Classes

UE12 P24 - Python

Problématique

Vous arrivez sur un projet inconnu et vous devez le modifier.

```python
def print_user(user: dict):
    # TODO
```

Comment savoir quelles clefs le dictionnaire `user` possède ?

- Le seul moyen : chercher tous les appels à la fonction
- Avec un fichier de 100 lignes ça va
- Avec 1 000 fichiers de 1 000 lignes c’est long

La solution : les classes

Si le projet contient des classes :

```python
def print_user(user: User):
    # TODO
```

Il suffit alors d'aller voir la définition de la classe `User` :

```python
class User:
    def __init__(self, index: int, name: str) -> None:
        self.index = index
        self.name = name
```

On va voir ce que signifient ces lignes.

Classes - La base

On définit un nouveau type d'objet avec le mot clef `class` :

```
1 class User: # On définit ce que c'est qu'un User
2     # ...
3
4 user = User() # On crée un utilisateur à partir de la définition de la clas
```

- La classe est le concept, la recette de cuisine
- L'objet (ou l'instance de classe) est le plat que vous mangez

> **Note**
>
> Par convention, le nom d'une classe commence par une majuscule

Classes - Le constructeur

L'appel `User()` fait 2 choses :

- Il crée un objet de type User
- Si elle existe, il appelle la méthode `__init__` de la classe User : on l'appelle constructeur

Classes - Exemple de constructeur

- Le premier argument de `__init__` est toujours `self` : il s'agit de l'objet lui-même
- Lorsque Python appelle `__init__`, il lui transmet automatiquement ce paramètre
  - C'est là qu'on va écrire la recette de cuisine

```
1 class User:
2     def __init__(self, name: str): # Créer un User demande un argument : na
3         self.name = name # Tout objet de la classe User aura une variable n
4
5 alice = User('Alice') # On n'utilise que le paramètre `name` :
6                       # Python s'occupe du paramètre `self`
7 print(alice.name)
```

```
Alice
```

Classes - `.`

On accède aux attributs (et méthodes) d'un objet avec `.`, vous l'avez déjà fait avec `numpy` ou `pandas` :

```python
import numpy as np

matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(matrix.shape) # shape est un attribut de l'objet matrix
```

```
(2, 3)
```

Classes et dictionnaires

En première approche, une classe est comme un dictionnaire qui ne peut contenir que des clefs prédéfinies.

```python
user_dict = {'id': 3, 'name': 'abc'}
print(user_dict['id'], user_dict['name'], user_dict['nam'], user_dict['age'])
# Un dictionnaire peut avoir n'importe quelle clef :
# Votre IDE ne peut pas dire s'il y aura une erreur
# sans lancer le programme

class User:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name

user = User(3, 'abc')
print(user.id, user.name, user.nam, user.age)
# Un objet User n'a pas de variable nam ni age
# Votre IDE détecte l'erreur tout de suite
```

Classes et types

Les classes ainsi créées peuvent être utilisées pour typer vos fonctions :

```python
class User:
    def __init__(self, name: str, age: int) -> None:
        self.name = name
        self.age = age

def greet_user(user: User) -> None:
   print(f'Salut {user.name} !')

alice = User('Alice', 43)
greet_user(alice)
```

```
Salut Alice !
```

Variables de classe et d'instance

Une variable 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) -> None:
5         self.content = content
6
7 print('A class variable:', Message.MAX_LENGTH)
8 print('An instance variable:', Message('Hi!').content)
```

```
A class variable: 140
An instance variable: Hi!
```

Classes, attributs, méthodes

Une classe a des attributs et des méthodes :

```python
class User:
    def __init__(self, name: str, age: int):
        self.name = name                # Un attribut
        self.age = age                  # Un 2e attribut

    def say_hi(self) -> None:           # Une méthode
        print(f'Hello, my name is {self.name}!')

bob = User('Bob', 22)
print('attributes:', bob.name, bob.age)
bob.say_hi()
```

```
attributes: Bob 22
Hello, my name is Bob!
```

Méthodes d’instance et de classe

Comme les variables, les méthodes peuvent être liées à l’instance ou à la classe.

