mardi 18 novembre 2025

INFORMATIQUE - 3ième année

R5.Devcloud.07

Développement de microservices

DataClass, Pydantic, FastAPI, Docker

Python 3.14

(Fascicule N° 1/2)

Prérequis:

M1105 : Base des systèmes d'exploitation (OS)	**Scripting Shell (Windows, Bash: Linux)**
M2102 : Administration système	**Scripting Shell**
M1207 : Bases de la programmation	**Python**
M2207, M309, M308 : Programmation Orientée Objet	**Java**
M1106 et M2105 : Développement WEB	**HTML, CSS, JS, PHP**

http://www.python.org

Bibliothèque http://docs.python.org/py3k/library/index.html

https://docs.python.org/3.14

jean-claude.nunes@univ-rennes1.fr

Définition de fonction avec passage d'aguments

Définition de fonction avec passage d'aguments et typage

- On précise :
  - les types d'arguments de la fonction (Déclaration de variables),
  - le type de la valeur retournée.

```
#!/usr/bin/env python3

def age_discount(age: int, is_premium: bool, price: float) -> float:
    if age >= 60:
        discount = 0.15
    elif is_premium :
        discount = 0.10
    else:
        discount = 0.05

    return price  * (1 - discount)

discount = age_discount(70,False, 100)
print("Après rabais, le prix est de: "+str(discount) +" euros")

discount2 = age_discount(25,False, 100)
print("Après rabais, le prix est de: "+str(discount2) +" euros")


discount3 = age_discount(25,True, 100)
print("Après rabais, le prix est de: "+str(discount3) +" euros")
```

Définition d'une fonction avec trois arguments d'entrée et retournant un float

remise de 15%

remise de 10%

remise de 5%

Appel de la fonction avec 3 arguments un int, un booléen et un string

Appel de la fonction avec 3 arguments un int , un booléen et un string

Résultat:

```
Après rabais, le prix est de: 85.0 euros
Après rabais, le prix est de: 95.0 euros
Après rabais, le prix est de: 90.0 euros
```

Classe en Python

Définition de classe et instanciation

☞ **Classe et instanciation**

Définition d'une classe

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

pta = Point(1,2)
ptb = Point(3,4)
print("pta : x =", pta.x, "y =", pta.y)
print(f"ptb :  ({ptb.x}, {ptb.y})")

print(type(pta))

print("ptb est une instance de la Classe Point: ",isinstance(ptb, Point))
```

Définition du constructeur avec 2 arguments
`self` n'est pas considéré comme un argument
(équivalent à `this` en Java)

`self` correspond à ***this*** en java

arguments: variables locales

variables d'instances

Instanciation (appel du constructeur avec 2 arguments)

`pta.x` : accès à la variable la instance `x` de l'objet `pta`

interprète comme la valeur de la variable d'instance de l'objet

Résultat:

```
pta : x = 1 y = 2
ptb :  (3, 4)
<class '__main__.Point'>
ptb est une instance de la Classe Point:  True
```

Classe en Python

Définition de classe et instanciation

☞ **Classe et instanciation**

```
#!/usr/bin/env python3

class Student:

    def __init__(self,name,email,age,weight):
        self.name = name
        self.email = email
        self.age = age
        self.weight = weight

user8 = Student("Pablo","pablo@univ-rennes.fr",19,75.4)

print("Email of user8: "+user8.email)

print("age of user8: "+str(user8.age))

print("weight of user8: "+str(user8.weight))
```

Définition d'une classe

Définition du constructeur

self correspond à *this* en java

4 arguments lors de l'instanciation

Instanciation (appel du constructeur avec arguments)

accès à une variable d'instance

Résultat:

```
Email of user8: pablo@univ-rennes.fr
age of user8: 19
weight of user8: 75.4
```

Classe en Python

Définition de classe

☞ **Classe et instanciation**

```python
class Compte:
    banque = "Banque Populaire"

    def __init__(self, prenom, nom):
        self.prenom = prenom
        self.nom = nom

compte1 = Compte("Alice", "Dupont")
compte2 = Compte("Bob", "Martin")

print(f"{compte1.prenom} {compte1.nom} est à la {compte1.banque}.")
print(f"{compte2.prenom} {compte2.nom} est à la {compte2.banque}.")
```

Définition d'une classe

création d'une variable d'instance et initialisation

Définition du constructeur avec 2 arguments `self` n'est pas considéré comme un argument (équivalent à `this` en Java)

Instanciation (appel du constructeur avec 2 arguments)

Résultat:

```
Alice Dupont est à la Banque Populaire.
Bob Martin est à la Banque Populaire.
```

Supprimer une variable d'instance ou attribut : *del*

☞ **supprimer un attribut grâce au mot-clé del.**

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

pta = Point(1,2)

print(f"pta :  ({pta.x}, {pta.y})")

del pta.x
print(f"pta :  ({pta.x}, {pta.y})")
```

supprime la variable d'isntance x

Résultat:

```
classes_prog2.py", line 11, in <module>
   print(f"pta :  ({pta.x}, {pta.y})")
                    ^^^^^
AttributeError: 'Point' object has no attribute 'x'
```

Définition de méthode

☞ **Définition de méthode**

Définition d'une classe

Définition du constructeur

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def deplace(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

a = Point(1,2)
b = Point(3,4)
print("a : x =", a.x, "y =", a.y)
print("b : x =", b.x, "y =", b.y)

a.deplace(3, 5)
b.deplace(-1, -2)
print("a : x =", a.x, "y =", a.y)
print("b : x =", b.x, "y =", b.y)
```

Définition de la méthode avec 2 arguments, *self* n'est pas considéré comme un argument (équivalent à *this* en Java)

Instanciation (appel du constructeur avec arguments)

accès à une variable d'instance

Appel de la méthode avec 2 arguments

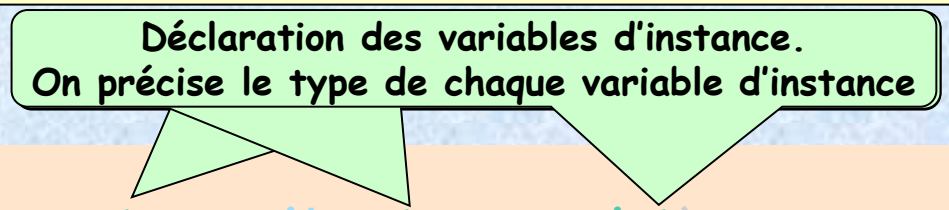
Résultat:

```
a : x = 1 y = 2
b : x = 3 y = 4
a : x = 4 y = 7
b : x = 2 y = 2
```

Classe en Python

Déclaration de variable d'instance

☞ **Déclaration de variable d'instance**




```
class User:
    def __init__(self, name: str, email: str, age: int):
        self.name = name
        self.email = email
        self.age = age

user128 = User( name="Pablo", email="pablo@univ-rennes.fr", age=18)
print(user128.email)
print(user128.age)
```

Résultat:

```
pablo@univ-rennes.fr
18
```

Classe en Python

Attributs ou méthodes protégés

☞ **Ces attributs ou méthodes sont destinés à être utilisés uniquement à l'intérieur de la classe et de ses classes dérivées. Un underscore unique (_) est utilisé avant leur nom.**

```
class Personne:
    def __init__(self, nom, prenom):
        self._nom = nom
        self._prenom = prenom

    def _get_nom(self):
        return self._nom

pers1 = Personne("Robert","Julien")
print(f"({pers1._nom},{pers1._prenom})")

print(pers1._get_nom())
```

variables d'instance protégées.

définition de méthode protégée

accès aux variables d'instance protégées.

définition de méthode protégée

Résultat:

```
(Robert,Julien)
Robert
```

Classe en Python

Variables d'instances privées

☞ **leur nom doit débuter par _ _ (deux fois le symbole underscore _)**

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def deplace(self, dx, dy):
        self.__x = self.__x + dx
        self.__y = self.__y + dy

    def affiche(self):
        print("abscisse =", self.__x, "ordonnee =", self.__y)

pta = Point(2, 4)
print(f"({pta.__x},{pta.__y})")

pta.affiche()
pta.deplace(1, 3)
pta.affiche()
```

variables d'instance privées.

Pour accéder aux données membres, on doit passer par des appels de méthode.

Résultat:

```
classes_prog5.py", line 14, in <module>
    print(f"({pta.__x},{pta.__y})")
                ^^^^^^
AttributeError: 'Point' object has no attribute
'__x'

abscisse = 2 ordonnee = 4
abscisse = 3 ordonnee = 7
```

Classe en Python

Variables d'instances privées

```
class CompteBancaire:
        def __init__(self, solde):
                self.__solde = solde

        def deposer(self, montant):
                if montant > 0:
                        self.__solde += montant

        def retirer(self, montant):
                if 0 < montant <= self.__solde:
                        self.__solde -= montant

        def afficher_solde(self):
                return self.__solde


compte = CompteBancaire(1000)
compte.deposer(500)
compte.retirer(200)
print(compte.afficher_solde())
```

variables d'instance privées.

Méthode publique pour accéder à l'attribut privé

Méthode publique pour accéder à l'attribut privé

Résultat:

1300

Classe en Python

Getter (Accesseur), Setter (mutateur)

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def get_x(self):
        return self.__x

    def set_x(self, x):
        self.__x = x

    def get_y(self):
        return self.__y

    def set_y(self, y):
        self.__y = y

a = Point(3, 7)
print("a : abscisse =", a.get_x())
print("a : ordonnee =", a.get_y())
a.set_x(6)
a.set_y(10)
print("a : abscisse =", a.get_x())
print("a : ordonnee =", a.get_y())
```

Résultat:

```
a : abscisse = 3
a : ordonnee = 7
a : abscisse = 6
a : ordonnee = 10
```

Classe en Python

Getter (Accesseur), Setter (mutateur)

```
class CompteBancaire:
        def __init__(self, solde):
                self.__solde = solde  # Attribut privé

        def get_solde(self):
                return self.__solde

        def set_solde(self, montant):
                if montant >= 0:
                        self.__solde = montant
                else:
                        print("Le solde ne peut pas être négatif.")


compte = CompteBancaire(1000)
print(compte.get_solde())
compte.set_solde(1500)
print(compte.get_solde())
```

Résultat:

```
1000
1500
```

Héritage

Classe mére

```
class Vehicule:
    def __init__(self, marque=None, vitesse_initiale=0):
        self.marque = marque
        self._vitesse = vitesse_initiale

    def vitesse(self):
        return self._vitesse

    def accelerer(self, delta_vitesse):
        self._vitesse += delta_vitesse

    def ralentir(self, delta_vitesse):
        self._vitesse -= delta_vitesse

class Voiture(Vehicule):
    def __init__(self, marque=None, vitesse_initiale=0, klaxon="tût tût !"):
        super().__init__(marque, vitesse_initiale)
        self.klaxon = klaxon

    def klaxonner(self):
        print(self.klaxon)

v = Voiture("Honda", 180.0)
print(v.marque)
print(v.vitesse)
v.accelerer(10)
print(v.vitesse)
v.klaxonner()
```

Classe fille de la classe mére *Vehicule* (pas de *extends*)

Résultat:

```
Honda
180.0
190.0
tût tût !
```

Polymorphisme ou redéfinition de variables d'instance

```
class Vehicule:
    roues = 4
    moteur = True

class Moto(Vehicule):
    roues = 2

class Velo(Vehicule):
    roues = 2
    moteur = False
    pedales = True

voiture = Vehicule()
print(f"Roues voiture : {voiture.roues}, Moteur voiture : {voiture.moteur}")

moto = Moto()
print(f"Roues moto : {moto.roues}, Moteur moto : {moto.moteur}")

velo = Velo()
print(f"Roues vélo : {velo.roues}, Moteur vélo : {velo.moteur}, Pédales vélo :
{velo.pedales}")
```

Classe mére

Attributs de classe (ou statique)

Classe fille

redéfinition d'Attributs de classe

Classe fille

redéfinition d'Attributs de classe

redéfinition d'Attributs de classe

Résultat:

