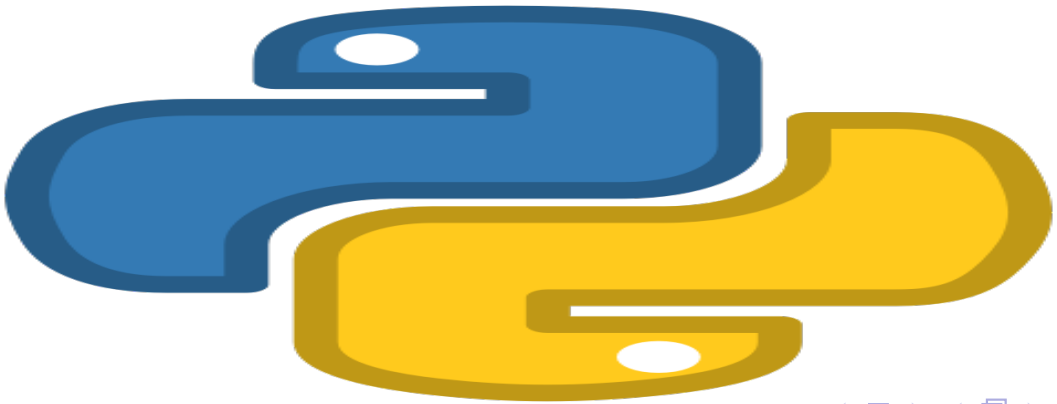
Python : programmation orientée-objet

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com



Plan

1. Introduction
2. Classe
   - `__str__()`
   - `__repr__()`
   - `__init__()`
   - Valeurs par défaut
   - Setter
   - Getter
3. Propriétés
   - `property()`
   - Deleter
   - `@property`
   - `__del__()`

Plan

4. Attribut et méthode statiques
   - Attribut statique
   - `@staticmethod`
   - `@classmethod`
   - `__new__()`
5. Association entre classes
   - Association unidirectionnelle
   - Association bidirectionnelle
6. Héritage
   - Héritage simple
   - Héritage multiple
   - MRO

Plan

7. Polymorphisme
   - Surcharge
   - Redéfinition
8. Classe et méthode abstraites
9. Indexeur
   - `__getitem__()`
   - `__setitem__()`
   - `__len__()`
10. Itérateur

Plan

11 Opérateur
- `__eq__()`
- `__gt__()`
- `__iadd__()`

12 Généricité

13 Méta-classe

14 Functor

15 Énumération

16 Introspection

17 Conventions

Python

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

© Achref EL M

Python

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Achref EL M

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)

Python

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Achref EL M

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe

Python

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

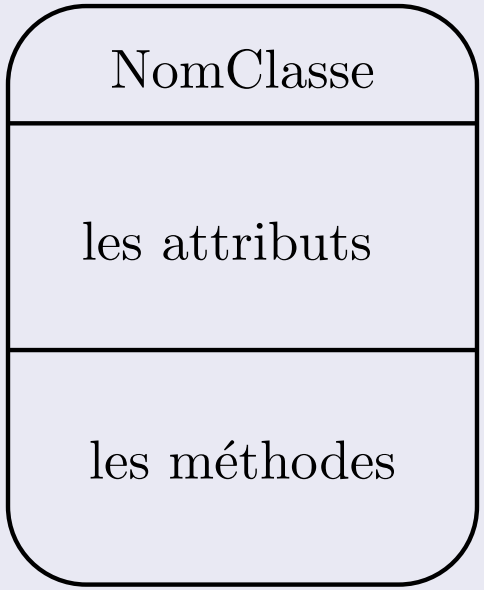
Achref EL M

Qu'est ce que c'est la notion d'instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe
- instance $\equiv$ objet

Python

De quoi est composé une classe ?



- Attribut : [visibilité] + type + nom
- Méthode : [visibilité] + valeur de retour + nom + arguments ≡ signature : exactement comme les fonctions en procédurale

Python

Particularité de **Python**

- Toutes les classes héritent implicitement d'une classe mère `object`.
- Le mot-clé `self` permet de désigner l'objet courant.

© Achr

Python

Méthodes magiques en **Python**

- Appelées aussi dunder methods, pour **d**ouble **under**score **methods**
- Méthodes ayant des noms entourés de doubles underscores __
- Permettant de définir ou de surcharger le comportement des objets comme opérations de
  - construction et de destruction d'objets
  - représentation
  - arithmétiques et de comparaison
  - ...

Python

Avant de commencer

- Créez un nouveau projet `cours_poo` dans votre espace de travail
- Créez un fichier `main.py`

© Ach

Python

Considérons la classe `Personne` définie dans `personne.py`

```
class Personne:
    pass
```

© Achref EL MOUELHI ©

Python

Considérons la classe `Personne` définie dans `personne.py`

```
class Personne:
    pass
```

Dans `main.py`, instancions la classe `Personne`

```
p = Personne()
```

© Achref EL MOUATEZ HI ©

Python

Considérons la classe `Personne` définie dans `personne.py`

```
class Personne:
    pass
```

Dans `main.py`, instancions la classe `Personne`

```
p = Personne()
```

N'oublions d'importer la classe `Personne`

```
from personne import Personne
```

© Achref EL MOUELHI ©

Python

Considérons la classe `Personne` définie dans `personne.py`

```
class Personne:
    pass
```

Dans `main.py`, instancions la classe `Personne`

```
p = Personne()
```

© Achref EL MOUELHI ©

N'oublions d'importer la classe `Personne`

```
from personne import Personne
```

Et si on affiche l'instance

```
print(p)
# affiche <personne.Personne object at 0x00D28160>
```

Dans une classe, on peut déclarer des attributs (et les initialiser)

```
class Personne:
    num: int
    nom: str
    prenom: str
```

© Achref EL MOUELHI ©

Dans une classe, on peut déclarer des attributs (et les initialiser)

```
class Personne:
    num: int
    nom: str
    prenom: str
```

Pour affecter des valeurs aux différents attributs

```
from personne import Personne

p = Personne()
p.num = 100
p.nom = 'wick'
p.prenom = "john"
```

HELHI ©

Dans une classe, on peut déclarer des attributs (et les initialiser)

```
class Personne:
    num: int
    nom: str
    prenom: str
```

Pour affecter des valeurs aux différents attributs

```
from personne import Personne

p = Personne()
p.num = 100
p.nom = 'wick'
p.prenom = "john"
```

Pour afficher la valeur d'un attribut

```
print(p.nom)
# affiche wick
```

HELHI ©

Python

Explication

Pour afficher les détails d'un objet, il faut que la méthode `__str__(self)` soit implémentée

© Achref EL MOUELHI ©

Python

Explication

Pour afficher les détails d'un objet, il faut que la méthode `__str__(self)` soit implémentée

HI ©

Définissons la méthode `__str__()`

```
class Personne:
    num: int
    nom: str
    prenom: str

    def __str__(self) -> str:
        return self.prenom + " " + self.nom
```

Python

Explication

Pour afficher les détails d'un objet, il faut que la méthode `__str__(self)` soit implémentée

Définissons la méthode `__str__()`

```
class Personne:
    num: int
    nom: str
    prenom: str

    def __str__(self) -> str:
        return self.prenom + " " + self.nom
```

Dans `main.py`, affichons les détails de notre instance `p`

```
print(p)
# affiche john wick
```

Python

Explication

- En **Python**, toutes les méthodes d'une classe, à l'exception des méthodes statiques, doivent déclarer un paramètre `self` comme premier paramètre.
- `self` est une référence à l'instance de l'objet sur lequel la méthode est appelée. Il permet d'accéder aux attributs et aux autres méthodes de l'instance.
- Le mot `self` n'est pas obligatoire mais c'est une convention fortement recommandée et largement adoptée.

