

Les méthodes spéciales

Table des matières

I. Méthodes spéciales visant les attributs d'une classe et items d'un dictionnaire	3
II. Exercice : Quiz	7
III. Méthodes spéciales d'opérations et de comparaisons	8
A. Quelques méthodes spéciales d'opérations	9
B. Quelques méthodes spéciales de comparaison	11
IV. Exercice : Quiz	12
V. Essentiel	14
VI. Auto-évaluation	15
A. Exercice	15
B. Test	15
Solutions des exercices	16

I. Méthodes spéciales visant les attributs d'une classe et items d'un dictionnaire

Durée : 1 H

Environnement de travail : REPL.IT

Contexte

En Python, les méthodes spéciales sont un ensemble de méthodes prédéfinies que vous pouvez utiliser pour enrichir vos classes. Elles sont faciles à reconnaître, car elles commencent et se terminent par des doubles tirets du 8 (`_`). On peut citer comme exemple les méthodes spéciales `__init__` ou `__str__`.

Chaque objet utilise un certain nombre de méthodes spéciales qui sont indispensables au bon fonctionnement du modèle objet de Python et qui regroupent toutes les fonctionnalités que partagent les objets de Python.

Ces méthodes spéciales sont associées à un fonctionnement interne particulier. Le fait d'avoir ces méthodes à disposition, de pouvoir y accéder, mais aussi de les redéfinir permet alors de maîtriser le fonctionnement interne de Python et d'agir dessus.

Les méthodes spéciales sont natives de Python. Un intérêt non négligeable des méthodes spéciales est qu'elles vont vous permettre de modifier le fonctionnement interne des méthodes Python. Nous allons voir plusieurs méthodes spéciales, leur intérêt, leur fonctionnement et leur syntaxe. Nous parlerons dans un premier lieu de méthodes spéciales qui visent les attributs d'une classe ainsi que les items d'un dictionnaire. Puis nous verrons plusieurs autres méthodes spéciales en introduisant la notion de surchargement des opérateurs. Vous verrez que les méthodes spéciales sont essentielles et elles vous apporteront des clés non négligeables dans l'abord de la programmation orientée objet.

Méthode

Quelques règles de syntaxe dans les classes et définition des méthodes

Objectifs

- Maîtriser les méthodes `__getattr__`, `__setattr__` et `__delattr__`
- Maîtriser les méthodes `__setitem__`, `__getitem__` et `__delitem__`

Complément

La méthode `__getattr__`

Pour commencer, créons une classe que nous utiliserons comme fil rouge.

Ligne de code :

```
1 class Account():
2     type = "humain"
3     def __init__(self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
```

On voit bien que cette classe peut être instanciée grâce à la méthode spéciale `__init__`. On peut donc créer un nombre infini d'objets qui correspondent à cette classe. Par exemple, on peut créer un objet ainsi :

Ligne de code :

```
1 class Account():
2     type = "humain"
3     def __init__(self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7
```

```

8 c1 = Account ("rcomputer", "blablabla@mail.com", 22)
9
10 print (c1.type) #humain
11 print (c1.identifiant) #rcomputer
12 print (c1.adressemail) #blablabla@mail.com
13 print (c1.age) #22

```

Ici on crée bien un objet `c1` qui possède plusieurs attributs, un attribut de classe (`type`) et des attributs d'instance (`self.identifiant`, `self.adressemail`, `self.age`). Mais à quoi va nous servir la méthode spéciale `__getattr__` ?

En fait, quand vous cherchez à récupérer un attribut qui n'existe pas, automatiquement, la fonction `__getattr__` est appelée par Python. Nous pouvons donc définir cette méthode spéciale pour qu'elle exécute une action lorsqu'un attribut recherché n'existe pas. Par exemple :

Ligne de code :

```

1 class Account ():
2     type = "humain"
3     def __init__ (self, identifiant, adressemail, age):
4         self.identifiant = identifiant
5         self.adressemail = adressemail
6         self.age = age
7     def __getattr__ (self, name):
8         print ("L'attribut {!r} n'existe pas".format (name))
9 c1 = Account ("rcomputer", "blablabla@mail.com", 22)
10
11 c1.numerotel #L'attribut 'numerotel' n'existe pas

```

Avec la commande `c1.numerotel`, on voit qu'on cherche avoir accès à un attribut de notre objet qui n'existe pas, l'attribut `numerotel`. Effectivement, il n'y a aucun attribut `numerotel` dans la classe `Account`. Donc, Python fait appel à la méthode spéciale `__getattr__`. Nous l'avons implémentée dans la classe `Account`, et on sait qu'elle exécutera l'instruction `print ("L'attribut {!r} n'existe pas".format (name))`. En fait, la méthode `__getattr__` est invoquée chaque fois que l'attribut non existant est accédé via la notation par points (`obj.name`) ou `getattr (obj, 'name')` ou `hasattr (obj, 'name')`. C'est pour cela qu'il est important d'insérer dans les paramètres de la méthode `__getattr__` lors de sa définition le mot clé `self`, et un autre argument (ici `name`) qui va correspondre à l'attribut non existant.

