# Classes

UE12 P24 - Python

#### Classes

#### Vous pouvez définir vos propres types d'objets :

```
1 class User:
2   def __init__(self, name: str, age: int):
3        self.name = name
4        self.age = age
5
6 def greet_user(user: User) -> None:
7   print(f'Salut {user.name} !')
8
9 alice = User('Alice', 43)
10 greet_user(alice)
```

Salut Alice !

#### Classes, attributs, méthodes

Une classe a des attributs et des méthodes :

attributes: Bob 22 Hello, my name is Bob!

#### Attributs d'objet et de classe

Un attribut peut être lié au concept (la classe), ou à un exemple de ce concept (un objet, une instance).

```
1 class Message:
2     MAX_LENGTH = 140
3
4     def __init__(self, content: str):
5         self.content = content
6
7 print('A class attribute:', Message.MAX_LENGTH)
8 print('An instance attribute:', Message('Hi!').content)
```

A class attribute: 140
An instance attribute: Hi!

#### Méthodes et fonctions de classe

Il en va de même pour les fonctions. Elles peuvent être liées à l'instance (on parle de méthode) ou à la classe.

```
1 class Message:
2   def get_possible_types():
3     return ['private', 'public']
4
5   def __init__(self, content: str, is_public: bool):
6     self.content = content
7     self.is_public = is_public
8
9   def get_type(self):
10     return 'public' if self.is_public else 'private'
11
12 print('A class function:', Message.get_possible_types())
13 print('A method:', Message('Hi!', False).get_type())
```

```
A class function: ['private', 'public']
A method: private
```

La différence est leur 1er paramètre : self pour une méthode.

#### Classes et conventions

Par convention, un attribut ou une méthode dont le nom commence par un \_ ne doit pas être utilisé hors de la classe.

### Méthodes magiques - 1

Les méthodes qui commencent et terminent par 2 \_ sont automatiquement appelées par Python dans certains cas.

#### Méthodes magiques - 2

Une classe nouvellement créée n'est pas très pratique :

```
1 class Channel:
2   def __init__(self, id: int, name: str):
3        self.id = id
4        self.name = name
5
6 print(Channel(1, 'First channel'))
<_ main__.Channel object at 0x0000022D25CECC20>
```

Heureusement, une méthode magique permet d'améliorer ça :

```
1 class Channel:
2   def __init__(self, id: int, name: str):
3         self.id = id
4         self.name = name
5         def __repr__(self) -> str:
7         return f'Channel(id={self.id}, name={self.name})'
8         print(Channel(1, 'First channel'))
Channel(id=1, name=First channel)
```

### Méthodes magiques - 3

Les méthodes magiques permettent d'implémenter toutes sortent de comportements :

```
1 class Channel:
2   def __init__(self, id: int, name: str):
3        self.id = id
4        self.name = name
5
6 print(Channel(1, 'First channel') == Channel(1, 'Changed name'))
```

False

```
1 class Channel:
2   def __init__(self, id: int, name: str):
3       self.id = id
4       self.name = name
5
6   def __eq__(self, other) -> str:
7       return self.id == other.id
8
9 print(Channel(1, 'First channel') == Channel(1, 'Changed name'))
```

#### Classes - Le minimum



Lorsque vous développez une nouvelle classe, implémentez toujours au moins \_\_init\_\_ et \_\_repr\_\_.

```
1 class Channel:
2   def __init__ (self, id: int, name: str):
3        self.id = id
4        self.name = name
5
6   def __repr__ (self) -> str:
7        return f'Channel(id={self.id}, name={self.name})'
```

#### **Dataclasses**

Le code précédent est un peu répétitif : le nom de chaque attribut apparait 4 fois !

Lorsqu'une classe est essentiellement un conteneur d'attributs, Python nous permet d'aller plus vite avec les dataclasses.

```
1 from dataclasses import dataclass
2
3 @dataclass
4 class Channel:
5    id: int
6    name: str
7
8 print(Channel(1, 'Town square'))
Channel(id=1, name='Town square')
```

## Héritage

Des propriétés partagées entre 2 classes peuvent être mutualisées en une classe mère commune dont les classes filles héritent.

### Héritage

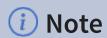
```
class Animal:
  2
        NUMBER OF LEGS: int
  3
        def move(self):
  4
             print(f"I'm moving thanks to my {self.NUMBER OF LEGS} legs!")
  5
    class Cat(Animal):
        NUMBER OF LEGS = 4
        def meow(self):
 8
  9
             print('Meow')
10
11
    class Snake(Animal):
12
        NUMBER OF LEGS = 0
13
14 \text{ cat} = \text{Cat}()
15 cat.move()
I'm moving thanks to my 4 legs!
    cat.meow()
Meow
    Snake().move()
I'm moving thanks to my 0 legs!
```

### Héritage et override

Si la fonction de la classe mère ne nous plaît pas, on peut la redéfinir dans la classe fille :

```
1 from typing import override
2
3 class Animal:
4    NUMBER_OF_LEGS: int
5    def move(self):
6        print(f"I'm moving thanks to my {self.NUMBER_OF_LEGS} legs!")
7 class Bird(Animal):
8    Goverride
9    def move(self):
10        print("I'm flying!")
11
12 Bird().move()
```

I'm flying!



- override n'est disponible que dans les versions récentes de Python.
- son usage n'est pas obligatoire

### Héritage et super

I'm moving thanks to my 2 legs!

I'm flying!

Lorsqu'on redéfinit une méthode dans une classe fille, on peut utiliser la méthode de la classe mère grâce à super :

```
from typing import override
   class Animal:
       NUMBER OF LEGS: int
 4
       def move(self):
 6
            print(f"I'm moving thanks to my {self.NUMBER OF LEGS} legs!")
   class Bird(Animal):
       NUMBER OF LEGS = 2
 8
 9
       @override
10
11
       def move(self):
12
            super().move()
13
            print("I'm flying!")
14
15 Bird().move()
```