Python

Remplaçons la méthode `__str__()` par `__repr__()`

```
class Personne:
    num: int
    nom: str
    prenom: str

    def __repr__(self) -> str:
        return self.prenom + " " + self.nom
```

© Acm

Python

Remplaçons la méthode `__str__()` par `__repr__()`

```
class Personne:
    num: int
    nom: str
    prenom: str

    def __repr__(self) -> str:
        return self.prenom + " " + self.nom
```

© Acm

Exécutons le `main.py` précédent sans le changer

```
print(p)
# affiche john wick
```

Python

Ajoutons la méthode `__str__()`

```
class Personne:
    num: int
    nom: str
    prenom: str

    def __str__(self) -> str:
        return "prénom : " + self.prenom + " " + "nom : " + self.nom

    def __repr__(self) -> str:
        return self.prenom + " " + self.nom
```

© ACH

Python

Ajoutons la méthode `__str__()`

```
class Personne:
    num: int
    nom: str
    prenom: str

    def __str__(self) -> str:
        return "prénom : " + self.prenom + " " + "nom : " + self.nom

    def __repr__(self) -> str:
        return self.prenom + " " + self.nom
```

© ACH

Exécutons le `main.py` précédent sans le changer

```
print(p)
# affiche prénom : john nom : wick
```

Python

Testons avec les chaînes formatées

```
from personne import Personne

p = Personne()
p.num = 100
p.nom = 'wick'
p.prenom = "john"

print(f"{p}")
# prénom : john nom : wick

print(f"{p!r}")
# john wick

print(str(p))
# prénom : john nom : wick

print(repr(p))
# john wick
```

Python

Convention sur `__str__()` et `__repr__()`

- Utilisez `__str__()` pour un affichage client
- Utilisez `__repr__()` pour le débogage (phase de développement)
- Plus de détails dans la documentation officielle : `https://docs.python.org/3/reference/datamodel.html#object.__repr__`

Python

Remarques

- Par défaut, toute classe en **Python** a un constructeur par défaut sans paramètre
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe

© Achref EL M

Python

Remarques

- Par défaut, toute classe en **Python** a un constructeur par défaut sans paramètre
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe

Achref EL M

Les constructeurs avec **Python**

- On le déclare avec le mot-clé `__init__()`
- Il peut contenir la visibilité des attributs si on veut simplifier la déclaration

Le constructeur de la classe `Personne` acceptant trois paramètres

```
class Personne:
    def __init__(self, num: int, nom: str, prenom: str):
        self.num = num
        self.nom = nom
        self.prenom = prenom
    def __str__(self) -> str:
        return self.prenom + " " + self.nom
```

© Achref EL MOUELHI

Le constructeur de la classe `Personne` acceptant trois paramètres

```
class Personne:
    def __init__(self, num: int, nom: str, prenom: str):
        self.num = num
        self.nom = nom
        self.prenom = prenom
    def __str__(self) -> str:
        return self.prenom + " " + self.nom
```

En testant le `main` précédent, une erreur sera générée

```
from personne import Personne

p = Personne()
p.num = 100
p.nom = 'wick'
p.prenom = "john"

print(p)
# affiche TypeError: __init__() missing 3 required positional arguments
   : 'num', 'nom', and 'prenom'
```

Python

Explication

Le constructeur par défaut a été écrasé (il n'existe plus)

© Achref EL MOUELHI

Python

Explication

Le constructeur par défaut a été écrasé (il n'existe plus)

Comment faire ?

- **Python** n'autorise pas la présence de plusieurs constructeurs (la surcharge)
- On peut utiliser les valeurs par défaut

Python

Le nouveau constructeur avec les valeurs par défaut

```
class Personne:

    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        self.num = num
        self.nom = nom
        self.prenom = prenom

    def __str__(self) -> str:
        return self.prenom + " " + self.nom
```

Python

Pour tester les deux constructeurs

```
from personne import Personne

p = Personne()
p.num = 100
p.nom = 'wick'
p.prenom = "john"

p2 = Personne(100, 'wick', 'john')

print(p2)
# affiche john wick
```

Python

Type de donnée	Valeur par défaut
`bool`	`False`
`int`	`0`
`float`	`0.0`
`str`	`""` (chaîne vide)
`list`	`[]` (liste vide)
`tuple`	`()` (tuple vide)
`dict`	{} (dictionnaire vide)
`set`	`set()` (ensemble vide)
Objet personnalisé	`None` si non initialisé

© Achref EL MOUELHI ©

Python

Attention

- Utiliser `[]` ou `{}` directement comme valeur par défaut peut entraîner des comportements inattendus.
- En effet, l'initialisation est évaluée une seule fois lors de la définition de la classe.
- Toutes les instances utilisent donc la même liste en mémoire.
- Pour éviter ce comportement indésirable, utiliser `None` et affecter la valeur mutable dans le constructeur.

Python

Exemple

```
class Person:
    def __init__(self, name="", hobbies=[]):
        self.name = name
        self.hobbies = hobbies
        # Liste mutable partagée entre toutes les instances

p1 = Person("Wick")
p2 = Person("Dalton")

# ajout d'un hobby à p1
p1.hobbies.append("Lecture")

# vérifions les hobbies des deux instances
print(p1.hobbies)  # ['Lecture']
print(p2.hobbies)  # ['Lecture'] => La liste est partagée
```

Python

Exemple

```
class Person:
    def __init__(self, name="", hobbies=None):
        self.name = name
        self.hobbies = hobbies if hobbies is not None else []
        # Nouvelle liste à chaque instance


p1 = Person("Wick")
p2 = Person("Dalton")

# ajout d'un hobby à p1
p1.hobbies.append("Lecture")

# vérifions les hobbies des deux instances
print(p1.hobbies)  # ['Lecture']
print(p2.hobbies)  # [''] => Chaque instance a sa propre liste
```

Python

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

© Achref EL MOUELHI

Python

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

Démarche

1. Bloquer l'accès direct aux attributs (mettre la visibilité à `private`)
2. Définir des méthodes qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

Python

Particularité de **Python**

- Le mot-clé `private` n'existe pas
- On préfixe les attributs par deux underscores pour indiquer qu'ils sont privés
- On préfixe les attributs par un underscore pour indiquer qu'ils sont protégés

Ajoutons deux underscrores à tous les attributs et définissons un setter pour chaque attribut

```
class Personne:

    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        self.__num = num
        self.__nom = nom
        self.__prenom = prenom

    def set_num(self, num: int) -> None:
        if num > 0:
            self.__num = num
        else:
            self.__num = 0

    def set_nom(self, nom: str) -> None:
        self.__nom = nom

    def set_prenom(self, prenom: str) -> None:
        self.__prenom = prenom

    def __str__(self) -> str:
        return str(self.__num) + " " + self.__prenom + " " + self.__nom
```

Python

Pour tester dans `main.py`

```
from personne import Personne

p = Personne(100, 'wick', 'john')

print(p)
# affiche 100 john wick

p.set_num(-100)

print(p)
# affiche 0 john wick
```

Python

Cependant, le constructeur accepte toujours les valeurs négatives

```
from personne import Personne

p = Personne(-100, 'wick', 'john')

print(p)
# affiche -100 john wick

p.set_num(-100)

print(p)
# affiche 0 john wick
```

Pour résoudre le problème précédent, on appelle le setter dans le constructeur

```
class Personne:

    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        self.set_num(num)
        self.__nom = nom
        self.__prenom = prenom

    def set_num(self, num: int) -> None:
        if num > 0:
            self.__num = num
        else:
             self.__num = 0

    def set_nom(self, nom: str) -> None:
        self.__nom = nom

    def set_prenom(self, prenom: str) -> None:
        self.__prenom = prenom

    def __str__(self) -> str:
        return str(self.__num) + " " + self.__prenom + " " + self.__nom
```

Python

Les valeurs négatives ne passent plus par le constructeur ni par le setter

```
from personne import Personne

p = Personne(-100, 'wick', 'john')

print(p)
# affiche 0 john wick

p.set_num(-100)

print(p)
# affiche 0 john wick
```

Python

Question

Comment récupérer les attributs (privés) de la classe `Personne` ?

© Achref EL MOU

Python

Question

Comment récupérer les attributs (privés) de la classe `Personne` ?