Méthode La méthode `__setattr__`

`__setattr__ (name, value)` est toujours appelée lors d'une tentative de définition de l'attribut nommé. Chaque fois qu'une tentative est faite pour définir la valeur d'attribut via la notation par points `obj.name = value` ou via `setattr (obj, "name", value)`, la méthode `__setattr__` est invoquée. Pour faire simple, revenons à notre exemple :

Ligne de code :

```

1 class Account ():
2     type = "humain"
3     def __init__ (self, identifiant, adressemail, age):
4         self.identifiant = identifiant
5         self.adressemail = adressemail
6         self.age = age
7     def __setattr__ (self, name, value):
8         print ("Attribut créé")
9
10 c1 = Account ("rcomputer", "blablabla@mail.com", 22) #Attribut créé      Attribut créé Attribut
    créé
11 c1.numerotel = "06889930" #Attribut créé
12 setattr (c1, "adresse", "Paris") #Attribut créé

```

Ici, on voit qu'à chaque fois qu'on crée un attribut à notre objet, tout d'abord lors de l'instanciation où on crée 3 attributs, puis avec la ligne `c1.numerotel = "06889930"` et enfin avec `setattr(c1, "adresse", "Paris")`, à chaque fois, la méthode spéciale `__setattr__` est appelée et exécute l'instruction `print ("Attribut créé")`. À l'intérieur de `__setattr__`, la valeur d'un attribut est modifiée en transmettant les paramètres reçus à la méthode de la classe d'objets `__setattr__`. Le paramètre `name` correspond au nom de l'attribut (par exemple l'attribut `numerotel`) et le paramètre `value` a sa valeur (par exemple dans ce dernier cas, la valeur « 06889930 »).

Complément La méthode `__delattr__`

Cette méthode va être appelée lorsqu'on va chercher à supprimer un attribut avec l'instruction `del`. Python transmet implicitement le nom de l'attribut à supprimer au format chaîne à la méthode `__delattr__`. Voici un exemple :

Ligne de code :

```
1 class Account():
2     type = "humain"
3     def __init__(self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7     def __delattr__(self, name):
8         print ("L'attribut {!r} n'existe plus".format (name))
9
10 c1 = Account ("rcomputer", "blablabla@mail.com", 22) #Attribut créé      Attribut créé
11                                     Attribut créé
12 del (c1.identifiant) #L'attribut 'identifiant' n'existe plus
```

Ici, on voit qu'on cherche à supprimer l'attribut `identifiant` de l'objet `c1`. Alors, la méthode `__delattr__` est appelée, et elle exécute le code `print ("L'attribut {!r} n'existe plus".format (name))`, car c'est l'instruction que nous avons intégrée à la définition de la méthode spéciale `__delattr__`.

Les méthodes `__setitem__`, `__getitem__` et `__delitem__`

À quoi vont servir les méthodes `__setitem__` et `__getitem__` ? Reprenons notre classe et créons à présent un attribut d'instance `self.metiers`, qui stockera tous les métiers exercés par la personne détenant le compte créé. Essayons de décortiquer ce code :

Ligne de code :

```
1 class Account():
2     type = "humain"
3     def __init__(self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7         self.metiers = {}
8     def __setitem__(self, key, value):
9         self.metiers [key] = value
10    def __getitem__(self, index):
11        return self.metiers [index]
12
13 c1 = Account ("rcomputer", "blablabla@mail.com",22)
14
15 c1["metier1"] = "boulangier" #fait appel à __setitem__
16 c1["metier2"] = "pompier" #fait appel à __setitem__
17
18 print (c1["metier2"]) #fait appel à __getitem__
```

Ici on voit bien qu'on crée un attribut d'instance `self.metiers` qui va stocker un dictionnaire (grâce à l'instruction `self.metiers = {}`). Puis on va définir les méthodes `__setitem__` et `__getitem__`.

La méthode spéciale `__setitem__` est appelée lorsqu'on va ajouter un item à l'objet créé (puisque la méthode est définie dans la classe). Comme l'objet `c1` n'est pas un dictionnaire, il faut modifier le fonctionnement de la méthode spéciale `__setitem__` pour lui indiquer qu'on attend d'elle qu'elle cherche un item dans le dictionnaire métiers. Donc on définit la méthode `__setitem__`. On indique comme paramètres `self`, `key` qui va correspondre à l'index que nous allons créer et `value` qui va correspondre à la valeur que nous créons. On indique dans notre définition d'ajouter au dictionnaire métiers la valeur souhaitée pour l'index spécifié avec l'instruction `self.metiers[key] = value`, avec les instructions suivantes :

Ligne de code :

```
1 c1["metier1"] = "boulangier"
2 c1["metier2"] = "pompier"
```

On fait 2 fois appel à la méthode `__setitem__`. C'est comme si on écrivait :

Ligne de code :

```
1 c1.__setitem__("metier1", "boulangier")
2 c1.__setitem__("metier2", "pompier")
```

Dans le premier cas par exemple, le paramètre `key` de la méthode `__setitem__` correspond à « *metier1* » et le paramètre `value` à « *boulangier* ». Dans le second cas, le paramètre `key` correspond à « *metier2* » et le paramètre `value` à « *pompier* ».

La méthode `__getitem__` quant à elle est appelée lorsqu'on cherche à obtenir un item de l'objet créé. Le problème est le même, `c1` n'est pas un dictionnaire. Il nous faut donc modifier le comportement de la méthode spéciale `__getitem__`. On définit donc la méthode `__getitem__` et on indique comme paramètres `self` et `index` (qui correspondra alors à l'index de l'item recherché). On demande de retourner la valeur de l'item appartenant au dictionnaire `metier` et ayant pour index la clé mentionnée avec l'instruction `return self.metiers[index]`. Puis, hors de la classe, on écrit l'instruction `c1["metier2"]` pour faire appel à la méthode `__getitem__`. C'est comme si on écrivait :

Ligne de code :

```
1 print (c1["metier2"])
```

Le paramètre `index` va donc correspondre à « *metier2* ». Il va donc chercher dans le dictionnaire métiers l'item d'index « *metier2* ». Que va donc imprimer Python avec l'instruction `print(c1.__getitem__("metier2"))` ?

