Python et la programmation Orientée-Objet

Séna APEKE

UL/BERIIA

17/08/2021

1. **Introduction**
2. **__str__**
3. **Les constructeurs**
   - Setter
   - Getter
   - Property()
   - Deleter
4. **Décorateur**
   - Introduction
   - __del__()
   - Attributs et méthodes statiques
5. **Héritage**
   - Héritage simple
   - Héritage multiple
6. **Polymorphisme**
   - Surchare
7. **Classe et méthodes abstraites**
   - Des méthodes abstraites avec des paramètres
   - Bonus : Créer des propriétés abstraites – @property
8. **Indexeur**
   - __getitem__()

Class

Qu'est ce qu'une classe en POO?

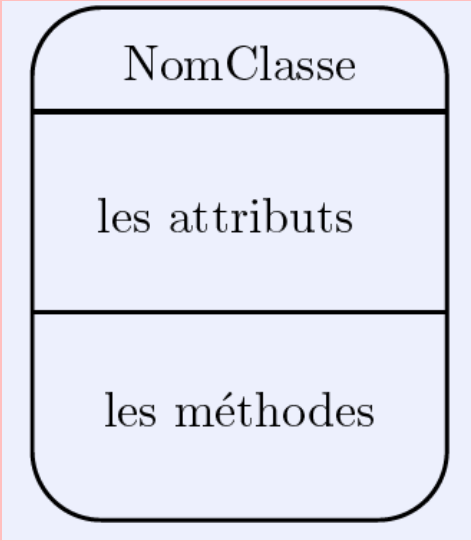
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétées statiques (attributs) et dynamiques (méthodes)

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

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

Class

Qu'est ce qu'une classe en POO?



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

Particularité de Python

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

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
```

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
```

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
```

__str__

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

Définissons la méthode __str__()

Définition

```
class Personne:
    num: int
    nom: str
    prenom: str
    def __str__(self) → str :
        return self.prenom + " " + self.nom
```

Affichons les détails de notre instance p

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

__repr__()

Remplaçons la méthode __str__() par __repr__()

```
class Personne:
    num: int
    nom: str
    prenom: str
    def __repr__(self → str :
        return self.prenom + " " + self.nom
```

Exécutons le main.py précédent sans le changer

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

Rajoutons 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
```

Exécutons le main.py précédent sans le changer

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

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.htmlobject.__repr__

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

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
```

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'
```

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 des valeurs par défaut

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
```

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
```

Setter

Hypothèse

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

Setter

Hypothèse

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

Démarche

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

Setter

Hypothèse

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

Démarche

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

Rappel

Setter : Ce sont les méthodes utilisées dans la fonctionnalité OOPS qui permet de définir la valeur des attributs privés dans une classe.

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
```

Pour tester :

```
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
```

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
```

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
```

Question

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

Question

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

Démarche

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

Question

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

Démarche

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

Rappel

Getter : Ce sont les méthodes utilisées dans la programmation orientée objet (OOPS) qui permet d'accéder aux attributs privés d'une classe.

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
```

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
```

La méthode property() permet

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

Ajoutons property 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
    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
```

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
```

Le deleter permet

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

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

```
class Personne:
    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 del_prenom(self) → None :
        del self.__prenom

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

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'
```

Le décorateur (annotation) property() permet

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

De quoi s'agit-il ?

En programmation, le décorateur est un modèle de conception qui ajoute dynamiquement des responsabilités supplémentaires à un objet. En Python, une fonction est l'objet de premier ordre. Ainsi, un décorateur en Python ajoute dynamiquement des responsabilités/fonctionnalités supplémentaires à une fonction sans modifier une fonction.

En Python, une fonction peut être passée en argument à une autre fonction. Il est également possible de définir une fonction à l'intérieur d'une autre fonction, et une fonction peut retourner une autre fonction.

Ainsi, un décorateur en Python est une fonction qui reçoit une autre fonction en argument. Le comportement de la fonction argument est étendu par le décorateur sans réellement le modifier. La fonction décorateur peut être appliquée sur une fonction en utilisant la syntaxe @decorator.

Comprenons pas à pas le décorateur en Python.

Considérez que nous avons la fonction greet (), comme indiqué ci-dessous.

