Langage Python

ISUP, Sorbonne Université

Etienne Guével
Ingénieur de Recherche - SCAI
etienne.guevel@sorbonne-universite.fr

Septembre-Novembre 2025

Partie 2

Programmation objet

Concepts de la programmation objet

Introduction

En python

- Les données sont représentées sous forme d'objets ou par des relations entre les objets

Introduction

En python

- Les données sont représentées sous forme d'objets ou par des relations entre les objets
- Chaque objet possède un identifiant, un type et une valeur

Introduction

En python

- Les données sont représentées sous forme d'objets ou par des relations entre les objets
- Chaque objet possède un identifiant, un type et une valeur
- La fonction `id` revoie l'entier représentant cet identifiant

Introduction

En python

- Les données sont représentées sous forme d'objets ou par des relations entre les objets
- Chaque objet possède un identifiant, un type et une valeur
- La fonction `id` revoie l'entier représentant cet identifiant
- L'identifiant d'un objet ne change jamais après sa création et représente l'adresse mémoire où est stockée l'objet

Introduction

En python

- Les données sont représentées sous forme d'objets ou par des relations entre les objets
- Chaque objet possède un identifiant, un type et une valeur
- La fonction `id` revoie l'entier représentant cet identifiant
- L'identifiant d'un objet ne change jamais après sa création et représente l'adresse mémoire où est stockée l'objet
- L'opérateur `is` compare les identifiants de deux objets

Introduction

En python

- Les données sont représentées sous forme d'objets ou par des relations entre les objets
- Chaque objet possède un identifiant, un type et une valeur
- La fonction `id` revoie l'entier représentant cet identifiant
- L'identifiant d'un objet ne change jamais après sa création et représente l'adresse mémoire où est stockée l'objet
- L'opérateur `is` compare les identifiants de deux objets
- Les objets créés par l'utilisateur sont **mutables** par défaut

Introduction

Définition

En informatique, un objet est un conteneur symbolique qui contient des informations et des mécanismes concernant un sujet

C'est une sorte d'abstraction du monde réel définie par un **ensemble de caractéristiques** (attributs et méthodes).

Par exemple, un oiseau est un canard s'il vole comme un canard, s'il cancane comme un canard et s'il nage comme un canard.

Programmation objet vs procédurale

Programmation procédurale

- Basée sur des procédures (séquences d'instructions, appels de fonctions)

Programmation objet

- Basée sur des classes

Programmation objet vs procédurale

Programmation procédurale

- Basée sur des procédures (séquences d'instructions, appels de fonctions)
- Exécutions étape par étape

Programmation objet

- Basée sur des classes

Programmation objet vs procédurale

Programmation procédurale

- Basée sur des procédures (séquences d'instructions, appels de fonctions)
- Exécutions étape par étape
- Les données et méthodes sont susceptibles de changer durant l'exécution

Programmation objet

- Basée sur des classes
- Les données sont fixées dans la définition même des classes

Programmation objet vs procédurale

Programmation procédurale

- Basée sur des procédures (séquences d'instructions, appels de fonctions)
- Exécutions étape par étape
- Les données et méthodes sont susceptibles de changer durant l'exécution

Programmation objet

- Basée sur des classes
- Les données sont fixées dans la définition même des classes
- C'est une manière de stocker de l'information

Concepts de la programmation objet

Concepts abordés dans cette partie

- Classe

Concepts de la programmation objet

Concepts abordés dans cette partie

- Classe
- Constructeur, attributs et méthodes

Concepts de la programmation objet

Concepts abordés dans cette partie

- Classe
- Constructeur, attributs et méthodes
- Principe d'encapsulation

Concepts de la programmation objet

Concepts abordés dans cette partie

- Classe
- Constructeur, attributs et méthodes
- Principe d'encapsulation
- Héritage

Classe vs objet

Définition

Une classe est un bloc de code définissant l'ensemble des caractéristiques communes à plusieurs objets.
C'est une sorte de plan permettant de créer plusieurs objets.

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

Contructeur

Définition

Le constructeur est une fonction appelée lors de la création de l'objet. Elle permet d'allouer la mémoire nécessaire à l'objet et d'initialiser ses attributs.

Le constructeur est une fonction définie par le mot clé `__init__`

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

Instanciation

Définition

Une instance de classe est un objet dont le comportement et l'**état** sont définis dans la classe.

