



Ressourcesinformatiques

Apprendre la Programmation Orientée Objet avec le langage Python

(avec exercices pratiques
et corrigés)

Vincent BOUCHENY

Fichiers complémentaires
à télécharger



Chapitre 1

L'émergence de la POO

- 1. Ancêtres de la POO 9
- 2. Besoin d'un langage de plus haut niveau 11

Chapitre 2

Les concepts de la POO

- 1. Modélisation 13
- 2. Objet et classe 16
- 3. Encapsulation 18
- 4. Agrégation et composition 21
 - 4.1 Agrégation 21
 - 4.2 Composition 22
- 5. Interface 23
- 6. Énumération 24
- 7. Héritage 25
 - 7.1 Héritage simple 25
 - 7.2 Classe abstraite 27
 - 7.3 Héritage multiple 28
 - 7.4 Du bon usage de l'héritage 31
- 8. Diagramme UML 32
 - 8.1 Structure vs comportement 32
 - 8.2 Diagramme de cas d'utilisation 32
 - 8.3 Diagramme de séquence 34
- 9. Exercice corrigé 35
 - 9.1 Repérage des classes 35
 - 9.2 Contenants et contenus 36
 - 9.3 Membres 42

Chapitre 3

Présentation de l'environnement Python

1. Python 2 ou Python 3 ?	49
2. Installation	51
2.1 python.org	51
2.2 Windows	52
2.3 Mac OS X	56
2.4 Unix/Linux	57
3. Outillage	57
3.1 pip	57
3.2 IDLE	58
3.3 PyCharm	59
4. Quelques concepts de base de Python	61
4.1 Introduction	61
4.2 Philosophie	61
4.3 Langage interprété et compilé	62
4.4 Duck typing	63
4.5 Modules	64

Chapitre 4

Les concepts de la POO avec Python

1. Classe	67
1.1 Déclaration	67
1.2 Instance	69
1.3 Membres d'une classe	72
1.3.1 Attribut	72
1.3.2 Méthode	75
1.4 Constructeur	79
1.5 Destructeur	82

- 1.6 Exercices 84
 - 1.6.1 Palindrome - méthode de classe 84
 - 1.6.2 Palindrome - méthode d'instance 85
 - 1.6.3 Puzzle 86
- 2. Héritage 88
 - 2.1 Construction 88
 - 2.2 Polymorphisme 93
 - 2.3 Héritage multiple 96
 - 2.4 Exercices 100
 - 2.4.1 Héritage « simple » 100
 - 2.4.2 Puzzle 103
- 3. Agrégation et composition 105
 - 3.1 Agrégation 105
 - 3.2 Composition 108
 - 3.3 Exercices 110
 - 3.3.1 Le jour d'après 110
 - 3.3.2 Immortel ? 113
- 4. Exception 114
 - 4.1 Levée 114
 - 4.2 Rattrapage 118
 - 4.3 Éviter le masquage d'exception 122
 - 4.4 Exception personnalisée 123
- 5. Concepts de la POO non natifs 125
 - 5.1 Classe abstraite 125
 - 5.2 Interface 129
 - 5.3 Encapsulation 130
- 6. Énumération 131
- 7. Duck typing 134

4 _____ Apprendre la POO

avec le langage Python

Chapitre 5

Un aperçu de quelques design patterns

1. Introduction	137
2. Singleton	139
3. Visiteur	146
3.1 Présentation	146
3.2 Exercice	152
4. Modèle - Vue - Contrôleur (MVC)	155
4.1 Présentation	155
4.2 Exercice	159
5. Abstract Factory	162

Chapitre 6

Plus loin avec Python

1. Introduction	167
2. XML	168
2.1 Présentation	168
2.2 DOM	169
2.2.1 Lecture	169
2.2.2 Méthode par accès direct	170
2.2.3 Méthode par analyse hiérarchique	170
2.2.4 Écriture	172
2.3 SAX	174
2.3.1 Lecture	174
2.3.2 Écriture	176
3. IHM	177
3.1 Tkinter	177
3.1.1 Création d'une fenêtre	178

- 3.1.2 Ajout de widgets. 180
- 3.1.3 Gestion des événements. 182
- 3.2 Qt. 184
 - 3.2.1 Présentation 184
 - 3.2.2 Installation 184
 - 3.2.3 Création d'une fenêtre 185
 - 3.2.4 Ajout de widgets. 187
 - 3.2.5 Gestion des événements. 189
- 4. Bases de données. 191
 - 4.1 Présentation 191
 - 4.2 SQLite 193
- 5. Multithreading 198
 - 5.1 Présentation 198
 - 5.2 Python et la programmation concurrente. 200
 - 5.3 Utilisation du module threading 201
 - 5.4 Synchronisation 204
 - 5.5 Interblocage. 206
- 6. Développement web. 208
 - 6.1 Présentation 208
 - 6.2 Création d'un projet Django 209
 - 6.3 Développement web MVC 211
 - 6.4 Quelques utilitaires intéressants 220

