Préparation d'un sujet d'examen

11. Python OOP : classes, instances, attributs, méthodes, héritage

Auteur : Jérémy Werro, Gymnase du Bugnon, classe 3MOCINFO

Date de Rendu : Vendredi 31 Mai 2019

Dernière modification : 31.05.2019

Les diagrammes sont issus de pensez-python

La POO (Programmation Orientée Objet)

Introduction

Dans le langage python, il existe un type pour chaque valeur (ou objet) assignée à une variable. Il y a par exemple le type `int` qui représente toute les valeurs de nombre entier, le type `float` qui représente les valeurs de nombre à virgule flottante, ou encore le type `str` qui représente les chaînes de caractère comme du texte. Ces types de valeur sont vérifiables dans l'interpréteur.

In [1]:

```
a = 1
b = 1.5
c = 'Hello'
```

In [2]:

```
type(a)
```

Out[2]:

```
int
```

In [3]:

```
type(b)
```

Out[3]:

```
float
```

In [4]:

```
type(c)
```

Out[4]:

```
str
```

- La programmation orientée objet utilise des objets dont le type est créé par le programmeur. Cela va permettre de créer des valeurs, ou objet, avec des caractéristiques et des utilisations plus complexe que de simple nombre ou chaîne. La POO permet une gestion des données plus efficace et propre.
- Pour créer un type nous avons besoin de plusieurs éléments. Comme pour faire des biscuits, il nous faut un moule, que l'on va appeler une classe, et de la matière, de la pâte qui va caractériser chaque objet et qu'on appelle des attributs.
- Avec un moule, on peut créer plusieurs biscuits de même forme mais avec un goût différent. Avec une classe, c'est la même chose, on va pouvoir créer plusieurs objets de même type mais avec des attributs différents qui rendent chaque objet unique. Un objet créé grâce à une classe s'appelle une instance.

- L’avantage des instances de classe est qu’elles peuvent elles-mêmes devenir des attributs pour la création d’un autre type d’instance. Cela permet de créer des objets très complexes.
- Ces objets complexes, lorsqu’ils sont utilisés dans des fonctions, peuvent provoquer des lignes de code très chargée. Cela nous oblige à appliquer un développement par conception afin de rendre notre programme plus claire et plus léger.
- Dans la définition des classes nous pouvons ajouter aussi des fonctions, appelées méthode, liées à la classe. Celles-ci utilisent directement l’instance et peuvent être appelées de manière plus intuitive.
- Enfin, il est aussi possible de créer des classes qui héritent des caractéristiques d’autre classe. Cet héritage permet d’avoir des classes semblables mais qui ont des rôles différents.

Les classes

Une classe est un moule qui permet de créer des objets d'un type particulier. Ces objets sont semblables mais les valeurs qui les composent sont différentes.

Voici comment créer une classe :

In [5]:

```
class Point:
    """Représente un point dans l'espace 2D."""
```

- `class` nous premet de créer un type qu'on défini.
- `Point` est le nom de la classe, et par conséquent le nom de type des objet créés.
- Le docstring décris la représentation réelle des objets créés. Il décrit également ce que représent les attributs.

Les instances

Pour créer un objet de classe il faut l'instancier, autrement dit créer une instance. Comme ceci :

In [6]:

```
point = Point()
```

Comme pour une affectation, on crée une variable `point` à laquelle on affecte la classe, ce qui crée une instance nommée `point` de type `Point` :

In [7]:

```
type(point)
```

Out[7]:

```
__main__.Point
```

La valeur attribuée à la variable `point` est une référence à un objet de type `Point`. Pour le vérifier, il suffit d'entrer la variable. Cela vous donne sa référence et sa place de stockage en valeur hexadécimale :

In [8]:

```
point
```

Out[8]:

```
<__main__.Point at 0x5d88fd0>
```

Les attributs

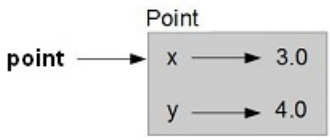
Les attributs sont les différente caractéristique d'une instance.

