Atelier d'informatique

Épisode VI : Programmation orientée objet

27 février 2017

Introduction

Lors du premier chapitre de ce cours, nous avions 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 avions 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`.

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

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`.

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 méthide `__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.

Exemple 2

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

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.

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

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()
```

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.

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 `__repr__` suivante, qui convertit un objet de type `Fraction` en chaîne de caractère :

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 `__repr__` suivante, qui convertit un objet de type `Fraction` en chaîne de caractère :

```
def __repr__(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 `__repr__` suivante, qui convertit un objet de type `Fraction` en chaîne de caractère :

```
def __repr__(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).

Exercice 2 (Vecteurs)

On va maintenant créer une classe `Vecteur` qui servira à représenter les vecteurs. On commencera par des vecteurs du plan dans un repère $\mathfrak{R} = (O, \vec{\imath}, \vec{\jmath})$ fixé.

- Définir la classe `Vecteur`. Son constructeur (méthode `__init__`) prendra en arguments deux nombres réels `x` et `y` et les stockera dans l'attribut `coord` qui sera un `tuple`.

Exercice 2 (Vecteurs)

On va maintenant créer une classe `Vecteur` qui servira à représenter les vecteurs. On commencera par des vecteurs du plan dans un repère $\mathfrak{R} = (O, \vec{\imath}, \vec{\jmath})$ fixé.

- Définir la classe `Vecteur`. Son constructeur (méthode `__init__`) prendra en arguments deux nombres réels `x` et `y` et les stockera dans l'attribut `coord` qui sera un `tuple`.
- Définir une addition `__add__`. On rappelle que la somme $\vec{u} + \vec{v}$ a pour coordonnées les sommes de celles de $\vec{u}$ et $\vec{v}$.

Exercice 2 (Vecteurs)

On va maintenant créer une classe `Vecteur` qui servira à représenter les vecteurs. On commencera par des vecteurs du plan dans un repère $\mathfrak{R} = (O, \vec{\imath}, \vec{\jmath})$ fixé.

- Définir la classe `Vecteur`. Son constructeur (méthode `__init__`) prendra en arguments deux nombres réels `x` et `y` et les stockera dans l'attribut `coord` qui sera un `tuple`.
- Définir une addition `__add__`. On rappelle que la somme $\vec{u} + \vec{v}$ a pour coordonnées les sommes de celles de $\vec{u}$ et $\vec{v}$.
- Définir un produit par un réel `mult`, tel que `u.mult(k)` renvoie le vecteur $k\vec{u}$.

Exercice 2 (Vecteurs)

On va maintenant créer une classe `Vecteur` qui servira à représenter les vecteurs. On commencera par des vecteurs du plan dans un repère $\mathfrak{R} = (O, \vec{\imath}, \vec{\jmath})$ fixé.

- Définir la classe `Vecteur`. Son constructeur (méthode `__init__`) prendra en arguments deux nombres réels `x` et `y` et les stockera dans l'attribut `coord` qui sera un `tuple`.
- Définir une addition `__add__`. On rappelle que la somme $\vec{u} + \vec{v}$ a pour coordonnées les sommes de celles de $\vec{u}$ et $\vec{v}$.
- Définir un produit par un réel `mult`, tel que `u.mult(k)` renvoie le vecteur $k\vec{u}$.
- Définir une méthode de représentation `__repr__`. Elle renverra la chaîne constituée du mot « `Vecteur` » suivie des coordonnées.

Exercice 2 (Vecteurs)

On va maintenant créer une classe `Vecteur` qui servira à représenter les vecteurs. On commencera par des vecteurs du plan dans un repère $\mathfrak{R} = (O, \vec{\imath}, \vec{\jmath})$ fixé.

- Définir la classe `Vecteur`. Son constructeur (méthode `__init__`) prendra en arguments deux nombres réels `x` et `y` et les stockera dans l'attribut `coord` qui sera un `tuple`.
- Définir une addition `__add__`. On rappelle que la somme $\vec{u} + \vec{v}$ a pour coordonnées les sommes de celles de $\vec{u}$ et $\vec{v}$.
