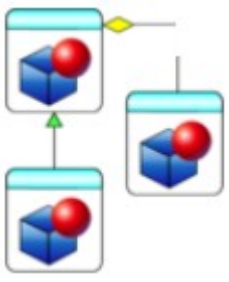
Programmation orientée objet

Table des matières

La programmation orientée objet (POO) est un paradigme de programmation inventé au début des années 1960. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique. Il possède une structure interne et un comportement. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues.

Sources : Vincent le Goff, wikipedia, stackoverflow.com



1. Introduction

À la différence de la programmation impérative, un programme écrit dans un langage objet répartit l'effort de résolution de problèmes sur un ensemble d'objets collaborant par envoi de messages. Chaque objet se décrit par un ensemble d'attributs (caractéristiques de l'objet) et un ensemble de méthodes portant sur ces attributs (fonctionnalités de l'objet).

La POO[1] résulte de la prise de conscience des problèmes posés par l'industrie du logiciel ces dernières années en termes de complexité accrue et de stabilité dégradée. L'objet solutionne certains de ces problèmes à travers :

- l'encapsulation des attributs qui empêche toute modification externe accidentelle
- l'héritage qui permet la ré utilisabilité du code

Il est inconséquent d'opposer la programmation impérative à l'OO car, in fine, toute programmation des méthodes reste tributaire des mécanismes de programmation procédurale et structurée. On y rencontre des variables, des arguments, des boucles, des arguments de fonction, des instructions conditionnelles, tout ce que l'on trouve classiquement dans les boîtes à outils impératives.

2. Définitions

2.1. Classe

Une classe est la modélisation informatique d'un concept. Si dans la vie courante vous vous dite : ceci est un concept, une notion bien précise possédant plusieurs aspects, alors la représentation informatique de cette notion, le sera sous forme de classe. Une classe est l'équivalent d'un type de donnée (abstrait).

Une classe permet de définir les données relatives à une notion, ainsi que les actions qu'y s'y rapportent. Il sera possible, au sein d'une seule et même structure informatique, de regrouper le pendant informatique des données - des variables - et le pendant informatique des actions, à savoir des procédures et des fonctions. Les variables à l'intérieur d'une classe seront appelées *attributs*, et les procédures et les fonctions seront appelées *méthodes*. Les attributs et les méthodes devront posséder des noms utilisables en temps normal pour des variables ou des procédures/fonctions.

Monde réel	**Programmation objet**	**Programmation impérative**
Concept	Classe	(rien)
Manipulation de concept	Objet	(rien)
Donnée	Attribut/Membre	Variable
Action	Méthode/Propriété/Interface	Procédure/Fonction

2.2. Objet

Nous avons vu précédemment que les classes étaient le pendant informatique des concepts. Les objets sont pour leur part le pendant informatique de la représentation réelle des concepts, à savoir des occurrences bien réelles de ce qui ne serait sans cela que des notions sans application possible. Une classe n'étant que la modélisation informatique d'un concept, elle ne se suffit pas à elle-même et ne permet pas de manipuler les occurrences de ce concept. Pour cela, on utilise les objets.

Pour manipuler une occurrence du concept modélisé par une classe, on utilise un *objet*. Un objet est dit d'*une classe* particulière, ou *instance* d'une classe donnée. En programmation objet, un objet ressemble à une variable et on pourrait dire en abusant un petit peu que son type serait une classe définie auparavant.

1 Programmation Orientée Objet

L'instanciation est l'action d'instancier, de créer un objet à partir d'un modèle. Elle est réalisée par la composition de deux opérations : l'allocation et l'initialisation. L'allocation consiste à réserver un espace mémoire au nouvel objet. L'initialisation consiste à fixer l'état du nouvel objet. Cette opération fait par exemple appel à l'un des constructeurs de la classe de l'objet à créer.

On construit ensuite un objet en faisant référence à cette classe. Construire un objet s'appelle l'instanciation.