L'instanciation est l'action de créer un objet à partir d'un modèle (c-à-d la classe) via deux opérations :

- l'allocation qui consiste à réserver un espace mémoire pour stocker l'objet
- l'initialisation qui consiste à fixer l'**état** de l'objet par l'appel au constructeur de la classe

```
point = Point(x=1, y=2)
```

Attributs d'instance

Définition

Les attributs sont des entités qui définissent les propriétés d'un objet. Dans l'exemple, `x` et `y` sont les attributs. Ils sont stockés dans la classe via le mot clé `self`

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

Attributs d'instance

Définition

Les attributs sont des entités qui définissent les propriétés d'un objet. Dans l'exemple, `x` et `y` sont les attributs. Ils sont stockés dans la classe via le mot clé `self`

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

On parle d'attribut d'instance

Attributs de classe : voir ici

Méthodes d'instance

Définition

Les sont des fonctions membres d'une classe. Elles permettent d'**intéragir avec l'état** de l'objet, en particulier le modifier.

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

    def norm(self):
        return math.sqrt(self.x**2 + self.y**2)
```

Méthodes de classe : voir ici

Méthodes statiques

Définition

Les méthodes statiques n'ont accès ni aux attributs de classe, ni aux attributs d'instance. Elles ont un lien logique avec la classe et l'objet mais ne nécessitent pas la connaissance des états.

Par exemple, elles peuvent servir de fonctions utilitaires

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

    @staticmethod
    def to_string(x, y):
        return f"""
        x = {x}
        y = {y}
        """
```

Principe d'encapsulation

Principe d'encapsulation

Le principe d'encapsulation est un des concepts fondammentaux en programmation objet. Il s'agit de regrouper des données dans un objet pour en limiter la manipulation, qui se fait uniquement par des méthodes dédiées.

Les méthodes sont ainsi vues comme une interface destinée à l'utilisateur, ou une sorte de couche de protection vis-à-vis de l'accès aux données

Principe d'encapsulation

Le principe d'encapsulation est un des concepts fondammentaux en programmation objet. Il s'agit de regrouper des données dans un objet pour en limiter la manipulation, qui se fait uniquement par des méthodes dédiées.

Les méthodes sont ainsi vues comme une interface destinée à l'utilisateur, ou une sorte de couche de protection vis-à-vis de l'accès aux données

Le principe d'encapsulation est fortement lié à la portée des méthodes et des attributs (publique, protégée ou privée).

Principe d'encapsulation

Accès et modification des attributs

Selon le principe d'encapsulation, il est nécessaire de définir des méthodes permettant de manipuler les attributs (accès, modification, suppression).

Par exemple, un attribut privé peut être modifié s'il y a une méthode le permettant.

Principe d'encapsulation

Accès et modification des attributs

Selon le principe d'encapsulation, il est nécessaire de définir des méthodes permettant de manipuler les attributs (accès, modification, suppression).

Par exemple, un attribut privé peut être modifié s'il y a une méthode le permettant.

On a en particulier :

- les accesseurs (ou getters) : accès à un attribut
- les mutateurs (ou setters) : modification d'un attribut

Principe d'encapsulation

Problème

Python n'applique pas le principe d'encapsulation : les attributs et les méthodes ne sont pas réellement privés et peuvent être modifiés très facilement y compris leur type !

Exemple d'accès à un attribut

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

```
>>> point = Point(1, 2)
>>> point.x
1
```

Exemple d'accès à un attribut

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

```
>>> point = Point(1, 2)
>>> getattr(point, "x")
1
```

Exemple d'accès à un attribut

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

    def get_x(self):
        return self.x
```

```
>>> point = Point(1, 2)
>>> point.get_x()
1
```

Exemple de modification d'un attribut

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

```
>>> point = Point(1, 2)
>>> point.x += 1
>>> point.x
2
```

Exemple de modification d'un attribut

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

```
>>> point = Point(1, 2)
>>> setattr(point, "x", 2)
>>> point.x
2
```

Exemple de modification d'un attribut

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

    def set_x(self, value):


        self.x = value
```

```
>>> point = Point(1, 2)
>>> point.set_x(2)
>>> point.x
2
```

Exemple de modification d'un attribut

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

    def set_x(self, value):


        self.x = value
```

```
>>> point = Point(1, 2)
>>> point.set_x("azerty")
>>> point.x
"azerty"
```

Exemple de modification d'un attribut

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

    def set_x(self, value):
        if type(value) not in [int, float]:
            sys.exit("Wrong type for value argument")
        self.x = value
```

```
>>> point = Point(1, 2)
>>> point.set_x("azerty")
SystemExit: Wrong type for value argument
```

Comment rendre les attributs privés 1/3

Pour rendre les attributs privés, préfixer le nom avec un “_”

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

Comment rendre les attributs privés 1/3

Pour rendre les attributs privés, préfixer le nom avec un “_”

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

```
>>> point = Point(1, 2)
>>> point._x
???
```

Comment rendre les attributs privés 1/3

Pour rendre les attributs privés, préfixer le nom avec un “_”

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

```
>>> point = Point(1, 2)
>>> point._x
1
```

L'utilisation de “_” n'est qu'une convention d'écriture !