Ligne de code :

```
1 pompier
```

L'item d'index « *metier2* » a pour valeur `pompier`, car nous avons créé cet item précédemment avec l'instruction `c1.__setitem__("metier2", "pompier")`.

Définition **Dictionnaire**

Un dictionnaire est un type de donnée indexé par des clés (chaînes de caractères ou nombres). Dans notre cas, on va créer une valeur « *boulangier* » qui a un index « *metier1* » et une valeur « *pompier* » qui a un index « *metier2* ».

Et maintenant, parlons de la méthode `__delitem__`. La méthode spéciale `__delitem__` va nous permettre de détruire un item. Voici un code exemple, et vous pouvez l'interpréter pour vous entraîner, c'est le même principe que la méthode `__delattr__`, mais pour un item :

Ligne de code :

```
1 class Account():
2     type = "humain"
3     def __init__(self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
```

```
6         self.age = age
7         self.metiers = {}
8     def __setitem__(self, key, value):
9         self.metiers[key] = value
10    def __getitem__(self, index):
11        return self.metiers[index]
12    def __delitem__(self, index):
13        del self.metiers[index]
14    c1 = Account("rcomputer", "blablabla@mail.com", 22)
15
16    c1["metier1"] = "boulangier" #fait appel à __setitem__
17    c1["metier2"] = "pompier" #fait appel à __setitem__
18
19    del (c1["metier1"])
20
21    print (c1["metier1"])
```

Python renvoie une erreur, car nous demandons d'afficher un item qui a été effacé.

Exercice : Quiz

[solution n°1 p.17]

Question 1

À l'aide de quel(s) caractère(s) pouvez-vous identifier les méthodes spéciales ?

- ☐ « __ »
- ☐ « @ »
- ☐ « # »
- ☐ « class »

Question 2

Quand est appelée la méthode spéciale `__setattr__` ?

- ☐ Lorsqu'on définit un attribut
- ☐ Lorsqu'on accède à l'attribut de l'objet
- ☐ Lorsqu'on vérifie si un attribut existe ou non
- ☐ Lorsqu'on efface un attribut

Question 3

Quand est appelée la méthode spéciale `__delattr__` ?

- ☐ Lorsqu'on définit un attribut
- ☐ Lorsqu'on accède à l'attribut de l'objet
- ☐ Lorsqu'on vérifie si un attribut existe ou non
- ☐ Lorsqu'on supprime un attribut

Question 4

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6     def __getattr__ (self, name):
7         print ("L'attribut {!r} n'existe pas".format (name))
8 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
9
10 print (o1.stockage)
11 print (o1.ram)
```

- ☐ L'attribut "stockage" n'existe pas None 8
- ☐ L'attribut "stockage" n'existe pas
- ☐ 8
- ☐ None

Question 5

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6         self.peripheriques = {"clavier":"azerty", "souris":"Logitech mouse"}
7     def __setitem__ (self, key, value):
8         self.peripheriques [key] = value
9     def __getitem__ (self, index):
10         return self.peripheriques [index]
11 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
12 print (o1 ["clavier"])
13 o1 ["casque"] = "jbl"
14 print (o1 ["casque"])
```

- ☐ azerty jbl
- ☐ jbl
- ☐ Error
- ☐ azerty jbl azerty

III. Méthodes spéciales d'opérations et de comparaisons

Méthode	Comment Python appelle-t-il les méthodes ?
---------	--

A. Quelques méthodes spéciales d'opérations

La méthode `__add__`

À quoi va servir la méthode `__add__` ? Comme vous pouvez le deviner en traduisant, elle va servir à ajouter des éléments. En fait, c'est lorsqu'on utilise l'opérateur « + » qu'on appelle la méthode spéciale `__add__`. Par exemple, si dans un code, vous écrivez :

Ligne de code :

```
1 a = 10
2 b = 30
3 c = a + b
```

En réalité, Python exécute :

Ligne de code :

```
1 a = 10
2 b = 30
3 c = a.__add__(b)
```

En quoi cela va-t-il nous servir ? Reprenons notre exemple et créons un nouveau compte :

Ligne de code :

```
1 class Account ():
2     type = "humain"
3     def __init__ (self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7 c1 = Account ("rcomputer", "blablabla@mail.com", 22)
8 c2 = Account ("pronet11", "blabla@mail.com", 28)
```

Pensez-vous qu'il sera possible d'additionner les 2 comptes `c1` et `c2` ? Non, c'est évident puisqu'il ne s'agit pas de nombres, mais d'objets. Ce qu'on aimerait bien, c'est modifier le comportement de la méthode `__add__` pour qu'elle nous permette d'obtenir le total des âges des 2 clients. On peut procéder ainsi :

Ligne de code :

```
1 class Account ():
2     type = "humain"
3     def __init__ (self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7     def __add__ (self, value):
8         return self.age + value.age
9 c1 = Account ("rcomputer", "blablabla@mail.com", 22)
10 c2 = Account ("pronet11", "blabla@mail.com", 28)
11
12 print (c1 + c2) #équivalent à c1.__add__ (c2)
```