Exemple :

```
moi = Personne()          # instanciation de la classe Personne
toi = Personne()          # autre instanciation
```

Un objet est un élément variable possédant un exemplaire personnel de chaque attribut défini par sa classe. Il est impossible de lire ou d'écrire la valeur d'un attribut d'une classe, mais il devient possible de lire ou d'écrire la valeur d'un attribut d'un objet. Le fait de modifier la valeur d'un attribut pour un objet ne modifie pas la valeur du même attribut d'un autre objet de même classe.

2.2.1. Les constructeurs

Parmi les différents types de méthode, il existe un type particulier : les constructeurs. Ces constructeurs sont des méthodes qui construisent l'objet désigné par la classe au moment de instanciation de la classe. Un constructeur porte le nom __init__.

Exemple :

```
class Personne:
      """Classe définissant une personne caractérisée par :
      - son nom
      - son prénom
      - son âge"""

      def __init__(self, nom : str, prenom : str):    # le constructeur
            """ Pour l'instant, on ne va définir que 3 attributs """
            self.nom    = nom
            self.prenom = prenom
            self.age    = 33

qui = Personne('Dupont', 'Jean')
print("Je suis {0} {1}, j'ai {2} ans." . format(qui.prenom, qui.nom, qui.age))
```

- La définition de la classe est constituée du mot-clé class, du nom de la classe et des deux points « : ».
- La définition du constructeur consiste en une définition « classique » d'une fonction. Elle a pour nom__init__. En Python, tous les constructeurs s'appellent ainsi. Les noms de méthodes entourés de part et d'autre de deux signes soulignés (__nommethode__) sont des méthodes spéciales. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé self.
- Dans le constructeur, on crée des variables self.nom, self.prenom et self.age que l'on initialise avec les paramètres passés au constructeur lors de l'instanciation.

Quand on instancie Personne(), on appelle le constructeur de la classe Personne. Celui-ci prend en paramètre la variable self qui représente l'objet en train de se créer.

2.2.2. Les destructeurs

En programmation orientée objet, le destructeur d'une classe est une méthode spéciale lancée lors de la destruction d'un objet afin de récupérer les ressources (principalement la mémoire vive) réservée dynamiquement lors de l'instanciation de l'objet. Un constructeur porte le nom __del__.

Le destructeur est appelé implicitement à la sortie du programme, ou explicitement à travers l'instruction del.

Exemple :

```
class Personne:
      """Classe définissant une personne caractérisée par :
      - son nom
      - son prénom
      - son âge"""

      def __init__(self, nom : str, prenom : str):    # le constructeur
            self.nom    = nom
            self.prenom = prenom
            self.age    = 33
            print("Voici {0} {1}" . format(self.prenom, self.nom))

      def __del__(self):      # le destructeur
            print("décédé(e) à {0} ans". format(self.age))


moi = Personne('Dupont', 'Jean')
del moi
toi = Personne('Durant', 'Jean')
```

2.2.3. Attributs de classe

Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet. Ils sont propres à l'objet : si vous créez plusieurs objets, les attributs nom, prenom,… de chacun ne seront pas forcément identiques d'un objet à l'autre. Mais on peut aussi définir des attributs dans notre classe.

Exemple :

```
class Personne:
      """Classe définissant une personne caractérisée par :
      - son nom
      - son prénom"""
      population = 0

      def __init__(self, nom : str, prenom : str):
            self.nom    = nom
            self.prenom = prenom
            Personne.population += 1

moi = Personne('Dupont', 'Jean')
toi = Personne('Durant', 'Jean')
print(Personne.population)
```

On définit notre attribut de classe directement dans le corps de la classe avant la définition du constructeur. Quand on veut l'appeler dans le constructeur, on préfixe le nom de l'attribut de classe par le nom de la classe. Et on y accède de cette façon également, en dehors de la classe.

À chaque fois qu'on crée un objet de type Personne, l'attribut de classe population s'incrémente de 1. Cela peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques.