```
1  class User:
2      def __init__(self, name: str) -> None:
3          self.name = name
4      @classmethod
5      def from_dict(cls, user_dict: dict) -> 'User':
6          return cls(user_dict['name'])
7
8  alice = User.from_dict({'id': 34, 'name': 'Alice'})
9  print(alice.name)
```

```
Alice
```

- Pour une méthode de classe, c'est la classe qui est le premier paramètre implicite (`cls`) et non l'objet (`self`)
- `@classmethod` est un décorateur, un mot-clef qui modifie le comportement de la méthode

Méthodes de classe et méthodes statiques

- Contrairement aux variables, il y a 2 types de méthodes liées à la classe :
  - Les méthodes de classe, qui dépendent de la classe
  - Les méthodes statiques, qui ne dépendent de rien du tout

Méthodes statiques

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

```
A static method: ['private', 'public']
An instance method: private
```

Classes et méthodes

```
 1 class ClassWithMethods:
 2     def an_instance_method(arg1, arg2):
 3         print('arg1 is:', arg1)
 4 
 5     @classmethod
 6     def a_class_method(arg1, arg2):
 7         print('arg1 is:', arg1)
 8 
 9     @staticmethod
10     def a_static_method(arg1):
11         print('arg1 is:', arg1)
12 
13 ClassWithMethods().an_instance_method('Custom arg')
14 ClassWithMethods().a_class_method('Custom arg')
15 ClassWithMethods().a_static_method('Custom arg')
```

```
arg1 is: <__main__.ClassWithMethods object at 0x0000026A1864B8C0>
arg1 is: <class '__main__.ClassWithMethods'>
arg1 is: Custom arg
```

Classes et méthodes

> **Note**
>
> Vous pouvez appelez les `staticmethod` et les `classmethod` sur la classe comme sur l'instance

```
1  class Message:
2      @staticmethod
3      def get_possible_types():
4          return ['private', 'public']
5
6  print(Message.get_possible_types())
7  a_message_instance = Message()
8  print(a_message_instance.get_possible_types())
```

```
['private', 'public']
['private', 'public']
```

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.

```python
class Server:
    def __init__(self, users: 'list[dict]', channels: 'list[dict]'):
        self._users = users      # Cette variable est "privée"
        self.channels = channels

    def get_users(self):
        return self._users
```

L'idée est de cacher l'implémentation à l'utilisateur de la classe : il n'a pas besoin de savoir si `get_users` utilise une variable interne ou un appel réseau.

Méthodes magiques - 1

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

```python
class Channel:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name
```

Vous n’appellerez jamais `__init__` explicitement.

Méthodes magiques - 2

Une classe nouvellement créée n'est pas très pratique :

```python
class Channel:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name

print(Channel(1, 'First channel'))
```

```
<__main__.Channel object at 0x0000026A1864BA10>
```

Heureusement, une méthode magique permet d'améliorer ça :

```python
class Channel:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name

    def __repr__(self) -> str:
        return f'Channel(id={self.id}, name={self.name})'

print(Channel(1, 'First channel'))
```

```
Channel(id=1, name=First channel)
```

Méthodes magiques - 3

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

```python
class Channel:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name

print(Channel(1, 'First channel') == Channel(1, 'Changed name'))
```

```
False
```

```python
class Channel:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name

    def __eq__(self, other: Channel) -> bool:
        return self.id == other.id

print(Channel(1, 'First channel') == Channel(1, 'Changed name'))
```

```
True
```

Classes - Le minimum

> **Tip**
>
> Lorsque vous développez une nouvelle classe, implémentez toujours au moins `__init__` et `__repr__`.

```python
class Channel:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name

    def __repr__(self) -> str:
        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`.

```python
from dataclasses import dataclass

@dataclass
class Channel:
    id: int
    name: str

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

```python
class Animal:
    NUMBER_OF_LEGS: int
    def move(self) -> None:
        print(f"I'm moving thanks to my {self.NUMBER_OF_LEGS} legs!")

class Cat(Animal):
    NUMBER_OF_LEGS = 4
    def meow(self) -> None:
        print('Meow')

class Snake(Animal):
    NUMBER_OF_LEGS = 0

cat = Cat()
cat.move()
```

```
I'm moving thanks to my 4 legs!
```

```python
cat.meow()
```

```
Meow
```

```python
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 :

```python
from typing import override

class Animal:
    NUMBER_OF_LEGS: int

    def move(self) -> None:
        print(f"I'm moving thanks to my {self.NUMBER_OF_LEGS} legs!")

class Bird(Animal):
    @override
    def move(self) -> None:
        print("I'm flying!")

Bird().move()
```

```
I'm flying!
```

Héritage et override

Note

- `override` n'est disponible que dans les versions récentes de Python (>= 3.12)
- son usage n'est pas obligatoire
- mais il permet de documenter son intention, en montrant qu'on a pas redéfini la méthode par erreur

Héritage et `super`

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

```python
from typing import override

class Animal:
    NUMBER_OF_LEGS: int
    def move(self) -> None:
        print(f"I'm moving thanks to my {self.NUMBER_OF_LEGS} legs!")
class Bird(Animal):
    NUMBER_OF_LEGS = 2

    @override
    def move(self) -> None:
        super().move()
        print("I'm flying!")

Bird().move()
```

```
I'm moving thanks to my 2 legs!
I'm flying!
```

Références

https://docs.python.org/3/tutorial/classes.html