Que va afficher Python ? Réfléchissons, écrire `(c1 + c2)`, ça veut dire au final `c1.__add__(c2)`. On s'intéresse donc à l'objet `c1` et on va lui appliquer la méthode `__add__` avec pour argument `c2`. On voit que dans la définition de `__add__`, on demande de retourner la somme de l'attribut `age` de `self` (donc de `c1`) et de l'attribut `age` de `value` (qui ici est donc `c2`, étant placé en argument lors de l'appel de la fonction `__add__`). Donc, on additionne finalement les âges de `c1` et de `c2`. En fait on dit qu'on a surchargé l'opérateur « + », ce qui revient à modifier la définition de la méthode `__add__`. On a donc :

```
1 50
```

Complément La méthode `__sub__`

À quoi va nous servir la méthode `__sub__` ? En fait, la méthode spéciale `__sub__` est automatiquement appelée lorsque l'opérateur « - » est utilisé. Donc concrètement, quand on écrit :

Ligne de code :

```
1 a = 30
2 b = 10
3 c = a - b
```

En réalité, Python exécute :

Ligne de code :

```
1 a = 30
2 b = 10
3 c = a.__sub__(b)
```

Donc admettons maintenant que je cherche à connaître la différence entre l'âge du compte c2 et du compte c1. Je veux le savoir en écrivant : `c2 - c1`. Comment faire ? Nous pouvons procéder ainsi :

Ligne de code :

```
1 class Account():
2     type = "humain"
3     def __init__(self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7     def __sub__(self, value):
8         return self.age - value.age
9 c1 = Account("rcomputer", "blablabla@mail.com", 22)
10 c2 = Account("pronet11", "blabla@mail.com", 28)
11
12 print(c2 - c1) #équivalent à c2.__sub__(c1)
```

C'est tout aussi simple que quand on a modifié la définition de la méthode `__add__` pour les objets de notre classe. Nous venons de surcharger l'opérateur « - ».

La méthode `__mul__`

Qu'est-ce que la méthode `__mul__` ? C'est la méthode appelée lors d'une multiplication, c'est-à-dire quand on utilise l'opérateur « * ». Par exemple, si on écrit :

Ligne de code :

```
1 a = 30
2 b = 10
3 c = a * b
```

En réalité, Python exécute :

Ligne de code :

```
1 a = 30
2 b = 10
3 c = a.__mul__(b)
```

Nous pouvons évidemment surcharger l'opérateur « * ». Pour reprendre l'exemple simple que nous avons utilisé, si cette fois on souhaite multiplier l'âge respectif de c1 et de c2 lorsqu'on écrit c1*c2, on peut écrire :

Ligne de code :

```
1 class Account ():
2     type = "humain"
3     def __init__ (self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7     def __mul__ (self, value):
8         return self.age * value.age
9 c1 = Account ("rcomputer", "blablabla@mail.com", 22)
10 c2 = Account ("pronet11", "blabla@mail.com", 28)
11
12 print (c1 * c2) #équivalent à c1.__mul__ (c2)
```

Évidemment, le terminal imprime :

Ligne de code :

```
1 616
```

B. Quelques méthodes spéciales de comparaison

Complément La méthode __eq__

La méthode `__eq__` est la méthode appelée lorsqu'on utilise l'opérateur de comparaison « == ». Comme pour les autres méthodes, on peut évidemment décider de la redéfinir. Par exemple, dans notre script, nous aimerions comparer 2 comptes. On va dire que si l'identifiant de 2 comptes est identique, alors ces 2 comptes sont égaux (même si les autres attributs diffèrent). Le problème, c'est que si on écrit :

Ligne de code :

```
1 class Account ():
2     type = "humain"
3     def __init__ (self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
7 c1 = Account ("rcomputer", "blablabla@mail.com", 22)
8 c2 = Account ("rcomputer", "blabla@mail.com", 28)
9
10 print (c1 == c2)
```

Python imprime :

Ligne de code :

```
1 False
```

Python considère que c1 et c2 sont différents, car ce sont des objets stockés à des adresses différentes. Python ne compare pas l'attribut identifiant respectif de c1 et c2. Nous allons donc modifier la définition de la méthode pour la classe Account :

Ligne de code :

```
1 class Account ():
2     type = "humain"
3     def __init__ (self, identifiant, adresseemail, age):
4         self.identifiant = identifiant
5         self.adresseemail = adresseemail
6         self.age = age
```

```

7     def __eq__ (self, value):
8         if self.identifiant == value.identifiant:
9             reponse = True
10        else:
11            reponse = False
12        return reponse
13 c1 = Account ("rcomputer", "blablabla@mail.com", 22)
14 c2 = Account ("rcomputer", "blabla@mail.com", 28)
15
16 print (c1 == c2)

```

Vous voyez, c'est tout simple, dans la définition de la méthode `__eq__`, on compare l'identifiant du paramètre `self` (qui correspondra donc à `c1` dans notre cas) et l'identifiant du paramètre `value` (qui correspondra alors à `c2`). Le terminal renvoie donc :

```
1 True
```

Les méthodes `__ne__`, `__gt__`, `__lt__`, `__ge__`, `__le__`

Maintenant, vous avez bien compris le principe des méthodes spéciales. Vous savez comment modifier leur définition et donc surcharger leur opérateur. Maintenant, voici une liste non exhaustive d'autres méthodes spéciales relatives à des opérateurs de comparaison :

- `__ne__` : appelée lorsqu'on utilise l'opérateur de différence : « `!=` »
- `__gt__` : appelée lorsqu'on utilise l'opérateur de comparaison : « `>` »
- `__lt__` : appelée lorsqu'on utilise l'opérateur de différence : « `<` »
- `__ge__` : appelée lorsqu'on utilise l'opérateur de différence : « `>=` »
- `__le__` : appelée lorsqu'on utilise l'opérateur de différence : « `<=` »

Rappel

Vous l'aurez compris, lorsqu'on utilise un opérateur, par exemple l'opérateur « `>` », Python va faire appel à la méthode spéciale qui lui correspond. Par exemple, `a > b` revient à `a.__gt__(b)`.