Démarche

Définir des méthodes qui retournent les valeurs des attributs (les `getter`)

Ajoutons les getters dans la classe `Personne`

```
class Personne:

    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        self.set_num(num)
        self.__nom = nom
        self.__prenom = prenom

    def get_num(self) -> int:
        return self.__num

    def set_num(self, num: int) -> None:
        if num > 0:
           self.__num = num
        else:
            self.__num = 0

    def get_nom(self) -> str:
        return self.__nom

    def set_nom(self, nom: str) -> None:
        self.__nom = nom

    def get_prenom(self) -> str:
        return self.__prenom

    def set_prenom(self, prenom: str) -> None:
        self.__prenom = prenom

    def __str__(self) -> str:
        return str(self.__num) + " " + self.__prenom + " " + self.__nom
```

Python

Pour tester

```
from personne import Personne

p = Personne(100, 'wick', 'john')

print(p.get_num(), p.get_nom(), p.get_prenom())
# affiche 100 wick john
```

Python

La méthode `property()` permet

- d'indiquer les setter et getter
- de les utiliser comme un attribut

© ACH

Python

Ajoutons `property` dans la classe `Personne`

```
class Personne:
    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        self.num = num
        self.__nom = nom
        self.__prenom = prenom

    def get_num(self) -> int:
        return self.__num
    def set_num(self, num: int) -> None:
        if num > 0:
            self.__num = num
        else:
            self.__num = 0
    num = property(get_num, set_num)

    def get_nom(self) -> str:
        return self.__nom
    def set_nom(self, nom: str) -> None:
        self.__nom = nom
    nom = property(get_nom, set_nom)

    def get_prenom(self) -> str:
        return self.__prenom
    def set_prenom(self, prenom: str) -> None:
        self.__prenom = prenom
    prenom = property(get_prenom, set_prenom)

    def __str__(self) -> str:
        return str(self.__num) + " " + self.__prenom + " " + self.__nom
```

Python

Pour tester

```
from personne import Personne
p = Personne(100, 'wick', 'john')

print(p)
# affiche 100 john wick

p.num = -100

print(p.num, p.nom, p.prenom)
# affiche 0 wick john
```

Python

Le deleter permet

- de supprimer un attribut d'un objet
- de ne plus avoir accès à un attribut

© ACH

Python

Définissons le deleter pour l'attribut `prenom` de la classe `Personne`

```
    def get_prenom(self) -> str:
        return self.__prenom

    def set_prenom(self, prenom: str) -> None:
        self.__prenom = prenom

    def del_prenom(self) -> None:
        del self.__prenom

    prenom = property(get_prenom, set_prenom, del_prenom)
```

Python

Pour tester

```
from personne import Personne

p = Personne(100, 'wick', 'john')

print(p)
# affiche 100 wick john

del p.prenom

print(p)
# affiche AttributeError: 'Personne' object has no attribute '__prenom'
```

Python

Le décorateur (annotation) `property()` permet

- de simplifier la déclaration des getters et setters
- de ne pas déclarer les getter et setter avec dans `property()`

© AC

Python

Pour générer les propriétés avec **PyCharm**

- saisissez `prop`
- choisissez si vous voulez le getter, getter et setter, getter, setter et deleter
- validez en cliquant sur `entrée`

© Achref EL MOUE

Python

Pour générer les propriétés avec **PyCharm**

- saisissez `prop`
- choisissez si vous voulez le getter, getter et setter, getter, setter et deleter
- validez en cliquant sur `entrée`

Pour générer les propriétés avec **VSC**

- installez l'extension **Python Getter Setter**
- sélectionnez les attributs
- cliquez sur `F1`
- saisissez `Generate Getter Setter for Python` et enfin cliquez sur `ENTREE`

Modifions la classe `Personne` et utilisons les décorateurs

```
class Personne:

    # le code précédent

    @property
    def num(self) -> int:
        return self.__num
    @num.setter
    def num(self, num) -> None:
        if num > 0:
            self.__num = num
        else:
            self.__num = 0

    @property
    def nom(self) -> str:
        return self.__nom
    @nom.setter
    def nom(self, nom) -> None:
        self.__nom = nom

    @property
    def prenom(self) -> str:
        return self.__prenom
    @prenom.setter
    def prenom(self, prenom) -> None:
        self.__prenom = prenom
    @prenom.deleter
    def prenom(self) -> str:
        del self.__prenom
```

Python

Pour tester

```
from personne import Personne

p = Personne(100, 'wick', 'john')

print(p)
# affiche 100 john wick

p.num = -100

print(p.num, p.nom, p.prenom)
# affiche 0 wick john
```

Python

__del__()

- destructeur : exécuté à la destruction de l'objet
- peut être implicitement (lorsque l'objet n'est plus référencé) ou explicitement avec le mot clé `del`

Python

Ajoutons le destructeur dans la classe `Personne`

```
class Personne:

    def __init__(self,  num: int = 0, nom: str = '', prenom:
       str = ''):

        self.num = num
        self._nom = nom
        self._prenom = prenom

    def __del__(self):
        print("destructeur appelé")

# + le contenu précédent
```

Python

Pour tester dans le `main`

```
from personne import Personne

p = Personne(100, 'wick', 'john')

print(p)
# affiche 100 john wick

del p
# affiche destructeur appelé
```

Python

Le destructeur sera appelé implicitement si l'objet n'est plus référencé

```
from personne import Personne

p = Personne(100, 'wick', 'john')

print(p)
# affiche 100 john wick

print("fin du programme")
# affiche fin du programme
# destructeur appelé
```

Python

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs/propriétés mais pas les mêmes valeurs

© Achref EL MOUELHI ©

Python

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs/propriétés mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (par exemple, le nombre d'objets instanciés de la classe `Personne`)

© H. OUELHI ©

Python

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs/propriétés mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (par exemple, le nombre d'objets instanciés de la classe `Personne`)

Solution : attribut statique ou attribut de classe

Un attribut dont la valeur est partagée par toutes les instances de la classe.

Python

Exemple

- Si on voulait créer un attribut contenant le nombre d'objets créés à partir de la classe `Personne`
- Notre attribut doit être déclaré `static`, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut

© Achref EL

Python

Exemple

- Si on voulait créer un attribut contenant le nombre d'objets créés à partir de la classe `Personne`
- Notre attribut doit être déclaré `static`, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut

© Achref EL

Particularité de **Python**

- En **Python**, un attribut qui n'est pas déclaré dans le constructeur est un attribut static
- Pas de mot-clé `static` comme dans la plupart des LOO

Python

Ajoutons un attribut statique `nbr_personnes` à la liste d'attributs de la classe `Personne`

```
nbr_personnes = 0
```

© Achref EL MOUELHI ©

Python

Ajoutons un attribut statique `nbr_personnes` à la liste d'attributs de la classe `Personne`

```
nbr_personnes = 0
```

Incrémentons notre compteur de personnes dans le constructeur

```
    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        self.num = num
        self.__nom = nom
        self.__prenom = prenom
        Personne.nbr_personnes += 1
```

© AU... HI ©

Python

Ajoutons un attribut statique `nbr_personnes` à la liste d'attributs de la classe `Personne`

```
nbr_personnes = 0
```

Incrémentons notre compteur de personnes dans le constructeur

```
    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        self.num = num
        self.__nom = nom
        self.__prenom = prenom
        Personne.nbr_personnes += 1
```

© AC HI ©

Décrémentons dans le destructeur

```
    def __del__(self):
        Personne.nbr_personnes -= 1
        print("destructeur appelé")
```

Python

Testons cela dans `main.py`

```
from personne import Personne

print(Personne.nbr_personnes)
# affiche 0

p = Personne(100, 'wick', 'john')

print(Personne.nbr_personnes)
# affiche 1
```

Python

Pour définir une méthode statique, il faut

- omettre `self` : une méthode définie sans `self` serait simplement une fonction qui ne peut pas accéder ni modifier les données de l'instance ou de la classe sans être explicitement passée une référence.
- utiliser
  - soit le décorateur `@staticmethod`,
  - soit le décorateur `@classmethod`.

Python

Remarques

- `@staticmethod` et `@classmethod` sont nécessaires pour indiquer explicitement à **Python** (et aux développeurs) qu'il s'agit d'une méthode statique ou une méthode de classe.
- Sans l'un de ces décorateurs, une fonction définie dans une classe est considérée par défaut comme une méthode d'instance, qui attend `self` comme premier argument.
- En omettant `self`, **Python** lèvera une erreur car il tentera de passer l'instance comme premier argument à une méthode qui n'est pas conçue pour le recevoir.