- Définir un produit par un réel `mult`, tel que `u.mult(k)` renvoie le vecteur $k\vec{u}$.
- Définir une méthode de représentation `__repr__`. Elle renverra la chaîne constituée du mot « `Vecteur` » suivie des coordonnées.
  Indication On peut concaténer deux chaînes en utilisant l'addition « $+$ », et convertir un `tuple` en chaîne via `str`.

Exercice 3 (Vecteurs, bis)

On va ajouter plus de méthodes à notre classe `Vecteur`, permettant de faire de la géométrie plus avancée.

- Définir une méthode `colineaire` testant la colinéarité de deux vecteurs. `u.colineaire(v)` renverra `True` si $\vec{u}$ et $\vec{v}$ sont colinéaires, et `False` sinon.

Exercice 3 (Vecteurs, bis)

On va ajouter plus de méthodes à notre classe `Vecteur`, permettant de faire de la géométrie plus avancée.

- Définir une méthode `colineaire` testant la colinéarité de deux vecteurs. `u.colineaire(v)` renverra `True` si $\vec{u}$ et $\vec{v}$ sont colinéaires, et `False` sinon.
- (**Première S**) Définir une méthode `dot(self,other)` qui calcule le produit scalaire $\vec{u} \cdot \vec{v}$.
  Rappel Si $\vec{u} = \begin{pmatrix} x \\ y \end{pmatrix}$ et $\vec{v} = \begin{pmatrix} x' \\ y' \end{pmatrix}$ sont des vecteurs, leur produit scalaire (canonique) est défini par

$$\vec{u} \cdot \vec{v} = xx' + yy'.$$

- Définir une méthode `orthogonal`: `u.orthogonal(v)` teste si les vecteurs $\vec{u}$ et $\vec{v}$ sont orthogonaux. On rappelle que c'est le cas si et seulement si $\vec{u} \cdot \vec{v} = 0$. On utilisera la méthode `dot`.
- Définir une méthode `norm` qui calcule la norme d'un vecteur. On rappelle qu'elle est définie par $\|\vec{u}\| = \sqrt{x^2 + y^2}$.

Exercice 4 (Vecteurs en dimension supérieure)

En général, un vecteur peut avoir n coordonnées, où n est un entier ≥ 2 appelé *dimension* (un vecteur de dimension 1 n'est qu'un nombre réel).

Modifiez la classe `Vecteur` de l'exercice 2 pour qu'elle représente des vecteurs à n coordonnées. Le constructeur définira un attribut `dim` correspondant à la dimension du vecteur.

Exercice 5 (Polynômes)

Un *polynôme* est une fonction $P : \mathbb{R} \longrightarrow \mathbb{R}$ de la forme

$$P(x) = a_n x^n + \cdots + a_1 x + a_0.$$

où les $a_i, 0 \leq i \leq n$ sont des réels appelés *coefficients* de f, et lorsque $a_n \neq 0$, l'entier naturel n est appelé *degré* de f.

Exercice 5 (Polynômes)

Un *polynôme* est une fonction $P : \mathbb{R} \longrightarrow \mathbb{R}$ de la forme

$$P(x) = a_n x^n + \cdots + a_1 x + a_0.$$

où les $a_i, 0 \leq i \leq n$ sont des réels appelés *coefficients* de f, et lorsque $a_n \neq 0$, l'entier naturel n est appelé *degré* de f.

On peut les représenter par un vecteur des *coefficients* des termes x^n,

$$(a_0, \ldots, a_n)$$

L'ensemble des polynômes est noté $\mathbb{R}[x]$.

Exercice 5 (Polynômes)

Un *polynôme* est une fonction $P : \mathbb{R} \longrightarrow \mathbb{R}$ de la forme

$$P(x) = a_n x^n + \cdots + a_1 x + a_0.$$

où les $a_i, 0 \leq i \leq n$ sont des réels appelés *coefficients* de f, et lorsque $a_n \neq 0$, l'entier naturel n est appelé *degré* de f.

On peut les représenter par un vecteur des *coefficients* des termes x^n,

$$(a_0, \ldots, a_n)$$

L'ensemble des polynômes est noté $\mathbb{R}[x]$.

Définissez une classe `Polynome`, qui sera une sous-classe de la classe `Vecteur`. On définira la méthode `func(self,x)`, telle que `P.func(x)` calcule $P(x)$.

Exercice 5 (Polynômes)

Un *polynôme* est une fonction $P : \mathbb{R} \longrightarrow \mathbb{R}$ de la forme

$$P(x) = a_n x^n + \cdots + a_1 x + a_0.$$

où les $a_i, 0 \leq i \leq n$ sont des réels appelés *coefficients* de f, et lorsque $a_n \neq 0$, l'entier naturel n est appelé *degré* de f.

On peut les représenter par un vecteur des *coefficients* des termes x^n,

$$(a_0, \ldots, a_n)$$

L'ensemble des polynômes est noté $\mathbb{R}[x]$.

Définissez une classe `Polynome`, qui sera une sous-classe de la classe `Vecteur`. On définira la méthode `func(self,x)`, telle que `P.func(x)` calcule $P(x)$.

On définira aussi une méthode `graphe(self,a,b)` qui dessine la courbe représentative $\mathcal{C}_f$ du polynôme sur l'intervalle $[a, b]$.

Exerices

Corrections

Correction de l'exercice 1

