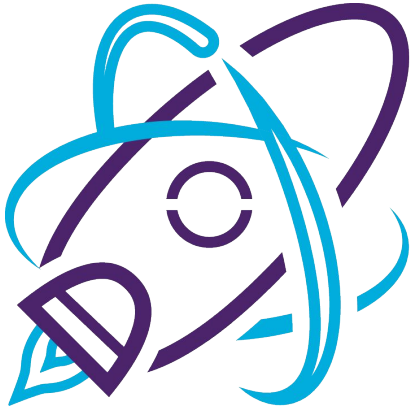


ReactIT

Formation & Recrutement
www.react-it.fr

Formation : Python

ReactIT

Votre formateur

- **Loïc Guillois**
- Développeur Web full stack depuis plus de 10 ans
- Expérience en environnement
  - SSII
  - Grand comptes Banque / Assurance / La Poste
  - Startup (Gamific.TV, La Fourchette.com, Akeneo)
  - Indépendant / freelance
- Enseignant / formateur depuis 5 ans
  - Développement
  - No SQL
  - Devops
  - Objets connectés

Faisons connaissance

Objectifs pédagogiques

- A l'issue de cette formation, vous serez en mesure de :
  - Connaître les possibilités du langage Python
  - Réaliser une application en s'appuyant sur des fonctionnalités avancées
  - Apprendre à écrire des programmes ou scripts grâce au langage Python

Prérequis du module:

  - Connaître un langage de programmation objet

Programme détaillé

Formation sur 3 jours:

- Découverte du langage
- La programmation orientée objet avec Python
- Les bibliothèques de Python
- Fonctionnalités avancées

Théorie et pratique.

Classe: définition

Une **classe** est un type permettant de regrouper dans la même **structure** : les informations (**champs**, propriétés, attributs) relatives à une entité ; les procédures et fonctions permettant de les manipuler (**méthodes**). Champs et méthodes constituent les **membres** de la classe.

Classe et objet

Termes techniques:

- « Classe » est la structure
- « Objet » est une instance de la classe (variable obtenue après **instanciation**)
- « Instanciation » correspond à la création d'un objet
- L'objet est une référence (traité par le garbage collector, destruction explicite inutile)

Classe et objet

Ce n'est pas obligatoire, mais on a toujours intérêt à définir les classes dans des modules. On peut avoir plusieurs classes dans un module.

Première classe

Personne.py

```
#début définition
class Personne :
     """Classe Personne"""

     #constructeur
     def __init__(self):
          #lister les champs
          self.nom = ""
          self.age = 0
          self.salaire = 0.0
     #fin constructeur

#fin définition
```

Première classe

main.py

```
import ModulePersonne as MP

p = MP.Personne()

print(dir(p))

p.nom = input("Nom : ")
p.age = int(input("Age : "))
p.salaire = float(input("Salaire : "))

print(p.nom,", ",p.age,", ",p.salaire)
```

Méthode de classe

On ajoute dans Personne.py

```
#saisie des infos
def saisie(self):
     self.nom = input("Nom : ")
     self.age = int(input("Age : "))
     self.salaire = float(input("Salaire : "))
#fin saisie

#affichage des infos
def affichage(self):
     print("Son nom est ", self.nom)
     print("Son âge : ", self.age)
     print("Son salaire : ", self.salaire)
#fin affichage
```

Méthode de classe

main.py

```
import Personne as MP

p = MP.Personne()

p.saisie()

p.affichage()
```

Gérer une collection d'objets

Python propose des outils pour la gestion des collections d'objets hétérogènes (**tuple, liste, dictionnaire**).

Ils sont opérationnels pour les instances de nos classes.

```
liste = []
a = MP.Personne()
a.saisie()
liste.append(a)
```

Gérer une collection d'objets

- Il est judicieux d'élaborer une classe dédiée à la gestion de la collection
- Les objets de la collection peuvent être différents. Quand ce sont des instances de classes héritières du même ancêtre, on parle de liste **polymorphe**
- Le gestionnaire de collection peut être un dictionnaire, le mécanisme « clé – valeur » (« clé – objet » en l'occurrence pour nous) ouvre des possibilités immenses (ce mécanisme est très en vogue, ex. bases NoSQL)

Héritage

L'héritage permet de construire une **hiérarchie de classes**. Les classes héritières héritent des champs et méthodes de la classe ancêtre.

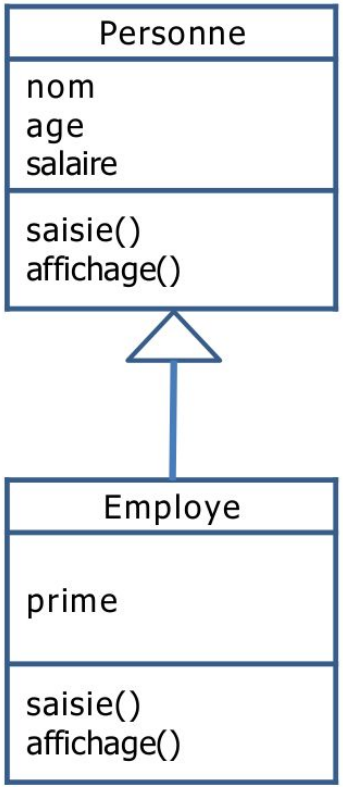
Ce mécanisme nécessite des efforts de modélisation et de conception.

