

Atelier d'informatique

Épisode VI : Programmation orientée objet

26 février 2017

Introduction

Définir une classe

Mot-clé `class`

Créer un objet

Définir des attributs

Définir des méthodes

Constructeur

Exercices

Sous-classes

① Introduction

② Définir une classe

Mot-clé `class`

Créer un objet

Définir des attributs

Définir des méthodes

Constructeur

③ Exercices

④ Sous-classes

Introduction

Lors du premier chapitre de ce cours, nous avons vu la notion de *type* d'une variable dans Python. Par exemple, l'utilisation de `type(42)` renvoyait `int` pour « integer » (entier, en anglais). De même, `type("coin")` renvoyait `str` pour « string » (chaîne de caractères).

Introduction

Lors du premier chapitre de ce cours, nous avons vu la notion de *type* d'une variable dans Python. Par exemple, l'utilisation de `type(42)` renvoyait `int` pour « integer » (entier, en anglais). De même, `type("coin")` renvoyait `str` pour « string » (chaîne de caractères).

Le type d'une variable renvoie au nom de sa *classe*. En fait, on ne parle pas vraiment de variable, mais d'*objet*. En Python, tout est un *objet*, et sa *classe* définit ses attributs et les fonctions qui lui sont propres permettant de le manipuler.

Introduction

Par exemple, définissez l'objet `x = 3`. Ensuite, écrivez `x` suivi d'un point, et regardez le menu défilant qui s'affiche dans Pyzo (si vous n'utilisez pas Pyzo mais que vous avez iPython, appuyez sur Tab). Il s'agit de la liste des attributs et *méthodes* (des fonctions attachées à l'objet) que possède l'objet (ou sa classe en soi).

Introduction

Par exemple, définissez l'objet `x = 3`. Ensuite, écrivez `x` suivi d'un point, et regardez le menu défilant qui s'affiche dans Pyzo (si vous n'utilisez pas Pyzo mais que vous avez iPython, appuyez sur Tab). Il s'agit de la liste des attributs et *méthodes* (des fonctions attachées à l'objet) que possède l'objet (ou sa classe en soi).

Les attributs de la forme `__truc__` sont des attributs spéciaux. Par exemple `__add__` est la méthode définissant l'addition, et réservant l'opérateur « `+` » à son utilisation.

Introduction

Par exemple, définissez l'objet `x = 3`. Ensuite, écrivez `x` suivi d'un point, et regardez le menu défilant qui s'affiche dans Pyzo (si vous n'utilisez pas Pyzo mais que vous avez iPython, appuyez sur Tab). Il s'agit de la liste des attributs et *méthodes* (des fonctions attachées à l'objet) que possède l'objet (ou sa classe en soi).

Les attributs de la forme `__truc__` sont des attributs spéciaux. Par exemple `__add__` est la méthode définissant l'addition, et réservant l'opérateur « `+` » à son utilisation.

Exemple 1

Ainsi, `x+2` et `x.__add__(2)` veulent dirent strictement la même chose. La méthode `__str__` permet de convertir l'objet en chaîne de caractère : entrez `x.__str__()`.

Définir une classe

Mot-clé `class`

Pour définir une classe d'objet, on utilise un mot-clé qui n'est pas `def`, réservé pour définir des fonctions (qui sont elles aussi des objets), mais `class`:

Définir une classe

Mot-clé `class`

Pour définir une classe d'objet, on utilise un mot-clé qui n'est pas `def`, réservé pour définir des fonctions (qui sont elles aussi des objets), mais `class`:

```
class MaClasse:  
    pass
```

Définir une classe

Mot-clé `class`

Pour définir une classe d'objet, on utilise un mot-clé qui n'est pas `def`, réservé pour définir des fonctions (qui sont elles aussi des objets), mais `class`:

```
class MaClasse:  
    pass
```

Pour l'instant, la classe `MaClasse` ne fait pas grand chose, comme l'indique le mot-clé `pass`, unique instruction dans sa définition (de même, pour faire une fonction qui ne sert à rien, on peut mettre `pass` dans sa définition). Mais on peut quand même définir des objets de cette classe.

Définir une classe

Créer un objet

Pour cela, on *instancie* l'objet (on le crée) en utilisant le constructeur `MaClasse`, qui fonctionne un peu comme une fonction :

```
x = MaClasse()
```

définit un objet de type `MaClasse` et l'affecte à une variable `x`.

L'objet `x` est ce qu'on appelle une *instance* de la classe `MaClasse`.

Définir une classe

Créer un objet

Exemple 2

Vérifiez le type de x. Que renvoie Python quand vous évaluez x ? Et `print(x)` ?

Définir une classe

Créer un objet

Exemple 2

Vérifiez le type de `x`. Que renvoie Python quand vous évaluez `x` ? Et `print(x)` ?