Python

Définissons une méthode statique avec le décorateur (@staticmethod) pour incrémenter `nbr_personnes`

```
@staticmethod
def increment() -> None:
    Personne.nbr_personnes += 1
```

© Achref EL MOUEL

Python

Définissons une méthode statique avec le décorateur (@staticmethod) pour incrémenter `nbr_personnes`

```
@staticmethod
def increment() -> None:
    Personne.nbr_personnes += 1
```

bref EL MOUEL

Et l'utiliser dans le constructeur

```
def __init__(self, num: int = 0, nom: str = '', prenom:
   str = ''):
    self.num = num
    self.__nom = nom
    self.__prenom = prenom
    Personne.increment()
```

Python

Testons cela dans `main.py`

```
from personne import Personne

print(Personne.nbr_personnes)
# affiche 0

p = Personne(100, 'wick', 'john')

print(Personne.nbr_personnes)
# affiche 1
```

Python

On peut aussi définir une méthode statique avec le décorateur (@classmethod) et pour incrémenter `nbr_personnes` on injecte `cls`

```
@classmethod
def increment(cls) -> None:
    cls.nbr_personnes += 1
```

© Achref EL MOUEL

Python

On peut aussi définir une méthode statique avec le décorateur (@classmethod) et pour incrémenter `nbr_personnes` on injecte `cls`

```
@classmethod
def increment(cls) -> None:
    cls.nbr_personnes += 1
```

Rien à modifier dans le constructeur

```
def __init__(self, num: int = 0, nom: str = '', prenom:
   str = ''):
    self.num = num
    self.__nom = nom
    self.__prenom = prenom
    Personne.increment()
```

Python

Pour tester, rien à modifier dans `main.py`

```
from personne import Personne

print(Personne.nbr_personnes)
# affiche 0

p = Personne(100, 'wick', 'john')

print(Personne.nbr_personnes)
# affiche 1
```

Python

Pour conclure

- Une méthode d'instance reçoit le mot-clé `self` comme premier paramètre.
- Une méthode de classe reçoit le mot-clé `cls` comme premier paramètre.
- Une méthode static ne reçoit ni `self` ni `cls` comme premier paramètre.

Python

__new__()

- Méthode statique
- Appelée avant `__init__()`
- Utilisés pour les modèles de conception comme les singletons

Python

__new__() vs __init__()

- __new__()
  - responsable de la création d'une nouvelle instance d'une classe
  - retourne l'instance de la classe
- __init__()
  - responsable de l'initialisation d'une instance d'une classe
  - ne retourne rien (None)

Python

Exemple avec `__new__()` et `__init__()`

```
from typing import Self

class Personne:
    nbr_personnes = 0

    def __new__(cls, *args) -> Self:
        print('__new__', args)
        return super().__new__(cls)

    def __init__(self, num: int = 0, nom: str = '', prenom: str = ''):
        print('__init__')
        self.num = num
        self.__nom = nom
        self.__prenom = prenom
        Personne.increment()

    # + le reste du code
```

Python

__new__()

- prend comme premier paramètre `cls` (parce qu'elle est statique).
- prend comme deuxième paramètre `*args` si le constructeur serait appelé avec des paramètres.
- retourne l'instance sinon `__init__()` ne sera jamais exécutée.

Python

Exercice 1

- Définissez une classe `Adresse` avec trois attributs privés `rue`, `code_postal` et `ville` de type chaîne de caractère
- Définissez un constructeur avec trois paramètres, les getters et setters
- Dans la classe `Personne`, ajouter un attribut `adresse` (de type `Optional[Adresse]`) dans le constructeur : `Optional[Adresse]` indique que `adresse` peut être soit de type `Adresse` soit `None`
- Générez les getter et setter de ce nouvel attribut
- Dans `main.py`, créez deux objets : un objet `personne` (de type `Personne`) et `adresse` (de type `Adresse`) et affectez le à l'objet `personne`
- Affichez tous les attributs de l'objet `personne`

Python

Exercice 1

- Définissez une classe `Adresse` avec trois attributs privés `rue`, `code_postal` et `ville` de type chaîne de caractère
- Définissez un constructeur avec trois paramètres, les getters et setters
- Dans la classe `Personne`, ajouter un attribut `adresse` (de type `Optional[Adresse]`) dans le constructeur : `Optional[Adresse]` indique que `adresse` peut être soit de type `Adresse` soit `None`
- Générez les getter et setter de ce nouvel attribut
- Dans `main.py`, créez deux objets : un objet `personne` (de type `Personne`) et `adresse` (de type `Adresse`) et affectez le à l'objet `personne`
- Affichez tous les attributs de l'objet `personne`

N'oublions pas d'importer `Optional`

```
from typing import Optional
```

Python

Exercice 2

- Transformez l'association entre `Personne` et `Adresse` en association bidirectionnelle
- Pour éviter les imports croisés (Circular imports), n'importez pas `Personne` dans `Adresse` ni `Adresse` dans `Personne`
- Importez `Personne` et `Adresse` uniquement dans `main.py`

© Achref EL

Python

Exercice 2

- Transformez l'association entre `Personne` et `Adresse` en association bidirectionnelle
- Pour éviter les imports croisés (Circular imports), n'importez pas `Personne` dans `Adresse` ni `Adresse` dans `Personne`
- Importez `Personne` et `Adresse` uniquement dans `main.py`

© Achref EL

N'oublions pas d'importer `annotations` dans `personne.py` et `adresse.py` : l'utilisation de `annotations` reste importante dans chaque fichier où les objets sont typés, afin de permettre l'utilisation des "forward declarations"

```
from __future__ import annotations
```

Python

Explication

- Avec from `__future__ import annotations`, tous les types (annotations de type) sont stockés comme des chaînes de caractères au lieu d'être évaluées immédiatement.
- Ainsi, **Python** ne tente pas d'importer ou d'accéder à l'objet typé au moment de la définition de la classe

Python

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est **une [sorte de]** `Classe2`

© Achref EL M

Python

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une `Classe1` est **une [sorte de]** `Classe2`

© Achref EL M

Forme générale

```
class ClasseFille (ClasseMère):
    # code
```

Python

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire

Python

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau

Python

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne

Python

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom` et `prénom`

Python

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom` et `prénom`
- Donc, on peut peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom` et `prénom`

Python

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que `numéro`, `nom` et `prénom`
- Donc, on peut peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom` et `prénom`
- Les classes `Étudiant` et `Enseignant` hériteront donc de la classe `Personne`

Python

Contenu de `etudiant.py`

```
from personne import Personne


class Etudiant(Personne):

    def __init__(self,  num: int = 0, nom: str = '', prenom: str = '', niveau: str = ''):
        super().__init__(num, nom, prenom)
        self.__niveau = niveau

    @property
    def niveau(self) -> str:
        return self.__niveau

    @niveau.setter
    def niveau(self, niveau) -> None:
        self.__niveau = niveau
```

© AC

Python

Contenu de `etudiant.py`

```
from personne import Personne


class Etudiant(Personne):

    def __init__(self,  num: int = 0, nom: str = '', prenom: str = '', niveau: str = ''):
        super().__init__(num, nom, prenom)
        self.__niveau = niveau

    @property
    def niveau(self) -> str:
        return self.__niveau

    @niveau.setter
    def niveau(self, niveau) -> None:
        self.__niveau = niveau
```

© AU

Remarques

- `class A (B)` : permet d'indiquer que la classe `A` hérite de la classe `B`
- `super()` permet d'appeler une méthode de la classe mère

Python

Contenu de `enseignant.py`

```
from personne import Personne


class Enseignant(Personne):

    def __init__(self, num: int = 0, nom: str = '',
       prenom: str = '', salaire: int = 0):
          super().__init__(num, nom, prenom)
          self.__salaire = salaire

    @property
    def salaire(self) -> int:
        return self.__salaire

    @salaire.setter
    def salaire(self, salaire) -> None:
        self.__salaire = salaire
```

Python

Pour créer deux objets `Enseignant` **et** `Etudiant` **dans** `main.py`

```
from etudiant import Etudiant
from enseignant import Enseignant

etudiant = Etudiant(200, 'maggio', 'toni', "licence")
print(etudiant)
# affiche 200 toni maggio

enseignant = Enseignant(300, 'dalton', 'jack', 1700)
print(enseignant)
# affiche 300 jack dalton
```

Python

Pour créer deux objets `Enseignant` **et** `Etudiant` **dans** `main.py`

```
from etudiant import Etudiant
from enseignant import Enseignant

etudiant = Etudiant(200, 'maggio', 'toni', "licence")
print(etudiant)
# affiche 200 toni maggio

enseignant = Enseignant(300, 'dalton', 'jack', 1700)
print(enseignant)
# affiche 300 jack dalton
```

Remarque

Le salaire et le niveau ne sont pas affichés.