```
Roues voiture : 4, Moteur voiture : True
Roues moto : 2, Moteur moto : True
Roues vélo : 2, Moteur vélo : False, Pédales vélo : True
```

Polymorphisme ou redéfinition de variables d'instance

Exécution de Notebook sous Colab de Google

Heritage_prog1.ipynb

Fichier Modifier Affichage Insérer Exécution Outils Aide

Partager

Commandes + Code + Texte Tout exécuter

RAM Disque

[8] 0 s

```
class Vehicule:
  roues = 4
  moteur = True

class Moto(Vehicule):
  roues = 2

class Velo(Vehicule):
  roues = 2
  moteur = False
  pedales = True

voiture = Vehicule()
print(f"Roues voiture : {voiture.roues}, Moteur voiture : {voiture.moteur}")

moto = Moto()
print(f"Roues moto : {moto.roues}, Moteur moto : {moto.moteur}")

velo = Velo()
print(f"Roues vélo : {velo.roues}, Moteur vélo : {velo.moteur}, Pédales vélo : {velo.pedales}")
```

```
Roues voiture : 4, Moteur voiture : True
Roues moto : 2, Moteur moto : True
Roues vélo : 2, Moteur vélo : False, Pédales vélo : True
```

Polymorphisme ou redéfinition de méthodes

```
class Animal:
    def sexprime(self):
        pass

class Oiseau(Animal):
    def sexprime(self):
        print("Piou piou!")

class Chien(Animal):
    def sexprime(self):
        print("Wouf!")

class Chat(Animal):
    def sexprime(self):
        print("Miaou!")

def faire_sexprimer(animal):
    animal.sexprime()

faire_sexprimer(Oiseau())
faire_sexprimer(Chien())
faire_sexprimer(Chat())
```

Définition de la méthode au sein de la classe mère

redéfinition de la méthode

redéfinition de la méthode

redéfinition de la méthode

Résultat:

```
Piou piou!
Wouf!
Miaou!
```

Polymorphisme ou redéfinition de méthodes

```
class Vehicule:
    def __init__(self, marque):
        self.marque = marque
    def se_deplacer(self):
        print(f"Le véhicule {self.marque} se déplace.")

class Voiture(Vehicule):
    def __init__(self, marque, modèle, nb_roues=4):
        super().__init__(marque)
        self.modèle = modèle
        self.nb_roues = nb_roues
    def se_deplacer(self):
        print(f"La voiture {self.marque} {self.modèle} roule sur {self.nb_roues} roues.")

ma_voiture = Voiture("Porsche", "Cayenne")
ma_voiture.se_deplacer()

class Moto(Vehicule):
    def __init__(self, marque, modèle, nb_roues=2):
        super().__init__(marque)
        self.modèle = modèle
        self.nb_roues = nb_roues
    def se_deplacer(self):
        print(f"La moto {self.marque} {self.modèle} roule sur {self.nb_roues} roues.")

ma_voiture = Moto("Yamaha", "MT-07")
ma_voiture.se_deplacer()
```

Définition de la méthode au sein de la classe mère

redéfinition de la méthode

redéfinition de la méthode

Résultat:

```
La voiture Porsche Cayenne roule sur 4 roues.
La moto Yamaha MT-07 roule sur 2 roues.
```

Polymorphisme ou redéfinition de méthodes

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def se_presenter(self):
        print(f"Je suis un animal du nom de {self.nom}.")

class Chien(Animal):
    def __init__(self, nom, couleur):
        super().__init__(nom)
        self.couleur = couleur

    def se_presenter(self):
        super().se_presenter()
        print(f"Je suis un chien de couleur {self.couleur}.")

mon_chien = Chien("Milo", "marron")
mon_chien.se_presenter()
```

Définition de la méthode au sein de la classe mère

appel explicite du constructeur de la classe mère

redéfinition de la méthode

appel de la méthode de la classe mère

Résultat:

```
Je suis un animal du nom de Milo.
Je suis un chien de couleur marron.
```

Héritage multiple

```
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def manger(self):
        print(f"{self.nom} est en train de manger.")

class Predateur:
    def chasser(self):
        print("Je chasse.")

class Proie:
    def fuir(self):
        print("Je fuis.")

class Lion(Animal, Predateur):
    pass

class Lapin(Animal, Proie):
    pass

simba = Lion("Simba")
simba.manger()
simba.chasser()

panpan = Lapin("Panpan")
panpan.manger()
panpan.fuir()
```

classe mère

classe mère

classe mère

Classe fille hérite de deux classes mère

Classe fille hérite de deux classes mère

Résultat:

```
Simba est en train de manger.
Je chasse.
Panpan est en train de manger.
Je fuis.
```

Méthodes spéciales

manipuler les variables d'instance d'un objet

- manipuler les attributs d'un objet grâce aux fonctions
- ***hasattr(o, n)*** savoir si un objet dispose d'un attribut,
- ***getattr(o, n)*** récupérer la valeur d'un attribut
- ***setattr(o, n, v)*** modifier un attribut
- ***delattr(o, n)*** supprimer un attribut

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

pta = Point(100,2000)

print(f"pta :  ({pta.x}, {pta.y})")
print(hasattr(pta, "x"))

setattr(pta, "x", 800)
print(f"pta :  ({pta.x}, {pta.y})")
print(hasattr(pta, "x"))
print(getattr(pta, "x"))

delattr(pta, "x")
print(hasattr(pta, "x"))
```

Résultat:

```
pta :  (100, 2000)
True

pta :  (800, 2000)
True
800

False
```

Méthodes spéciales

manipuler les variables d'instance d'un objet

☞ **manipuler les attributs d'un objet grâce aux fonctions**

```
class Produit:
    def __init__(self, nom, prix):
        self.nom = nom
        self.prix = prix
        if prix > 50:
            self.reduction = 0.10 # 10%

produit_luxe = Produit("Parfum", 120)
produit_standard = Produit("Gel douche", 4.5)

def afficher(produit):
    print(f"Produit : {produit.nom}, Prix : {produit.prix}€")

    if hasattr(produit, 'reduction'):
        prix_reduit = produit.prix * (1 - produit.reduction)
        print(f"  -> Prix avec réduction : {prix_reduit:.2f}€")

afficher(produit_luxe)

afficher(produit_standard)
```

Remise à cette condition

Vérification si l'attribut 'reduction' existe avant de l'appliquer

Résultat:

```
Produit : Parfum, Prix : 120€
  -> Prix avec réduction : 108.00€
Produit : Gel douche, Prix : 4.5€
```

Méthodes spéciales

manipuler les variables d'instance d'un objet

☞ **manipuler les attributs d'un objet grâce aux fonctions**

```
class Utilisateur:
    def __init__(self, nom, permissions=None):
        self.nom = nom
        if permissions:
            for perm in permissions:
                setattr(self, f"peut_{perm}", True)

admin = Utilisateur("Alice", permissions=["modifier", "supprimer"])
editeur = Utilisateur("Bob", permissions=["modifier"])

action_requise = input("Quelle action vérifier (ex: modifier, supprimer) ? ")
permission_a_verifier = f"peut_{action_requise}"

def peut_executer(user, permission):
    if hasattr(user, permission):
        print(f"Oui, {user.nom} a la permission : '{permission}'")
    else:
        print(f"Non, {user.nom} n'a PAS la permission : '{permission}'")

peut_executer(admin, permission_a_verifier)
peut_executer(editeur, permission_a_verifier)
```

utiliser *setattr()* pour créer dynamiquement les attributs

Création d'utilisateurs avec des permissions différentes

L'action à vérifier est demandée à l'utilisateur

Construit dynamiquement "peut_supprimer", "peut_modifier", etc.

On utilise la variable 'permission' pour vérifier l'attribut

Si l'utilisateur saisit "supprimer", le programme affiche :
Oui, Alice a la permission : 'peut_supprimer'
Non, Bob n'a PAS la permission : 'peut_supprimer'

Résultat:

```
Quelle action vérifier (ex: modifier, supprimer) ? modifier
Oui, Alice a la permission : 'peut_modifier'
Oui, Bob a la permission : 'peut_modifier'
```

Méthodes spéciales

Définition de la documentation d'une classe

☞ **Définition de méthode __doc__**

__doc__

Commentaire spécial comme Javadoc

```
class Point:
    def __init__(self, x, y):
        """Point class represents a point
in 2D space with x and y coordinates."""
        self.x = x
        self.y = y

pta = Point(100,2000)

print(f"pta :  ({pta.x}, {pta.y})")

help(pta)

pta.__doc__
```

Résultat:

```
pta :  (100, 2000)

Help on Point in module __main__ object:

class Point(builtins.object)
 |  Point(x, y)
 |
 |  Methods defined here:
 |
 |  __init__(self, x, y)
 |      Point class represents a point in
2D space with x and y coordinates.
 |
 |  ----------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables
 |
 |  __weakref__
 |      list of weak references to the object
```

Méthodes spéciales

Redéfinition de la méthode `str()`

☞ **Définition de la méthode `str()`**

`__str__`

```
class Employer:
    def __init__(self, age, name, male):
        self.age = age
        self.name = name
        self.male = male

    def __str__(self):
        return f"Je suis un Employer avec age={self.age}, name={self.name} and
male={self.male}."

emp12=Employer(62, "Robert", True)
print(str(emp12))

emp45=Employer(23, "Gaëlle", False)
print(str(emp45))
```

Définition d'une classe

Définition du constructeur

redéfinition de la méthode `str()` pour convertir un objet en string

Instanciation
(appel du constructeur avec arguments)

appel de la méthode `str()`

Résultat:

```
Je suis un Employer avec age=62, name=Robert and male=True.
Je suis un Employer avec age=23, name=Gaëlle and male=False.
```

Méthodes spéciales

Définition de la méthode *repr()*

☞ **Définition de la méthode** *__repr__()*

__repr__

```
class Employer:
    def __init__(self, age, name, male):
        self.age = age
        self.name = name
        self.male = male

    def __repr__(self):
        return f"Employer({self.age}, name={self.name} and male={self.male})"

emp12=Employer(62, "Robert", True)
print(repr(emp12))

emp45=Employer(23, "Gaëlle", False)
print(repr(emp45))
```

Définition d'une classe

Définition du constructeur

redéfinition de la méthode *repr()* pour convertir un objet en string

Instanciation
(appel du constructeur avec arguments)

appel de la méthode *repr()*

Résultat:

Employer(62, name=Robert and male=True)
Employer(23, name=Gaëlle and male=False)

Méthodes spéciales

Redéfinition de l'opérateur ==

☞ **Définition de la méthode __eq__()**

__eq__

```
class Employer:
    def __init__(self, age, nom):
        self.age = age
        self.nom = nom

    def __eq__(self,autre):
        return self.nom == autre.nom and self.age == autre.age

emp1=Employer(62, "Robert")
print(f"pta :  ({emp1.nom}, {emp1.age})")

emp2=Employer(23, "Gaëlle")
print(f"emp2 :  ({emp2.nom}, {emp2.age})")

emp3=Employer(23, "Gaëlle")
print(f"emp3 :  ({emp3.nom}, {emp3.age})")

print(emp1 == emp2)
print(emp2 == emp3)
```

Définition d'une classe

Définition du constructeur

redéfinition de l'opérateur == au travers de la méthode __eq__()

appel implicite de la méthode __eq__()

Résultat:

```
emp2 :  (Gaëlle, 23)
emp3 :  (Gaëlle, 23)
False
True
```

Méthodes spéciales

Redéfinition de l'opérateur ==

☞ Définition de la méthode __eq__()

__eq__

```
#!/usr/bin/env python3
class Voiture:

    def __init__(self, marque, modele, annee, couleur, kilometrage):
        self.marque = marque
        self.modele = modele
        self.annee = annee
        self.couleur = couleur
        self.kilometrage = kilometrage

    def __repr__(self):
        return f"Voiture(marque='{self.marque}', modele='{self.modele}', annee={self.annee},
couleur='{self.couleur}', kilometrage={self.kilometrage} )"

    def __eq__(self, other):
        if not isinstance(other, Voiture):
            return False
        return (self.marque == other.marque and
                self.modele == other.modele and
                self.annee == other.annee and
                self.couleur == other.couleur and
                self.kilometrage == other.kilometrage)

voiture1 = Voiture("Toyota", "Celica", 2024, "Blanche", 25000)
voiture2 = Voiture("Toyota", "Yaris", 2010, "Noire", 150000)
print("Resultat de comparaison des voitures: "+str(voiture1==voiture2))

voiture3 = Voiture("Toyota", "Yaris", 2010, "Noire", 150000)
print("Resultat de comparaison des voitures: "+str(voiture2==voiture3))
```

Définition d'une classe

Définition du constructeur avec variables d'instance.

Definition de la méthode de comparaison d'objets appelées en utilisant l'opérateur == .

isinstance() équivalent à l'opérateur instanceof du java

Appel explicite de la méthode __eq__() par l'opérateur de comparaison ==

Appel explicite de la méthode __eq__() par l'opérateur de comparaison ==

Résultat:

```
Resultat de comparaison des voitures: False
Resultat de comparaison des voitures: True
```

Exemple 1/2

Redéfinition de l'itérateur

☞ **Définition de la méthode `__iter__()`**

`__iter__`

```
class Employer:
    def __init__(self, age, nom):
        self.age = age
        self.nom = nom

class GestionnaireEmployers:
    def __init__(self):
        self.employers = []
        self._index = 0

    def ajouter_employer(self, employer):
        self.employers.append(employer)

    def __iter__(self):
        self._index = 0
        return self

    def __next__(self):
        if self._index < len(self.employers):
            employer = self.employers[self._index]
            self._index += 1
            return employer
        else:
            raise StopIteration
```

Définition d'une classe

Définition du constructeur

Définition d'une classe

Définition du constructeur

définition la méthode `ajouter_employer()`

définition la méthode `__iter__()`

définition de la méthode `__next__()`

Exemple 2/2

Redéfinition de l'itérateur

☞ **Définition de la méthode `__iter__()`**

`__iter__`

```
emp1=Employer(62, "Robert")
emp2=Employer(23, "Gaëlle")

gestionnaire = GestionnaireEmployers()

gestionnaire.ajouter_employer(emp1)
gestionnaire.ajouter_employer(emp2)
gestionnaire.ajouter_employer(Employer(30,"Etienne"))

for travailleur in gestionnaire:
    print(f"trav :  ({travailleur.nom}, {travailleur.age})")
```

Appel du constructeur

Appel du constructeur

Appel de la méthode

appel implicite de la méthode `__iter__()`

Résultat:

```
trav :  (Robert, 62)
trav :  (Gaëlle, 23)
trav :  (Etienne, 30)
```

Exemple 1/2

Redéfinition de l'opérateur +

☞ **Définition de la méthode `__add__()`**

`__add__`

```
class Employer:
    def __init__(self, age, nom):
        self.age = age
        self.nom = nom

class GestionnaireEmployers:
    def __init__(self):
        self.employers = []
        self._index = 0

    def ajouter_employer(self, employer):
        self.employers.append(employer)

    def __iter__(self):
        self._index = 0
        return self

    def __next__(self):
        if self._index < len(self.employers):
            employer = self.employers[self._index]
            self._index += 1
            return employer
        else:
            raise StopIteration

    def __add__(self, autre):
        nouveau_gestionnaire = GestionnaireEmployers()
        nouveau_gestionnaire.employers = self.employers + autre.employers
        return nouveau_gestionnaire
```

Définition d'une classe

Définition du constructeur

Définition d'une classe

Définition du constructeur

définition la méthode `ajouter_employer()`

définition la méthode `__iter__()`

définition de la méthode `__next__()`

redéfinition de l'opérateur + par la méthode `__add__()`

Méthodes spéciales — Exemple 1/2

Redéfinition de l'opérateur +

☞ **Définition de la méthode *__add__()***

___add___

```
entreprise1 = GestionnaireEmployers()
emp1=Employer(62, "Robert")
emp2=Employer(23, "Gaëlle")
entreprise1.ajouter_employer(emp1)
entreprise1.ajouter_employer(emp2)

entreprise2 = GestionnaireEmployers()
entreprise2.ajouter_employer(Employer(30,"Etienne"))
entreprise2.ajouter_employer(Employer(45,"Sophie"))

gestionnaire_combiné = entreprise1 + entreprise2

for travailleur in gestionnaire_combiné:
    print(f"trav :  ({travailleur.nom}, {travailleur.age})")
```

Appel du constructeur

Appel du constructeur

Appel de la méthode

Appel du constructeur

Appel de la méthode

appel implicite de la méthode __add__() en utilisant l'opérateur +

Résultat:

```
trav :  (Robert, 62)
trav :  (Gaëlle, 23)
trav :  (Etienne, 30)
trav :  (Sophie, 45)
```

Méthodes spéciales

Définition de la méthode *del*

__del__

☞ **Définition de la méthode *__del__()***

```
class Employer:
    def __init__(self, age, name, male):
        self.age = age
        self.name = name
        self.male = male

    def __del__(self):
        print(f"Employer({self.age}, name={self.name} and male={self.male} is being destroyed.")

emp12=Employer(62, "Robert", True)
print(f"Employer : ({emp12.name}, {emp12.age}, { emp12.male})")

emp45=Employer(23, "Gaëlle", False)
print(f"Employer : ({emp45.name}, {emp45.age}, { emp45.male})")

chaine= "Some text data"
print(f"Chaine : {chaine}")
#chaine
emp12=None
print("emp12 a été mis à None. Puis, ensuite, le ramasse-miettes va détruire l'autre objet Employer.")
del emp45  # Explicitly deleting the object
```

Définition d'une classe

Définition du constructeur

redéfinition de la méthode *del* au travers de la méthode *__del__()*
C'est le destructeur.

appel implicite de la méthode *__del__()* pour supprimer l'objet et libérer les ressources

appel implicite de la méthode *__del__()* pour supprimer l'objet et libérer les ressources

Résultat:

```
Employer : (Robert, 62, True)
Employer : (Gaëlle, 23, False)
Chaine : Some text data
Employer(62, name=Robert and male=True is being destroyed.
emp12 a été mis à None. Puis, ensuite, le ramasse-miettes va détruire l'autre objet Employer.
Employer(23, name=Gaëlle and male=False is being destroyed.
```

Méthodes spéciales

Redéfinition de la méthode *call*

___call___

☞ **Définition de la méthode *__call__()***

```
class Test:
    def __init__(self, value=0):
        print('constructor called')
        self.value = value
    def __call__(self, x):
        print('call method called')
        return self.value + x

T = Test(50)
print(f"{T.value}")
print(T(10))
```

Définition d'une classe

Définition du constructeur

définition de la méthode *__call__()*

Appel du constructeur

appel de la méthode *__call__()*

Résultat:

```
constructor called
50
call method called
60
```

Méthodes spéciales

Autres fonctions utiles

☞ ***__contains__*** : **Pour vérifier l'appartenance avec l'opérateur in.**

```
class MaListAges:
    def __init__(self, ages):
        self.ages = ages

    def __contains__(self, item):
        return item in self.ages

my_list_ages = MaListAges([18, 17, 20, 22, 25, 30, 19])
print(22 in my_list_ages)
print(6 in my_list_ages)
```

Redéfinition de l'opérateur *in*

Appel de la méthode *__contains__()*

Résultat:

```
True
False
```

Méthodes spéciales

Autres fonctions utiles

- ☞ *__len__* : Pour définir le comportement de la fonction *len()*.
- ☞ *__getitem__* : Pour accéder aux éléments via l'indexation.

```
class GestionnaireEmployers:
    def __init__(self):
        self.employers = []
        self._index = 0

    def ajouter_employer(self, employer):
        self.employers.append(employer)

    def __iter__(self):
        self._index = 0
        return self

    def __next__(self):
        if self._index < len(self.employers):
            employer = self.employers[self._index]
            self._index += 1
            return employer
        else:
            raise StopIteration

    def __len__(self):
        return len(self.employers)

    def __getitem__(self, index):
        return self.employers[index]
```

redéfinition de la méthode *len()* au travers de la méthode *__len__()*

définition de la méthode *__getitem__()*

Méthodes spéciales

Autres fonctions utiles

```
gestionnaire = GestionnaireEmployers()
emp1=Employer(62, "Robert")
emp2=Employer(23, "Gaëlle")

gestionnaire.ajouter_employer(emp1)
gestionnaire.ajouter_employer(emp2)
gestionnaire.ajouter_employer(Employer(30,"Etienne"))

for travailleur in gestionnaire:
    print(f"trav :  ({travailleur.nom}, {travailleur.age})")

print(len(gestionnaire))
print(gestionnaire[0])
```

Appel de la méthode *len()*

Appel implicite de la méthode *__getitem_()*

Résultat:

```
trav :  (Robert, 62)
trav :  (Gaëlle, 23)
trav :  (Etienne, 30)
3
Je suis un Employer avec age=62, name=Robert.
```

Les classes exceptions

- une exception est un objet qui est directement ou indirectement une instance de la classe ***BaseException***.
- ***BaseException***
  - ***SystemExit***
  - ***KeyboardInterrupt***
  - ***GeneratorExit***
  - ***Exception***
    - ***StopIteration***
    - ***StopAsyncIteration***
    - ***ArithmeticError***
      - ***FloatingPointError***
      - ***OverflowError***
      - ***ZeroDivisionError***
    - ***AssertionError***
    - ***AttributeError***
    - ***BufferError***
    - ***EOFError***
    - ***ImportError***
      - ***ModuleNotFoundError***
    - ***LookupError***
      - ***IndexError***
      - ***KeyError***
    - ***MemoryError***

Les classes exceptions

- *NameError*
  - *UnboundLocalError*
- *OSError*
  - *BlockingIOError*
  - *ChildProcessError*
  - *ConnectionError*
    - *BrokenPipeError*
    - *ConnectionAbortedError*
    - *ConnectionRefusedError*
    - *ConnectionResetError*
  - *FileExistsError*
  - *FileNotFoundError*
  - *InterruptedError*
  - *IsADirectoryError*
  - *NotADirectoryError*
  - *PermissionError*
  - *ProcessLookupError*
  - *TimeoutError*
- *ReferenceError*
- *RuntimeError*
  - *NotImplementedError*
  - *RecursionError*

Les classes exceptions

- *SyntaxError*
  - *IndentationError*
    - *TabError*
- *SystemError*
- *TypeError*
- *ValueError*
  - *UnicodeError*
    - *UnicodeDecodeError*
    - *UnicodeEncodeError*
    - *UnicodeTranslateError*
- *Warning*
  - *DeprecationWarning*
  - *PendingDeprecationWarning*
  - *RuntimeWarning*
  - *SyntaxWarning*
  - *UserWarning*
  - *FutureWarning*
  - *ImportWarning*
  - *UnicodeWarning*
  - *BytesWarning*
  - *ResourceWarning*

Les classes exceptions

Les classes exceptions

- une exception est un objet qui est directement ou indirectement une instance de la classe ***BaseException***.
- il est très simple de créer ses propres exceptions.
- Il est recommandé de créer des exceptions en héritant de ***Exception*** ou d'une classe héritant de ***Exception***.
- ***Exception*** est une classe qui hérite de ***BaseException***.
- Pour simplifier l'implémentation, ***BaseException*** définit l'attribut ***args*** qui contient tous les paramètres passés au constructeur.