Méthode Quelques autres méthodes spéciales

Exercice : Quiz

[solution n°2 p.18]

Question 1

À quelle méthode spéciale fait appel l'opérateur « `*` » ?

- ☐ `__mul__`
- ☐ `__repr__`
- ☐ `__ge__`
- ☐ `__init__`

Question 2

Que va imprimer le code suivant ?

Ligne de code :

```

1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6         self.peripheriques = {"clavier":"azerty", "souris":"Logitech mouse"}
7     def __add__ (self, value):
8         return self.ram + value.ram
9 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
10 o2 = Ordinateur ("acer", "intel core i5", 16)
11
12 print (o1+o2)

```

- ☐ 24
- ☐ Error
- ☐ « o1+o2 »
- ☐ 8

Question 3

Que va imprimer le code suivant ?

Ligne de code :

```

1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
7 o2 = Ordinateur ("acer", "intel core i5", 16)
8
9 print (o1*o2)

```

- ☐ 128
- ☐ 16
- ☐ Une erreur
- ☐ « o1*o2 »

Question 4

Que va imprimer le code suivant ?

Ligne de code :

```

1 class Velo ():
2     def __init__ (self, marque, nbvitesses, freins):
3         self.marque = marque
4         self.nbvitesse = nbvitesses
5         self.freins = freins
6     def __eq__ (self, value):
7         if self.marque == value.marque and self.nbvitesse == value.nbvitesse:
8             reponse = True
9         else:
10            reponse = False

```

```

11         return reponse
12 v1 = Velo ("Rockrider", "16", "plaquettes")
13 v2 = Velo ("Rockrider", "16", "disques")
14
15 if v1 == v2:
16     print ("ce sont les mêmes vélos")
17 else:
18     print ("ce ne sont pas les mêmes vélos")

```

- ☐ « Ce sont les mêmes vélos »
- ☐ « Ce ne sont pas les mêmes vélos »
- ☐ Une erreur

Question 5

Que va imprimer le code suivant ?

Ligne de code :

```

1 class Velo ():
2     def __init__ (self, marque, nbvitesses, freins):
3         self.marque = marque
4         self.nbvitesse = nbvitesses
5         self.freins = freins
6
7 v1 = Velo ("Rockrider", "16", "disques")
8 v2 = Velo ("Rockrider", "16", "disques")
9
10 if v1 == v2:
11     print ("ce sont les mêmes vélos")
12 else:
13     print ("ce ne sont pas les mêmes vélos")

```

- ☐ « Ce sont les mêmes vélos »
- ☐ « Ce ne sont pas les mêmes vélos »
- ☐ Rien

V. Essentiel

Les méthodes spéciales sont des méthodes natives de Python qui sont automatiquement appelées lorsqu'une instruction précise est exécutée. Ces méthodes se reconnaissent par leur syntaxe : un mot clé précédé et suivi de 2 «_» (du 8). Le constructeur `__init__` par exemple est une méthode spéciale. La méthode spéciale `__getattr__` est appelée lorsqu'on cherche à obtenir un attribut de l'objet. `__setattr__` est appelée quand on cherche à définir ou créer un attribut. `__delattr__` est appelée quand on cherche à supprimer un attribut. Les méthodes `__getitem__`, `__setitem__` et `__delitem__` fonctionnent à l'identique, mais portent quant à elles sur les items d'un dictionnaire par exemple.

Qui plus est, chaque opérateur utilisé appelle une méthode spéciale. Par exemple, l'opérateur « + » appelle la méthode `__add__`, l'opérateur « - » appelle la méthode `__sub__`, l'opérateur « * » appelle la méthode `__mul__`. C'est aussi le cas pour les opérateurs de comparaisons. Par exemple, « == » appelle la méthode `__eq__`, l'opérateur « != » appelle la méthode `__ne__`, « > » appelle la méthode `__gt__`, « < » appelle `__lt__`, « >= » appelle `__ge__` et « <= » appelle `__le__`.

Ce qui est particulièrement intéressant, c'est qu'on peut définir ces fonctions. Dans le cadre des méthodes spéciales appelées par un opérateur, on va pouvoir traiter des cas d'opérations entre objets, bien que n'étant pas des valeurs numériques. Nous pourrions par exemple indiquer dans la définition de ces méthodes spéciales qu'on cherche à appliquer l'opération uniquement à un attribut de chaque objet. En fait, l'utilisation des méthodes spéciales ouvre le champ des possibilités en programmation orientée objet.

VI. Auto-évaluation

A. Exercice

Vous êtes un vendeur de matériel informatique et vous cherchez à écrire un script Python permettant de créer des objets relatifs aux produits que vous vendez. Vous vous concentrez sur la vente de disques durs.