Python

Redéfinissons `__str__` dans `etudiant.py`

```
    def __str__(self):
        return super().__str__() + " " + self.
            __niveau
```

© Achref EL MOU

Python

Redéfinissons `__str__` **dans** `etudiant.py`

```
    def __str__(self):
        return super().__str__() + " " + self.
            __niveau
```

Et dans `enseignant.py`

```
    def __str__(self):
        return super().__str__()  + " " + str(self.
            __salaire)
```

Achref EL MOU

Python

Re-testons tout cela dans `main.py`

```
from etudiant import Etudiant
from enseignant import Enseignant

etudiant = Etudiant(200, 'maggio', 'toni', "licence")
print(etudiant)
# affiche 200 toni maggio licence

enseignant = Enseignant(300, 'dalton', 'jack', 1700)
print(enseignant)
# affiche 300 jack dalton 1700
```

Python

Remarques

- `super(Etudiant, self)` est une écriture **Python** simplifiée et remplacée en `Python 3` par `super()`.
- En remplaçant `super()` dans `Etudiant` ou `Enseignant` par `Personne`, le résultat restera le même.
- `super()` est conseillé pour appeler les méthodes de la classe mère du niveau suivant tant dis que `Personne` (ou le nom d'une classe mère d'un niveau quelconque) permet de préciser le niveau de la classe mère qu'on cherche à appeler.

TypeScript

Remarque

Pour connaître la classe d'un objet, on peut utiliser le mot-clé `isinstance`

© Achref EL MOUELHI ©

TypeScript

Remarque

Pour connaître la classe d'un objet, on peut utiliser le mot-clé `isinstance`

MOUELHI ©

Exemple

```
print(isinstance(enseignant, Enseignant))
# affiche True

print(isinstance(enseignant, Personne))
# affiche True

print(isinstance(p, Enseignant))
# affiche False
```

Exercice

1. Créer un objet de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne` stocker les tous dans un seul tableau.
2. Parcourir le tableau et afficher pour chacun soit le `numero` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

© Achref EL MOUELHI ©

Exercice

1. Créer un objet de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne` stocker les tous dans un seul tableau.
2. Parcourir le tableau et afficher pour chacun soit le `numero` s'il est personne, soit le `salaire` s'il est enseignant ou soit le `niveau` s'il est étudiant.

MOUELHI ©

Pour parcourir un tableau, on peut faire

```
from personne import Personne
from etudiant import Etudiant
from enseignant import Enseignant

p = Personne(100, 'wick', 'john')
etudiant = Etudiant(200, 'maggio', 'toni', "licence")
enseignant = Enseignant(300, 'dalton', 'jack', 1700)

list = [p, etudiant, enseignant]

for elt in list:
    pass
```

Python

Solution

```
for elt in list:
    if isinstance(elt, Etudiant):
        print(elt.niveau)
    elif isinstance(elt, Enseignant):
        print(elt.salaire)
    else:
        print(elt.num)
```

Python

Particularité du **Python**

- Héritage multiple : une classe peut hériter simultanément de plusieurs autres
- L'héritage multiple est autorisé par certains langages comme **Python**, **C++**...

© Ach

Considérons la classe `Doctorant` qui hérite de `Enseignant` et `Etudiant`

```
from personne import Personne
from etudiant import Etudiant
from enseignant import Enseignant

class Doctorant(Enseignant, Etudiant):

    def __init__(self, num: int = 0, nom: str = '', prenom: str = '',
       salaire: int = 0, niveau: str = '', annee: str = ''):
        Enseignant.__init__(self, num, nom, prenom, salaire)
        Etudiant.__init__(self, num, nom, prenom, niveau)
        self.__annee = annee

    @property
    def annee(self) -> str:
        return self.__annee

    @annee.setter
    def annee(self, annee) -> None:
        self.__annee = annee

    def __str__(self) -> str:
        return Personne.__str__(self)  + " " + str(self.salaire) + " "
          + self.niveau + " " + self.__annee
```

Python

Dans `main.py`**, créons un objet e type** `Doctorant`

```
from doctorant import Doctorant

doctorant = Doctorant (300, 'cooper', 'austin', 1700, 'doctorat', '1 ère année')

print(doctorant)
# affiche 300 austin cooper 1700 doctorat 1 ère année
```

© Achr... HI ©

Python

Attention

- L'héritage multiple peut introduire des ambiguïtés : si deux classes parentes ont une méthode du même nom.
- Le **MRO** permet de résoudre ces conflits en suivant un ordre bien défini.

© Achref EL M

Python

Attention

- L'héritage multiple peut introduire des ambiguïtés : si deux classes parentes ont une méthode du même nom.
- Le **MRO** permet de résoudre ces conflits en suivant un ordre bien défini.

Achref EL M

MRO : **M**ethod **R**esolution **O**rder

Un mécanisme **Python** qui détermine l'ordre dans lequel les classes sont parcourues pour rechercher une méthode ou un attribut

Python

Comment fonctionne le **MRO** ?

- **Linearisation C3** : algorithme utilisé par **Python** pour déterminer l'ordre de recherche.
  - Une classe est prioritaire sur ses parents.
  - L'ordre des parents dans la déclaration de la classe est respecté.
  - Une classe ne peut être visitée qu'une seule fois.
- **Ordre de résolution**
  - **Python** commence par rechercher dans la classe elle-même.
  - Puis, il suit l'ordre des classes parentes selon le **MRO**.
  - Enfin, il remonte vers la classe racine `object`.

Python

Exemple : héritage unique

```
class A:
    def greet(self):
        return "Hello from A"

class B(A):
    def greet(self):
        return "Hello from B"

class C(B):
    pass

print(C.mro())  # affiche l'ordre MRO
# [<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]

c = C()
print(c.greet())
# affiche Hello from B
```

Python

Exemple : héritage multiple

```
class A:
    def greet(self):
        return "Hello from A"

class B:
    def greet(self):
        return "Hello from B"

class C(A, B):
    pass

print(C.mro())
# [<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]

c = C()
print(c.greet())
# affiche Hello from A
```

Python

Exemple : héritage multiple

```
class A:
    def __str__(self):
        return "A"


class B(A):
    def __str__(self):
        return super().__str__() + "B"


class C(A):
    def __str__(self):
        return super().__str__() + "C"


class D(C, B):
    def __str__(self):
        return C.__str__(self) + "D"


d = D()
print(d)
# affiche ABCD
```

Python

Explication

- Pour la classe `D`, l'ordre **MRO** est : `D` → `C` → `B` → `A` → `object`
- Dans `C.__str__`, `super().__str__()` n'appelle pas `A.__str__`, mais suit l'ordre **MRO**. Le prochain élément dans l'ordre après `C` est `B`.
- Résumé du flux :
  - `D.__str__` → `C.__str__(self)` + `D`.
  - `C.__str__` → `B.__str__(self)` + `C`.
  - `B.__str__` → `A.__str__(self)` + `B`.
  - `A.__str__` → `A`.
- La concaténation des chaînes dans cet ordre produit `ABCD`, qui est affiché.

Python

Surcharge (overload)

- Définir dans une classe plusieurs méthodes avec
  - le même nom
  - une signature différente
- Si dans une classe, on a deux méthodes avec le même nom, **Python** remplace toujours la précédente par celle qui est définie après.
- Donc pas de surcharge réelle en **Python**.

Python

Redéfinition (override)

- Définir une méthode dans une classe qui est déjà définie dans la classe mère
- Possible avec **Python**
- Deux manières de redéfinir une méthode
  - Proposer une nouvelle implémentation dans la classe fille différente et indépendante de celle de la classe mère (*simple comme si on définit une nouvelle méthode*)
  - Proposer une nouvelle implémentation qui fait référence à celle de la classe mère

Python

Redéfinition (override)

- Définir une méthode dans une classe qui est déjà définie dans la classe mère
- Possible avec **Python**
- Deux manières de redéfinir une méthode
  - Proposer une nouvelle implémentation dans la classe fille différente et indépendante de celle de la classe mère (*simple comme si on définit une nouvelle méthode*)
  - Proposer une nouvelle implémentation qui fait référence à celle de la classe mère

Exemple

Définir une méthode `afficherDetails()` dans `Personne` et la redéfinir dans `Etudiant` : une fois sans faire référence à celle de la classe `Personne` et une autre en faisant référence.