```
def greet():
    print('Hello! ', end='')
```

Maintenant, nous pouvons étendre la fonctionnalité de la fonction ci-dessus sans la modifier en la passant à une autre fonction, comme indiqué ci-dessous.

```
def mydecorator(fn):
    fn()
    print('How are you?')
```

Ci-dessus, la fonction **mydecorator()** prend une fonction comme argument. Il appelle la fonction argument et affiche également des éléments supplémentaires. Ainsi, il étend la fonctionnalité de la fonction **greet()** sans la modifier. Cependant, ce n'est pas le décorateur réel.

```
>>> mydecorator(greet)
Hello! How are you?
```

mydecorator() n'est pas un décorateur en Python. Le décorateur en Python peut être défini sur n'importe quelle fonction appropriée en utilisant la syntaxe @decorator_function_name pour étendre les fonctionnalités de la fonction sous-jacente.
Ce qui suit définit le décorateur pour la fonction greet() ci-dessus.

```
def mydecorator(fn):
    def inner_function():
        fn()
        print('How are you?')
    return inner_function
```

La fonction **mydecorator()** est la fonction décoratrice qui prend une fonction (toute fonction qui ne prend aucun argument) comme argument. La fonction interne inner_function() peut accéder à l'argument de la fonction externe, elle exécute donc du code avant ou après pour étendre la fonctionnalité avant d'appeler la fonction argument. La fonction mydecorator renvoie une fonction interne.

Maintenant, nous pouvons utiliser mydecorator comme décorateur à appliquer sur une fonction qui ne prend aucun argument, comme indiqué ci-dessous.

```
@mydecorator
def greet():
    print('Hello! ', end='')
```

Maintenant, appeler la fonction greet() ci-dessus donnera la sortie suivante.

```
>>> greet()
Hello! How are you?
```

Le mydecorator peut être appliqué à n'importe quelle fonction qui ne nécessite aucun argument. Par exemple :

```
@mydecorator
def dosomething():
    print('I am doing something.', end='')
```

```
>>> dosomething()
I am doing something. How are you?
```

La fonction de décorateur typique ressemblera à :

```
def mydecoratorfunction(some_function): # decorator function
    def inner_function():
        # write code to extend the behavior of some_function()
        some_function()  call some_function
        # write code to extend the behavior of some_function()
    return inner_function  return a wrapper function
```

La bibliothèque Python contient de nombreux décorateurs intégrés comme raccourci pour définir les propriétés, les méthodes de classe, les méthodes statiques, etc.

Decorator	Description
@property	Declares a method as a property's setter or getter methods.
@classmethod	Declares a method as a class's method that can be called using the class name.
@staticmethod	Declares a method as a static method.

Modifions la classe Personne et utilisons les décorateurs

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

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

Pour tester

```
from personne import Personne

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

p.num = -100

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

__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**

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):
        pinrt("destructeur appelé")
```

Avec le contenu précédent

Pour tester

```
from personne import Personne

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

del p
# affiche destructeur appelé
```

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 wick john

print("fin du programme")

# affiche fin du programme
# affiche destructeur appelé
```

Récapitulatif

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

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs 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)

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs 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.

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
- En Python, un attribut qui n'est pas déclaré dans le constructeur est un attribut static

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
- En Python, un attribut qui n'est pas déclaré dans le constructeur est un attribut static

Particularité de Python

- Un attribut qui n'est pas déclaré dans le constructeur est un attribut statique
- Pas de mot-clé static comme dans la plupart des LOO

Ajoutons un attribut statique nbr personnes à la liste d'attributs de la classe Personne

```
nbr_personnes = 0
```

```
class Personne:
    # 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

    # Décrémentons dans le destructeur
    def __del__(self):
        Personne.nbr_personnes -= 1
        pinrt("destructeur appelé")
```

Avec le contenu précédent

Pour tester

```
from personne import Personne

print(Personne.nbr_personnes)
# affiche 0

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

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

Pour définir une méthode statique, on utilise

- soit le décorateur staticmethod
- soit le décorateur classmethod

Définissons une méthode statique avec le décorateur (@staticmethod) pour incrémenter nbr personnes

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

Et 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()
```