2.2.4. Les méthodes

Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions agissant sur l'objet. Créer une méthode d'instance, revient à créer une fonction ayant comme premier paramètre le mot clef self.

Exemple :

```
class Personne:
      """Classe définissant une personne caractérisée par :
      - son nom
      - son prénom
      - son âge
      - son lieu de résidence"""

      def __init__(self, nom : str, prenom : str):    # le constructeur
            """ on ajoute un attribut lieu de résidence... """
            self.nom       = nom
            self.prenom    = prenom
            self.age       = 33
            self.residence = "Paris"

      def ma_residence(self):
            """ ...et la méthode associée au lieu de résidence """
            return "J'habite {0}." . format(self.residence)

qui = Personne('Dupont', 'Jean')
print("Je suis {0} {1}, j'ai {2} ans." . format(qui.prenom, qui.nom, qui.age))
print(qui.ma_residence())
```

Pour appeler une méthode de l'instance Personne, il suffit donc d'écrire instance.méthode().

3. Les trois fondamentaux

La POO est dirigée par trois fondamentaux qu'il convient de toujours garder à l'esprit : encapsulation, héritage et polymorphisme.

3.1. Encapsulation

Derrière ce terme se cache le concept même de l'objet : réunir sous la même entité les données et les moyens de les gérer, à savoir les attributs et les méthodes.

L'encapsulation introduit donc une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.

Sous ce nouveau concept se cache également un autre élément à prendre en compte : pouvoir masquer aux yeux d'un programmeur extérieur tous les rouages d'un objet et donc l'ensemble des procédures et fonctions destinées à la gestion interne de l'objet, auxquelles le programmeur final n'aura pas à avoir accès. L'encapsulation permet donc de masquer un certain nombre de attributs et méthodes tout en laissant visibles d'autres attributs et méthodes.

L'encapsulation permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

On va définir des méthodes un peu particulières, appelées des accesseurs et mutateurs. Les accesseurs donnent accès à l'attribut. Les mutateurs permettent de le modifier. Concrètement, au lieu d'écrire mon_objet.mon_attribut, il faut écrire mon_objet.get_mon_attribut(). De la même manière, pour modifier l'attribut ce sera mon_objet.set_mon_attribut(valeur) et non pas mon_objet.mon_attribut = valeur.

3.1.1. Attributs et méthodes publics

Comme leur nom l'indique, les attributs et méthodes dits publics sont accessibles depuis tous les descendants et dans tous les modules.

On peut considérer que les éléments publics n'ont pas de restriction particulière.

Exemple :

```
class Personne:
      """Classe définissant une personne caractérisée par :
      - son nom
      - son prénom
      - son âge"""

      def __init__(self, nom : str, prenom : str, age=33):
            self.nom    = nom
            self.prenom = prenom
            self.age    = age
```

Un attribut ne devrait être public que si sa modification n'entraîne pas de changement dans le comportement de l'objet. Dans le cas contraire, il faut passer par une méthode. Modifier un attribut "manuellement" et ensuite appeler une méthode pour informer de cette modification est une violation du principe d'encapsulation.

3.1.2. Attributs et méthodes privés

La visibilité privée restreint la portée d'un attribut ou d'une méthode au module où il ou elle est déclaré(e). Ainsi, si un objet est déclaré dans une unité avec un attribut privé, alors cet attribut ne pourra être accédé qu'à l'intérieur même de l'unité.

Il est possible de les déclarer privés grâce au double souligné __ pour que les éléments ne soient accessibles qu'à la classe elle-même.

Très souvent, les accesseurs en lecture verront leur nom commencer par get quand leurs homologues en écriture verront le leur commencer par set.