Question 1

[solution n°3 p.21]

Créez une classe permettant de créer des disques durs. Chaque disque dur peut avoir une marque, une couleur et un espace de stockage différents. Créez 2 disques durs différents.

Question 2

[solution n°4 p.21]

Vous souhaitez que lorsque vous additionnez 2 objets, le résultat renvoyé soit le total du stockage disponible sur les 2 disques durs. Qui plus est, vous souhaitez que lorsque vous créez un nouvel attribut d'un objet, Python imprime la référence de l'objet concerné, le nom du nouvel attribut et sa valeur. Quel code écririez-vous ?

B. Test

Exercice 1 : Quiz

[solution n°5 p.22]

Question 1

Qu'est-ce qu'une méthode spéciale ?

- ☐ Une méthode native de Python qui est appelée lorsqu'un type d'instruction précis est exécuté
- ☐ Une variable globale
- ☐ Une méthode accessible uniquement via une bibliothèque

Question 2

La méthode `__ne__` est appelée quand :

- ☐ L'opérateur « `==` » est utilisé
- ☐ L'opérateur « `!=` » est utilisé
- ☐ Un attribut est défini

Question 3

Dans ce code, que faut-il rajouter dans la définition de la méthode `__add__` pour que l'addition des 2 objets retourne la somme des attributs `prix` ?

Ligne de code :

```
1 class Panier ():
2     def __init__ (self, fruit, nombre, prix):
3         self.fruit = fruit
4         self.nombre = nombre
5         self.prix = prix
6     def __add__ (self, value):
7         #Ecrire ICI
8 panier1 = Panier ("Pastèque", "1", 5)
9 panier2 = Panier ("Melon", "2", 4)
```

```
10
11 print (panier1+panier2)
```

- ☐ return self.prix + self.value
- ☐ return self.prix + value.prix
- ☐ Il ne faut rien écrire

Question 4

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Voiture ():
2     def __init__ (self, marque, portes, cv):
3         self.marque = marque
4         self.portes = portes
5         self.cv = cv
6     def __add__ (self, value):
7         return self.cv + value.cv
8 v1 = Voiture ("Chevrolet", 5, "100")
9 v2 = Voiture ("Peugeot", 3, "90")
10
11 print (v1+v2)
```

- ☐ 190
- ☐ 10090
- ☐ Une erreur

Question 5

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Voiture ():
2     def __init__ (self, marque, portes, cv):
3         self.marque = marque
4         self.portes = portes
5         self.cv = cv
6     def __getattr__ (self, name):
7         print ("cet attribut n'existe pas")
8 v1 = Voiture ("Chevrolet", 5, "100")
9 v2 = Voiture ("Peugeot", 3, "90")
10 del (v2.cv)
11
12 print (v1.cv)
```

- ☐ 100
- ☐ Une erreur
- ☐ « Cet attribut n'existe pas »

Solutions des exercices

Exercice p. 7 Solution n°1**Question 1**

À l'aide de quel(s) caractère(s) pouvez-vous identifier les méthodes spéciales ?

☒ « `__` »

☐ « `@` »

☐ « `#` »

☐ « `class` »

☒ Une méthode spéciale est précédée et suivie de 2 « `_` » (tirets du 8).

Question 2

Quand est appelée la méthode spéciale `__setattr__` ?

☒ Lorsqu'on définit un attribut

☐ Lorsqu'on accède à l'attribut de l'objet

☐ Lorsqu'on vérifie si un attribut existe ou non

☐ Lorsqu'on efface un attribut

☒ La méthode spéciale `__setattr__` est automatiquement appelée lorsqu'on définit un attribut. Des paramètres correspondent à son nom et à sa valeur.

Question 3

Quand est appelée la méthode spéciale `__delattr__` ?

☐ Lorsqu'on définit un attribut

☐ Lorsqu'on accède à l'attribut de l'objet

☐ Lorsqu'on vérifie si un attribut existe ou non

☒ Lorsqu'on supprime un attribut

☒ La méthode spéciale `__delattr__` est automatiquement appelée lorsqu'on supprime un attribut, c'est-à-dire quand on utilise l'instruction `del (#objet.attribut)`.

Question 4

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6     def __getattr__ (self, name):
7         print ("L'attribut {!r} n'existe pas".format (name))
8 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
9
```

```
10 print (o1.stockage)
11 print (o1.ram)
```

- ☒ L'attribut "stockage" n'existe pas None 8
- ☐ L'attribut "stockage" n'existe pas
- ☐ 8
- ☐ None

Q On cherche à imprimer un attribut qui n'existe pas avec l'instruction `print (o1.stockage)`. La méthode `__getattr__` est donc automatiquement appelée. Dans sa définition, nous avons indiqué que nous voulons imprimer `("L'attribut {!r} n'existe pas".format (name))`. Qui plus est, la dernière instruction du script imprime `(o1.ram)`. Donc, Python imprime « *L'attribut 'stockage' n'existe pas* », la méthode `__getattr__` étant appelée, car l'attribut `stockage` n'existe pas, puis `None`, car il n'y a aucun attribut `stockage` et enfin `8` avec l'instruction `print (o1.ram)`.

Question 5

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6         self.peripheriques = {"clavier":"azerty", "souris":"Logitech mouse"}
7     def __setitem__ (self, key, value):
8         self.peripheriques [key] = value
9     def __getitem__ (self, index):
10         return self.peripheriques [index]
11 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
12 print (o1 ["clavier"])
13 o1 ["casque"] = "jbl"
14 print (o1 ["casque"])
```