Mais au final, on améliore la lisibilité et la réutilisabilité du code.

Héritage

La classe Employe est une Personne, avec le champ supplémentaire prime. Cela va nécessiter la reprogrammation des méthodes `saisie()` et `affichage()`.

On peut éventuellement ajouter d'autres méthodes spécifiques à `Employe`.



Héritage

```
class Employe(Personne):

     def __init__ (self):
          Personne.__init__(self)
          self.prime = 0.0

     def saisie(self):
          Personne.saisie(self)
          self.prime = float(input("Prime : "))

     def affichage(self):
          Personne.affichage(self)
          print("Sa prime : ", self.prime)
```

Héritage

```
import Personne as MP

e = MP.Employe()
e.saisie()

print(">>Affichage")
e.affichage()
```

Liste polymorphe

Une collection peut contenir des objets de type différent.

Cette caractéristique prend un sens particulier quand les objets sont issus de la même lignée.

Liste polymorphe

```
import Personne as MP
liste = []
n = int(input("Nb de pers : "))
for i in range(0,n):
    code = input("1 Personne, 2 Employé : ")
    if (code == "1"):
        m = MP.Personne()
    else:
        m = MP.Employe()
for p in liste:
    p.saisie()
for p in liste:
    p.affichage()
```

Variable de classe

Une variable de classe est un champ directement accessible sur la classe, et qui est **partagée par toutes les instances de la classe**.

```
class Personne:
    # variable de classe
    compteur = 0

    def __init__ (self):
        self.nom = ""
        self.age = 0
        self.salaire = 0.0
        Personne.compteur += 1
```

La programmation orientée objet

Le mot clef super

Lorsqu'une méthode est redéfini par une classe fille, c'est celle ci qui est appelé. Exemple:

```
Employe.saisie()
```

Si pour une raison particulière vous souhaitez appeler l'implémentation de la classe parente, on utilise le mot clef `super`:

```
super(Personne, employeA).saisie()
```

Classe vide

Pendant la phase de développement , vous pouvez avoir besoin de créer une classe vide, sans attributs ou méthodes supplémentaires. Vous pouvez utiliser le mot clef `pass`.

```
class A():
    pass
```

Classe abstraite

Cette caractéristique prend un sens particulier quand les objets sont issus de la même lignée.

Python ne propose pas d'implémenter un tel type de classe dans son langage.

Classe abstraite

On peut simuler un comportement de classe abstraite grâce aux exceptions.

```
class Animal():

    def manger(self):

        raise NotImplementedError()
```

Héritage multiple

Il est possible pour une classe d'hériter d'une ou plusieurs classes:

```
class A:
    def foo(self):
        return '!'
class B:
    def bar(self):
        return '?'
class C(A, B):
    pass
```

Héritage multiple

L’ordre dans lequel on hérite des parents est important.

Il détermine dans quel ordre les méthodes seront recherchées dans les classes mères. Ainsi, dans le cas où la méthode existe dans plusieurs parents, celle de la première classe sera conservée.

Héritage multiple

Cet ordre dans lequel les classes parentes sont explorées pour la recherche des méthodes est appelé **Method Resolution Order(MRO)**.

On peut le connaître à l'aide de la méthode `mrod` es classes.

```
A.mro()
```

Les mixins

Les mixins sont des classes dédiées à une fonctionnalité particulière, utilisable en héritant d'une classe de base et de ce mixin.

Certains framework comme Django, propose des mixins.

Les mixins

Par exemple, un mixin qui pourrait nous être utile serait une classe avec une méthode `reverse()` pour nous retourner l'objet inversé d'une chaîne de caractère (`str`):

```
class Reversable:
    def reverse(self):
        return self[::-1]

class ReversableStr (Reversable, str):
    pass
```

Les mixins

En pratique, le mixin s'utilise donc par l'instanciation de la classe mixin:

```
s = ReversableStr('abc')

s.reverse()
```

Des questions ?

Restons en contact

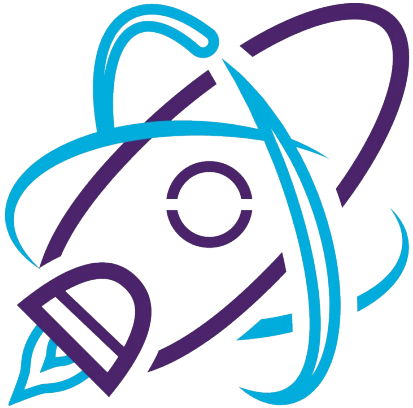
Votre contact:

- React IT
  - Loïc Guillois, président
  - hello@react-it.fr

www.react-it.fr

JUNIOR JOBS

Découvrez aussi http://junior.jobs



ReactIT

Formation & Recrutement
www.react-it.fr

Formation : Python