



Module Informatique Deuxième Année

Cours1: Programmation orienté objet



AU: 2021/2022

Classe et objet

Introduction

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 (ordonnée) : *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'>
```



Tous les types sont des classes en Python.

Notion de classe

- Une classe modélise un ensemble d'**objets** et leurs **comportements**.
- **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 à ses objets :
 - i) Quelles sont les caractéristiques des objets de la classe ?

Attributs

- ii) Quelles sont les opérations pertinentes sur ces objets ?

Méthodes

Définition d'une classe

- On veut définir la classe des **cercles**, Les cercles seront les objets de cette classe.
- Tous les objets d'une classe ont les mêmes attributs [propriétés].

L'attribut *attr* d'un objet *obj* sera noté **obj.attr** en Python.

cercle1.couleur $\longrightarrow$ la couleur du cercle1

cercle2.centre $\longrightarrow$ le centre du cercle2

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

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

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

La méthode spéciale `__init__` : Constructeur

- Un objet de la classe Cercle aura 3 attributs : centre, rayon et couleur
- 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 :                                # avec C majuscule
    """La classe des Cercles """
    def __init__ ( self,centre,rayon,couleur) :
        self.centre = centre                  # tuple
        self.rayon = rayon                    # int
        self.couleur = couleur                # str
```

Notion d'objet : instance d'une classe

- Pour construire un objet particulier (une instance) de la classe Cercle, on 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 :

- Pour demander l'accès aux attributs de cercle :

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

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

- Pour modifier les attributs de cercle (ils sont dits publics) :

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

Notion d'objet : instance d'une classe

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

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

`__repr__` : méthode spéciale cachée

- 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)>
```

Définitions des méthodes d'instances

- Les méthodes d'instances seront rédigés dans le corps de la classe des objets :
- A quoi vont-elles servir ? A envoyer un message à un objet de la classe.

Par exemple :

o demander à un cercle de bien vouloir calculer et retourner son aire :

cercle.aire() **# avec résultat**

o lui demander de grossir en multipliant son rayon par k :

cercle.zoom(k) **# sans résultat**

NB : *En rédigeant le corps d'une classe, il faut imaginer quels sont les messages que nous aurons à envoyer aux objets, et comment ceux-ci répondront.*

Définir les **fonctionnalités** de la classe ?

Définitions des méthodes d'instances

```
import math

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

Attributs de classes et Attributs d'instances

```
class Compteur:
    """Cette classe possède un attribut de classe qui s'incrémente à chaque
    fois que l'on crée un objet de ce type"""

    objets_crees = 0 # Attribut de classe, le compteur vaut 0 au départ
    def __init__(self, typeC):
        self.typeCompteur=typeC # Attribut d'instance ou objet: Objet.attribut d'objet
        """À chaque fois qu'on crée un objet, on incrémente le compteur"""
        Compteur.objets_crees += 1 # nom de la classe.Attribut de la classe
```

```
c1=Compteur("eau")
c2=Compteur("électrique")
c3=Compteur("électrique")
print(c3.typeCompteur)
print(c1.typeCompteur) # Objet.attribut d'objet
print(Compteur.objets_crees) # nom de la classe.Attribut de la classe
```

Application: Création des classes Pile et File

Pile

Structure:

En informatique, une pile (stack) est une structure de données fondée sur le principe suivant :

« dernier arrivé, premier sorti » (ou **LIFO**) ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.

Fonctionnement:

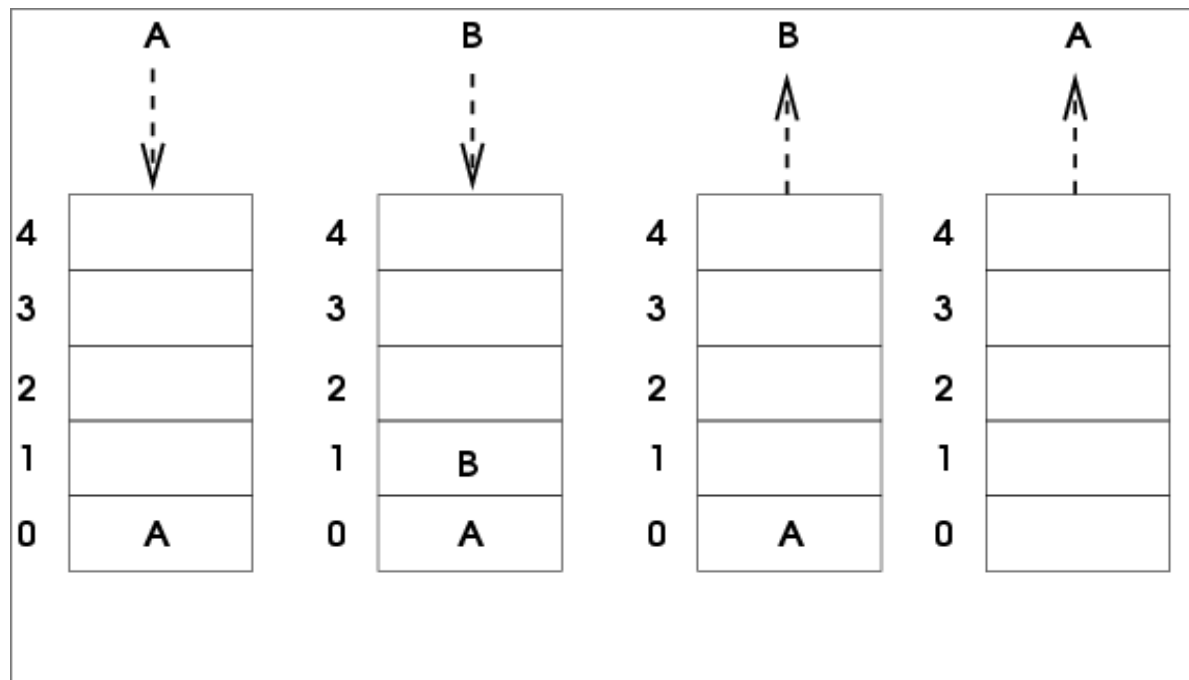
- Le fonctionnement correspond à l'image d'une pile de carte ou d'assiettes posée sur une table .



- On peut accéder qu'au dernier élément ajouté, qu'on appelle le **sommet de la pile**.

Fonctionnement

- Sur la figure suivante :
 - on ajoute A puis B : c'est **empiler** A puis B
 - ensuite on retire dans l'ordre B, puis A : c'est **dépiler** B puis A



Opérations

Trois opérations principales :

- ***créer_pile*** : renvoie une nouvelle pile de capacité c , initialement vide.
- ***empiler*** : ajoute un élément dans la pile : « Push ».
- ***dépiler*** : dépile et renvoie le sommet de la pile : « Pop ».

Les opérations dépiler et empiler modifient le contenu de la pile

D'autres opérations sont disponibles, mais ne modifient pas la pile :

- ***est_pleine*** : indique si la pile est pleine.
- ***est_vide*** : indique si la pile est vide
- ***sommet*** : renvoie le sommet de la pile , sans modifier la pile.

Pile à capacité finie et Pile non bornée

- **Les piles à capacité finie:**

- La manière la plus simple de réaliser une pile consiste à utiliser un tableau de taille N .

0	1	...	n	N
n	éléments			place disponible

- Le premier élément du tableau contient le nombre d'éléments n de la pile.
- Les cases d'indice 1 à n du tableau contiennent alors les éléments de la pile.
- Le sommet de la pile se trouve à l'indice n .

creer_pile de capacité 10

empiler A dans la pile

empiler B dans la pile

depiler de la pile

0	1	2	3	...	10
0	...				
1	A	...			
2	A	B	...		
1	A	...			

- **Les piles non bornées** n'ont pas de limite de taille. Elle n'a pas une case pour indiquer la taille.