Chapitre 7
Quelques bonnes pratiques

- 1. Introduction 221
- 2. S'assurer avec des bases solides 222
 - 2.1 De l'importance du socle 222
 - 2.2 Structures de données et algorithmes communs 222

6 --- Apprendre la POO

avec le langage Python

2.3	Un problème, plusieurs solutions	224
2.4	Choisir et maîtriser le bon outil	225
3.	Rester concis et simple	226
3.1	DRY	226
3.2	KISS	228
4.	Harmoniser l'équipe	229
4.1	Rédiger des conventions d'écriture	230
4.2	Revoir le code, tous ensemble	231
4.3	Documenter	232
4.4	Tester	232
4.5	Installer un environnement de déploiement continu	233
5.	Rejoindre une communauté	234
5.1	Participer à des conférences et groupes utilisateurs	235
5.2	Écouter la sagesse des foules	235
6.	Maîtriser les problèmes récurrents	237
6.1	Débogage	237
6.2	Traces	238
6.3	Monitoring	238
6.4	Performance	239
6.5	ETL	240
6.6	Bases de données : relationnelles ou non	241
6.7	Bases de données et ORM	242
6.8	Intégrations	243
6.9	Autres environnements logiciels	244
7.	Poursuivre sa croissance personnelle	244
7.1	Savoir poser des questions	245
7.2	Open source	246
7.3	Architecture	247

Table des matières _____ 7

8. Poursuivre sa croissance professionnelle	248
8.1 L'humain	248
8.2 Développement rapide et itérations	248
9. Conclusion	249
Index	251

Chapitre 4

Les concepts de la POO avec Python

1. Classe

1.1 Déclaration

Une classe est la définition d'un concept métier, elle contient des attributs (des valeurs) et des méthodes (des fonctions).

En Python, le nom d'une classe ne peut commencer par un chiffre ou un symbole de ponctuation, et ne peut pas être un mot-clé du langage comme `while` ou `if`. À part ces contraintes, Python est très permissif sur le nom des classes et variables en autorisant même les caractères accentués. Cette pratique est cependant extrêmement déconseillée à cause des problèmes de compatibilité entre différents systèmes.

Voici l'implémentation d'une classe en Python ne possédant aucun membre : ni attribut, ni méthode.

```
class MaClass:
    # Pour l'instant, la classe est déclarée vide,
    # d'où l'utilisation du mot-clé 'pass'.
    pass
```

Le mot-clé `class` est précédé du nom de la classe. Le corps de la classe est lui indenté comme le serait le corps d'une fonction. Dans un corps de classe, il est possible de définir :

- des fonctions (qui deviendront des méthodes de la classe) ;
- des variables (qui deviendront des attributs de la classe) ;
- des classes imbriquées, internes à la classe principale.

Les méthodes et les attributs seront présentés dans les sections suivantes éponymes.

Il est possible d'organiser le code de façon encore plus précise grâce à l'imbrication de classes. Tout comme il est possible, voire conseillé, de répartir les classes d'une application dans plusieurs fichiers (qu'on appelle « modules » en Python), il peut être bien plus lisible et logique de déclarer certaines classes dans une classe « hôte ». La déclaration d'une classe imbriquée dans une autre ressemble à ceci :

```
# Classe contenante, déclarée normalement.
class Humain :

    # Classes contenues, déclarées normalement aussi,
    # mais dans le corps de la classe contenante.

    class Femme :
        pass

    class Homme:
        pass
```

La seule implication de l'imbrication est qu'il faut désormais passer par la classe `Humain` pour utiliser les classes `Femme` et `Homme` en utilisant l'opérateur d'accès « point » : `Humain.Femme` et `Humain.Homme`. Exactement comme on le ferait pour un membre classique.

L'imbrication n'impacte en rien le comportement du contenant ou du contenu.

1.2 Instance

Si l'on exécute le code précédent de déclaration de classe vide, rien ne s'affiche. Ceci est normal puisque l'on n'a fait que déclarer une classe. Après cette exécution, l'environnement Python sait qu'il existe désormais une classe nommée `MaClasse`. Maintenant, il va falloir l'utiliser.

Pour rappel : une classe est une définition, une abstraction de l'esprit. C'est elle qui permet de présenter, d'exposer les données du concept qu'elle représente. Afin de manipuler réellement ces données, il va falloir une représentation concrète de cette définition. C'est l'instance, ou l'objet, de la classe.

Une instance (ou un objet) est un exemplaire d'une classe. L'instanciation, c'est le mécanisme qui permet de créer un objet à partir d'une classe. Si l'on compare une instance à un vêtement, alors la classe est le patron ayant permis de le découper, et la découpe en elle-même est l'instanciation. Par conséquent, une classe peut générer de multiples instances, mais une instance ne peut avoir comme origine qu'une seule classe.