Pour tester

```
from personne import Personne

print(Personne.nbr_personnes)
# affiche 0

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

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

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()
```

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
```

Pour conclure

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

Exercice

- Définissez une classe Adresse avec trois attributs privés rue, code postal et ville de type chaıˆne de caractère
- Définissez un constructeur avec trois paramètres, les getters et setters
- Dans la classe Personne, ajouter un attribut adresse (de type Adresse) dans le constructeur et 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

L'héritage, quand ?

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

Forme générale :

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

Exemple

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

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
```

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

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
```

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.

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)
```

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
```

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

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 ´ etudiant.

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
```

Solution

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

Héritage multiple

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

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
```

Dans main.py, créons un objet de 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
```

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.

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 définir afficherDetails() dans Personne

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

Commençons par 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)
```

Commençons par 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
```

On peut aussi utiliser l'implémentation de la classe Personne

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

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
```

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

Classe abstraite

Les classes abstraites sont des classes qui ne peuvent pas être instanciées, elles contiennent une ou plusieurs méthodes abstraites. C'est un modèle pour d'autres classes qui héritent un ensemble de méthodes et de propriétés.

Classe abstraite

Les classes abstraites sont des classes qui ne peuvent pas être instanciées, elles contiennent une ou plusieurs méthodes abstraites. C'est un modèle pour d'autres classes qui héritent un ensemble de méthodes et de propriétés.

Une méthode abstraite est une méthode déclarée, mais qui n'a pas d'implémentation. Toutefois, une classe abstraite nécessite des sous-classes qui fournissent des implémentations pour les méthodes abstraites.

Pourquoi une classe abstraite ?

L'abstraction est très utile lors de la conception de systèmes complexes pour limiter la répétition et assurer la cohérence. C'est similaire à une interface dans d'autres langages. De plus, les classes abstraites nous donnent une manière standard de développer le code même si vous avez plusieurs développeurs travaillant sur un projet.

Une classe abstraite agit comme un modèle pour les sous-classes. C'est la classe principale qui spécifie comment les classes enfants doivent se comporter. En un sens, une classe abstraite est la « classe pour créer des classes ».

Par exemple, Shape pourrait être une classe abstraite qui implémente une méthode abstraite area(). Ensuite, les sous-classes Circle, Triangle et Rectangle fournissent leurs propres implémentations de cette méthode en fonction de leurs caractéristiques en géométrie.

Ainsi, les classes abstraites permettent d'appliquer des règles pour créer des classes enfants liées.

Un exemple d'une classe abstraite en Python

Tout d'abord, commençons par définir trois classes Lion, Panda et Singe :

```
class Lion:
    def nourrir(self):
        print("Nourrir le lion avec de la viande crue!")

class Panda:
    def nourrir_animal(self) :
        print("Nourrir le panda avec du bambou!")

class Singe:
    def nourriture(self):
        print("Nourrir le singe avec des bananes!")
```

Notre travail consiste à nourrir tous les animaux en utilisant un script Python.

code

```
# Les animaux du zoo
leo = Lion()
po = Panda()
tok = Singe()

# Nourrir les animaux du zoo
leo.nourrir()
po.nourrir_animal()
tok.nourriture()
```

Le résultat du code

```
Nourrir le lion avec de la viande crue!
Nourrir le panda avec du bambou!
Nourrir le singe avec des bananes!
```

Mais imaginez combien de temps cela prendrait pour chaque animal d'un grand zoo, en répétant le même processus et en codant des centaines de fois. Cela rendrait également le code plus difficile à maintenir.
En fait, afin d'optimiser le processus, la structure de notre programme pourrait ressembler à ceci :

```
# mettre les animaux dans une liste
zoo = [leo, po, tok]