Exemple :

```
class Personne:
    """ Classe représentant une personne """
    def __init__(self, nom : str, prenom : str, age=33):
        self.__nom    = nom
        self.__prenom = prenom
        self.__age    = age
    def get_name(self):
        return self.__nom

### Programme principal ###
qui = Personne('Dupont', 'Jean')
print(qui.get_name())
print(qui.__nom)              # lève l'exception AttibuteError
qui.__nom = 'Durant'          # ne modifie pas l'attribut
print(qui.get_name())
```

3.1.3. Propriétés

Les propriétés sont un moyen transparent de manipuler des attributs d'objet. Elles permettent de dire à Python : « Quand un utilisateur souhaite modifier cet attribut, fais cela ». De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable. Ou encore, on peut faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.

Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut. C'est dans la définition de la classe que l'on précise que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.

Un constructeur porte le nom property. Elle attend quatre paramètres, tous optionnels :

- la méthode donnant accès à l'attribut ;
- la méthode modifiant l'attribut ;
- la méthode appelée quand on souhaite supprimer l'attribut ;
- la méthode appelée quand on demande de l'aide sur l'attribut.

En pratique, on utilise surtout les deux premiers paramètres : ceux définissant les méthodes d'accès et de modification, autrement dit les accesseur et mutateur d'objet.

Exemple :

```
class Personne:
    """ Classe représentant une personne """
    def __init__(self, nom : str, prenom : str, age=33):
        self.__nom    = nom
        self.__prenom = prenom
        self.__age    = age
    def __get_age(self):
        return self.__age
    def __set_age(self, age : int):
        if 100 > age > 17:
            self.__age = age

    age = property(__get_age, __set_age)

### Programme principal ###
qui = Personne('Dupont', 'Jean')
print(qui.age)
qui.age = 10            # ne modifie pas l'attribut
print(qui.age)
```

En Python, rien n'empêche le programmeur de créer un attribut de façon dynamique. Cela est évidemment source de bugs en plus de bafouer le principe d'encapsulation. Il existe toutefois une méthode spéciale qui permet de limiter en partie cette source d'ennui grâce à la méthode __getattr__.

Cette méthode spéciale permet de définir une méthode d'accès aux attributs plus large que celle que Python propose par défaut. En fait, cette méthode est appelée quand on tape objet.attribut (non pas pour modifier l'attribut mais simplement pour y accéder). Python recherche l'attribut et, s'il ne le trouve pas dans l'objet et si une méthode __getattr__ existe, il va l'appeler en lui passant en paramètre le nom de l'attribut recherché, sous la forme d'une chaîne de caractères.

```
class Personne:
    """ Classe représentant une personne """
    def __init__(self, nom : str, prenom : str, age=33):
        self.__nom    = nom
        self.__prenom = prenom
        self.__age    = age
    def __getattr__(self, name : str):
        return None
    def __get_age(self):
        return self.__age
    def __set_age(self, age : int):
        if age > 17:
            self.__age = age

    age = property(__get_age, __set_age)
```

```
### Programme principal ###
qui = Personne('Dupont', 'Jean')
print(qui.age)
qui.age = 10            # ne modifie pas l'attribut
print(qui.age)
qui.__name, qui.__age = 'Albert', 10          # à proscrire !!
print(qui.__name, qui.__age)
print(qui.age)          # résultat très étonnant...
```

3.2. Héritage

L'héritage est l'un des fondements de la programmation objet qui permet une réutilisation d'éléments déjà programmés dans un cadre général. L'héritage est une fonctionnalité objet qui permet de déclarer que telle classe sera elle-même modelée sur une autre classe, qu'on appelle la classe parente, ou la classe mère. Concrètement, si une classe B hérite de la classe A, les objets créés sur le modèle de la classe B auront accès aux méthodes et attributs de la classe A, on dit que la classe A est la fille de la classe B et que la classe B est le parent (ou la superclasse) de la classe A.

3.2.1. L'héritage simple

On oppose l'héritage simple, dont nous venons de voir les aspects théoriques, à l'héritage multiple que nous verrons prochainement.

Il est temps d'aborder la syntaxe de l'héritage. Nous allons définir une première classe Personne et une seconde classe AgentSpecial qui hérite de Personne.