```
class MonException(Exception):
    pass

try:
    raise MonException("Ma première exception")
except MonException as e:
    print(e.args)
    print(e)
```

définition d'une classe exception qui hérite de *Exception*

bloc de surveillance des intructions succeptibles de déclencher une exception

Levée (déclenchement) explicite d'une exception

On ne doit la lever que si il y a un problème. Penser à mettre une condition avant! *Throw* en Java.

Capture d'une exception particulière

Résultat:

```
('Ma première exception',)
Ma première exception
```

Les classes exceptions

Les classes exceptions

☞ une exception est un objet qui est directement ou indirectement une instance de la classe ***BaseException***.

```
class CustomException(Exception):
    def __init__(self, value):
        self.parameter = value

    def __str__(self):
        return repr(self.parameter)
try:
    raise CustomException('My Useful Error Message!')
except CustomException as instance:
    print('Caught: ' + instance.parameter)
```

définition d'une classe exception qui hérite de *Exception*

Levée (déclenchement) explicite d'une exception. *Throw* en Java

On doit la lever que si il y a un problème. Penser à mettre une condition avant!

Capture d'une exception particulière

Résultat:

Caught: My Useful Error Message!

Les classes exceptions

1.5

Les classes exceptions

☞ une exception est un objet qui est une instance de la classe *BaseException* ou de ses descendantes.

définition d'une classe exception qui hérite de *Exception*

```
class TemperatureError(Exception):
    """Exception levée lorsque la température est invalide."""
    pass

def verifier_temperature(temp):
    if temp < -273.15:
        raise TemperatureError("Température ne peut être < au zéro absolu.")

try:
    verifier_temperature(-300)
except TemperatureError as e:
    print(f"Erreur : {e}")
```

Levée (déclenchement) explicite d'une exception. *Throw* en Java

bloc de surveillance des intructions succeptibles de déclencher une exception

Capture d'une exception particulière

Résultat:

Erreur : Température ne peut être < au zéro absolu.

Les classes exceptions

Les classes exceptions

☞ une exception est un objet qui est directement ou indirectement une instance de la classe ***BaseException***.

définition d'une classe exception qui hérite de *Exception*

```
class InvalidAgeException(Exception):
    "Raised when the input value is less than 18"
    pass

number = 18

try:
    input_num = int(input("Enter a number: "))
    if input_num < number:
        raise InvalidAgeException
    else:
        print("Eligible to Vote")

except InvalidAgeException:
    print("Exception occurred: Invalid Age")
```

bloc de surveillance des intructions succeptibles de déclencher une exception

Levée (déclenchement) explicite d'une exception

Capture d'une exception particulière

Résultat:

```
Enter your age: -25
Only positive integers are allowed
```

Les classes exceptions

Les classes exceptions

☞ une exception est un objet qui est directement ou indirectement une instance de la classe ***BaseException***.

définition d'une classe exception qui hérite de *Exception*

```
class NegativeNumberException(RuntimeError):
    def __init__(self, age):
        super().__init__()
        self.age = age

def enterage(age):
    if age < 0:
        raise NegativeNumberException('Only positive integers are allowed')

    if age % 2 == 0:
        print('Age is Even')

    else:
        print('Age is Odd')

try:
    num = int(input('Enter your age: '))
    enterage(num)
except NegativeNumberException:
    print('Only positive integers are allowed')
except:
    print('Something is wrong')
```

Levée (déclenchement) explicite d'une exception

bloc de surveillance des intructions succeptibles de déclencher une exception

Capture d'une exception particulière

Résultat:

```
Enter your age: -25
Only positive integers are allowed
```

Les classes exceptions

Les classes exceptions

☞ une exception est un objet qui est directement ou indirectement une instance de la classe ***BaseException***.

définition d'une classe exception qui hérite de *Exception*

```
class SalaryNotInRangeError(Exception):
    def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)

try:
    salary = int(input("Enter salary amount: "))
    if not 5000 < salary < 15000:
        raise SalaryNotInRangeError(salary)
except SalaryNotInRangeError:
    print('SalaryNotInRangeError: Salary is not in (5000, 15000) range')
```

Levée (déclenchement) explicite d'une exception

Capture d'une exception particulière

Résultat:

```
Enter salary amount: 2000
SalaryNotInRangeError: Salary is not in (5000, 15000) range
```

Décorateurs

Définition du wrapper

☞ définition d'une fonction avec un "emballage" (wrapper) contenant du code supplémentaire.

```
def simple_decorator(func):
    def wrapper():
        print("Before the function call")
        func()
        print("After the function call")
    return wrapper

@simple_decorator
def hello():
    print("Hello!")

@simple_decorator
def bonjour():
    print("Bonjour!")


hello()

print("---")

bonjour()
```

Définition d'un décorateur

Définition d'un wrapper

Appel de la fonction passée en argument du décorateur

application du décorateur à la fonction qui suit

Définition d'une fonction

application du décorateur à la fonction qui suit

Définition d'une fonction

Appel de fonction

Appel de fonction

Résultat:

```
Before the function call
Hello!
After the function call
---
Before the function call
Bonjour!
After the function call
```

Décorateurs

Décorer des fonctions avec arguments

☞ définition d'une fonction avec un "emballage" (wrapper) contenant du code supplémentaire.

```
def mon_decorateur(func):
    def wrapper(*args, **kwargs):
        print("Avant la fonction")
        result = func(*args, **kwargs)
        print("Après la fonction")
        return result
    return wrapper

@mon_decorateur
def bonjour(nom):
    nom = nom.capitalize()
    print(f"Bonjour, {nom} !")

@mon_decorateur
def age(annee_naissance):
    age = 2025 - annee_naissance
    print(f"Votre age est {age} !")

bonjour("albert")

print("---")

age(2000)
```

Définition d'un décorateur

Définition d'un wrapper

Appel de la fonction passée en argument du décorateur

application du décorateur à la fonction qui suit

Définition d'une fonction

Mettre en majuyscule la première lettre

application du décorateur à la fonction qui suit

Définition d'une fonction

Appel de fonction

Appel de fonction

@

Résultat:

```
Avant la fonction
Bonjour, Albert !
Après la fonction
---
Avant la fonction
Votre age est 25 !
Après la fonction
```

Décorateurs

Décorer des fonctions avec arguments

☞ définition d'une fonction avec un "emballage" (wrapper) contenant du code supplémentaire.

```
import time

def temps_exec(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Durée de {func.__name__}: {end - start:.5f} secondes")
        return result
    return wrapper

@temps_exec
def gros_calcul():
    time.sleep(3)
    print("Calcul terminé !")

gros_calcul()

print("---")

gros_calcul()
```

Définition d'un décorateur

Définition d'un wrapper

Appel de la fonction passée en argument du décorateur

On récupère le nom de la fonction

application du décorateur à la fonction qui suit

Définition d'une fonction

Simule un calcul lourd, par exemple : Appel d'une API!!!!!!

Appel de fonction

Appel de fonction

@

Résultat:

```
Calcul terminé !
Durée de gros_calcul: 3.00059 secondes
---
Calcul terminé !
Durée de gros_calcul: 3.00091 secondes
```

Décorateurs

Conserver la documentation malgré l'utilisation d'un décorateur

☞ **Problème: Le décorateur masque la documentation de la fonction moyenne**

```
def trace(func):
    def decorateur(*args, **kwargs):
        print("Début d'appel à", func.__name__)
        resultat = func(*args, **kwargs)
        print("Fin d'appel à", func.__name__)
        return resultat
    return decorateur

@trace
def moyenne(x, *args):
    """Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres."""
    nb = 1 + len(args)
    somme = x
    for y in args:
        somme += y
    return somme / nb

help(moyenne)

print(moyenne(10, 20, 30, 40))
```

Documentation de la fonction moyenne

Résultat:

```
Help on function decorateur in module __main__:

decorateur(*args, **kwargs)

Début d'appel à moyenne
Fin d'appel à moyenne
25.0
```

Le décorateur masque la documentation de la fonction moyenne

wraps

2.1

Conserver la documentation malgré l'utilisation d'un décorateur

☞ Conserve la documentation malgré l'utilisation d'un décorateur

```
from functools import wraps
def trace(func):
    @wraps(func)
    def decorateur(*args, **kwargs):
        print("Début d'appel à", func.__name__)
        resultat = func(*args, **kwargs)
        print("Fin d'appel à", func.__name__)
        return resultat
    return decorateur

@trace
def moyenne(x, *args):
    """Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres."""
    nb = 1 + len(args)
    somme = x
    for y in args:
        somme += y
    return somme / nb

help(moyenne)

print(moyenne(10, 20, 30, 40))
```

Commentaire spécial lié à la Documentation de la fonction moyenne

@

Documentation de la fonction moyenne

Résultat:

```
Help on function moyenne in module __main__:

moyenne(x, *args)
    Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres.

Début d'appel à moyenne
Fin d'appel à moyenne
25.0
```

Appliquer plusieurs décorateurs

☞ **ajouter plusieurs décorateurs sur une même fonction :**

```
def decorateur_1(func):
    def wrapper(*args, **kwargs):
        print("Décorateur 1 : Avant la fonction")
        result = func(*args, **kwargs)
        print("Décorateur 1 : Après la fonction")
        return result
    return wrapper

def decorateur_2(func):
    def wrapper(*args, **kwargs):
        print("Décorateur 2 : Avant la fonction")
        result = func(*args, **kwargs)
        print("Décorateur 2 : Après la fonction")
        return result
    return wrapper

@decorateur_1
@decorateur_2
def fonction3():
    print("Exécution de la fonction3")

fonction3()
```

1er décorateur

2d décorateur

1er décorateur

2d décorateur

Définition de fonction

Appel de fonction

@

Résultat:

```
Décorateur 1 : Avant la fonction
Décorateur 2 : Avant la fonction
Exécution de la fonction3
Décorateur 2 : Après la fonction
Décorateur 1 : Après la fonction
```

Appliquer plusieurs décorateurs

☞ **ajouter plusieurs décorateurs sur une même fonction :**

```
import time

def log_execution(func):
    def wrapper(*args, **kwargs):
        print(f"Appel de la fonction {func.__name__} avec {args} et {kwargs}")
        return func(*args, **kwargs)
    return wrapper

def chronometre(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Temps d'exécution de {func.__name__} : {end - start:.4f} secondes")
        return result
    return wrapper

@log_execution
@chronometre
def calcul(n):
    time.sleep(n)
    print("calcul terminé !")

calcul(2)
```

1er décorateur

2d décorateur

1er décorateur

2d décorateur

Définition de fonction

Appel de fonction

@

Résultat:

Appel de la fonction wrapper avec (2,) et {}
calcul terminé !
Temps d'exécution de calcul : 2.0008 secondes

wrapper 2.1

Décorateurs et Classes

☞ **ajouter plusieurs décorateurs sur une même fonction :**

```
def verifier_permissions(func):
    def wrapper(self, *args, **kwargs):
        if not self.est_admin:
            print("Accès refusé.")
            return
        return func(self, *args, **kwargs)
    return wrapper

def log_execution(func):
    def wrapper(*args, **kwargs):
        print(f"Exécution de {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

class Systeme:
    def __init__(self, est_admin):
        self.est_admin = est_admin

    @verifier_permissions
    @log_execution
    def supprimer_donnees(self):
        print("Données supprimées.")