Commençons par la définir `afficherDetails()` dans `Personne`

```
    def afficher_details(self) -> None:
        print(self.__prenom + " " + self.__nom)
```

© Achref EL MOUELHI ©

Commençons par la définir `afficherDetails()` dans `Personne`

```
    def afficher_details(self) -> None:
        print(self.__prenom + " " + self.__nom)
```

Ensuite, on la redéfinit dans `Etudiant`

```
    def afficher_details(self) -> None:
        print(self.__prenom + " " + self.__nom + " " + self.__niveau)
```

© Achref EL MOUEL

Commençons par la définir `afficherDetails()` dans `Personne`

```
    def afficher_details(self) -> None:
        print(self.__prenom + " " + self.__nom)
```

Ensuite, on la redéfinit dans `Etudiant`

```
    def afficher_details(self) -> None:
        print(self.__prenom + " " + self.__nom + " " + self.__niveau)
```

Testons tout cela dans le `main`)

```
from personne import Personne
from etudiant import Etudiant

p = Personne(100, 'wick', 'john')
p.afficher_details()
# affiche john wick

etudiant = Etudiant(200, 'maggio', 'toni', "licence")
etudiant.afficher_details()
# affiche toni maggio licence
```

Python

On peut aussi utiliser l'implémentation de la classe `Personne`

```
    def afficher_details(self) -> None:
        Personne.afficher_details(self)
        print(self.__niveau)
```

© Achref EL MOUELHI

Python

On peut aussi utiliser l'implémentation de la classe `Personne`

```
    def afficher_details(self) -> None:
        Personne.afficher_details(self)
        print(self.__niveau)
```

En testant, le résultat est le même

```
etudiant = Etudiant(200, 'maggio', 'toni', "licence")
etudiant.afficher_details()
# affiche toni maggio
licence
```

Python

On peut aussi utiliser l'implémentation de la classe `Personne`

```
    def afficher_details(self) -> None:
        Personne.afficher_details(self)
        print(self.__niveau)
```

En testant, le résultat est le même

```
etudiant = Etudiant(200, 'maggio', 'toni', "licence")
etudiant.afficher_details()
# affiche toni maggio
licence
```

Refaire la même chose pour `Enseignant`

Python

Classe abstraite

- hérite de la classe `ABC` (**Abstract Base Class**) du module `abc`
- ne peut être instanciée si elle contient une ou plusieurs méthodes abstraites

© Achref EL MOU

Python

Classe abstraite

- hérite de la classe `ABC` (**Abstract Base Class**) du module `abc`
- ne peut être instanciée si elle contient une ou plusieurs méthodes abstraites

Si on déclare la classe `Personne` abstraite

```
from abc import ABC

class Personne(ABC):

    # code précédent
```

Python

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

© Achref EL MOUEL

Python

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Déclarons la méthode `afficherDetails()` comme abstraite dans `Personne`

```
@abstractmethod
def afficher_details(self):
    pass
```

Python

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Déclarons la méthode `afficherDetails()` comme abstraite dans `Personne`

```
    @abstractmethod
    def afficher_details(self):
        pass
```

Sans oublier d'importer `abstractmethod`

```
from abc import ABC, abstractmethod
```

Python

En testant, le résultat est le même

```
etudiant = Etudiant(200, 'maggio', 'toni', "licence")
etudiant.afficher_details()
# affiche toni maggio
licence
```

© Achi HI ©

Python

Définition

- Concept utilisé en programmation pour faciliter l'accès à un tableau d'objet défini dans un objet.
- (Autrement dit) Utiliser une "classe" comme un tableau.

Python

Considérons la classe `ListePersonnes` suivante qui contient un tableau de `Personne`

```
class ListePersonnes:
    def __init__(self, personnes: list) -> None:
        self.__personnes = personnes
```

© Achref EL MOU

Python

Considérons la classe `ListePersonnes` suivante qui contient un tableau de `Personne`

```
class ListePersonnes:
    def __init__(self, personnes: list) -> None:
        self.__personnes = personnes
```

Comment faire pour enregistrer des personnes dans le tableau `personnes` et les récupérer facilement en faisant `mes_amis[i]`

```
p = Personne(100, 'wick', 'john')
p2 = Personne(101, 'dalton', 'jack')
mes_amis = ListePersonnes([p, p2])
print(mes_amis[0])
```

© Achraf EL MOU

Python

Pour pouvoir accéder à un élément d'indice `i`, on ajoute la méthode `__getitem__()`

```
class ListePersonnes:
    def __init__(self, personnes: list) -> None:
        self.__personnes = personnes

    def __getitem__(self, i):
        return self.__personnes[i]
```

Python

Dans `main.py`

```
from liste_personnes import ListePersonnes
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(101, 'dalton', 'jack')
mes_amis = ListePersonnes([p, p2])

print(mes_amis[0])
# affiche 100 john wick
```

Python

En définissant un indexeur, l'opérateur de slicing `[:]` est systématiquement surchargé

```
from liste_personnes import ListePersonnes
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(101, 'dalton', 'jack')
mes_amis = ListePersonnes([p, p2])

print(mes_amis[0:1])
# affiche 100 john wick
```

Python

Pour pouvoir modifier un élément en utilisant son indice, on ajoute la méthode `__setitem__()`

```
class ListePersonnes:
    def __init__(self, personnes: list) -> None:
        self.__personnes = personnes

    def __getitem__(self, i):
        return self.__personnes[i]

    def __setitem__(self, key, value):
        self.__personnes[key] = value
```

Python

Dans `main.py`

```
from liste_personnes import ListePersonnes
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(101, 'dalton', 'jack')
mes_amis = ListePersonnes([p, p2])
mes_amis[1] = Personne(102, 'hadad', 'karim')

print(mes_amis[1])
# affiche 102 karim hadad
```

Python

Pour pouvoir utiliser la fonction `len()`, on ajoute la méthode `__len__()`

```
class ListePersonnes:
    def __init__(self, personnes: list) -> None:
        self.__personnes = personnes

    def __getitem__(self, i):
        return self.__personnes[i]

    def __setitem__(self, key, value):
        self.__personnes[key] = value

    def __len__(self):
        return len(self.__personnes)
```

Python

Dans `main.py`

```
from liste_personnes import ListePersonnes
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(101, 'dalton', 'jack')
mes_amis = ListePersonnes([p, p2])

print(len(mes_amis))
# affiche 2
```

Python

Autres opérateurs pour l'indexeur et méthodes à implémenter

- `in : __contains__()`
- `del : __delitem__()`
- ...

Python

Exercice

- Dans `Personne`, définir un indexeur sur les adresses
- Dans le `Main`, vérifier qu'il est possible d'accéder à l'adresse d'une personne de `ListePersonnes` en faisant par exemple `mes_amis[0][0]`

Python

Itérateur ?

- Concept utilisé dans plusieurs langages de programmation comme **C++** et **Java**
- Objet utilisé pour itérer sur des objets **Python** itérables comme les listes, les tuples, les dictionnaires, les ensembles et les chaînes de caractères
- Initialisé avec la méthode `__iter__()`
- Utilisant la méthode `__next__()` pour récupérer l'élément suivant

Python

Considérons la liste suivante

```
liste = [2, 3, 8, 5]
```

© Achref EL MOUELHI

Python

Considérons la liste suivante

```
liste = [2, 3, 8, 5]
```

EL MOUELH

Pour parcourir la liste, plusieurs solutions possibles

- utiliser une boucle `while` ou `for`
- utiliser un itérateur

Python

Déclarons l'itérateur

```
iterateur = iter(liste)
```

© Achref EL MOUELHI ©

Python

Déclarons l'itérateur

```
iterateur = iter(liste)
```

Pour demander la première valeur

```
print(iterateur.__next__())
# affiche 2
```

© Achr JOUELHI ©

Python

Déclarons l'itérateur

```
iterateur = iter(liste)
```

Pour demander la première valeur

```
print(iterateur.__next__())
# affiche 2
```

© Ach JOUELHI ©

Ou en plus simple

```
print(next(iterateur))
# affiche 5
```

Python

Utilisons une boucle `while` pour itérer sur tous les éléments de la liste

```
while True:
    print(next(iterateur))
# affiche
# 2
# 3
# 8
# 5
# Traceback (most recent call last):
#   File "C:/Users/elmou/PycharmProjects/cours-poo/main.
  py", line 6, in <module>
#     print(next(iterateur))
# StopIteration
```

Python

Utilisons une boucle `while` pour itérer sur tous les éléments de la liste

```
while True:
    print(next(iterateur))
# affiche
# 2
# 3
# 8
# 5
# Traceback (most recent call last):
#   File "C:/Users/elmou/PycharmProjects/cours-poo/main.
  py", line 6, in <module>
#     print(next(iterateur))
# StopIteration
```

Une exception sera levée lorsqu'il n'y a plus de valeurs à retourner

Python

Pour résoudre le problème précédent

```
while True:
    try:
        print(next(iterateur))
    except StopIteration:
        break
# affiche
# 2
# 3
# 8
# 5
```

Python

Pour rendre un objet itérable

- Soit on définit un indexeur pour l'objet
- Soit on implémente les méthodes `__iter__()` et `__next__()`

© ACI

Python

Pour rendre la classe itérable, implémentons les méthodes `__iter__()` et `__next__()`

```
class Nombre:
  def __iter__(self):
    self.valeur = 1
    return self

  def __next__(self):
    if self.valeur <= 20:
      x = self.valeur
      self.valeur += 1
      return x
    else:
      raise StopIteration
```

© Achref El

Python

Pour rendre la classe itérable, implémentons les méthodes `__iter__()` et `__next__()`

```
class Nombre:
  def __iter__(self):
    self.valeur = 1
    return self

  def __next__(self):
    if self.valeur <= 20:
      x = self.valeur
      self.valeur += 1
      return x
    else:
      raise StopIteration
```

© Achref El

Explication

- La méthode `__iter__()` retourne `self` pour indiquer que l'on souhaite rendre l'objet courant itérable
- La méthode `__next__()` lève une exception quand l'objet n'est plus itérable

Python

La classe `Nombre` retourne 20 valeurs incrémentales commençant par 1

```
nombre = Nombre()
iter = iter(nombre)

for val in iter:
  print(val, end=" ")
# affiche 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
   18 19 20
```

Python

Pour la suite

Commentez la méthode `__getitem__()` dans `ListePersonnes`.