Exemple :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom : str, prenom : str):
        self.__nom    = nom
        self.__prenom = prenom
    def get_identity(self):
        return self.__prenom + " " + self.__nom

class AgentSpecial(Personne):
    """Classe définissant un agent spécial.
    Elle hérite de la classe Personne"""
    def __init__(self, nom : str, prenom : str, matricule : str):
        """Un agent se définit par son nom et son matricule"""
        Personne.__init__(self, nom, prenom)    # appel explicite au constructeur
        self.__matricule = matricule
    def get_matricule(self):
        return self.__matricule

### Programme principal ###
qui = AgentSpecial('Dupont', 'Jean', '007')
print("{0} : {1}".format(qui.get_identity(), qui.get_matricule()))
```

3.2.2. L'héritage multiple

Python inclut un mécanisme permettant l'héritage multiple. L'idée est en substance très simple : au lieu d'hériter d'une seule classe, on peut hériter de plusieurs. Assez souvent, on utilisera l'héritage multiple pour des classes qui ont besoin de certaines fonctionnalités définies dans une classe mère.

Au lieu de préciser, comme dans les cas d'héritage simple, une seule classe mère entre parenthèses, on indique plusieurs, séparées par des virgules.

```
class SuperHero(Personne, Pouvoirs):
```

La recherche des méthodes se fait dans l'ordre de la définition de la classe. Dans l'exemple ci-dessus, si on appelle une méthode d'un objet issu de SuperHero, on va d'abord chercher dans la classe SuperHero. Si la méthode n'est pas trouvée, on la cherche d'abord dans Personne. Encore une fois, si la méthode n'est pas trouvée, on cherche dans toutes les classes mères de la classe Personne, si elle en a, et selon le même système. Si, encore et toujours, on ne trouve pas la méthode, on la recherche dans Pouvoirs et ses classes mères successives.

C'est donc l'ordre de définition des classes mères qui importe. On va chercher la méthode dans les classes mères de gauche à droite. Si on ne trouve pas la méthode dans une classe mère donnée, on remonte dans ses classes mères, et ainsi de suite.

3.3. Polymorphisme

Afin de mieux cerner cette notion, il suffit d'analyser la structure du mot : poly comme plusieurs et morphisme comme forme. Le polymorphisme traite de la capacité de l'objet à posséder plusieurs formes.

Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, un objet va hériter des attributs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire, ou de la compléter.

Le concept de polymorphisme ne doit pas être confondu avec celui d'héritage multiple. En effet, l'héritage multiple permet à un objet d'hériter des membres (attributs et méthodes) de plusieurs objets à la fois, alors que le polymorphisme réside dans la capacité d'un objet à modifier son comportement propre et celui de ses descendants au cours de l'exécution.

3.3.1. Polymorphisme statique : surcharge de méthodes

La plupart du temps, lorsque l'on surcharge une méthode, le but n'est pas d'écraser l'ancienne, mais de la compléter de façon à apporter de nouvelles fonctionnalités. Il est donc nécessaire de pouvoir appeler la méthode ancêtre.

Il n'est pas nécessaire de surcharger ou de redéfinir une méthode ! Ainsi, si un objet ne surcharge pas une méthode, c'est celle du premier ancêtre la définissant ou la surchargeant qui sera appelée.

De fait, il n'est pas nécessaire pour un objet de réécrire un constructeur (ou un destructeur) si celui de son ancêtre suffit à son initialisation.

Exemple :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom : str, prenom : str):
        self.__nom    = nom
        self.__prenom = prenom
    def get_identity(self):
        return self.__prenom + " " + self.__nom

class AgentSpecial(Personne):
    """Classe définissant un agent spécial.
    Elle hérite de la classe Personne"""
    def __init__(self, nom : str, prenom : str, matricule : str):
        """Un agent se définit par son nom et son matricule"""
        Personne.__init__(self, nom, prenom)    # appel explicite au constructeur
        self.__matricule = matricule
    def get_identity(self):
        return self.__matricule

### Programme principal ###
moi = AgentSpecial('Dupont', 'Jean', '007')
```

```
print("identité : {0}".format(moi.get_identity()))

toi = Personne('Durant', 'Jean')
print("identité : {0}".format(toi.get_identity()))
```

Cette surcharge s'effectue plus naturellement à travers la méthode spéciale __str__ qui est appelée par la fonction print.