utilisateur = Systeme(est_admin=False)
utilisateur.supprimer_donnees()

admin = Systeme(est_admin=True)
admin.supprimer_donnees()
```

1er décorateur

2d décorateur

1er décorateur

2d décorateur

Définition de fonction

Appel de fonction

@

Résultat:

Accès refusé.
Exécution de supprimer_donnees
Données supprimées.

@staticmethod 2.1

Décorateurs et Classes

☞ ajouter plusieurs décorateurs sur une même fonction :

```
class MathUtils:
    @staticmethod
    def addition(a, b, c, d):
        return a + b + c + d

print(MathUtils.addition(3, 5, 10, 2))
```

Précise que la méthode définie ensuite sera statique

Définition de fonction

Appel de fonction statique sans instanciation préalable

Résultat:

20

```
class Compteur:
    total = 0

    @classmethod
    def incrementer(cls):
        cls.total += 1

Compteur.incrementer()
Compteur.incrementer()
print(Compteur.total)
```

Précise que la méthode définie ensuite sera statique

Définition de fonction

Appel de fonction statique sans instanciation préalable

Résultat:

2

@property

2.1

Décorateurs et Getter/Setter

☞ **gestion des propriétés avec @property. Cela permet de définir des getters et setters sans appeler explicitement des méthodes.**

```
class Personne:
    def __init__(self, nom, prenom):
        self._nom = nom
        self._prenom = prenom

    @property
    def nom(self):
        return self._nom

    @nom.setter
    def nom(self, nouveau_nom):
        if not nouveau_nom:
            raise ValueError("Le nom ne peut pas être vide.")
        self._nom = nouveau_nom

p = Personne("Sophie", "Martin")
print(p.nom)

p.nom = "Durand"
print("({p.nom}, {p._prenom})")
```

Getter

Setter

Appel du constructeur

appel du getter

Modification du nom via le setter

Résultat:

```
2
```

DataClass

- permettent d'alléger le code.
- nous n'avons plus besoin d'écrire le code des fonctions *__init__*, *__repr__*, et *__eq__*.
- se fait via un décorateur du même nom qui doit être importé.

```
class User:
    def __init__(self, name: str, email: str, age: int):
        self.name = name
        self.email = email
        self.age = age

user128 = User( name="Pablo", email="pablo@univ-rennes.fr", age=18)
print(user128.email)
print(user128.age)
```

Déclaration des variables d'instance et Définition du constructeur.

Résultat:

```
pablo@univ-rennes.fr
18
```

```
from dataclasses import dataclass

@dataclass
class User:
    name: str
    email: str
    age: int

user128 = User( name="Pablo", email="pablo@univ-rennes.fr", age=18)
print(user128.email)
print(user128.age)
```

Décorateur

Déclaration des variables d'instance et Définition du constructeur.

@

Résultat:

```
pablo@univ-rennes.fr
18
```

DataClass

@dataclass

2.2

```python
from dataclasses import dataclass
from typing import List
from typing import Tuple

@dataclass
class Person:
    name: str
    age: int
    height: float
    email: str
    house_coordinates: Tuple

@dataclass
class People:
    people: List[Person]

joe = Person('Joe', 25, 1.85, 'joe@gmail.fr.io', (40.7, -73.9))
print(joe)

mary = Person('Mary', 43, 1.67, 'mary@gmail.fr', (-73.9, 40.7))
print(mary)
print("_____")
print(People([joe, mary]))
```

Définition d'une classe

Définition d'une classe

@

appel de la méthode *str()* pour convertir un objet en *str*

Résultat:

```
Person(name='Joe', age=25, height=1.85, email='joe@gmail.fr.io', house_coordinates=(40.7, -73.9))
Person(name='Mary', age=43, height=1.67, email='mary@gmail.fr', house_coordinates=(-73.9, 40.7))
_____
People(people=[Person(name='Joe', age=25, height=1.85, email='joe@gmail.fr.io', house_coordinates=(40.7, -73.9)), Person(name='Mary', age=43, height=1.67, email='mary@gmail.fr', house_coordinates=(-73.9, 40.7))])
```

DataClass

☞ **Rédéfinition de la méthode *str()***

```
#!/usr/bin/env python3
from dataclasses import dataclass
from dataclasses import field

@dataclass
class Employer:
    age: int
    name: str
    male: bool

    def __str__(self):
        return f"Employer avec age={self.age}, name={self.name} andmale={self.male}."

emp12=Employer(62, "Robert", True)

print(str(emp12))

emp45=Employer(23, "Gaëlle", False)
print(str(emp45))
```

Définition d'une classe

Redéfinition de la méthode *str(...)*

Instanciation
(appel du constructeur avec arguments)

appel explicite de la méthode *str()*

Résultat:

"Employer avec age=62, name=Robert and male=True"
"Employer avec age=23, name= Gaëlle and male=False"

post_init__()

☞ *post_init__()* est appelée systématiquement par le constructeur *__init__()*.

```
from dataclasses import dataclass, field

@dataclass
class Employer:
    name: str
    emp_id: str
    age: int
    city: str
    check_age: bool = field(init=False)

    def __post_init__(self):
        if self.age >= 30:
            self.check_age = True
        else:
            self.check_age = False


emp = Employer("Emilie", "emilie35400", 21, 'Rennes')
print(f"Âge >30: {emp.check_age}")
```

Définition d'une classe

Définition de cette méthode qui est appelée automatiquement et de manière implicite par le constructeur __init__().

Résultat:

Âge >30: False

post_init__()

☞ *post_init__()* est appelée systématiquement par le constructeur *__init__()*.

```
from dataclasses import dataclass

@dataclass
class User:
    first_name: str
    last_name: str
    age: int

    def __post_init__(self):
        self.full_name = f"{self.first_name} {self.last_name}"

emp1 = User("Emilie","Andre", 21)
emp2 = User("Julie", "Robert", 29)
print(emp1)
print(emp2)
```

Définition d'une classe

Définition de cette méthode qui est appelée automatiquement et de manière implicite par le constructeur *__init__()*.

Résultat:

```
User(first_name='Emilie', last_name='Andre', age=21)
User(first_name='Julie', last_name='Robert', age=29)
```

conversion en dictionnaire ou en tuple

- ☞ ***asdict()*** **convertit en dictionnaire**
- ☞ ***astuple()*** **convertit en tuple.**

```
from dataclasses import dataclass, asdict, astuple

@dataclass
class Product:
    name: str
    price: float

p1 = Product("Pen", 2.50)
print(asdict(p1))
print(astuple(p1))

p2 = Product("Notebook", 5.00)
print(asdict(p2))
print(astuple(p2))
```

Définition d'une classe

pour convertir en dictionnaire

pour convertir en tuple

Résultat:

```
{'name': 'Pen', 'price': 2.5}
('Pen', 2.5)
{'name': 'Notebook', 'price': 5.0}
('Notebook', 5.0)
```

Dataclasses — Classes de Données

Les paramètres optionnels de @dataclass sont :

- ***init*** **(valeur par défaut *True*)** — **définit la méthode *__init__()***
- ***repr*** **(valeur par défaut *True*)** — **définit la méthode *__repr__()***
- ***eq*** **(valeur par défaut *True*)** — **définit la méthode *__eq__()* pour des comparaisons avec ==**
- ***frozen*** **(valeur par défaut *False*)** — **définir les objets immuables**
- ***unsafe_hash*** **(valeur par défaut *False*)** — **définit la méthode *__hash__()* (à n'utiliser que si les objets sont immuables)**
- ***order*** **(valeur par défaut *False*)** — **définit les méthodes de comparaison (*__lt__*, *__le__*, etc.) en considérant les objets comme des tuples**
- ***kw_only*** **(valeur par défaut *False*)** — **obliger à utiliser une initialisation des objets à l'aide d'un passage de paramètre nommé**

```
@dataclass(unsafe_hash=True)
```

```
@dataclass(frozen=True)
```

Dataclass immutable

☞ **dataclass immuable (*frozen=True*)**

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Client:
    name: str
    age: int

c = Client("John", 30)
print(c)

c.age = 31
```

Dataclass immuable

Définition d'une classe

Attempting to modify an attribute will raise a FrozenInstanceError

@

Résultat:

```
Client(name='John', age=30)

Traceback (most recent call last):
  File "dataclass_prog5.py", line 12, in <module>
    c.age = 31
    ^^^^^
  File "<string>", line 16, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'age'
```

Dataclass immutable

☞ **dataclass immuable (*frozen=True*)**

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Transaction:
    source: str
    destination: str
    content: str

def is_valid(self):
    return self.source != "" and self.destination != "" \
        and self.source != self.destination

@dataclass(frozen=False)
class Person:
    name: str
    firstname: str
    city: str

tAB = Transaction('PersA', 'PersB', 'transactionA->B!')
print(tAB)

print(f"Transaction tAB est elle valide  ? tAB.is_valid()")

tCA = Transaction('PersC', 'PersA', 'transactionC->A!')
print(tCA)

t2 = Transaction('PersA', 'PersB', 'transactionA->B!')
print(f"tAB==t2? {tAB==t2}")

caro = Person('Robert', 'Caroline', 'Nantes')
print(f"tAB == caro ? {tAB == caro}")
```

Dataclass immuable

Définition d'une classe

Définition d'une classe

@

Résultat:

```
Transaction(source='PersA',
destination='PersB',
content='transactionA->B!')

Transaction tAB est elle valide  ?
tAB.is_valid()

Transaction(source='PersC',
destination='PersA',
content='transactionC->A!')

tAB==t2? True

tAB == caro ? False
```

Comparaison et tri

☞ ***order*** **: Par défaut, un dataclass ne peut pas être trié ou comparé avec <, >, <=, >=.**

☞ **Pour activer ces opérations, nous pouvons spécifier *order=True* .**

```
from dataclasses import dataclass

@dataclass(order=True)
class Product:
    price: float
    name: str

p1 = Product(19.99, "Widget")
p2 = Product(29.99, "Gadget")
p3 = Product(9.99, "Thingamajig")

list_of_products = [p2, p1, p3]
print(list_of_products)
sorted_products = sorted(list_of_products)
print(sorted_products)

print(p1 < p2)  # True, because 19.99 < 29.99
print(p3 > p1)  # False, because 9.99 is not greater than 19.99
print(p1 == Product(19.99, "Widget"))  # True, same price and name
```

order=True

Définition d'une classe A

Résultat:

```
[Product(price=29.99, name='Gadget'),
Product(price=19.99, name='Widget'),
Product(price=9.99, name='Thingamajig')]
[Product(price=9.99, name='Thingamajig'),
Product(price=19.99, name='Widget'),
Product(price=29.99, name='Gadget')]
True
False
True
```

unsafe_hash

☞ ***unsafe_hash*: If true (par défaut), definition méthode *__hash__()*.**

```
from dataclasses import dataclass, field
@dataclass(unsafe_hash=True)
class Employer:
    name: str
    age: int
    emp_id: str
    city: str = "Rennes"

emp1 = Employer("Emilie", 21,"emilie35400")
print(emp1)
emp2 = Employer("Julie", 22, "julie35000")
print(emp2)
print(f"hash(emp1):{hash(emp1)}")
print(f"hash(emp2):{hash(emp2)}")
emp2.city = "Nantes"
print(f"After modification, hash(emp2):{hash(emp2)}")
```

unsafe_hash=True

Définition d'une classe A

Résultat:

```
Employer(name='Emilie', age=21, emp_id='emilie35400', city='Rennes')
Employer(name='Julie', age=22, emp_id='julie35000', city='Rennes')
hash(emp1):-1731291069975006080
hash(emp2):-7110212641519120722
After modification, hash(emp2):-6656704564142972706
```

@dataclass, field 2.2.3

Fields: Données membres statiques

☞ **Données membres (attributs, Fields) statiques**

Nom de l'attribut	Valeur
compare	Si ***compare=True*** définit les méthodes ***__eq__()*** et ***__lt__()***
default	valeur par défaut de cette variable d'instance
default_factory	Génère dynamiquement une valeur par défaut
hash	<member 'hash' of 'Field' objects>
init	Si ***init=False***, pas besoin de fournir cet argument (varaible d'instance) au constructeur ***__init__()***
kw_only	<member 'kw_only' of 'Field' objects>
metadata	<member 'metadata' of 'Field' objects>
name	<member 'name' of 'Field' objects>
repr	Si ***repr=True*** définit la méthode ***__repr__()***
type	type de variable d'instance

default

☞ **Champs de classe de données initialisés automatiquement**

```
from dataclasses import dataclass, field

@dataclass
class Employer:
    name: str
    emp_id: str
    age: int
    city: str = field(default="Rennes")


emp = Employer("Emilie", "Emilie35", 21)
print(emp)
```

Définition d'une classe

default signifie la valeur par défaut.

Résultat:

Employer(name='Emilie', emp_id='Emilie35', age=21, city='Rennes')

@dataclass, field 2.2.3

default_factory

- *default_factory* permet des valeurs modifiables ou nécessitant un calcul quelconque.
- est très utile pour éviter les pièges liés aux valeurs mutables partagées comme les listes ou les dictionnaires.