© Achref El Mouelhi ©

Python

Et si on veut parcourir les objets `Personne` de la classe `ListePersonnes` comme si c'était une vraie liste, c'est-à-dire :

```
from liste_personnes import ListePersonnes
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(101, 'dalton', 'jack')
mes_amis = ListePersonnes([p, p2])

for personne in mes_amis:
  print(personne)
```

Un message d'erreur nous informe que `ListePersonnes` n'est pas itérable

Python

Exercice

Ajoutez les méthodes `__iter__()` et `__next__()` dans `ListePersonnes` pour qu'elle soit itérable.

© Achr

Python

Solution

```
from personne import Personne

class ListePersonnes:
    def __init__(self, personnes: list[Personne]) -> None:
        self.__personnes = personnes
        self.__indice = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.__indice >= len(self.__personnes):
            raise StopIteration
        value = self.__personnes[self.__indice]
        self.__indice += 1
        return value
```

Python

Définition

- Concept introduit initialement par le langage **C++** (surcharge d'opérateurs)
- Permettant de surcharger les opérateurs de comparaison ou les opérateurs arithmétiques déjà utilisés pour les types prédéfinis

Python

Opérateurs de comparaison et méthodes à implémenter

- `==` : `__eq__()`
- `!=` : `__ne__()`
- `>=` : `__ge__()`
- `<=` : `__le__()`
- `>` : `__gt__()`
- `<` : `__lt__()`

Python

Opérateurs arithmétiques et méthodes à implémenter

- `+` : `__add__()`
- `-` : `__sub__()`
- `*` : `__mul__()`
- `**` : `__pow__()`
- `/` : `__truediv__()`
- `//` : `__floordiv__()`
- `%` : `__mod__()`
- ...

Python

Opérateurs arithmétiques composés et méthodes à implémenter

- `+=` : `__iadd__()`
- `-=` : `__isub__()`
- `*=` : `__imul__()`
- `**=` : `__ipow__()`
- `/=` : `__itruediv__()`
- `//=` : `__ifloordiv__()`
- `%=` : `__imod__()`
- ...

Python

Exemple

```
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(100, 'wick', 'john')
if p == p2:
    print(True)
else:
    print(False)
# affiche False
```

© Achref EL

Python

Exemple

```
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(100, 'wick', 'john')
if p == p2:
    print(True)
else:
    print(False)
# affiche False
```

© Achref EL

Remarques

- Les deux objets précédents ne sont pas égaux car ils occupent deux espaces mémoires différents
- Pour définir une nouvelle règle de comparaison d'objets, on doit implémenter la méthode `__eq__()` dans la classe `Personne`

Python

Ajoutons l'implémentation de la méthode `__eq__()` dans la classe `Personne`

```
def __eq__(self, p: "Personne") -> bool:
    return p.__nom == self.__nom and p.__prenom == self.
       __prenom and p.__num == self.__num
```

© Achref EL MOUELHI ©

Python

Ajoutons l'implémentation de la méthode `__eq__()` dans la classe `Personne`

```
def __eq__(self, p: "Personne") -> bool:
    return p.__nom == self.__nom and p.__prenom == self.
       __prenom and p.__num == self.__num
```

MOUELHI ©

En testant le `main` précédent, le résultat est

```
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(100, 'wick', 'john')
if p == p2:
    print(True)
else:
    print(False)
# affiche True
```

Python

Ajoutons l'implémentation de la méthode `__gt__()` dans la classe `Personne`

```
def __gt__(self, other: "Personne"):
    return self._num > other._num
```

© Ach JOUELHI©

Python

testons cela dans le `main`

```
from personne import Personne

p = Personne(100, 'wick', 'john')
p2 = Personne(101, 'wick', 'john')

if p2 > p:
    print(True)
else:
    print(False)
# affiche True

if p > p2:
    print(True)
else:
    print(False)
# affiche False
```

Python

Ajoutons l'implémentation de la méthode `__iadd__()` dans la classe `Enseignant`

```
    def __iadd__(self, i: int):
        self.__salaire += i
        return self
```

© Achref EL MOUELHI ©

Python

Ajoutons l'implémentation de la méthode `__iadd__()` dans la classe `Enseignant`

```
    def __iadd__(self, i: int):
        self.__salaire += i
        return self
```

MOUELHI ©

testons cela dans `main.py`

```
from enseignant import Enseignant

enseignant = Enseignant(300, 'dalton', 'jack', 1700)

print(enseignant)
# affiche 300 jack dalton  1700

enseignant += 200
print(enseignant)
# affiche 300 jack dalton  1900
```

Python

Généricité

- Concept connu dans plusieurs LOO (**Java**, **TypeScript**, **C#**)
- Objectif : définir des fonctions, classes s'adaptant avec plusieurs types de données

Python

Exemple

- si on a besoin d'une classe dont les méthodes effectuent les mêmes opérations quel que soit le type d'attributs
  - **somme** pour **entiers** ou **réels**,
  - **concaténation** pour **chaînes de caractères**,
  - **ou logique** pour **booléens**...
  - ...
- **Impossible sans définir plusieurs classes (une pour chaque type)**

Python

Solution avec la généricité

```
from typing import TypeVar, Generic

T = TypeVar('T')

class Operation (Generic[T]):

    def __init__(self, var1: T, var2: T):
        self.__var1 = var1
        self.__var2 = var2

    def plus(self):
        if type(self.__var1) is str :
            return self.__var1 + self.__var2
        elif type(self.__var1) is int and  type(self.__var2) is int:
            return self.__var1 + self.__var2;
        elif type(self.__var1) is bool and  type(self.__var2) is bool:
            return self.__var1 | self.__var2
        else:
            raise TypeError("error")
```

Python

Nous pouvons donc utiliser la même méthode pour plusieurs types différents

```
from operation import Operation

try:
    operation1  =  Operation[int](5, 3)
    print(operation1.plus())
    # affiche 8

    operation2 =  Operation[str]("bon", "jour")
    print(operation2.plus())
    # affiche bonjour

    operation3 =  Operation[bool](True, False)
    print(operation3.plus())
    # affiche true

    operation4 =  Operation[bool](2.8, 4.9)
    print(operation4.plus())
    # affiche problème de type

except TypeError:
    print ("problème de type")
```

Python

On peut aussi spécifier la liste des types autorisés

```
from typing import TypeVar, Generic

T = TypeVar('T', int, float, str, bool)

class Operation (Generic[T]):

    def __init__(self, var1: T, var2: T):
        self.__var1 = var1
        self.__var2 = var2

    def plus(self):
        if type(self.__var1) is str :
            return self.__var1 + self.__var2
        elif type(self.__var1) is int and  type(self.__var2) is int:
            return self.__var1 + self.__var2;
        elif type(self.__var1) is bool and  type(self.__var2) is bool:
            return self.__var1 | self.__var2
        else:
            raise TypeError("error")
```

Python

Depuis Python 3.12, la définition d'une classe générique peut être simplifiée ainsi

```
class Operation[T]:

    def __init__(self, var1: T, var2: T):
        self.__var1 = var1
        self.__var2 = var2

    def plus(self):
        if type(self.__var1) is str :
            return self.__var1 + self.__var2
        elif type(self.__var1) is int and  type(self.__var2) is int:
            return self.__var1 + self.__var2;
        elif type(self.__var1) is bool and  type(self.__var2) is bool:
            return self.__var1 | self.__var2
        else:
            raise TypeError("error")
```

Python

Méta-classe

- est une classe utilisée pour créer des classes.
- contrôle le comportement des classes elles-mêmes, de la même manière qu'une classe contrôle le comportement des objets.