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom : str, prenom : str):
        self.__nom    = nom
        self.__prenom = prenom
    def __str__(self):
        return self.__prenom + " " + self.__nom

### Programme principal ###
toi = Personne('Durant', 'Jean')
print("identité :", toi)
```

3.3.2. Polymorphisme statique : surcharge d'opérateurs

La surcharge d'opérateur permet d'avoir une signification spécifique quand ils sont appliqués à des types spécifiques. Surcharger les opérateurs standards permet de tirer parti de l'intuition des utilisateurs de la classe. L'utilisateur va en effet pouvoir écrire son code en s'exprimant dans le langage du domaine plutôt que dans celui de la machine.

Pour surcharger l'addition, la méthode spéciale à redéfinir est __add__. Elle prend en paramètre l'objet que l'on souhaite ajouter. Il existe évidemment d'autres méthodes :

- __sub__ : surcharge de l'opérateur –
- __mul__ : surcharge de l'opérateur *
- __truediv__ : surcharge de l'opérateur /
- __floordiv__ : surcharge de l'opérateur // (division entière)
- __mod__ : surcharge de l'opérateur % (modulo)
- __pow__ : surcharge de l'opérateur ** (puissance)
- …

que vous pouvez consulter sur le site web de Python.

De même pour la surcharge des opérateurs de comparaison : ==,!=,<,>,<=,>=.

Ces méthodes sont appelées si vous tentez de comparer deux objets entre eux. Donc, si on veut comparer des durées, par exemple, on doit redéfinir certaines méthodes. Elles devront prendre en paramètre l'objet à comparer à self, et renvoyer un booléen (True ou False).

- __eq__ : surcharge l'opérateur ==
- __ne__ : surcharge l'opérateur !=
- __gt__ : surcharge l'opérateur >
- __ge__ : surcharge l'opérateur =>=
- __lt__ : surcharge l'opérateur <
- __le__ : surcharge l'opérateur <=
- …

Exemple :

```
class Duree:
    """Classe contenant des durées sous la forme d'un nombre de minutes
    et de secondes"""
    def __init__(self, duree = 0.0):
        min, sec = str(duree).split('.')
        self.__min, self.__sec = int(min), int(sec)
    def __str__(self):
        return "{0:02}:{1:02}".format(self.__min, self.__sec)

    def __add__(self, duree : float):
        """L'objet à ajouter est un entier, le nombre de secondes"""
        nouvelle_duree = Duree()

        min, sec = str(duree).split('.')
        nouvelle_duree.__min  = self.__min + int(min)
        nouvelle_duree.__sec  = self.__sec + int(sec)

        if nouvelle_duree.__sec >= 60:
            nouvelle_duree.__min += nouvelle_duree.__sec // 60
            nouvelle_duree.__sec  = nouvelle_duree.__sec % 60

        return nouvelle_duree

    def __eq__(self, autre_duree):
        """Test si self et autre_duree sont égales"""
        return self.__sec == autre_duree.__sec and self.__min == autre_duree.__min

    def __gt__(self, autre_duree):
        """Test si self > autre_duree"""
        nb_sec1 = self.__sec + self.__min * 60
        nb_sec2 = autre_duree.__sec + autre_duree.__min * 60
        return nb_sec1 > nb_sec2