```
from dataclasses import dataclass, field

@dataclass
class Email:
    subject: str
    body: str
    sender: str
    recipients: list = field(default_factory=list)
    cc: list = field(default_factory=list)

email1 = Email("Lunch?", "I'm hungry.", "lauren@gmail.fr")
email2 = Email("Meeting", "Let's meet tomorrow", "nicole@gmail.fr")

email1.recipients.append("kevin@gmail.fr")
email1.recipients.append("charlie@gmail.fr")
print(email1.recipients)

email2.recipients.append("rupert@gmail.fr")
print(email2.recipients)
```

Définition d'une classe

default_factory pour des valeurs modifiables ou nécessitant un calcul quelconque

Résultat:

```
['kevin@gmail.fr', 'charlie@gmail.fr']
['rupert@gmail.fr']
```

@dataclass, field 2.2.3

default_factory

☞ *default_factory* permet des valeurs modifiables ou nécessitant un calcul quelconque.

```python
from dataclasses import dataclass, field
from random import choice

def get_default_language():
    languages = ['Python3', 'Java', "CPP", "JavaScript", "GoLang"]
    return choice(languages)

@dataclass
class Livre:
    title: str
    author: str
    language: str = field(default_factory = get_default_language)

livre4 = Livre("DataClass", "Alfred")
print(livre4)

livre5 = Livre("Pydantic", "Robert")
print(livre5)
```

Définition de fonction

Tirage aléatoire dans la liste

Définition d'une classe

default_factory pour des valeurs modifiables ou nécessitant un calcul quelconque

Appel de fonction

Résultat:

```
Livre(title=Node.JS', author='Alfred', language='JavaScript')
Livre(title='Pydantic', author='Robert', language='Python3')
```

@dataclass, field 2.2.3

default_factory

☞ *default_factory* permet des valeurs modifiables ou nécessitant un calcul quelconque.

```
from dataclasses import dataclass, field
from typing import List

@dataclass
class Student:
    name: str
    grades: List[int] = field(default_factory=list)


student1 = Student(name="Alice")
student2 = Student(name="Bob", grades=[90, 85])

student1.grades.append(95)

print(student1)
print(student2)
```

Définition d'une classe

default_factory pour des valeurs modifiables ou nécessitant un calcul quelconque

Résultat:

```
Student(name='Alice', grades=[95])
Student(name='Bob', grades=[90, 85])
```

@dataclass, field 2.2.3

default_factory

☞ *default_factory* permet des valeurs modifiables ou nécessitant un calcul quelconque.

```
from dataclasses import dataclass, field

def get_emp_id():
    id = 3300035400
    return id

@dataclass
class Employer:
    name: str
    age: int

    emp_id: str = field(default_factory=get_emp_id)
    city: str = field(default="Rennes")

emp = Employer("Sarah", 21)
print(emp)
```

Définition d'une fonction pour initialiser une variable d'instance

Définition d'une classe

default_factory pour des valeurs modifiables ou nécessitant un calcul

appel de fonction pour initialiser la variable d'instance *emp_id*

Résultat:

```
Employer(name='Sarah', age=21, emp_id=3300035400, city='Rennes')
```

@dataclass, field 2.2.3

init

☞ ***init*** **à** ***false*** **signifie que l'on n'a pas besoin de fournir cet argument au constructeur**

```
from dataclasses import dataclass, field

@dataclass
class Employer:
    name: str
    age: int
    emp_id: str
    city: str = field(init=False, default="Rennes")


emp = Employer("Emilie", "emilie35400", 21)
print(emp)
```

Définition d'une classe

init à *false* usignifie que l'on n'a pas besoin de fournir cet argument au constructeur

Pas d'argument de ville au constructeur

Résultat:

```
Employer(name='Emilie', age='emilie35400', emp_id=21, city='Rennes')
```

@dataclass, field 2.2.3

repr

☞ ***repr*: si *True*, retourne cette variable d'instance par la méthode *__repr__()* en convertissant cet objet en *str*.**

```
from dataclasses import dataclass, field
@dataclass
class Employer:
    name: str
    age: int
    emp_id: str
    city: str = field(init=False, default="Rennes", repr=True)

emp = Employer("Emilie", 21, "emilie35400"),
print(emp)
```

Définition d'une classe

repr si *True*, retourne cette variable d'instance

Résultat:

(Employer(name='Emilie', age=21, emp_id='emilie35400', ***city='Rennes'***),)

```
from dataclasses import dataclass, field
@dataclass
class Employer:
    name: str
    age: int
    emp_id: str
    city: str = field(init=False, default="Rennes", repr=False)

emp2 = Employer("Emilie", 21, "emilie35400"),
print(emp2)
```

Définition d'une classe

repr si *False*, ne retrourne pas cette variable d'instance

repr si *False*, ne retrourne pas cette variable d'instance

Résultat:

(Employer(name='Emilie', age=21, emp_id='emilie35400'),)

@dataclass, field 2.2.3

compare

☞ ***compare*: If *True* (par défaut), définit les méthodes de comparaison: equalité et inégalité (*__eq__()*, *__gt__()*.)**

```
from dataclasses import dataclass, field

@dataclass()
class Employer:
    name: str
    age: int
    emp_id: str
    city: str = field(default="Rennes", compare=True)

emp1 = Employer("Emilie", "emilie35400", 21)
emp2 = Employer("Julie", "julie35000", 22)
print(f"Deux instances sont elles égales? {emp1 == emp2}");

emp3 = Employer("Emilie", "emilie35400", 21)
print(f"Deux instances sont elles égales? {emp1 == emp3}");
```

Définition d'une classe A class for holding an Employers content

compare à *True* permet la comparaison avec ==

Résultat:

```
Deux instances sont elles égales? False
Deux instances sont elles égales? True
```

compare

☞ ***compare*: If *True* (par défaut), définit les méthodes de comparaison: equalité et inégalité (*__eq__()*, *__gt__()*.)**

```
from dataclasses import dataclass, field

@dataclass(unsafe_hash=True)
class Employer:
    name: str
    age: int
    emp_id: str
    city: str = field(init=False, default="Rennes", repr=True, compare=True)

emp1 = Employer("Emilie", "emilie35400", 21)
emp2 = Employer("Julie", "julie35000", 22)

print(f"Deux instances sont elles égales? {emp1 == emp2}")
```

unsafe_hash=True

Définition d'une classe A

compare permet la comparaison en définissant les méthodes de comparaison

Appel implicite de la méthode *__eq__()*

Résultat:

Deux instances sont elles égales? False

hash

☞ **hash: si *True* définit la méthode *__hash__()*.**

```
# hash
from dataclasses import dataclass, field

def get_emp_id():
    id = 2345
    return id

@dataclass(unsafe_hash=True)
class Employer:
    name: str
    age: int
    emp_id: str = field(default_factory=get_emp_id)
    city: str = field(init=False, default="Rennes", repr=True, hash=True)

emp = Employer("Sarah", 21)
print(hash(emp))
```

Définition d'une classe

hash à True, définit la méthode __hash__()

Pas d'argument de id ni ville au constructeur

Résultat:

```
-1067144175145615364
```

metadata

☞ ***metadata*** **(métadonnées) : Il peut s'agir d'un mappage ou de None. None est traité comme un dictionnaire vide.**

```
from dataclasses import dataclass, field

@dataclass(unsafe_hash=True)
class Employer:
    name: str
    age: int
    emp_id: str
    city: str = field(init=False, default="Rennes", repr=True,
                      metadata={'format': 'State'})

emp = Employer("Emilie", "emilie35400", 21)
print(emp.__dataclass_fields__['city'].metadata['format'])
```

unsafe_hash=True

Définition d'une classe A class for holding an Employers content

field(...) use init , default, repr, hash, compare fields

__dataclass_fields__ use

Résultat:

State

Héritage

☞ **permet de définir une classe qui reprend toutes les fonctionnalités d'une classe parente.**

```
from dataclasses import dataclass, field

@dataclass
class Staff:
    name: str
    emp_id: str
    age: int

@dataclass
class Employer(Staff):
    salary: int


emp = Employer("Emilie", "emilie35400", 21, 45000)
print(emp)
```

Définition d'une classe mère

Définition d'une classe fille

Résultat:

```
Employer(name='Emilie', emp_id='emilie35400', age=21, salary=45000)
```

Pourquoi Pydantic ?

☞ Pas de déclaration en python → ce qui peut poser des problèmes!

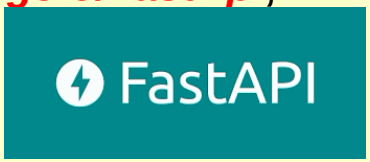
Pydantic

☞ est une bibliothèque de validation de données en Python.

☞ une des plus utilisées en Python.

☞ certaines des API les plus connues dépendent de *Pydantic*:

- ☞ *tiangolo/FastApi*, framework web pour la construction d'API



- ☞ *hwchase17/LangChain*, création d'appli. à l'aide de grands modèles de langage (LLM) tels que chatbots

- ☞ *huggingface/Transformers* ensemble de modèles d'IA préentrainés de LLM,

- ☞ *apache/AirFlow* outil open-source puissant pour automatiser, orchestrer et surveiller vos workflows complexes

Intérêt de Pydantic

- IDE tye hints
- autocompletion
- data validation
- JSON serialization: sérialiser des objets

Pydantic

- bibliothèque de validation de données la plus populaire de Python, qui permet de transformer les indications de type en règles de validation à l'exécution.
- Au lieu d'écrire des dizaines de if isinstance() et des fonctions de validation personnalisées, vous définissez votre structure de données une seule fois en utilisant la syntaxe familière de Python.
- validation des données entrantes, conversion des types le cas échéant et fourniture de messages d'erreur clairs en cas d'échec de la validation.

- Installer *Pydantic*

$ pip3 install pydantic

Lors de l'installation de *FastAPI*, installation de *pydantic* est aussi réalisée

