

# La programmation objet avec Python

# Principe général

- La programmation objet est un paradigme dans lequel :
  - L'idée est de définir des briques logicielles appelées **objets**,
  - Un objet représente un concept, une idée ou toute entité du monde physique,
  - Les objets ont une structure interne (**implémentation**) et un comportement (**interface**).
- Le but de cette méthode :
  - permettre une meilleure modélisation des problèmes, et donc une meilleure programmation,
  - obtenir des briques logiciel facilement réutilisables en dissociant l'implémentation de l'interface (**encapsulation**).

# Langage

- On peut programmer orienté objet dans n'importe quel langage.
- Mais certains ont un support natif de la POO : Smalltalk, Objective-C, Java, C++, Go, PHP, Python, Self, JavaScript, ...
- Ils ne fonctionnent pas tous de manières différentes :
  - soit avec des classes qu'on instancie (C++, Java, Python) ;
  - soit avec des prototypes qu'on clone (Self, JavaScript) ;
  - ou alors avec d'autres choses moins clairement nommées (Go).

# La classe

- Une classe est la déclaration d'un type d'objet.
- Les données internes de la classe appelées **attributs**.
- Les traitements sur ces données et/ou d'autres données sont appelées **méthodes**.
- On parle d'une **instance** d'une classe lors de la création d'un objet qui a cette classe pour type.

# Objet

- Un **objet** est l'**instance** d'un type :
  - `12` est une instance de ***int***
  - `'bonjour'` est une instance ***str***
- Ne pas confondre créer une classe et **instancier** une classe :
  - Créer une classe = définir une nouvelle classe, écrire ses attributs et méthodes.
  - Instancier une classe = créer un exemplaire d'un objet.  
Par exemple `i=1`.

# Créer une classe

- On déclare une classe avec le mot-clé `class` suivi du nom de cette classe :

```
class nom_classe :  
    ■ attributs et méthodes  
    # suite du programme
```

- Les attributs sont des données qui vont être partagées par l'ensemble des instances de la classes

```
class Point :  
    ■ x = 0  
    ■ y = 0  
    # suite du programme
```

- Si x est modifié par une instance de Point, alors elle modifiées pour toutes les autres instances !

# Fonctions - Méthodes

- Les méthodes et les fonctions sont fondamentalement de même nature.
- On les appelle méthodes quand elles sont implémentées dans une classe.
- et fonctions dans le reste du code.
- En python les fonctions sont généralement, comme en mathématiques, préfixes : fonction(argument).
- Et les méthodes sont généralement postfixes : objet.méthode()

# Constructeur

- Lorsqu'une méthode est appelée sur une instance, elle reçoit cette instance en argument.
  - Dans certains langages, le passage de cet argument est entièrement implicite, et son nom est alors le plus souvent le mot-clef *this*.
  - En Python, c'est le premier argument de la méthode et on le nomme **self**, par convention.
- Il existe une méthode spéciale qu'on appelle le constructeur de la classe et qui est chargée de l'**initialisation** des attributs.
  - Parfois c'est la méthode qui a le même nom que la classe.
  - En Python, c'est la méthode qui s'appelle **\_\_init\_\_**.
  - C'est la méthode qui est appelée quand on **instancie** la classe.

# Constructeur

- Le constructeur est une méthode spéciale qui permet d'instancier un objet.

```
class Point :  
    def __init__(self, x, y):  
        print("création d'un objet point")  
        self.x = x  
        self.y = y  
# suite du programme  
A = Point (2, 1)  
print (A.x)
```

- `x` et `y` sont des attributs d'instance de la classe `Point`. Elles reçoivent ici les valeurs passées en paramètres du constructeur.
- Pour offrir plusieurs constructeurs, on donne une valeur par défaut aux variables d'instances
- On interagit avec les attributs d'un objet à l'aide du `.`

# Encapsulation : principe

- Une des forces de la programmation orientée objet est l'**encapsulation**.
- Il s'agit de cacher les détails de l'implémentation d'une classe à ses utilisateurs, et de leur offrir une interface stable.
- Ainsi :
  - l'implémentation de la classe peut changer sans impact sur l'utilisateur,
  - les objets sont seuls responsables du maintien de leur cohérence interne !

# Accesseurs

- Pour cette raison l'interface d'une classe offre généralement des **accesseurs**.
- Un accesseur est une méthode qui permet de lire (généralement nommée *get\_nom\_attribut*) ou écrire (*set\_nom\_attribut*) dans un attribut.
- Cela permet à la classe de contrôler la bonne formation de ses objets, tout en offrant une interface stable aux utilisateurs.

# Héritage

- L'héritage consiste à faire profiter tacitement la classe dérivée B des attributs et des méthodes de la classe de base A.
- Une classe qui hérite d'une autre récupère ses membres et peut en définir de nouveaux ou redéfinir ceux qui sont hérités.
- En Python, une classe peut hériter de plusieurs classes.
- La classe héritante est appelée classe dérivée (ou sous classe).
- Les classes héritées sont appelées classes de base directes (ou super classe).
- Une classe de base d'une classe dérivée est soit une classe de base directe, soit une classe de base directe d'une classe de base.
- On parle d'héritage simple quand il n'y a qu'une seule classe de base directe.
- Sinon, on parle d'héritage multiple.

# Accessibilité des membres

- Pour mettre en œuvre l'encapsulation, la POO propose de restreindre l'accès aux membres d'une classe.
- Il existe trois niveaux de restriction :
  - **publique** : accès libre par tous,
  - **privée** : accès réservé aux méthodes de la classe propriétaire,
  - **protégée** : niveau intermédiaire que l'on verra plus tard en étudiant les notions d'héritage et de hiérarchie de classes.
- En Python ces notions ne sont pas directement supportées, mais il existe une convention :
  - quand son nom commence par `__`, un membre est privé,
  - quand son nom commence par `_`, un membre est protégé,
  - sinon, un membre est public.

# Polymorphisme

- Si une classe B dérive de A, alors l'interface de A est accessible sur les instances de B.
- On peut donc utiliser un objet de type B là où un A est attendu.

# Classe Abstraite - Interface

- Parfois, on veut définir un comportement (i.e., une interface), mais sans spécifier son implémentation, en laissant le soin de le faire aux classes dérivées.
- Dans ce cas on peut définir ce qu'on appelle des interfaces ou classes abstraites .
- Cette notion n'est pas directement supportée en Python non plus, mais il existe une convention : on fait remonter une erreur aux méthodes qui doivent être redéfinies : *raise NotImplementedError()*
- Une classe qui implémente toutes les méthodes abstraites de ses classes parentes est dite **concrète**.