Évaluer un objet fait appel à sa méthode de *représentation* `__repr__`. Par défaut, les objets d'une classe n'ayant pas de telle méthode précisée renvoient une chaîne donnant le type de l'objet et son emplacement mémoire.

Définir une classe

Créer un objet

Exemple 2

Vérifiez le type de `x`. Que renvoie Python quand vous évaluez `x` ? Et `print(x)` ?

Évaluer un objet fait appel à sa méthode de *représentation* `__repr__`. Par défaut, les objets d'une classe n'ayant pas de telle méthode précisée renvoient une chaîne donnant le type de l'objet et son emplacement mémoire.

Lorsque `print` est appelé sur un objet, il le convertit en chaîne via sa méthode `__str__` et imprime la chaîne. Si elle n'est pas précisée, elle renvoie la même chose que `__repr__` par défaut.

Définir une classe

Créer un objet

Maintenant, si vous écrivez `x` suivi d'un point, vous verrez défiler quand même une liste d'attributs... Ce sont en fait les attributs de base communs à tous les objets en Python, qui assurent que les classes définies par l'utilisateur fonctionnent bien.

Définir une classe

Créer un objet

Maintenant, si vous écrivez `x` suivi d'un point, vous verrez défiler quand même une liste d'attributs... Ce sont en fait les attributs de base communs à tous les objets en Python, qui assurent que les classes définies par l'utilisateur fonctionnent bien.

En particulier, la méthode `__init__` correspond au constructeur utilisé pour définir `x`. Quand vous écrivez `MaClasse()` pour instancier un objet de type `MaClasse`, il fait appel à cette méthode pour créer l'objet.

Définir une classe

Définir des attributs

On distingue deux types d'attributs et méthode : les attributs et méthodes *de classe*, définis sur la classe en soi et accessibles depuis la classe ou ses instances, et les attributs et méthodes *d'objet*, définis et accessibles sur les objets.

Définir une classe

Définir des attributs

On distingue deux types d'attributs et méthode : les attributs et méthodes *de classe*, définis sur la classe en soi et accessibles depuis la classe ou ses instances, et les attributs et méthodes *d'objet*, définis et accessibles sur les objets.

Les attributs d'objet peuvent donc varier d'une instance à l'autre d'une même classe. Les attributs de classe, non.

Pour définir des attributs d'objet, on peut le faire par affectation directe en écrivant quelque chose comme `obj.attribut = valeur`, mais ce n'est pas forcément très utile.

Pour définir des attributs d'objet, on peut le faire par affectation directe en écrivant quelque chose comme `obj.attribut = valeur`, mais ce n'est pas forcément très utile.

Souvent, on définit des attributs au sein des méthodes. Les méthodes sont très importantes, car c'est là que réside tout l'intérêt des objets, sans lesquelles ce ne seraient alors que des tableaux de valeurs étiquetées.

Pour définir des attributs sur la classe en soi, hérités par toute instance de cette classe, on fait des affectations dans la définition de la classe :

```
class MaClasse:  
    sujet = "Coin."
```

Pour définir des attributs sur la classe en soi, hérités par toute instance de cette classe, on fait des affectations dans la définition de la classe :

```
class MaClasse:  
    sujet = "Coin."
```

Maintenant, définir `x = MaClasse()` et évaluer `x.sujet` renvoie toujours la chaîne `"Coin."`.

Comme il s'agit d'un attribut de classe, on peut aussi y accéder en écrivant `MaClasse.sujet`.

Définir une classe

Définir des méthodes

Pour définir une méthode, on définit une fonction au sein de la classe :

Définir une classe

Définir des méthodes

Pour définir une méthode, on définit une fonction au sein de la classe :

```
class MaClasse:  
    def methode(*args, **kwargs):  
        < instructions >
```

et on accède à la méthode via `MaClasse.methode`. Sous cette forme, cela a un intérêt limité : elle n'interagit pas avec les objets et leurs attributs. On ne peut même pas l'utiliser sur une instance.

Définir une classe

Définir des méthodes

Pour définir une méthode, on définit une fonction au sein de la classe :

```
class MaClasse:  
    def methode(*args, **kwargs):  
        < instructions >
```

et on accède à la méthode via `MaClasse.methode`. Sous cette forme, cela a un intérêt limité : elle n'interagit pas avec les objets et leurs attributs. On ne peut même pas l'utiliser sur une instance.

Exemple 3

Définissez une méthode sur `MaClasse` au sens ci-dessus. Définissez une instance `x` de `MaClasse`. Essayez de faire appel à votre méthode. Qu'obtenez vous ?