```
Windows PowerShell
PS C:\Nunes\Enseignements\PYTHON_PERL_Auto_Taches_3206\Python_R507_DevCloud\Partage> pip install pydantic
Collecting pydantic
  Downloading pydantic-2.11.9-py3-none-any.whl.metadata (68 kB)
Collecting annotated-types>=0.6.0 (from pydantic)
  Downloading annotated_types-0.7.0-py3-none-any.whl.metadata (15 kB)
Collecting pydantic-core==2.33.2 (from pydantic)
  Downloading pydantic_core-2.33.2-cp313-cp313-win_amd64.whl.metadata (6.9 kB)
Collecting typing-extensions>=4.12.2 (from pydantic)
  Downloading typing_extensions-4.15.0-py3-none-any.whl.metadata (3.3 kB)
Collecting typing-inspection>=0.4.0 (from pydantic)
  Downloading typing_inspection-0.4.1-py3-none-any.whl.metadata (2.6 kB)
Downloading pydantic-2.11.9-py3-none-any.whl (444 kB)
Downloading pydantic_core-2.33.2-cp313-cp313-win_amd64.whl (2.0 MB)
   ---------------------------------------- 2.0/2.0 MB 12.8 MB/s  0:00:00
Downloading annotated_types-0.7.0-py3-none-any.whl (13 kB)
Downloading typing_extensions-4.15.0-py3-none-any.whl (44 kB)
Downloading typing_inspection-0.4.1-py3-none-any.whl (14 kB)
Installing collected packages: typing-extensions, annotated-types, typing-inspection, pydantic-core, pydantic
Successfully installed annotated-types-0.7.0 pydantic-2.11.9 pydantic-core-2.33.2 typing-extensions-4.15.0 typing-inspec
tion-0.4.1
PS C:\Nunes\Enseignements\PYTHON_PERL_Auto_Taches_3206\Python_R507_DevCloud\Partage>
```

validate_call bibliothèque de validation de données

```
#!/usr/bin/env python3
from pydantic import validate_call

@validate_call
def division(a: int, b: int) -> float:
    print(f"le type de 'a' est '{type(a)}' et le type de b est '{type(b)}'.")
    return a / b

res1=division(10, 20)
print(res1)

print("_____")
res2=division(1000, "20")
print(res2)

print("_____")
res3=division(10, "coucou")
print(res3)
```

Le décorateur @validate_call va utiliser les annotations de types de chaque paramètre pour effectuer des validations.

Définition d'une fonction avec deux arguments d'entrée et retournant un float

Appel de la fonction avec deux arguments un int et un int

Appel de la fonction avec deux arguments un int et un string

Appel de la fonction avec deux arguments un int et un string

Résultat:

```
le type de 'a' est '<class 'int'>' et le type de b est '<class 'int'>'.
0.5
_____
le type de 'a' est '<class 'int'>' et le type de b est '<class 'int'>'.
50.0
_____
Traceback (most recent call last):
  File "c:\Nunes\Enseignements\PYTHON_PERL_Auto_Taches_3206\Python_R507_DevCloud\Partage\prog5.py", line 17, in <module>
    res3=division(10, "coucou")
  File "C:\Users\jeanc\AppData\Local\Programs\Python\Python313\Lib\site-packages\pydantic\_internal\_validate_call.py", line 39, in wrapper_function
    return wrapper(*args, **kwargs)
  File "C:\Users\jeanc\AppData\Local\Programs\Python\Python313\Lib\site-packages\pydantic\_internal\_validate_call.py", line 136, in __call__
    res = self.__pydantic_validator__.validate_python(pydantic_core.ArgsKwargs(args, kwargs))
pydantic_core._pydantic_core.ValidationError: 1 validation error for division
1
  Input should be a valid integer, unable to parse string as an integer [type=int_parsing, input_value='coucou', input_type=str]
    For further information visit https://errors.pydantic.dev/2.11/v/int_parsing
```

Erreur liée au passage d'arguments, le 2ème argument de type string "coucou".

Modèle de données

☞ **Modèle de données**

```
#!/usr/bin/env python3
from pydantic import BaseModel


class User(BaseModel):
    id: int
    name: str
    email: str
    age: int
    profession: str

employe128 = User(id=42, name="Pablo", email="pablo@univ-rennes.fr", age=45, profession="administrateur")

print(employe128.profession)
print("_____")

employe3 = User(id=4, name="Pedro", email="pedro@univ-rennes.fr", age=26, profession="developpeur")
print(employe3.email)
```

Définition d'une classe

Déclaration des variables d'instance.

On précise le type de chaque variable d'instance

Instanciation (appel du constructeur)

accès à une variable d'instance de l'objet employe128

Instanciation (appel du constructeur)

Résultat:

```
administrateur
_____
pedro@univ-rennes.fr
```

Modèle de données

☞ **Modèle de données**

```python
#!/usr/bin/env python3
from pydantic import BaseModel

class Employer(BaseModel):
    id: int
    name: str
    email: str=None
    age: int=None
    profession: str

employe3=Employer(id=3,name="Pedro",email="pedro@univ.fr",age=26,profession="developpeur")
print(employe3.model_dump())

employe4 = Employer(id=4, name="Pierre", profession="ebeniste")
print(employe4.model_dump())
```

Déclaration des variables d'instance.

sont facultatifs et ont comme valeur par défaut `None`.

Appel du construteur avec 5 arguments (les facultatifs compris)

Appel du construteur avec 3 arguments (sans les facultatifs)

Résultat:

```
{'id': 3, 'name': 'Pedro', 'email': 'pedro@univ.fr', 'age': 26, 'profession': 'developpeur'}
{'id': 4, 'name': 'Pierre', 'email': None, 'age': None, 'profession': 'ebeniste'}
```

Dataclasses — Classes de Données

☞ **Validation des valeurs des variables d'instance**

```
#!/usr/bin/env python3
from pydantic import BaseModel
class Personne(BaseModel):
    nom: str
    prenom: str
    annee_naissance: int

carole=Personne(nom="Dupont", prenom="Carole", annee_naissance=2000)
print(carole)

jules=Personne(nom="Merlin", prenom="Jules", annee_naissance="1988")
print(jules.model_dump())

try:
    paul=Personne(nom="Durand", prenom="Paul", annee_naissance="nineteen ninety five")
    print(paul.model_dump())
except Exception as e:
    print("Erreur lors de la création de l'objet Paul :", e)
```

conversion du str en int

appel de la méthode *model_dump()* convertissant une instance de Personne en dict

erreur de conversion du str en int

Résultat:

```
nom='Dupont' prenom='Carole' annee_naissance=2000
{'nom': 'Merlin', 'prenom': 'Jules', 'annee_naissance': 1988}
Erreur lors de la création de l'objet Paul : 1 validation error for Personne
annee_naissance
  Input should be a valid integer, unable to parse string as an integer [type=int_parsing,
input_value='nineteen ninety five', input_type=str]
    For further information visit https://errors.pydantic.dev/2.12/v/int_parsing
```

Computed Fields

***@computed_field*: décorateur permettant d'ajouter:**

- ☞ **des variables d'instance calculés à partir d'autres variables d'instance (*property*),**
- ☞ **des variables d'instance dont le calcul est coûteux et qui doivent être mises en cache *cached_property*.**

```
#!/usr/bin/env python3
from pydantic import BaseModel, computed_field

class Personne(BaseModel):
    nom: str
    prenom: str
    annee_naissance: int

    @computed_field
    @property
    def age(self) -> int:
        return 2025 - self.annee_naissance

print(Personne(nom="Dupont", prenom="Carole", annee_naissance=2000).model_dump())

print(Personne(nom="Merlin", prenom="Jules", annee_naissance=1988).model_dump())
```

Définition d'une classe avec 3 variables d'instance et 1 constructeur avec arguments

ajout d'une 4ème variable d'instance calculée automatiquement

Instanciation (appel du constructeur) avec seulement 3 arguments

4ème variable d'instance

Résultat:

```
{'nom': 'Dupont', 'prenom': 'Carole', 'annee_naissance': 2000, 'age': 25}
{'nom': 'Merlin', 'prenom': 'Jules', 'annee_naissance': 1988, 'age': 37}
```

Computed Fields

***@computed_field*: décorateur permettant d'ajouter:**

- ☞ **des variables d'instance calculés à partir d'autres variables d'instance (*property*),**
- ☞ **des variables d'instance dont le calcul est coûteux et qui doivent être mises en cache *cached_property*.**

```python
#!/usr/bin/env python3
from pydantic import BaseModel, computed_field

class Rectangle(BaseModel):
    width: int
    length: int

    @computed_field
    @property
    def area(self) -> int:
        return self.width * self.length

print(Rectangle(width=3, length=2).model_dump())

print(Rectangle(width=20, length=10).model_dump())
```

ajout d'une variable d'instance calculée

Instanciation (appel du constructeur) avec 2 arguments

3ème variable d'instance (calculée)

Résultat:

```
{'width': 3, 'length': 2, 'area': 6}
{'width': 20, 'length': 10, 'area': 200}
```

Computed Fields

@computed_field: décorateur permettant d'ajouter:

- ☞ des variables d'instance calculés à partir d'autres variables d'instance (***property***),
- ☞ des variables d'instance dont le calcul est coûteux et qui doivent être mises en cache ***cached_property***.

```python
from functools import cached_property
from pydantic import BaseModel, computed_field
class Square(BaseModel):
    width: float
    @computed_field
    def area(self) -> float:
        return round(self.width**2, 2)

    @area.setter
    def area(self, new_area: float) -> None:
        self.width = new_area**0.5

    @computed_field(alias='the magic number', repr=False)
    @cached_property
    def random_number(self) -> int:
        return random.randint(0, 1_000)

square = Square(width=1.3)

print(repr(square))
print(square.random_number)
print(square.model_dump())
square.area = 4
print(square.model_dump_json(by_alias=True))
```

Définition d'une classe

ajout d'une nouvelle variable d'instance `area` initialisée par `width` au carré avec 2 chiffres après la virgule.

Définition du setter de la variable d'instance `area`

ajout d'une nouvelle variable d'instance cachée `random_number` initialisée par une variable aléatoire entre 0 et 100.
On lui donne un alias avec le nom `the magic number`.

Appel du setter

Résultat:

```
Square(width=1.3, area=1.69)
613
{'width': 1.3, 'area': 1.69, 'random_number': 613}
{"width":2.0,"area":4.0,"the magic number":613}
```

Cette instance a 3 variables d'instance et pas une seule `width`.

Validators: validateurs de champs

- définir une règle de validation spécifique à chaque variable d'instance, telles que:
  - simple vérification de type.
  - vérifications de valeurs,
- ***@fieldvalidator('nom_variable_instance')***: Décorateur permettant d'ajouter une fonction de validation spécifique à la variable d'instance ***nom_variable_instance***.

```
from pydantic import BaseModel, field_validator

class UserProfile(BaseModel):
    name: str
    age: int
    email: str

    @field_validator('age')
    def check_age(cls, value):
        if value < 18:
            raise ValueError('Age must be at least 18')
        return value

u2=UserProfile(name="Noam Dupond", age=17, email="noam.dupond@univ-rennes.fr")
```

Définition d'une classe avec variables d'instance et constructeur

Ce n'est pas self !!!!!

Décorateur de fonction de validation spécifique de l'âge

Définition de fonction de vérification de l'âge

Levée d'une exception

Cette instanciation déclenche la validation et lève une exception.

Résultat:

```
Traceback (most recent call last):
  File pydantic_validator_prog1.py", line 17, in <module>
    u2=UserProfile(name="Noam Dupond", age=17, email="noam.dupond@univ-rennes.fr")
  File "C:\Users\jeanc\AppData\Local\Programs\Python\Python313\Lib\site-packages\pydantic\main.py", line 250, in __init__
    validated_self = self.__pydantic_validator__.validate_python(data, self_instance=self)
pydantic_core._pydantic_core.ValidationError: 1 validation error for UserProfile
age
  Value error, Age must be at least 18 [type=value_error, input_value=17, input_type=int]
    For further information visit https://errors.pydantic.dev/2.12/v/value_error
```

Validators

☞ ***@fieldvalidator('nom_variable_instance')***: **Décorateur permettant d'ajouter une fonction de validation spécifique à la variable d'instance *nom_variable_instance*.**

```
from pydantic import BaseModel, field_validator

class UserProfile(BaseModel):
    name: str
    age: int
    email: str

    @field_validator('age')
    def check_age(cls, value):
        if value < 18:
            raise ValueError('Age must be at least 18')
        return value

u1=UserProfile(name="Sébastien Dupond", age=42, email="sebastien.dupond@univ-
rennes.fr")
print(u1)

u2=UserProfile(name="Noam Dupond", age=17, email="noam.dupond@univ-rennes.fr")
print(u2)
```

Définition d'une classe avec variables d'instance et constructeur

Ce n'est pas self !!!!!

Définition de fonction de validation

Levée d'une exception

Cette instanciation déclenche la validation et lève une exception.