```
class Fraction:
    def __init__(self,a,b):
        # Teste si c'est bien une fraction
        if type(a) != int or type(b) != int:
            raise TypeError("Numérateur" +\
                "et dénominateur doivent être entiers")
        if b==0:
            mess = "Division par zéro !"
            raise ZeroDivisionError(mess)
        self.num = a
        self.den = b
```

Correction de l'exercice 1

```
class Fraction:
    def __init__(self,a,b):
        # Teste si c'est bien une fraction
        if type(a) != int or type(b) != int:
            raise TypeError("Numérateur" +\
                "et dénominateur doivent être entiers")
        if b==0:
            mess = "Division par zéro !"
            raise ZeroDivisionError(mess)
        self.num = a
        self.den = b

    def __float__(self):
        return self.num/self.den
```

Correction de l'exercice 1

```
class Fraction:
    def __init__(self,a,b):
        # Teste si c'est bien une fraction
        if type(a) != int or type(b) != int:
            raise TypeError("Numérateur" +\
                "et dénominateur doivent être entiers")
        if b==0:
            mess = "Division par zéro !"
            raise ZeroDivisionError(mess)
        self.num = a
        self.den = b

    def __float__(self):
        return self.num/self.den

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

Correction de l'exercice 1 (suite)

```
def __add__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*d+b*c,b*d)
```

Correction de l'exercice 1 (suite)

```
def __add__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*d+b*c,b*d)

def __mul__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*c,b*d)
```

Correction de l'exercice 1 (suite)

```
def __add__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*d+b*c,b*d)

def __mul__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*c,b*d)

def __sub__(self,other):
    # astuce consistant en multiplier la
    # 2eme fraction par -1 et ajouter
    f = Fraction(-1,1)*other
    return self + other
```

Correction de l'exercice 1 (suite)

```
def __add__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*d+b*c,b*d)

def __mul__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*c,b*d)

def __sub__(self,other):
    # astuce consistant en multiplier la
    # 2eme fraction par -1 et ajouter
    f = Fraction(-1,1)*other
    return self + other

def __truediv__(self,other):
    a,b = self.num,self.den
    c,d = other.num,other.den
    return Fraction(a*d,b*c)
```