Pour faire que les méthodes interagissent avec les objets, on passe en argument à la méthode l'objet sur lequel elle est définie :

Pour faire que les méthodes interagissent avec les objets, on passe en argument à la méthode l'objet sur lequel elle est définie :

```
class MaClasse:  
    def methode(self, *args, **kwargs):  
        < instructions >
```

Pour faire que les méthodes interagissent avec les objets, on passe en argument à la méthode l'objet sur lequel elle est définie :

```
class MaClasse:  
    def methode(self, *args, **kwargs):  
        < instructions >
```

Une fois donné un objet `obj`, on peut donc faire appel à la méthode en écrivant `obj.methode(*args,**kwargs)`, qui est en fait `MaClasse.methode(obj,*args,**kwargs)` puisque la méthode est rattachée à la classe.

Ainsi, pour donner des attributs à un objet au sein d'une méthode, on écrit `self.attribut = valeur`:

Ainsi, pour donner des attributs à un objet au sein d'une méthode, on écrit `self.attribut = valeur`:

Exemple 4

```
class Foo:
    def bar(self, x):
        self.attribut = x
        return x
```

Que fait `obj = Foo()` suivi de `obj.bar(42)` ?

Définir une classe

Définir le constructeur

Comme dit plus haut, écrire `MaClasse()` définit un objet de type `MaClasse`, et pour se faire fait appel à la méthode `MaClasse.__init__`, appelée *constructeur*.

Définir une classe

Définir le constructeur

Comme dit plus haut, écrire `MaClasse()` définit un objet de type `MaClasse`, et pour se faire fait appel à la méthode `MaClasse.__init__`, appelée *constructeur*.

Pour le modifier, notamment pour qu'il prenne des arguments, il faut définir cette méthode au sein de la classe :

```
class MaClasse:
    def __init__(self, *args, **kwargs):
        < instructions >
```

Parmi ces instructions, il y a notamment des définitions d'attributs.

Définir une classe

Définir le constructeur

Comme dit plus haut, écrire `MaClasse()` définit un objet de type `MaClasse`, et pour se faire fait appel à la méthode `MaClasse.__init__`, appelée *constructeur*.

Pour le modifier, notamment pour qu'il prenne des arguments, il faut définir cette méthode au sein de la classe :

```
class MaClasse:
    def __init__(self, *args, **kwargs):
        < instructions >
```

Parmi ces instructions, il y a notamment des définitions d'attributs.

Attention Le constructeur ne doit pas renvoyer de valeurs, donc il n'y a jamais de `return`.

Exemple 5

Que font les objets du type suivant ?

```
class Grapheur:
    def __init__(self, fonction):
        self.func = fonction

    def dessiner(self, a, b, N=100):
        X = np.linspace(a, b, N)
        Y = self.func(X)
        plt.figure(0)
        plt.grid(True)
        plt.plot(X, Y)
        plt.show()
```

Exercices

Exercice 1 (Fractions)

On va créer une classe `Fraction` pour représenter des fractions.

Exercices

Exercice 1 (Fractions)

On va créer une classe `Fraction` pour représenter des fractions.

- Définir une classe `Fraction`. Son constructeur prendra en arguments deux entiers `a` et `b` qui seront respectivement son numérateur et son dénominateur. On définira les attributs `num` et `den` qui correspondront au numérateur et dénominateur.

Exercices

Exercice 1 (Fractions)

On va créer une classe `Fraction` pour représenter des fractions.

- Définir une classe `Fraction`. Son constructeur prendra en arguments deux entiers `a` et `b` qui seront respectivement son numérateur et son dénominateur. On définira les attributs `num` et `den` qui correspondront au numérateur et dénominateur.
- Définir la méthode `__float__` qui convertit une fraction en nombre à virgule. On renverra donc la valeur a/b .

Exercices

Exercice 1 (Fractions)

On va créer une classe `Fraction` pour représenter des fractions.

- Définir une classe `Fraction`. Son constructeur prendra en arguments deux entiers `a` et `b` qui seront respectivement son numérateur et son dénominateur. On définira les attributs `num` et `den` qui correspondront au numérateur et dénominateur.
- Définir la méthode `__float__` qui convertit une fraction en nombre à virgule. On renverra donc la valeur `a/b`.
- Définir la méthode `__str__` suivante, qui convertit un objet de type `Fraction` en chaîne de caractère :

Exercice 1 (Fractions)

On va créer une classe Fraction pour représenter des fractions.

- Définir une classe Fraction. Son constructeur prendra en arguments deux entiers a et b qui seront respectivement son numérateur et son dénominateur. On définira les attributs num et den qui correspondront au numérateur et dénominateur.
- Définir la méthode `__float__` qui convertit une fraction en nombre à virgule. On renverra donc la valeur a/b.
- Définir la méthode `__str__` suivante, qui convertit un objet de type Fraction en chaîne de caractère :

```
def __str__(self):  
    return "{}/{ {}".format(self.num, self.den)
```

Exercices

Exercice 1 (Fractions)

On va créer une classe `Fraction` pour représenter des fractions.