```
name='Sébastien Dupond' age=42 email='sebastien.dupond@univ-rennes.fr'
Traceback (most recent call last):
  File pydantic_validator_prog1.py", line 17, in <module>
    u2=UserProfile(name="Noam Dupond", age=17, email="noam.dupond@univ-rennes.fr")
  File "C:\Users\jeanc\AppData\Local\Programs\Python\Python313\Lib\site-packages\pydantic\main.py", line 250, in __init__
    validated_self = self.__pydantic_validator__.validate_python(data, self_instance=self)
pydantic_core._pydantic_core.ValidationError: 1 validation error for UserProfile
age
  Value error, Age must be at least 18 [type=value_error, input_value=17, input_type=int]
    For further information visit https://errors.pydantic.dev/2.12/v/value_error
```

Validators

☞ **@fieldvalidator('nom_variable_instance')**: Décorateur permettant d'ajouter une fonction de validation spécifique à la variable d'instance ***nom_variable_instance***.

```python
from pydantic import BaseModel, field_validator

class UserProfile(BaseModel):
    name: str
    age: int
    email: str

    @field_validator('age')
    def check_age(cls, value):
        if value < 18:
            raise ValueError('Age must be at least 18')
        return value

try:
    u1=UserProfile(name="Romain Dupond", age=42, email="romain.dupond@univ-rennes.fr")
    print(u1)

    u2=UserProfile(name="Noam Dupond", age=17, email="noam.dupond@univ-rennes.fr")
    print(u2)
except Exception as e:
    print(f"Validation error: {e}")
```

Définition d'une classe avec variables d'instance et constructeur

Ce n'est pas self !!!!!

Définition de fonction de validation

Levée d'une exception

Résultat:

```
name='Romain Dupond' age=42 email='romain.dupond@univ-rennes.fr'
Validation error: 1 validation error for UserProfile
age
  Value error, Age must be at least 18 [type=value_error, input_value=17, input_type=int]
    For further information visit https://errors.pydantic.dev/2.12/v/value_error
```

Validators: validateurs de champs

☞ ***@fieldvalidator('nom_variable_instance')***: Décorateur permettant d'ajouter une fonction de validation spécifique à la variable d'instance ***nom_variable_instance***.

```python
from pydantic import BaseModel, field_validator
class UserProfile(BaseModel):
    name: str
    age: int
    email: str

    @field_validator('age')
    def check_age(cls, value):
        if value <= 0 or value > 118:
            raise ValueError('Age must be at least 0 and at most 118')
        return value

try:
    u2=UserProfile(name="Alice Smith", age=30, email="alice.univ-rennes.fr")
    print(u2)
    u1=UserProfile(name="Noam Dupond", age=-217, email="noam.dupond@univ-
rennes.fr")
    print(u1)
except Exception as e:
    print("Error:", e)
```

Définition d'une classe avec variables d'instance et constructeur

Ce n'est pas self !!!!!

Définition de fonction de validation de l'âge

Levée d'une exception

Cette instanciation déclenche la validation et lève une exception.

Résultat:

```
name='Alice Smith' age=30 email='alice.univ-rennes.fr'
Error: 1 validation error for UserProfile
age
  Value error, Age must be at least 0 and at most 118 [type=value_error, input_value=-217, input_type=int]
    For further information visit https://errors.pydantic.dev/2.12/v/value_error
```

Validators

☞ ***@model_validator()*: Décorateur permettant de valider chaque variable d'instance.**

```python
from pydantic import BaseModel, model_validator

class UserModel(BaseModel):
    name: str
    password1: str
    password2: str

    @model_validator(mode="after")
    def check_passwords_match(self) -> "UserModel":
        if self.password1 != self.password2:
            raise ValueError("passwords do not match")
        return self

try:
    print("Creating user with matching passwords!")
    u1=UserModel(name="Kevin",password1="abc", password2="abc")
    print(u1)
    u2=UserModel(name="Julie",password1="ab123#", password2="abcd#") #Passwords don't match
    print(u2)

except ValueError as e:
    print(e)  # This will raise a ValueError: Passwords do not match
```

Définition d'une classe avec variables d'instance et constructeur

Runs validation after the model is fully initialized.

Levée d'une exception

Résultat:

```
Creating user with matching passwords!
name='Kevin' password1='abc' password2='abc'
1 validation error for UserModel
  Value error, passwords do not match [type=value_error, input_value={'name': 'Julie', 'passwo...', 'password2': 'abcd#'}, input_type=dict]
    For further information visit https://errors.pydantic.dev/2.12/v/value_error
```

Validators

☞ **@model_validator(): Décorateur permettant de valider chaque variable d'instance.**

```python
from pydantic import BaseModel, model_validator

class Colis(BaseModel):
    name: str
    a_envoyer: bool
    adresse_destination: str

    @model_validator(mode="after")
    def check_colis(self):
        if self.a_envoyer and not self.adresse_destination:
            raise ValueError("Colis marked for sending must have a destination address.")
        return self

try:
    c1=Colis(name="Jouet",a_envoyer="True", adresse_destination="10 rue de la paix, Paris")
    print(f"c1={c1}")
    print("-----")
    c2=Colis(name="Livre",a_envoyer="True", adresse_destination="")
    print(f"c2={c2}")
    print("-----")
    c3=Colis(name="Cadeau",a_envoyer="False", adresse_destination="78 avenue des Champs, Nantes")
    print(f"c3={c3}")
    print("-----")
    c4=Colis(name="Parfum",a_envoyer="False", adresse_destination="")
    print(f"c4={c4}")

except ValueError as e:
    print(e)
```

Résultat:

```
c1=name='Jouet' a_envoyer=True adresse_destination='10 rue de la paix, Paris'
-----
1 validation error for Colis
  Value error, Colis marked for sending must have a destination address. [type=value_error, input_value={'name': 'Livre',
'a_envo...dresse_destination': ''}, input_type=dict]
    For further information visit https://errors.pydantic.dev/2.12/v/value_error
```

model validator 3.1

Validators

☞ ***@model_validator()*: Décorateur permettant de valider chaque variable d'instance.**

```python
from pydantic import BaseModel, model_validator, EmailStr
from typing import Optional

class UserProfile(BaseModel):
    name: str
    age: int
    email: EmailStr
    nickname: Optional[str] = None

    @model_validator(mode="after")
    def check_age(self):
        if self.age <= 0 or self.age > 118:
            raise ValueError('Age must be at least 0 and at most 118')
        return self

try:
    u2=UserProfile(name="Alice Smith", age=30, email="alice@rennes.fr")
    print(u2)
    u1=UserProfile(name="Noam Dupond", age=21, email="noam.dupond-rennes.fr")
    print(u1)
except ValueError as e:
    print("Error:", e)
```

Vérification de syntaxe d'un email

N'est pas un email

Résultat:

```
name='Alice Smith' age=30 email='alice@rennes.fr' nickname=None
Error: 1 validation error for UserProfile
email
  value is not a valid email address: An email address must have an @-sign.
[type=value_error, input_value='noam.dupond-rennes.fr', input_type=str]
```

Conversion d'une valeur dictionnaire (JSON) en liste d'arguments pour le constructeur

☞ opérateur **

```
#!/usr/bin/env python3
from pydantic import BaseModel

class Personne(BaseModel):
    nom: str
    prenom: str
    age: int

carol=Personne(nom="Dupont", prenom="Carole", age=20)
print(carol.model_dump_json())

jul_data = {
    'nom':'Merlin',
    'prenom': 'Jules',
     'age': 45
}
jul=Personne(**jul_data)
print(jul.model_dump_json())
```

Définition d'une classe avec variables d'instance et constructeur

Instanciation (appel du constructeur)

initialisation d'un dictionnaire

** valide et convertit la valeur JSON en liste d'arguments pour le construteur

**

Résultat:

```
{"nom":"Dupont","prenom":"Carole","age":20}
{"nom":"Merlin","prenom":"Jules","age":45}
```

Conversion d'une valeur dictionnaire (JSON) en liste d'arguments pour le constructeur

☞ opérateur **

```
from pydantic import BaseModel, EmailStr
from typing import Optional

class User(BaseModel):
    age: int
    email: EmailStr
    is_active: bool = True
    nickname: Optional[str] = None

user_data = {
    "age": "25",
    "email": "john@gmail.fr",
    "is_active": "true"
}

user = User(**user_data)
print(user.age)
print(user.model_dump())
```

Définition d'une classe avec variables d'instance et constructeur

Vérification de syntaxe d'un email

initialisation d'un dictionnaire

String convertie implicitement en int

String convertie implicitement en booléen

** valide et convertit la valeur dictionnaire (JSON) en liste d'arguments pour le construteur

Résultat:

```
25
{'age': 25, 'email': 'john@gmail.fr', 'is_active': True, 'nickname': None}
```

Conversion d'une valeur dictionnaire (JSON) en liste d'arguments pour le constructeur

☞ opérateur **

```
from pydantic import BaseModel, EmailStr
from typing import Optional
class User(BaseModel):
    name: str
    age: int
    email: EmailStr
    is_alive: bool = True
    nickname: Optional[str] = None

john = {
    "name": "John Travolta",
    "age": "71",
    "email": "john@movies.com",
    "is_alive": "true",
    "nickname": "Johnny"
}
user_john = User(**john)
print(user_john.model_dump())

victor = {
    "name": "Victor Hugo",
    "age": "83",
    "email": "victor@books.com",
    "is_alive": "false"
}
user_victor = User(**victor)
print(user_victor.model_dump())
```

Vérification de syntaxe d'un email

** valide et convertit la valeur dictionnaire (JSON) en liste d'arguments pour le construteur

**

**

Résultat:

{'name': 'John Travolta', 'age': 71, 'email': 'john@movies.com', 'is_alive': True, 'nickname': 'Johnny'}

{'name': 'Victor Hugo', 'age': 83, 'email': 'victor@gmail.fr', 'is_alive': False, 'nickname': None}

Conversion d'une valeur dictionnaire (JSON) en liste d'arguments pour le constructeur

☞ opérateur **

```
#!/usr/bin/env python3
from pydantic import BaseModel
class Personne(BaseModel):
    nom: str
    prenom: str
    age: int

carol_data = {
    'nom':'Dupont',
    'prenom': 'Carole',
    'age': 20
}
carol=Personne.model_validate(carol_data)
print(carol)

jul_data = {
    'nom':'Merlin',
    'prenom': 'Jules',
    'age': 45
}

jul=Personne(**jul_data)
print(jul)
```

Définition d'une classe avec variables d'instance et constructeur

initialisation d'un dictionnaire

valide et convertit la valeur dictionnaire (JSON) en liste d'arguments pour le construteur

initialisation d'un dictionnaire

** valide et convertit la valeur dictionnaire (JSON) en liste d'arguments pour le construteur

**

Résultat:

```
25
{'age': 25, 'email': 'john@gmail.fr', 'is_active': True, 'nickname': None}
```