# nourrir les animaux à travers une boucle
for animal in zoo:
        # Mais quelle méthode à utiliser ?
        # nourrir(), nourrir_animalounourriture?
        # Nous aurons : an AttributeError!
```

Le problème est que chaque classe a un nom de méthode différent, quand on nourrit un lion, c'est la méthode **nourrir()**, quand on nourrit un panda, c'est **nourrir_animal()** et c'est **nourriture()** pour le singe.
Ce code est un gâchis, car les méthodes qui font la même chose doivent avoir le même nom. Si nous pouvions seulement forcer nos classes à implémenter les mêmes noms de méthodes.
Il s'avère que la classe abstraite est ce dont nous avons besoin. Puisque'elle force ses sous-classes à implémenter toutes ses méthodes abstraites.

En créant une classe abstraite **Animal**, chaque classe enfant qui hérite de la classe Animal doit implémenter des méthodes abstraites de Animal, qui dans notre cas est la méthode **nourrir()**.

```
from abc import ABC, abstractmethod
# abc est un module python intégré, nous importons ABC et
abstractmethod

class Animal(ABC): # hériter de ABC(Abstract base class)
    @abstractmethod # un décorateur pour définir une méthode abstraite
     def nourrir(self):
         pass
```

Nous avons importé le module intégré abc. Lors de la définition d'une classe abstraite, nous avons hérité de la classe abstraite de base – ABC. Pour définir une méthode abstraite dans la classe abstraite, nous avons utilisé un décorateur **@abstractmethod**.

Si vous héritez de la classe **Animal**, mais n'implémentez pas les méthodes abstraites, vous obtiendrez une erreur. Si nous essayons d'instancier la classe, cela générera un **TypeError**, car nous ne pouvons pas instancier **Panda** sans implémenter une méthode abstraite **nourrir()**.

```
class Panda(Animal):
    def autre_nom(self):
        print("Nourrir le panda avec du bamboo!")

po = Panda()

TypeError: Can't instantiate abstract class Panda with abstract methods
nourrir
```

Donc, voici ce qu'il faudrait faire.

```
class Lion(Animal):
    def nourrir(self):
        print("Nourrir le lion avec de la viande crue!")

class Panda(Animal):
    def nourrir(self):
        print("Nourrir le panda avec du bambou!")

class Singe(Animal):
    def nourrir(self):
        print("Nourrir le singe avec des bananes!")
```

Puis, voici le code dont nous avons besoin pour créer et nourrir nos animaux.

```
zoo = [Lion(), Panda(), Singe()]

for animal in zoo:
    animal.nourrir()
```

Des méthodes abstraites avec des paramètres

Lorsque la sous-classe implémente une méthode, elle doit également contenir tous les paramètres de la classe de base. L'implémentation de la sous-classe peut également ajouter des paramètres supplémentaires si nécessaire.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def faire_quelque_chose(self, action):
        pass

class Lion(Animal):
    def faire_quelque_chose(self, action, time):
        print(f"action le lion à time")

class Panda(Animal):
```

La méthode **faire_quelque_chose** possède un paramètre **action**. De plus, **time** est un autre paramètre des sub-classes.

Code

```
zoo = [Lion(), Panda(), Singe()]

for animal in zoo:
    animal.faire_quelque_chose(action="nourrir", time="10h00")
```

L'exécution du code.

```
nourrir le lion à 10h00
nourrir le panda à 10h00
nourrir le singe à 10h00
```

Nous avons vu comment utiliser le décorateur @property.
Si nous voulons créer des propriétés abstraites et forcer notre sous-classe à implémenter ces propriétés en utilisant les décorateurs @property et @absctractmethod.

Comme les animaux ont souvent des régimes alimentaires différents, nous allons définir une méthode diet dans nos classes d'animaux. Puisque tous les animaux héritent de Animal, nous pouvons définir diet comme étant une propriété abstraite. De plus, nous aurons besoin d'une autre méthode aliment qui définit les aliments que nous donnerons aux animaux et qui sera une propriété. Et nous aurons un setter de aliment vérifiera si nous essayons de nourrir l'animal avec quelque chose qui n'est pas dans diet.