- Définir une classe `Fraction`. Son constructeur prendra en arguments deux entiers `a` et `b` qui seront respectivement son numérateur et son dénominateur. On définira les attributs `num` et `den` qui correspondront au numérateur et dénominateur.
- Définir la méthode `__float__` qui convertit une fraction en nombre à virgule. On renverra donc la valeur `a/b`.
- Définir la méthode `__str__` suivante, qui convertit un objet de type `Fraction` en chaîne de caractère :

```
def __str__(self):  
    return "{}/{ {}".format(self.num, self.den)
```

Définir la fraction `f = Fraction(1,3)`. Que fait `print(f)` ?

Exercice 1 (suite)

- Définir une méthode d'addition `__add__(self, other)`, où `other` est un autre objet de type `Fraction`, qui ajoute les deux fractions. On peut l'utiliser en écrivant `frac1 + frac2` où `frac1` et `frac2` sont deux fractions, plutôt que `frac1.__add__(frac2)`.

Exercice 1 (suite)

- Définir une méthode d'addition `__add__(self, other)`, où `other` est un autre objet de type `Fraction`, qui ajoute les deux fractions. On peut l'utiliser en écrivant `frac1 + frac2` où `frac1` et `frac2` sont deux fractions, plutôt que `frac1.__add__(frac2)`.

Indication On rappelle que

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

Exercice 1 (suite)

- Définir une méthode d'addition `__add__(self,other)`, où `other` est un autre objet de type `Fraction`, qui ajoute les deux fractions. On peut l'utiliser en écrivant `frac1 + frac2` où `frac1` et `frac2` sont deux fractions, plutôt que `frac1.__add__(frac2)`.

Indication On rappelle que

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

- Définir de même une soustraction `__sub__(self,other)`.

Exercice 1 (suite)

- Définir une méthode d'addition `__add__(self,other)`, où `other` est un autre objet de type `Fraction`, qui ajoute les deux fractions. On peut l'utiliser en écrivant `frac1 + frac2` où `frac1` et `frac2` sont deux fractions, plutôt que `frac1.__add__(frac2)`.

Indication On rappelle que

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

- Définir de même une soustraction `__sub__(self,other)`.
- Définir une multiplication `__mul__(self,other)` et de division `__truediv__(self,other)`.
- Implémenter une méthode qui simplifie une fraction (au sens où le numérateur et le dénominateur n'ont pas de facteurs communs).

Sous-classes

Le principe des sous-classes est de « spécialiser » une classe, en ajoutant des attributs et méthodes, mais en conservant la version initiale, par exemple pour définir une autre sous-classe.

Sous-classes

Le principe des sous-classes est de « spécialiser » une classe, en ajoutant des attributs et méthodes, mais en conservant la version initiale, par exemple pour définir une autre sous-classe.

Une sous-classe a la particularité d'*hériter* les méthodes et attributs de sa classe mère.

Sous-classes

Le principe des sous-classes est de « spécialiser » une classe, en ajoutant des attributs et méthodes, mais en conservant la version initiale, par exemple pour définir une autre sous-classe.

Une sous-classe a la particularité d'*hériter* les méthodes et attributs de sa classe mère.

Étant une classe initiale Foo, on crée la sous-classe Bar de la façon suivante :

```
class Foo(Bar):  
    < instructions >
```

La nouvelle classe Bar aura donc les attributs et méthodes définis par les instructions données, et ceux de la classe Bar.

Étant une classe initiale Foo, on crée la sous-classe Bar de la façon suivante :

```
class Foo(Bar):  
    < instructions >
```

La nouvelle classe Bar aura donc les attributs et méthodes définis par les instructions données, et ceux de la classe Bar.

Remarque Si une méthode ou un attribut défini dans Foo a le même nom qu'une méthode ou un attribut défini dans Bar, ceux de la sous-classe remplacent ceux de la classe mère.

Étant une classe initiale Foo, on crée la sous-classe Bar de la façon suivante :

```
class Foo(Bar):  
    < instructions >
```

La nouvelle classe Bar aura donc les attributs et méthodes définis par les instructions données, et ceux de la classe Bar.

Remarque Si une méthode ou un attribut défini dans Foo a le même nom qu'une méthode ou un attribut défini dans Bar, ceux de la sous-classe remplacent ceux de la classe mère.

Faites attention au constructeur si vous le redéfinissez. Si c'est le cas et que vous voulez quand même effectuer les instructions de la classe mère, écrivez `super().__init__(self,*args,**kwargs)`. `super()` fait référence à la classe mère.