Ainsi donc, si l'on veut créer une instance de `MaClasse` :

```
■ instance = MaClasse()
```

Les parenthèses sont importantes : elles indiquent un appel de méthode. Dans le cas précis d'une instanciation de classe, la méthode s'appelle `__init__` : c'est le constructeur de la classe. Si cette méthode n'est pas implémentée dans la classe, alors un constructeur par défaut est automatiquement appelé, comme c'est le cas dans cet exemple.

Il est désormais possible d'afficher quelques informations sur la console :

```
■ print(instance)
>>> <__main__.MaClasse object at 0x10f039f60>
```

Lorsqu'on affiche sur la sortie standard la variable `instance`, l'interpréteur informe qu'il s'agit d'un objet de type `__main__.MaClasse` dont l'adresse mémoire est `0x10f039f60`.

70 _____ Apprendre la POO

avec le langage Python

D'où vient ce `__main__` ? Il s'agit du module dans lequel `MaClasse` a été déclarée. En Python, un fichier source correspond à un module, et le fichier qui sert de point d'entrée à l'interpréteur est appelé `__main__`. Le nom du module est accessible via la variable spéciale `__name__`.

```
# Ceci est le module b.  
print("Nom du module du fichier b.py : " + __name__)  
chiffre = 42
```

b.py

```
# Ceci est le module a.  
print("Nom du module du fichier a.py : " + __name__)  
import b  
print(b.chiffre)
```

a.py

```
$> python a.py  
Nom du module du fichier a.py : __main__  
Nom du module du fichier b.py : b  
42
```

sortie standard

Le fichier `a.py` sert d'entrée à l'interpréteur Python : ce module se voit donc attribuer `__main__` comme nom. Lors de l'import du module `b.py`, le fichier est entièrement lu et affiche le nom du module qui correspond, lui, au nom du fichier.

Une classe faisant toujours partie d'un module, la classe `MaClasse` a donc été assignée au module `__main__`.

L'adresse hexadécimale de la variable instance correspond à l'emplacement mémoire réservé pour stocker cette variable. Cette adresse permet, entre autres, de différencier deux variables qui pourraient avoir la même valeur.

```
a = MaClasse()
print("Info sur 'a' : {}".format(a))
>>> Info sur 'a' : <__main__.MaClasse object at 0x10b61f908>

b = a
print("Info sur 'b' : {}".format(b))
>>> Info sur 'b' : <__main__.MaClasse object at 0x10b61f908>

b = MaClasse()
print("Info sur 'b' : {}".format(b))
>>> Info sur 'b' : <__main__.MaClasse object at 0x10b6797b8>
```

L'adresse de `a` est `0x10b61f908`. Lorsqu'on assigne `a` à `b` et qu'on affiche `b`, on constate que les variables pointent vers la même zone mémoire. Par contre, si l'on assigne à `b` une nouvelle instance de `MaClasse`, alors son adresse n'est plus la même : il s'agit d'une nouvelle zone mémoire allouée pour stocker un nouvel exemplaire de `MaClasse`.

Voici tout ce qu'il y a à savoir sur la variable instance. D'autres informations sont accessibles concernant la classe elle-même :

```
print(MaClasse)
>>> <class '__main__.MaClasse'>

classe = MaClasse
print(classe)
>>> <class '__main__.MaClasse'>
```

En Python, tout est objet, y compris les classes. N'importe quel objet peut être affecté à une variable, et les classes ne font pas exception. Il est donc tout à fait valide d'assigner `MaClasse` à une variable `classe`, et son affichage sur la sortie standard confirme bien que `classe` est une classe, et pas une instance. D'où l'importance des parenthèses lorsqu'on désire effectuer une instantiation de classe. En effet, les parenthèses précisent bien qu'on appelle le constructeur de la classe, et on obtient par conséquent une instance. L'omission des parenthèses signifie que l'on désigne la classe elle-même.

1.3 Membres d'une classe

1.3.1 Attribut

Un attribut est une variable associée à une classe. Pour définir un attribut au sein d'une classe, il suffit :

- d'assigner une valeur à cet attribut dans le corps de la classe :

```
class Cercle:
    # Déclaration d'un attribut de classe 'rayon'
    # auquel on assigne la valeur 2.
    rayon = 2

print(Cercle.rayon)
>>> 2
```

- d'assigner une valeur à cet attribut en dehors de la classe. On parle alors d'attribut dynamique :

```
class Cercle:
    # Le corps de la classe est laissé vide.
    pass

# Déclaration dynamique d'un attribut de classe 'rayon'
# auquel on assigne la valeur 2.
Cercle.rayon = 2

print(Cercle.rayon)
>>> 2
```

Les attributs définis ainsi sont appelés « attributs de classe » car ils sont liés à la classe, par opposition aux « attributs d'instance » dont la vie est liée à l'instance à laquelle ils sont rattachés. Les attributs de classe sont automatiquement reportés dans les instances de cette classe et deviennent des attributs d'instance.

```
c = Cercle()
print(c.rayon)
>>> 2
```