Par exemple, dans le cas d'un point dans l'espace 2-D, il faut indiquer ses coordonées sur l'axe x et y. L'objet `point` de type `Point` doit donc posséder un attribut x et un attribut y, appelé respectivement `point.x` et `point.y` auxquels ont affecte les valeurs souhaitées :

In [9]:

```
point.x = 3.0
point.y = 4.0
```

Voici à quoi ressemble le diagramme de cet objet :



Il est important de définir ces arguments dans le docstring de la classe :

In [10]:

```
class Point:
    """Représente un point dans l'espace 2D.

    attributs : x et y (int).
    """
```

En entrant l'attribut vous pouvez vérifier sa valeur :

In [11]:

```
point.x
```

Out[11]:

```
3.0
```

Ces attributs sont utilisable comme des variables normales. Mais vous pouvez aussi les utiliser dans une fonction en indiquant comme argument de fonction l'objet dont ils sont issus :

In [12]:

```
def afficher_point(p):
    print('({} ; {})'.format(p.x, p.y))
```

La fonction `afficher_point()` prend un objet `p` en argument et en affiche les attributs x et y.

In [13]:

```
afficher_point(point)
```

```
(3.0 ; 4.0)
```

Objet comme attribut

Un attribut peut également être une instance. Pour créer un objet rectangle il nous faut ses dimension, longueur et largeur, et sa position, donnée par un point.

In [14]:

```
class Rectangle:
```

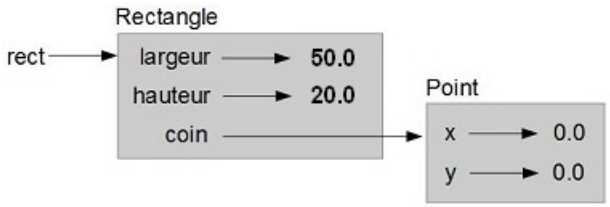
```
    """Représente un Rectangle.

    attributs: largeur (int), longueur (int), coin (Point).
    """
```

Les dimensions du rectangle sont des `int`, mais le coin de référence est un objet de type `Point`. Pour instancier un rectangle, il faut rentrer les valeurs de chaque attribut. Pour le cas du coin, il faut instancier l'attribut `coin` à la classe `Point` et ensuite affecter ses coordonées aux valeurs souhaitées:

In [15]:

```
rect = Rectangle()
rect.largeur = 50
rect.hauteur = 20
rect.coin = Point()
rect.coin.x = 0.0
rect.coin.y = 0.0
```

Voici à quoi ressemble le diagramme de cet objet :



Une instance est utilisable comme n'importe quelle valeur, en plus de posséder des valeurs internes que sont ses attributs. Ses attributs sont également modifiables, et ce, sans modifier le reste de l'objet :

In [16]:

```
rect.hauteur = 100
rect.largeur
```

Out[16]:

```
50
```

La largeur du rectangle ne change pas.

Copier

Avec la fonction `copy()`, importée du module `copy`, il est possible de créer une copie d'instance :

In [17]:

```
import copy

point = Point()
point.x = 6.0
point.y = 8.0

coin = copy.copy(point)

afficher_point(point)
afficher_point(coin)
```

```
(6.0 ; 8.0)
(6.0 ; 8.0)
```

Les deux points possède les mêmes valeurs d'attributs, mais ce sont bel et bien 2 points distincts :

In [18]:

```
point == coin
```

Out[18]:

```
False
```

C'est un détail important qui pose problème lorsqu'une instance possède comme attribut une autre instance, comme le rectangle par exemple. La fonction `copy()` ne copie que l'instance et ses références, par conséquent les objet attributs ne sont pas copiés et deviennent attribut de la nouvelle copie :

In [19]:

```
rect2 = copy.copy(rect)
rect == rect2
```

Out[19]:

```
False
```

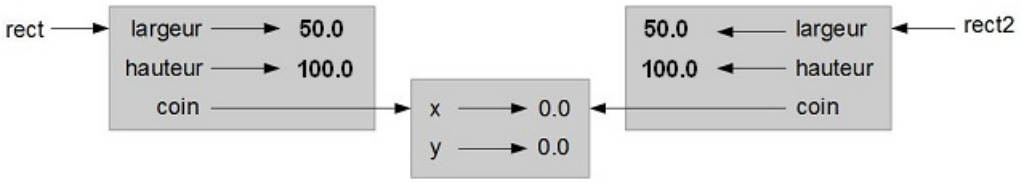
In [20]:

```
rect.coin == rect2.coin
```