- ☒ azerty jbl
- ☐ jbl
- ☐ Error
- ☐ azerty jbl azerty


Q Avec l'instruction `print (o1 ["clavier"])`, on cherche à accéder à un item « *clavier* », ce qui appelle automatiquement la méthode spéciale `__getitem__`. Dans la définition de `__getitem__`, on indique que le dictionnaire visé est contenu dans la variable `péripheriques`. Python affiche donc la valeur de l'item ayant pour index « *clavier* », c'est-à-dire « *azerty* ». Puis l'instruction `o1 ["casque"] = "jbl"` appelle automatiquement la méthode spéciale `__setitem__` qu'on a définie pour qu'elle puisse correctement définir un item dans le dictionnaire `péripheriques`. Un item ayant pour `key` (ici `key` revient au même qu'`index`) « *casque* » et valeur « *jbl* » est ajouté au dictionnaire. Puis l'instruction `print (o1 ["casque"])` fait appel à la méthode `__getitem__` qui imprime la valeur de l'item ayant pour index « *casque* », c'est-à-dire « *jbl* », item que nous venons d'ajouter au dictionnaire.

Exercice p. 12 Solution n°2

Question 1

À quelle méthode spéciale fait appel l'opérateur « * » ?

- ☒ `__mul__`
- ☐ `__repr__`
- ☐ `__ge__`
- ☐ `__init__`

 L'opérateur « * » fait automatiquement appel à la méthode spéciale `__mul__`. C'est la méthode qu'on appelle pour une multiplication.


Question 2

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6         self.peripheriques = {"clavier":"azerty", "souris":"Logitech mouse"}
7     def __add__ (self, value):
8         return self.ram + value.ram
9 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
10 o2 = Ordinateur ("acer", "intel core i5", 16)
11
12 print (o1+o2)
```

- ☒ 24
- ☐ Error
- ☐ « o1+o2 »
- ☐ 8

 On définit la méthode spéciale `__add__` dans la classe `Ordinateur ()`. On dit de retourner la somme de l'attribut `ram` de l'objet et de l'attribut `ram` de la valeur. En fait, l'instruction `print (o1+o2)` signifie `print (o1.__add__ (o2))`. L'objet traité est donc `o1` et la valeur indiquée en paramètre `o2`. L'opérateur « + » est surchargé et vise directement les attributs `ram` des 2 objets concernés. 24 est donc affiché ($8 + 16 = 24$).

Question 3

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Ordinateur ():
2     def __init__ (self, marque, processeur, ram):
3         self.marque = marque
4         self.processeur = processeur
5         self.ram = ram
6 o1 = Ordinateur ("hp", "amd ryzen 5", 8)
7 o2 = Ordinateur ("acer", "intel core i5", 16)
8
9 print (o1*o2)
```

- ☐ 128
- ☐ 16
- ☒ Une erreur
- ☐ « $o1*o2$ »

Q Ici, on cherche à multiplier 2 objets, et on ne définit pas la méthode `__mul__` pour qu'elle porte sur 2 valeurs numériques. Python imprime donc une erreur puisqu'on ne peut pas multiplier 2 objets, car ils ne sont pas des valeurs numériques.

Question 4

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Velo ():
2     def __init__ (self, marque, nbvitesses, freins):
3         self.marque = marque
4         self.nbvitesse = nbvitesses
5         self.freins = freins
6     def __eq__ (self, value):
7         if self.marque == value.marque and self.nbvitesse == value.nbvitesse:
8             reponse = True
9         else:
10            reponse = False
11        return reponse
12 v1 = Velo ("Rockrider", "16", "plaquettes")
13 v2 = Velo ("Rockrider", "16", "disques")
14
15 if v1 == v2:
16     print ("ce sont les mêmes vélos")
17 else:
18     print ("ce ne sont pas les mêmes vélos")
```

- ☒ « Ce sont les mêmes vélos »
- ☐ « Ce ne sont pas les mêmes vélos »
- ☐ Une erreur

Q On définit dans la classe la méthode spéciale `__eq__`. On lui demande finalement de retourner `True` si les attributs `marque` et `nbvitesses` sont égaux. Comme les 2 instances créent 2 objets de même marque et de même nbvitesses, la condition `if v1 == v2` qui appelle la méthode `__eq__` est validée et imprime « ce sont les mêmes vélos ».

Question 5

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Velo ():
2     def __init__ (self, marque, nbvitesses, freins):
3         self.marque = marque
4         self.nbvitesse = nbvitesses
5         self.freins = freins
6
7 v1 = Velo ("Rockrider", "16", "disques")
8 v2 = Velo ("Rockrider", "16", "disques")
9
```

```

10 if v1 == v2:
11     print ("ce sont les mêmes vélos")
12 else:
13     print ("ce ne sont pas les mêmes vélos")

```

- ☐ « Ce sont les mêmes vélos »
- ☒ « Ce ne sont pas les mêmes vélos »
- ☐ Rien

Q La condition `if` compare 2 objets. Ces objets ont une référence différente même si leurs attributs sont identiques. Étant donné que la méthode `__eq__` n'est pas redéfinie, l'opérateur « `==` » compare les 2 objets, et pas un ou plusieurs de leurs attributs. Comme les 2 objets n'ont pas la même référence, Python imprime « ce ne sont pas les mêmes vélos ».