Comment rendre les attributs privés 2/3

Pour rendre les attributs privés, préfixer le nom avec un “__”

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

    def get_x(self):
        return self.__x
```

Comment rendre les attributs privés 2/3

Pour rendre les attributs privés, préfixer le nom avec un “__”

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

    def get_x(self):
        return self.__x
```

```
>>> point = Point(1, 2)
>>> point.__x
???
```

Comment rendre les attributs privés 2/3

Pour rendre les attributs privés, préfixer le nom avec un “__”

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

    def get_x(self):
        return self.__x
```

```
>>> point = Point(1, 2)
>>> point.__x
AttributeError: "Point" object has no attribute "__x"
```

On a rendu privé l’accès à l’attribut `x`

Comment rendre les attributs privés 2/3

Pour rendre les attributs privés, préfixer le nom avec un “__”

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

    def get_x(self):
        return self.__x
```

```
>>> point = Point(1, 2)
>>> point.get_x()
1
```

Mais il reste accessible via le getter associé

Comment rendre les attributs privés 3/3

Mais en réalité

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

Comment rendre les attributs privés 3/3

Mais en réalité

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

```
>>> point = Point(1, 2)
>>> dir(point)
["_Point__x", "_Point__y", ...]
```

Comment rendre les attributs privés 3/3

Mais en réalité

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

```
>>> point = Point(1, 2)
>>> point._Point__x
1
```

L'attribut `x` n'est pas du tout privé en fait ! L'utilisation du préfixe "__" n'a fait que renommer la variable.

@property

Le décorateur `@property` est une autre manière de définir des getters

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

  @property
  def x(self):
      return self.__x
```

L'accès à l'attribut `__x` se fait par

```
>>> point = Point(1, 2)
>>> point.x
1
```

Surcharge des opérateurs

Surcharge des opérateurs : exemple de la fonction `print`

Exemple de la class `Complex`

- Implémenter une classe `Complex` qui stocke l'information d'un nombre complexe soit sa partie entière et sa partie immaginaire
- Implémenter une méthode `print` pour afficher le nombre complexe
- Comparer `c.print()` et `print(c)`

Surcharge des opérateurs : exemple de la fonction `print`

Exemple de la class `Complex`

```
>>> class Complex():
...     def __init__(self, real, imag):
...         self.real = real
...         self.imag = imag
...
...     def print(self):
...         print(f"{self.real} + {self.imag}i")

>>> c = Complex(1, 2)
>>> c.print()
1 + 2i
>>> print(c)
<__main__.Complex object at 0x1101db700>
>>> c + c
TypeError: unsupported operand type(s) for +: 'Complex' and
    'Complex'
```

Surcharge des opérateurs : exemple de la fonction `print`

Exemple de la class `Complex`

- Implémenter une classe `Complex` qui stocke l'information d'un nombre complexe soit sa partie entière et sa partie immaginaire
- Implémenter une méthode `print` pour afficher le nombre complexe
- Comparer `c.print()` et `print(c)`
- Modifier la méthode `print` pour quelle **retourne** la chaîne de caractère à afficher et la renommer en `__str__`. Exécuter à nouveau `print(c)`

Surcharge des opérateurs : exemple de la fonction `print`

Exemple de la class `Complex`

```
>>> class Complex():
...     def __init__(self, real, imag):
...         self.real = real
...         self.imag = imag
...
...     def print(self):
...         print(f"{self.real} + {self.imag}i")
...
...     def __str__(self):
...         return f"{self.real} + {self.imag}i"