La classe: Pile bornée

```
class PileB:
    def __init__(self, N):
        self.T=[0]+[None for i in range(1,N)]
    def __repr__(self) :
        ch ="Pile\n"
        for e in self.T[::-1] :
            ch += '|{}\n\_____/\\n'.format(e)
        return ch
    def est_pleine(self):
        return self.T[-1] is not None

    def est_vide(self):
        return self.T[0] == 0
    def sommet(self):
        try:
            assert not self.est_vide()
            indice_sommet = self.T[0]
            return self.T[indice_sommet]
        except:
            print('pile vide')
```

La classe: Pile bornée

```
def empiler(self, v):
    try:
        assert not self.est_pleine()
        n = self.T[0]+1
        self.T[n]=v
        self.T[0]+=1
    except AssertionError:
        print('la pile est pleine')
def depiler(self):
    try:
        if self.est_vide():
            raise
        n = self.T[0]
        vdep=self.sommet()      # vdep: la valeur à dépiler
        self.T[n]=None
        self.T[0]-=1
        return vdep
    except:
        print('la pile est vide')
```

La classe: Pile non bornée

```
class Pile:
    "classe pile non bornée"
    def __init__(self):
        """initialisation d'une pile vide"""
        self.L = []
    def __repr__(self):
        """Pour l'affichage"""
        ch='Pile \n'
        for i in self.L[::-1]:
            ch+='|{} \n\_____/ \n'.format(i)
        return ch
    def est_vide(self):
        """teste si la pile est vide"""
        return len(self.L) == 0
```

La classe: Pile non bornée

```
def sommet(self):  
    "renvoie le sommet de la pile"  
    assert not self.est_vide()  
    return self.L[-1]  
def depiler(self):  
    try:  
        assert not self.est_vide()  
        return self.L.pop()  
    except:  
        print("la pile est vide")  
def empiler(self, x):  
    "empile x sur la pile"  
    self.L.append(x)
```

Les files

Structure:

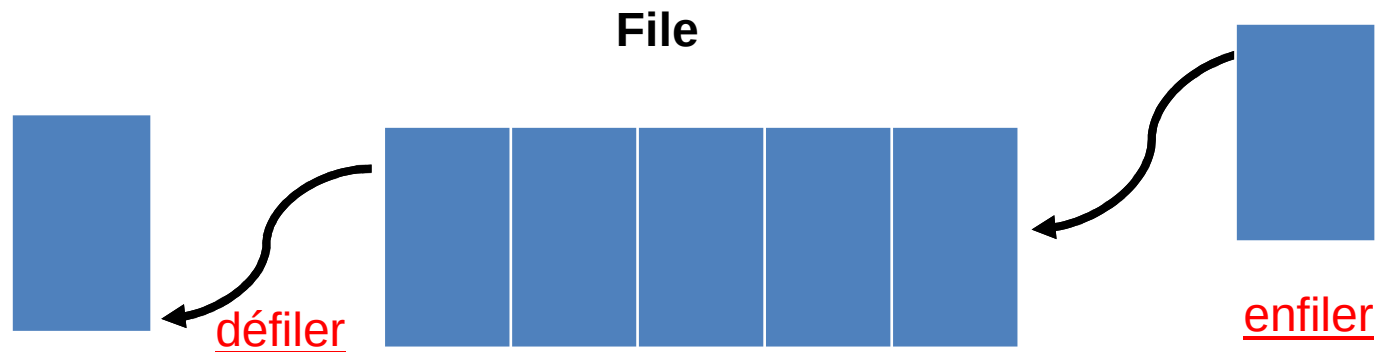
En informatique, une *file* (queue en anglais) est une structure de données basée sur le principe du :

« premier entré, premier sorti »
en anglais **FIFO** « **First In , First Out** »

Ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.

Fonctionnement:

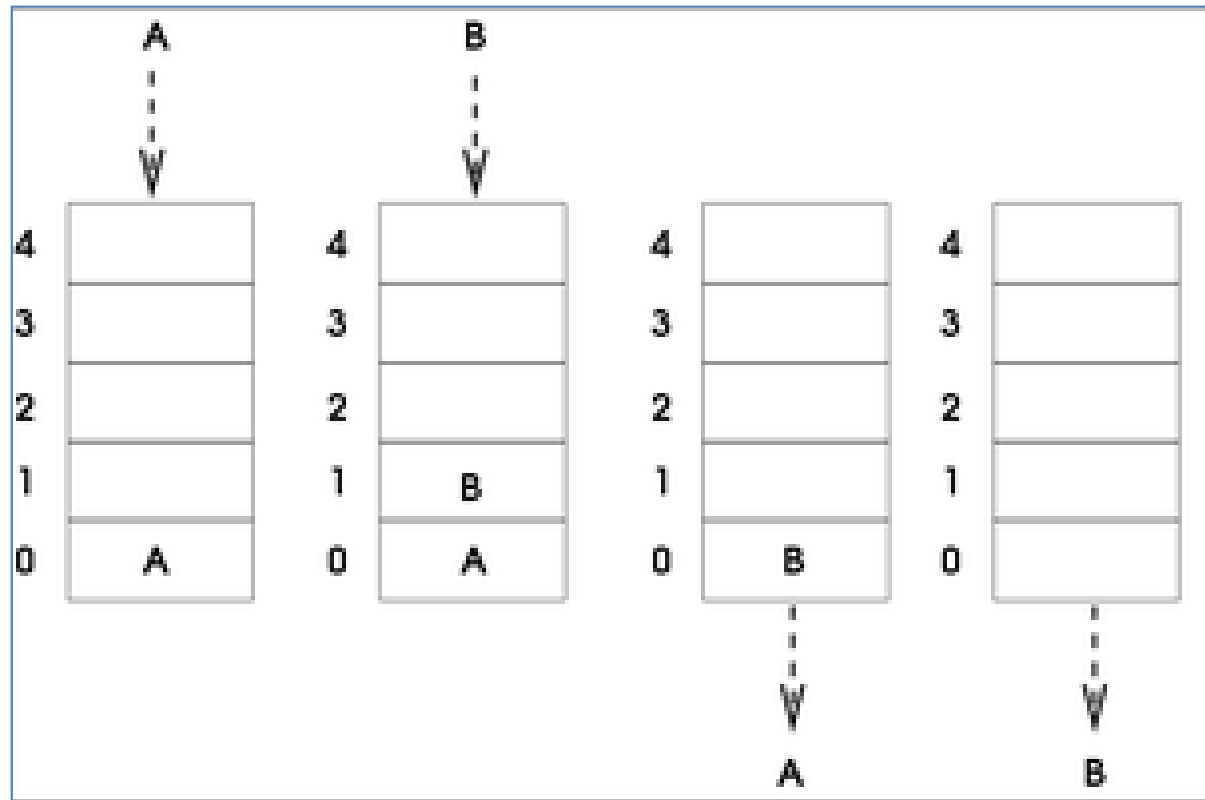
Le fonctionnement ressemble à une file d'attente à la poste : les premières personnes arrivées sont les premières personnes servies.



Fonctionnement

Sur la figure suivante :

- On ajoute **A puis B** : c'est **Enfiler**
- ensuite on retire dans l'ordre A puis B : c'est **Défiler**



Implémentation de la classe File

```
class File:
    def __init__(self):
        """implémentation d'une file"""
        self.L=[]
    def __repr__(self):
        """Pour l'affichage"""
        a_afficher = 'File \n'
        for e in self.L[::-1]:
            a_afficher += '|{} \n'.format(e)
            a_afficher += '\n'
        return a_afficher
    def est_vide(self):
        return len(self.L)==0
    def enfiler(self, element):
        """insérer un element en queue dela file"""
        self.L.append(element)
    def defiler(self):
        """supprimer l'élément de la tete de la file"""
        try:
            assert not self.est_vide()
            self.L.pop(0)
        except :
            print('la file est vide')
```

Héritage
Polymorphisme
Surcharge

Notion d'héritage

- Un avantage de la POO est qu'une classe Python peut toujours être spécialisée en **une classe fille (sous classe)** qui hérite alors de tous les attributs et méthodes de sa classe mère (**super classe**).
- La syntaxe pour la définition de sous classe est :

```
class SousClasse ( SuperClasse ) :
```

```
...
```

- Exemple :

```
class Carre ( Rectangle ) :
```

```
...
```

```
...
```

Notion d'héritage

```
class Rectangle:
    def __init__ ( self, longueur=30, largeur=15) :
        self.L, self.l, self.nom = longueur, largeur, 'rectangle'

class Carre ( Rectangle) :

    pass
```

C1= Carre(10,10)

Dans cet exemple, on voit bien que l'objet **C1** est une instance de la classe **Carré** mais qu'il a conservé les méthodes et attributs de la classe Rectangle sans qu'on ait besoin de les redéfinir:

Héritage et polymorphisme

- Le **polymorphisme** est la faculté pour une méthode portant le même nom mais appartenant à des classes distinctes héritées d'effectuer un travail différent.
- La Dans l'exemple suivant, la classe **Carre** hérite de la classe **Rectangle**, et la méthode **__init__** est polymorphe : on surcharge (redéfinit) la méthode **__init__**.

```
class Rectangle:
    def __init__( self, longueur=30, largeur=15) :
        self.L, self.l, self.nom = longueur, largeur, 'rectangle'

class Carre ( Rectangle) :
    def __init__( self, cote=10) :
        Rectangle.__init__( self, cote, cote)
        self.nom = 'carre'
```

Notion de Conception Orientée Objet

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes de deux façons possibles :

- **la composition** qui repose sur la relation **a-un** ou sur la relation **utilise-un** :
Il s'agit d'une collaboration de plusieurs classes distinctes via une association (utilise-un) ou une agrégation (a-un).
- **la dérivation** qui repose sur la relation **est-un** :
Il s'agit de la création de sous-classes par **spécialisation**.

Bien sûr, ces deux conceptions peuvent cohabiter, et c'est souvent le cas !

Notion de Conception Orientée Objet

Exemple de composition :

L'implémentation Python utilisée est généralement l'instanciation de classes dans le constructeur de la classe composite.

```
class Point:
    def __init__(self, x, y) :
        self.px, self.py = x, y

class Segment:
    "Classe composite utilisant la classe distincte Point"
    def __init__(self, x1, y1, x2, y2) :
        self.orig = Point ( x1, y1)  # Segment "a-un" Point origine,
        self.extrem = Point ( x2, y2)  # et "a-un" Point extrémité

    def __repr__(self) :
        return ( "Segment : [ ( { :g} , { :g} ) , ( { :g} , { :g} ) ]"
                .format ( self.orig.px, self.orig.py, self.extrem.px, self.extrem.py) )
```

Notion de Conception Orientée Objet

Exemple de Dérivation :

L'implémentation Python utilisée est généralement l'appel du constructeur de la super classe dans le constructeur de la sous classe, soit nommément, soit grâce à l'instruction **super**.

```
class Rectangle:
    def __init__(self, longueur=30, largeur=15) :
        self.L, self.l, self.nom = longueur, largeur, 'rectangle'

class Carre ( Rectangle) :
    def __init__(self, cote=10) :
        # appel au constructeur de la super-classe de Carre :
        super() .__init__(cote, cote)
        self.nom = 'carre'
```

Surcharge des méthodes spéciales

Les méthodes spéciales (magiques) portent des noms prédéfinis, précédés et suivis de deux caractères de soulignement.

Elles servent :

- à initialiser l'objet instancié : `__init__`
- à modifier son affichage : `__repr__`
- à redéfinir ses opérateurs :

Opérateur	Méthode spéciale	Utilisation
Opposé	<code>__neg__</code>	<code>- obj1</code>
Addition	<code>__add__</code>	<code>Obj1 + obj2</code>
Soustraction	<code>__sub__</code>	<code>Obj1 - obj2</code>
Multiplication	<code>__mul__</code>	<code>Obj1 * obj2</code>
Division	<code>__div__</code>	<code>Obj1 / obj2</code>
Division entière	<code>__floordiv__</code>	<code>Obj1 // obj2</code>
in	<code>__contains__</code>	<code>Obj1 in Obj2</code>

Méthodes spéciales

- Exemple de surcharge de l'opérateur ' + ' pour les objets Vecteurs :

```
class Vecteur2D:
    def __init__ ( self, x, y ) :
        self.x = x
        self.y = y
    def __add__ ( self, autre ) : # addition vectorielle
        return Vecteur2D ( self.x + autre.x, self.y + autre.y )
    def __repr__ ( self ) : # affichage d'un Vecteur2D
        return "Vecteur ( {} , {} ) ".format ( self.x, self.y )
```

```
>>> v1 = Vecteur2D ( 1.2, 2.3 )
>>> v2 = Vecteur2D ( 3.4, 4.5 )
>>> print ( v1 + v2 )
Vecteur ( 4.6, 6.8 )
```