Correction de l'exercice 2

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

Correction de l'exercice 2

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

    def __add__(self,other):
        x , y = self.coord
        x1,y1 = other.coord
        return Vecteur(x+x1,y+y1)
```

Correction de l'exercice 2

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

    def __add__(self,other):
        x , y = self.coord
        x1,y1 = other.coord
        return Vecteur(x+x1,y+y1)

    def mult(self,k):
        x, y = self.coord
        return Vecteur(k*x,k*y)
```

Correction de l'exercice 2

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

    def __add__(self,other):
        x , y = self.coord
        x1,y1 = other.coord
        return Vecteur(x+x1,y+y1)

    def mult(self,k):
        x, y = self.coord
        return Vecteur(k*x,k*y)

    def __repr__(self):
        return "Vecteur"+str(self.coord)
```

Correction de l'exercice 3

```
def colineaire(self,other):
    x, y = self.coord
    x1,y1 = other.coord
    return x*y1 - y*x1 == 0
```

Correction de l'exercice 3

```
def colineaire(self,other):
    x, y = self.coord
    x1,y1 = other.coord
    return x*y1 - y*x1 == 0

def dot(self,other):
    x, y = self.coord
    x1,y1 = other.coord
    return x*x1 + y*y1
```

Correction de l'exercice 3

```
def colineaire(self,other):
    x, y = self.coord
    x1,y1 = other.coord
    return x*y1 - y*x1 == 0

def dot(self,other):
    x, y = self.coord
    x1,y1 = other.coord
    return x*x1 + y*y1

def orthogonal(self,other):
    return self.dot(other) == 0
```

Correction de l'exercice 3

```
def colineaire(self,other):
    x, y = self.coord
    x1,y1 = other.coord
    return x*y1 - y*x1 == 0

def dot(self,other):
    x, y = self.coord
    x1,y1 = other.coord
    return x*x1 + y*y1

def orthogonal(self,other):
    return self.dot(other) == 0

def norm(self):
    from math import sqrt
    return sqrt(self.dot(self))
```

Correction de l'exercice 4

Voici ce qui est modifié :

```
def __init__(self,*args):
    self.dim = len(args)
    self.coord = tuple(args)
```

Correction de l'exercice 4

Voici ce qui est modifié :

```
def __init__(self,*args):
    self.dim = len(args)
    self.coord = tuple(args)

def __add__(self,other):
    # couples (x,y) où x est une coordonnée de
    # self et y la coordonnée correspondante de other
    coords = zip(self.coord,other.coord)
    # arguments à passer au constructeur,
    # somme des coordonnées des vecteurs
    args = (x+y for x,y in coords)
    return Vecteur(*args)
```

Correction de l'exercice 4

Voici ce qui est modifié :

```
def __init__(self,*args):
    self.dim = len(args)
    self.coord = tuple(args)

def __add__(self,other):
    # couples (x,y) où x est une coordonnée de
    # self et y la coordonnée correspondante de other
    coords = zip(self.coord,other.coord)
    # arguments à passer au constructeur,
    # somme des coordonnées des vecteurs
    args = (x+y for x,y in coords)
    return Vecteur(*args)

def mult(self,k):
    # arguments à passer au constructeur
    args = (k*x for x in self.coord)
    return Vecteur(args)
```

Correction de l'exercice 5

On sous-classe `Vecteur` et on garde le constructeur standard, qui permet de représenter le polynôme par un vecteur.

```
class Polynome(Vecteur):
    def func(self,x):
        coeffs = self.coords
        n = self.dim
        return sum(coeffs[i]*x**i for i in range(n))

    def graphe(self,a,b):
        from numpy import linspace
        import matplotlib.pyplot as plt
        Xar = linspace(a,b,100)
        valeurs = self.func(Xar)
        fig = plt.figure(0)
        plt.grid(True)
        plt.plot(Xar, valeurs)
        plt.show()
        return fig
```