les constructions qui seront celles des langages orientés objet à classes : classe, polymorphisme, héritage, etc. Mais c'est réellement par et avec Smalltalk 71 puis Smalltalk 80 (Dan Ingalls), inspiré en grande partie de Simula 67 et de Lisp, que les principes de la programmation par objets, résultat des travaux d'Alan Kay, sont véhiculés : objet, encapsulation, messages, typage et polymorphisme (via la sous-classification) ; les autres principes, comme l'héritage, sont soit dérivés de ceux-ci ou une implantation. Dans Smalltalk, tout est objet, même les classes. Il est aussi plus qu'un langage à objets, c'est un environnement graphique interactif complet.

À partir des années 1980, commence l'effervescence des langages à objets : Objective C (début des années 1980), C++ (C with classes) en 1983, Eiffel en 1984, Common Lisp Object System dans les années 1980, etc. Les années 1990 voient l'âge d'or de l'extension de la programmation par objet dans les différents secteurs du développement logiciel.

Depuis, la programmation par objet n'a cessé d'évoluer aussi bien dans son aspect théorique que pratique et différents métiers et discours mercatiques à son sujet ont vu le jour :

- l'analyse objet (AOO ou OOA en anglais) ;
- la conception objet (COO ou OOD en anglais) ;
- les bases de données objet (SGBDOO) ;
- les langages objets avec les langages à prototypes ;
- ou encore la méthodologie avec MDA (Model Driven Architecture).

Aujourd'hui, la programmation par objet est vue davantage comme un paradigme, le **paradigme objet**, que comme une simple technique de programmation. C'est pourquoi, lorsque l'on parle de nos jours de programmation par objets, on désigne avant tout la partie codage d'un modèle à objets obtenu par AOO et COO.