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @property
    def alimentation(self):
        return self._aliment

    @alimentation.setter
    def alimentation(self, aliment):
        if aliment in self.diet:
            self._aliment = aliment
        else:
            raise ValueError(f"Cet animal ne mange de faliment.")
    @property
    @abstractmethod
    def diet(self):
            pass
    @abstractmethod
    def nourrir(self, time):
            pass
```

```
class Lion(Animal):
    @property
    def diet(self):
        return ["cheval", "gazelle", "buffle"]

    def nourrir(self, time):
        print(f"Le lion mange de la viande de self._aliment à time")

class Serpent(Animal):
    @property
    def diet(self):
        return ["grenouille", "lapin"]

    def nourrir(self, time):
        print(f"Le serpent mange de la viande de self
```

Nous allons créer deux objets, définir la nourriture des animaux, puis appeler la méthode nourrir().

```
leo = Lion()
leo.alimentation = "buffle"
leo.nourrir("12h00")
jo = Serpent()
jo.alimentation = "grenouille"
jo.nourrir("12h20")
```

L'exécution du code.

Le lion mange de la viande de buffle à 12h00
Le serpent mange de la viande de grenouille à 12h20

Si nous essayons de nourrir un animal avec quelque chose qu'il ne mange pas.

```
leo = Lion()
leo.alimentation = "carottes"
leo.nourrir("12h00")

ValueError: Cet animal ne mange de carottes.
```

Si nous essayons de nourrir un animal avec quelque chose qu'il ne mange pas.

```
leo = Lion()
leo.alimentation = "carottes"
leo.nourrir("12h00")

ValueError: Cet animal ne mange de carottes.
```

En résumé :

- Les classes enfants doivent implémenter les méthodes et les propriétés définies dans la classe abstraite de base
- La classe abstraite de base ne peut pas être instanciée
- Nous utilisons @abstractmethod pour définir une méthode dans la classe abstraite et une combinaison de @property et @abstractmethod afin de définir une propriété abstraite.

Test avec la class Personne

Si on déclare la classe Personne abstraite

```
from abc import ABC

class Personne(ABC):
    # code précédent
```

Déclarons la méthode afficherDetails() comme abstraite dans Personne sans oublier d'importer abstractmethod

```
from abc import ABC, abstractmethod

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

En testant, le résultat est le même

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

Indexeur

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.

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 listePersonnes[i]

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

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]
```

Dans main.py

```
from listepersonnes 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
```

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
```

Dans main.py

```
from listepersonnes 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
```

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)
```

Dans main.py

```
from listepersonnes 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
```

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]

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

Considérons la liste suivante

liste = [2, 3, 8, 5]

Pour parcourir la liste, plusieurs solutions possibles

utiliser une boucle while ou for

utiliser un itérateur

Déclarons l'itérateur

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

Pour demander la première valeur

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

Ou en plus simple

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

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

Pour résoudre le problème précédent

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

Pour qu'une classe soit itérable, il faut implémenter les deux 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
```

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
```

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

```
from listepersonnes 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

Exercice

Ajoutez les méthodes__iter__() et __next__() dans ListePersonnes pour qu'elle soit itérable.

Solution

```
class ListePersonnes:
    def __init__(self, personnes: list) → None :
        self.__personnes = personnes
        self.__indice = 0

    def __iter__(self):
        return self

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

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

Opérateurs de comparaison et méthodes à implémenter

- $==$: __eq__()
- $!=$: __ne__()
- $\geq$: __ge__()
- $\leq$: __le__()
- $>$: __gt__()
- $<$: __lt__()

Opérateurs arithmétiques et méthodes à implémenter

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

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
```

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

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

```
def __eq__(self,p:"Personne")
→ bool :    p.__nom == self.__nomandp.__prenom == self.
      __prenom and p.__.num == self.__.num
```

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
```

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

```
def __gt__(self, other: "Personne"):
    return self.__num f ≤ other.__num
```

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
```

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

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

testons cela dans main.py

```
from enseignant import Enseignant

enseignant = Enseignant(300, 'dalton', 'jack', 1700)
print(enseignant)
# affiche 300 jack dalton 1700
print(enseignant + 200)
# affiche 300 jack dalton 1900
```

Généricité

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

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 ype)

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")
```

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")
```