d1 = Duree(12.8)
print(d1)
d2 = d1 + .54     # ajoute 54 secondes
print(d2)
print(d1 == d2)
print(d2 > d1)
```

3.3.3. Polymorphisme dynamique

Attention à ne pas confondre surcharge et polymorphisme. La surcharge consiste à définir des méthodes qui portent des noms identiques au sein d'une classe : c'est un polymorphisme statique. Alors que le polymorphisme dynamique porte sur les objets eux mêmes et s'effectue à l'exécution.

4. Décorateurs

Les décorateurs sont des fonctions de Python dont le rôle est de modifier le comportement par défaut d'autres fonctions ou classes. Pour schématiser, une fonction modifiée par un décorateur ne s'exécutera pas elle-même mais appellera le décorateur. C'est au décorateur de décider s'il veut exécuter la fonction et dans quelles conditions.

On déclare qu'une fonction doit être modifiée par un (ou plusieurs) décorateurs grâce à une (ou plusieurs) lignes au-dessus de la définition de fonction, comme ceci :

```
@nom_du_decorateur
```

```
def ma_fonction(...)
```

Le décorateur s'exécute au moment de la définition de fonction et non lors de l'appel. Il prend en paramètre une fonction (celle qu'il modifie) et renvoie une fonction (qui peut être la même).

Exemple :

```
def debug(fonction : callable):
    print("appel de la fonction {0}".format(fonction))
    return fonction

@debug
def factoriel(n : int) -> int:
    """ calcul de n! """
    if n < 2:
        return 1
    return n * factoriel(n-1)

print(factoriel(4))
```

Résultat :

```
appel de la fonction <function factoriel at 0x030C6C90>
24
```

On peut ainsi poursuivre le débogage et tracer les appels récursifs de la fonction factoriel().

Exemple :

```
def debug(fonction : callable):
    print("appel de la fonction {0}".format(fonction))

    def pile_appels(n : int):
        print("->", n)
        return fonction(n)
    return pile_appels

@debug
def factoriel(n : int) -> int:
    """ calcul de n! """
    if n < 2:
        return 1
    return n * factoriel(n-1)

print(factoriel(4))
```

Résultat :

```
appel de la fonction <function factoriel at 0x030C6C90>
-> 4
-> 3
-> 2
-> 1
24
```

Une grande utilité des décorateurs, mis à part le débogage, porte sur la mesure de performance des algorithme. Ainsi, il suffit de modifier légèrement le code ci-dessus.

Il est important de noter que les décorateurs peuvent s'utiliser avec des méthodes de classes.

Exemple :

```
import time
```

```
def perf(fonction : callable):
    def executer(*parametres : tuple):
        debut    = time.time()
        resultat = fonction(*parametres)
        print("temps d'exécution : {0} ms".format(time.time() - debut))
        return resultat
    return executer

class suite:
      @perf
      def fibonacci(self, n : int) -> int:
          """ calcul de la suite de Fibonacci """
          a, b = 0, 1
          for i in range(n):
              a, b = b, a + b
          return a

s = suite()
print(s.fibonacci(40000))
```

Résultat :

```
temps d'exécution : 0.7121801376342773 ms
```

De même qu'il peut être utile de « débrayer » le décorateur en mode production comme pour les tests unitaires. Dans ce cas, il est nécessaire de pouvoir passer un paramètre au décorateur. Cette fois, la fonction de décorateur prendra en paramètres non pas une fonction, mais les paramètres du décorateur et elle ne renverra pas une fonction de substitution, mais un décorateur.

Exemple :

```
import time

def perf(debug = True):
      def wrapper(fonction : callable):
          def executer(*parametres : tuple):
              if debug:
                    debut    = time.time()
                    resultat = fonction(*parametres)
                    print("temps d'exécution : {0} ms".format(time.time() - debut))
                    return resultat
              else:
                    return fonction(*parametres)
          return executer
      return wrapper

class suite:
      @perf(False)
      def fibonacci(self, n : int) -> int:
          """ calcul de la suite de Fibonacci """
          a, b = 0, 1
          for i in range(n):
              a, b = b, a + b
          return a

s = suite()
print(s.fibonacci(40000))
```