p. 15 Solution n°3

Ligne de code :

```

1 class Disque ():
2     def __init__ (self, marque, couleur, stockage):
3         self.marque = marque
4         self.couleur = couleur
5         self.stockage = stockage
6 d1 = Disque ("Samsung", "noir", 512)
7 d2 = Disque ("Thomson", "bleu", 256)

```

Dans ce code, on crée bien une classe avec le constructeur `__init__`. Il y a bien 3 attributs d'instance. On crée alors 2 objets avec des paramètres différents.

p. 15 Solution n°4

Ligne de code :

```

1 class Disque ():
2     def __init__ (self, marque, couleur, stockage):
3         self.marque = marque
4         self.couleur = couleur
5         self.stockage = stockage
6     def __add__ (self, value):
7         return self.stockage + value.stockage
8     def __setattr__ (self, name, value):
9         print (self)
10        print (name)
11        print (value)
12 d1 = Disque ("Samsung", "noir", 512)
13 d2 = Disque ("Thomson", "bleu", 256)

```

On voit que lorsqu'on définit la méthode `__add__`, on surcharge l'opérateur « `+` » pour qu'il additionne les 2 attributs d'instance `self.stockage` et `value.stockage`. Puis, on définit la méthode `__setattr__` qui va alors imprimer la référence de l'objet avec `print (self)`, le nom de l'attribut (avec `print (name)`) et sa valeur (avec `print (value)`). Au final, comme on crée 2 objets, et qu'il y'a donc affectation de valeur à des attributs, la méthode `__setattr__` est appelée 6 fois, puisque Python crée :

- `d1.marque = « Samsung »`
- `d1.couleur = « noir »`
- `d1.stockage = 512`

- d2.marque = « Thomson »
- d2.couleur = « bleu »
- d2.stockage = 256

Exercice p. 15 Solution n°5

Question 1

Qu'est-ce qu'une méthode spéciale ?

- ☒ Une méthode native de Python qui est appelée lorsqu'un type d'instruction précis est exécuté
- ☐ Une variable globale
- ☐ Une méthode accessible uniquement via une bibliothèque
- ☒ Une méthode spéciale est une méthode native de Python qui est appelée lorsqu'un type d'instruction précis est exécuté. On les reconnaît syntaxiquement à leur format, précédé et suivi de 2 tirets du 8 (_).

Question 2

La méthode `__ne__` est appelée quand :

- ☐ L'opérateur « `==` » est utilisé
- ☒ L'opérateur « `!=` » est utilisé
- ☐ Un attribut est défini
- ☒ La méthode `__ne__` est appelée quand l'opérateur « `!=` » est utilisé. Cet opérateur vérifie si 2 valeurs sont bien différentes.

Question 3

Dans ce code, que faut-il rajouter dans la définition de la méthode `__add__` pour que l'addition des 2 objets retourne la somme des attributs prix ?

Ligne de code :

```
1 class Panier ():
2     def __init__ (self, fruit, nombre, prix):
3         self.fruit = fruit
4         self.nombre = nombre
5         self.prix = prix
6     def __add__ (self, value):
7         #Ecrire ICI
8 panier1 = Panier ("Pasteque", "1", 5)
9 panier2 = Panier ("Melon", "2", 4)
10
11 print (panier1+panier2)
```

- ☐ `return self.prix + self.value`
- ☒ `return self.prix + value.prix`
- ☐ Il ne faut rien écrire
- ☒ Il faut écrire `return self.prix + value.prix`. L'instruction `panier1+panier2` signifie `panier1.__add__(panier2).self.prix` correspond donc à l'attribut `prix` du panier1 et `value.prix` correspond à l'attribut `prix` du panier2.


Question 4

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Voiture ():
2     def __init__ (self, marque, portes, cv):
3         self.marque = marque
4         self.portes = portes
5         self.cv = cv
6     def __add__ (self, value):
7         return self.cv + value.cv
8 v1 = Voiture ("Chevrolet", 5, "100")
9 v2 = Voiture ("Peugeot", 3, "90")
10
11 print (v1+v2)
```

- ☐ 190
- ☒ 10090
- ☐ Une erreur

 Ici, la définition de la méthode spéciale `__add__` additionne les attributs `cv` des 2 objets. Or, on voit lorsqu'on instancie la classe qu'on indique pour les 2 objets que le 3^e paramètre, c'est-à-dire celui qui correspond à l'attribut `cv` est une chaîne de texte, car 100 et 90 ont des guillemets. Or, lorsque la méthode `__add__` traite 2 chaînes de textes, elle les place l'une à la suite de l'autre. On parle de concaténation. Python imprime donc 10 090.


Question 5

Que va imprimer le code suivant ?

Ligne de code :

```
1 class Voiture ():
2     def __init__ (self, marque, portes, cv):
3         self.marque = marque
4         self.portes = portes
5         self.cv = cv
6     def __getattr__ (self, name):
7         print ("cet attribut n'existe pas")
8 v1 = Voiture ("Chevrolet", 5, "100")
9 v2 = Voiture ("Peugeot", 3, "90")
10 del (v2.cv)
11
12 print (v1.cv)
```

- ☒ 100
- ☐ Une erreur
- ☐ « Cet attribut n'existe pas »

 Ici, on supprime l'attribut `cv` de l'objet `v2`. Lorsque Python exécute `print (v1.cv)`, il imprime l'attribut `cv` de `v1` qui lui existe et n'a pas été supprimé. Python ne fait donc pas appel à la méthode spéciale `__getattr__`. Il imprime la valeur de `v1.cv` qui est 100.