>>> c = Complex(1, 2)
>>> c.print()
1 + 2i
>>> print(c)
1 + 2i
```

Surcharge des opérateurs

Définition

Le mécanisme de surcharge des opérateurs consiste à

- redéfinir des fonctions ou opérations déjà définies pour certains types
- pour les adapter à d'autres types d'objets

Quels sont les opérateurs concernés (1/2)

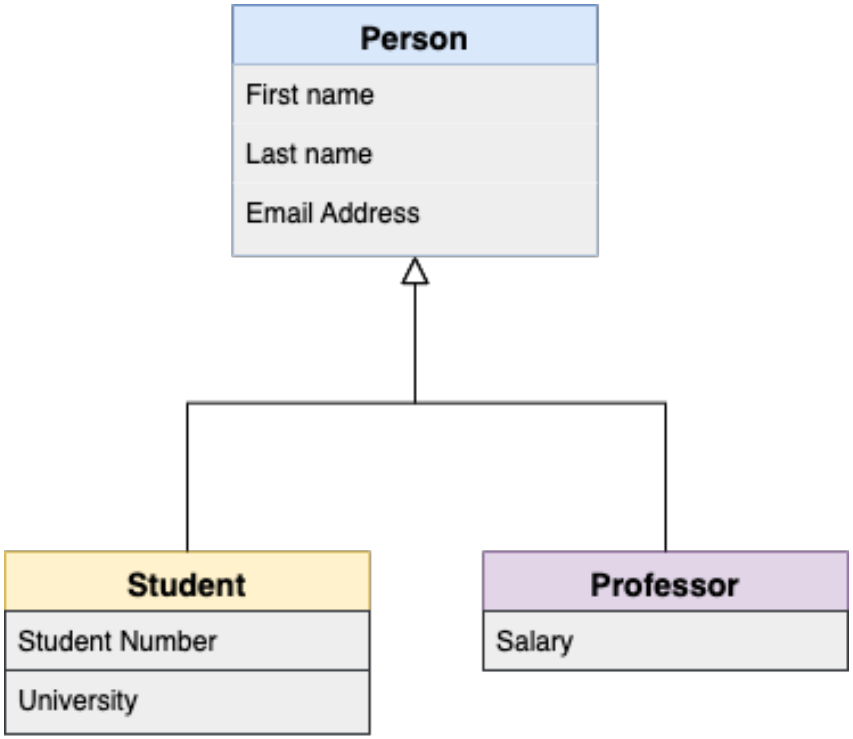
Opérateur	Expression	Interprétation Python
Addition	p1 + p2	p1.__add__(p2)
Soustration	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Puissance	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Division entière	p1 // p2	p1.__floordiv__(p2)
Modulo	p1 % p2	p1.__mod__(p2)
ET binaire	p1 & p2	p1.__and__(p2)
OU binaire	p1 | p2	p1.__or__(p2)

Quels sont les opérateurs concernés (2/2)

Opérateur	Expression	Interprétation Python
Inférieur à	p1 < p2	p1.__lt__(p2)
Inférieur ou égal	p1 <= p2	p1.__le__(p2)
Egal	p1 == p2	p1.__eq__(p2)
Différent	p1 != p2	p1.__ne__(p2)
Supérieur à	p1 > p2	p1.__gt__(p2)
Supérieur ou égal	p1 >= p2	p1.__ge__(p2)

Héritage

Exemple : un étudiant est une personne



Exemple : un étudiant est une personne

```
class Person():
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def __str__(self):
        return f"{self.firstname} {self.lastname}"

class Student():
    def __init__(self, firstname, lastname, university):
        self.firstname = firstname
        self.lastname = lastname
        self.university = university

    def __str__(self):
        return f"{self.firstname} {self.lastname}"

    def get_university(self):
        return self.university
```

Exemple : un étudiant est une personne

```
class Person():
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def __str__(self):
        return f"{self.firstname} {self.lastname}"

class Student():
    def __init__(self, firstname, lastname, university):


        self.university = university




    def get_university(self):
        return self.university
```

Définition

- L'héritage est un mécanisme qui nous permet de créer une nouvelle classe à partir d'une classe existante, en ajoutant de nouveaux attributs et méthodes
- La nouvelle classe est appelée classe fille et la classe existante, classe mère
- On dit que la classe fille "hérite de" ou "étend" la classe mère

```
class mere:
  # corps de la classe mere

class fille(mere):
  # corps de la classe fille
```

Fonction super

```
class Person():
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def __str__(self):
        return f"{self.firstname} {self.lastname}"

class Student():
    def __init__(self, firstname, lastname, university):


        self.university = university




    def get_university(self):
        return self.university
```

Fonction super

```
class Person():
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def __str__(self):
        return f"{self.firstname} {self.lastname}"

class Student(Person):
    def __init__(self, firstname, lastname, university):
        super().__init__(firstname, lastname)

        self.university = university




    def get_university(self):
        return self.university
```

Fonction super

```
class Person():
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    def __str__(self):
        return f"{self.firstname} {self.lastname}"

class Student(Person):
    def __init__(self, firstname, lastname, university):
        Person.__init__(self, firstname, lastname)

        self.university = university




    def get_university(self):
        return self.university
```

Ce qu'il faut retenir

Ce qu'il faut retenir de la programmation objet 1/2

- C'est un concept central en python : tout est objet
- Elle est équivalente à définir des types
- C'est un manière de stocker de l'information
- Les attributs et les méthodes sont systématiquement publiques
- Il est possible de surcharger des opérateurs mathématiques (somme, produit, opérateurs logiques, etc.) pour des types non standards
- L'héritage permet de partager des attributs et des méthodes entre plusieurs objets et éviter la duplication de code

Ce qu'il faut retenir de la programmation objet 2/2

Exemples pratiques en ML :

Data loader en traitement de l'image

keras.io/examples/vision/oxford_pets_image_segmentation

Définition d'un réseau de neurones avec `PyTorch`

pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

Pour aller plus loin

Concept de polymorphisme

w3schools.com/python/python_polymorphism

Classes abstraites

peps.python.org/pep-3119

Classes de données

docs.python.org/3/library/dataclasses