© Achref EL

Python

Méta-classe

- est une classe utilisée pour créer des classes.
- contrôle le comportement des classes elles-mêmes, de la même manière qu'une classe contrôle le comportement des objets.

© Achref EL

Remarque

Par défaut, la méta-classe de toutes les classes en **Python** est `type`.

Python

Méta-classe, pourquoi ?

- Pour modifier ou contrôler la manière dont les classes sont créées.
- Pour appliquer des règles globales à la définition des classes (par exemple, ajouter des méthodes ou valider les attributs).
- ...

© Achref EL

Python

Méta-classe, pourquoi ?

- Pour modifier ou contrôler la manière dont les classes sont créées.
- Pour appliquer des règles globales à la définition des classes (par exemple, ajouter des méthodes ou valider les attributs).
- ...

© Achref El

Hypothèse

Imaginons que nous voulons forcer toutes les classes à avoir une méthode spécifique appelée `saluer`.

Python

Solution avec la généricité

```
class VerificateurDeMethodes(type):
    def __new__(cls, name, bases, dct):
        if 'saluer' not in dct:
            raise TypeError(f"La classe {name} doit définir une méthode 'saluer'")
        return super().__new__(cls, name, bases, dct)
```

© Achref EL MOUELHI

Python

Solution avec la généricité

```
class VerificateurDeMethodes(type):
    def __new__(cls, name, bases, dct):
        if 'saluer' not in dct:
            raise TypeError(f"La classe {name} doit définir une méthode 'saluer'")
        return super().__new__(cls, name, bases, dct)
```

EL MOUELH

Méta-classe, pourquoi ?

- `cls` : la méta-classe elle-même.
- `name` : le nom de la classe en cours de création.
- `bases` : un tuple contenant les classes parentes.
- `dct` : le dictionnaire des attributs et méthodes définis dans la classe.

Python

Cette classe est valide car elle a une méthode `saluer`

```
class ClasseValide(metaclass=VerificateurDeMethodes):
    def saluer(self):
        print("Bonjour !")
```

© Achref EL MOU

Python

Cette classe est valide car elle a une méthode `saluer`

```
class ClasseValide(metaclass=VerificateurDeMethodes):
    def saluer(self):
        print("Bonjour !")
```

Celle-là lèvera une erreur car elle n'a pas de méthode `saluer`

```
class ClasseInvalide(metaclass=VerificateurDeMethodes):
    pass
```

Python

Functor

- Une classe qui définit la méthode `__call__()`
- Permet de rendre une instance d'une classe appelable : les objets pouvant être utilisés comme des fonctions.

Python

Déclarons le `functor` suivant

```
class Operation[T]:

    def __init__(self, var1: T, var2: T):
        self.__var1 = var1
        self.__var2 = var2

    def __call__(self):
        return self.__var1 + self.__var2
```

Python

Pour l'utiliser

```
operation1  =  Operation[int](5, 3)
print(operation1())
# affiche 8
```

© Ach... IEL HI ©

Python

Remarque

Un Functor peut être utilisé comme une callback.

© Achref

Python

Énumération

- ensemble de noms liés à des valeurs constantes (nom + valeur = membre)
- structure de données itérable

© AGH

Python

Exemple : définissons l'énumération `Animal`

```
from enum import Enum

Animal = Enum('Animal', 'CHAT CHIEN CHAMEAU CHEVAL SOURIS')
```

© Achref EL MOUEL

Python

Exemple : définissons l'énumération `Animal`

```
from enum import Enum

Animal = Enum('Animal', 'CHAT CHIEN CHAMEAU CHEVAL SOURIS')
```

Ou

```
from enum import Enum

Animal = Enum('Animal', ['CHAT', 'CHIEN', 'CHAMEAU', 'CHEVAL',
   'SOURIS'])
```

hbref EL MOUL

Python

Pour accéder à la valeur ou le nom d'un membre d'une énumération

```
print (Animal.CHIEN.value)
# affiche 2

print (Animal.CHIEN.name)
# affiche chien
```

© Achref EL MOU

Python

Pour accéder à la valeur ou le nom d'un membre d'une énumération

```
print (Animal.CHIEN.value)
# affiche 2

print (Animal.CHIEN.name)
# affiche chien
```

Pour récupérer un membre, les deux écritures suivantes sont équivalentes

```
print (Animal.CHIEN)
# affiche Animal.CHIEN

print (Animal['CHIEN'])
# affiche Animal.CHIEN
```

Python

Pour récupérer le nom associée à une valeur

```
print (Animal(1))
# affiche Animal.CHAT
```

© Achref EL MOU

Python

Pour récupérer le nom associée à une valeur

```
print (Animal(1))
# affiche Animal.CHAT
```

Pour itérer sur une énumération

```
for animal in Animal:
    print(animal)
```

Python

Pour modifier les valeurs par défaut associées aux noms, on déclare l'énumération comme une classe qui hérite de `Enum`

```
from enum import Enum

class Animal(Enum):
        CHAT = 1
        CHIEN = 3
        CHAMEAU = 2
        CHEVAL = 4
        SOURIS = 6
```

© AC

Python

Pour modifier les valeurs par défaut associées aux noms, on déclare l'énumération comme une classe qui hérite de `Enum`

```
from enum import Enum

class Animal(Enum):
        CHAT = 1
        CHIEN = 3
        CHAMEAU = 2
        CHEVAL = 4
        SOURIS = 6
```

© AC

Pour récupérer la valeur associée à un nom (rien ne change)

```
print(Animal.CHAMEAU.value)
# affiche 2
```

Python

Deux noms différents peuvent avoir la même valeur

```
from enum import Enum

class Animal(Enum):
        CHAT = 1
        CHIEN = 3
        CHAMEAU = 2
        CHEVAL = 2
        SOURIS = 6
```

© Achre

Python

Deux noms différents peuvent avoir la même valeur

```
from enum import Enum

class Animal(Enum):
        CHAT = 1
        CHIEN = 3
        CHAMEAU = 2
        CHEVAL = 2
        SOURIS = 6
```

© Achre

En accédant via une valeur, on récupère le premier membre ayant la valeur recherchée

```
print(Animal(2))
# affiche Animal.CHAMEAU
```

Python

Pour forcer l'unicité des valeurs, on utilise le décorateur `@unique` (ValueError : duplicate values found in <enum 'Animal'> : CHEVAL -> CHAMEAU)

```
from enum import Enum, unique

@unique
class Animal(Enum):
        CHAT = 1
        CHIEN = 3
        CHAMEAU = 2
        CHEVAL = 2
        SOURIS = 6
```

Python

Pour affecter des valeurs auto-increment aux différents noms

```
from enum import Enum, auto

class Animal(Enum):
        CHAT = auto()
        CHIEN = auto()
        CHAMEAU = auto()
        CHEVAL = auto()
        SOURIS = auto()

for animal in Animal:
    print(animal.value)
# affiche 1 2 3 4 5
```

Python

Fonction `dir()`

- sans paramètre : retourne la liste de méthodes et attributs disponibles dans le module
- avec paramètre (objet ou classe) : retourne la liste de méthodes et attributs définis (et hérités) de la classe

Python

Quelques constantes sur les fonctions

```
print(dir())
# affiche ['__annotations__', '__builtins__', '__cached__', '
   __doc__', '__file__', '__loader__', '__name__', '__package__
   ', '__spec__']

print(dir(personne))
# affiche ['__class__', '__delattr__', '__dict__', '__dir__', '
   __doc__', '__eq__', '__format__', '__ge__', '
   __getattribute__', '__gt__', '__hash__', '__init__', '
   __init_subclass__', '__le__', '__lt__', '__module__', '
   __ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__
   ', '__setattr__', '__sizeof__', '__str__', '__subclasshook__
   ', '__weakref__', 'increment', 'nbr_personnes', 'nom', 'num
   ', 'prenom']
```

Python

Conventions

- Une classe par fichier
- Deux lignes vides avant et après la déclaration d'une classe
- Une ligne vide avant et après la déclaration d'une méthode
- Un message de documentation pour chaque méthode publique à placer après la ligne contenant le mot-clé `def`
- Une classe de test par classe