Python

Bastien Gorissen & Thomas Stassin

Année 2016

Table des matières

Chapitre 1

Level 1

1.1 Introduction

Dans le premier cours ("Introduction à la programmation"), on avait toucher aux principes fondamentaux de la programmation dîtes *procédurale*. Dans ce cours, qui peut être vu comme une suite, nous irons plus en profondeur dans Python afin d'apprendre les bases de la programmation *orientée objets*.

1.1.1 Programmation orientée objets en Python

Ce cours, reste un cours de Python, donc, bien que nous verrons la programmation orientée objet, nous n'irons pas en dehors de ce que Python propose. J'entend par là que certains "principes" de l'orienté objets, ne correspondant pas à la philosophie de Python [1], ne sont pas appliqué par ce langage, et donc ne seront pas illustré par ce cours [2].

1. "Keep it simple" (Garde çà simple).
2. Ces principe seront surement abordé dans d'autres cours comme le cours de C#.

1.2 Classes ?, Objets ?

Des *Objets* vous en avez déjà rencontré pas mal sans les avoir nommé comme tel. J'en veux pour preuve que dans Python, tout est *objet*[3].

Exemple :

```
a = 1
```

Dans ce morceau de code, `a` est un *object*. D'ailleurs, tant qu'on y est, la classe de `a` est `int`. Vous allez me dire que `int` c'est son *type*. C'est vrai aussi, en Python le *type* et la *classe* sont assez similaire.

Bon cela ne nous dit pas vraiment ce qu'est une classe.

1.2.1 RPG Like

On peut faire le parallèle entre les *classes* d'un langage de programmation et celle d'un RPG[4] et pareille entre les *objets* et les personnages. Une *classe* regroupe toute les caractéristiques qui définisse un *type*, de la même façon que dans un RPG, la classe "mage" définit ce qu'est un mage, quels types de magie il peut utiliser et quels sont ces caractérisques.

Et de la même façon que dans un groupe de personnages de RPG vous pouvez avoir plusieurs mages, étant différent l'un de l'autre ; dans votre code vous aurez plusieurs *objets* partageant la même *classe*.

```
a = 1
b = 3
```

Ici, `a` et `b` ont tout les deux pour classe `int`. On dit que ce sont des *instances* de la *classe* `int`.

1.2.2 Attributs

L'une des caractéristiques d'une *classe* est qu'elle peut posséder des *attributs*, j'irais même plus loin : il est rare qu'elle n'en possède pas.

Mais qu'est-ce qu'un attribut ?

3. Ou du moins tout ce qui n'est pas une instruction
4. Role playing game (jeu de rôles)

Pour faire simple, un attribut est une *variable* de *classe*.

```
hero = Hero()
print(hero.position)
```

Dans l'exemple ci-dessus, on voit que la variable `hero` qui est une instance de la *class* `Hero`, possède un attribut `position`. Cet *attribut* définit la position d'un `Hero` et est donc l'une de ses caractéristique.

La plupart du temps, les attributs sont modifiables directement

```

hero = Hero()
hero.position = (3, 3)
```

Ci-dessus, le code modifie l'*attribut* `position` pour le rendre égale à `(3, 3)`.

1.2.3 Un peu de méthode

L'autre caractéristique d'une *classe* sont ses *méthodes*. Si un *attribut* peut être vu comme une *variable* de *classe*, une méthode peut être comparée à une *fonction*.

En clair, une *méthode* est une *fonction* qui ne concerne que la *classe* et qui n'affectera que l'*instance* qui l'appel. On peut voir les *méthodes* comme des actions actives ou passives, disponible pour une *classe*.

```
hero = Hero()
hero.move()
```

Dans cette exemple, on utilise la *méthode* `move` de la *classe* `Hero`. Cette *méthode* agit sur la position du héros.

4. Pour ce cours, çà sera tout le temps le cas.

1.2.4 Première classe

Il est bien beau de savoir ce qu'est une classe (même si avouons-le, c'est encore flou). Mais il est encore mieux de savoir en faire soi-même.

```
class Hero:

    def __init__(self):
        self.position = (0, 0)

    def move(self)
        x, y = self.position
        self.position = x + 1, y
```

Voici la *classe* `Hero` enfin dévoilée.

Regardons-la de plus prêt :

La première ligne est la déclaration de la classe. Pour écrire une *classe*, il faut écrire "`class`" suivit du nom de la *classe*[5]. Comme souvent en Python, tout le code lié à la *classe*, sera indenté d'un rang par rapport à la déclaration de la *classe*.

Ensuite nous avons la définition de la première méthode, nommée `__init__`[6]. Cette première méthode est nommée "*constructeur*". Le constructeur est une méthode un peu à part. En effet, elle n'est appelée qu'une seule fois, lors de la création de l'*objet*.

Dans le cas de la classe `Hero` le *construteur* définit l'*attribut* `position` comme valant `(0, 0)`.

En gros, c'est le *constructeur* qui construit l'objet.

```
hero = Hero()
```

C'est au moment de l'*affectation* de la variable `hero` par une *instance* de la classe `Hero` que le constructeur est appelé.

Une deuxième méthode est ensuite définie, c'est `move` qui fait en sorte que la position du héros bougent de 1 sur l'axe des x.

5. Le nom des classes que vous créerez prendra toujours une majuscle
6. Vous noterez qu'une méthode se définit comme une fonction.

1.2.5 "self", le paramêtre fantome

Si vous avez bien fait attention, vous aurez remarqué que chacune des *méthodes* de la classe `Hero`, possède un paramêtre `self`. Vous aurez aussi remarqué que lors de l'appel de la *méthode* `move`, on n'avait fait appel à aucun argument. Quel est donc ce mystérieux paramêtre ?

En Python[7], lors de l'appel d'une méthode, l'*instance* de la *classe* elle-même est passée en première argument, de manière implicite. C'est l'interpréteur lui-même qui le rajoute de manière automatique.

donc quand vous écrivez ceci :

```
hero = Hero()
hero.move()
```

L'interpréteur, lui va exécuter ceci :

```

hero = Hero(hero)
hero.move(hero)
```

Donc, les *méthodes* doivent toujours avoir un paramêtre, qui par convention est nommé `self` (parce que c'est l'objet lui-même), afin d'accueillir cette argument "fantôme" ajouté par l'interpréteur.

Ce paramêtre a d'ailleurs beaucoup d'utilité, car il permet d'accéder aux *attributs* de l'*objet* au sein des méthodes.

```
class Hero:

    def __init__(self):
        self.position = (0, 0)

    def move(self)
        x, y = self.position
        self.position = x + 1, y
```

Dans la méthode `move` d'ailleurs, intervient sur la position du héros grâce à l'attribut `position`. Elle accède a cette attribut via le paramêtre `self`.

Bien-sûr, il est possible d'avoir des *méthodes* avec d'autres arguments que `self`[8], mais lui, je le répète, doit toujours être présent dans la déclaration

7. Vous ne trouverez pas cette spécificité dans d'autre langage.
8. Y compris dans le *constructeur*

d'une méthode.