Out[20]:

```
True
```

Voici à quoi ressemble le diagramme de ces objets :



Si vous souhaitez faire une copie complète, le module `copy` vous propose la fonction `deepcopy()` qui effectue une copie "profonde" de l'instance :

In [21]:

```
rect3 = copy.deepcopy(rect)
rect == rect3
```

Out[21]:

```
False
```

In [22]:

```
rect.coin == rect3.coin
```

Out[22]:

```
False
```

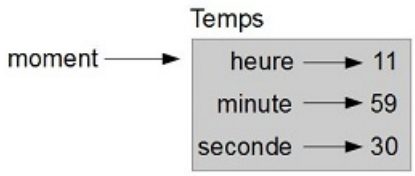
Modificateur

Lorsqu'on a des instances, on peut utiliser leur attributs dans des fonction qui se servent des valeurs pour, par exemple recréer une nouvelle instance. Avec une classe `Temps` qui créer des instances temporelles avec comme attributs `heure`, `minutes`, `seconde` :

In [23]:

```
class Temps:
    """Représente le moment de la journée.

    attributs : heure, minute, seconde
    """

instant = Temps()
instant.heure = 11
instant.minute = 59
instant.seconde = 30
```

Voici à quoi ressemble le diagramme de cet objet :



Et voici une fonction qui ajoute du temps avec comme argument deux instances temps :

In [24]:

```
def ajouter_temps(t1, t2):
    total = Temps()
    total.heure = t1.heure + t2.heure
    total.minute = t1.minute + t2.minute
    total.seconde = t1.seconde + t2.seconde
    return total
```

Les instances utilisée par la fonction ne sont pas modifiées. Ces fonctions sont appelées fonction pure.

In [25]:

```
chrono = Temps()
chrono.heure = 9
chrono.minute = 12
chrono.seconde = 10

temps = ajouter_temps(instant, chrono)
print(temps.heure, temps.minute, temps.seconde)
```

```
20 71 40
```

Le problème de cet fonction est qu'elle crée une troisième instance. C'est pour ça qu'il est parfois préférable d'utiliser des fonctions qui modifient des instances et qu'on appelle modificateurs.

In [26]:

```
def temps_vers_int(temps):
    minutes = temps.heure * 60 + temps.minute
    secondes = minutes * 60 + temps.seconde
    return secondes
```

La fonction `temps_vers_int()` permet de transformer une instance `Temps` en valeur `int` et ainsi l'utiliser plus facilement, dans une addition par exemple.

Voici la fonction inverse :

In [27]:

```
def int_vers_temps(secondes):
```

```
        temps = Temps()
        minutes, temps.seconde = divmod(secondes, 60)
        temps.heure, temps.minute = divmod(minutes, 60)
        return temps
```

Avec ces deux modificateurs, la fonction `ajouter_temps()` devient bien plus facile et précise :

In [28]:

```
def ajouter_temps(t1, t2):
    secondes = temps_vers_int(t1) + temps_vers_int(t2)
    return int_vers_temps(secondes)
```

In [29]:

```
temps = ajouter_temps(instant, chrono)
print(temps.heure, temps.minute, temps.seconde)
```

```
21 11 40
```

Il est parfois judicieux de bien comprendre les opérations qu'on souhaite effectuer et le concept général de nos fonctions. En appliquant un développement par conception, on peut concevoir un code plus efficace, facile à debogué et/ou à modifier.

Méthodes

Dans la définition d'une classe, il est possible d'y intégrer des fonctions propres à la classe. Ces fonctions, appelées méthodes, vont traiter les relations et interactions entre les différentes instances. Une méthode, à la différence d'une simple fonction, est définie dans la définition de sa classe et son invocation est plus intuitive.

In [30]:

```
class Temps:
    """Représente le moment de la journée.

    attributs : heure, minute, seconde
    """

    def afficher_temps(self):
        print('%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde))
```

Voilà comment intégrer la fonction `afficher_temps()` dans la classe `Temps` et en faire ainsi une méthode. Le premier argument d'une méthode est appelé par convention `self`. Il s'agit de l'instance elle-même.

Pour appeler la méthode, la syntaxe la plus courante est la suivante :

In [31]:

```
timer = Temps()
timer.heure = 3
timer.minute = 56
timer.seconde = 22

timer.afficher_temps()
```

```
03:56:22
```

Dans cet appelle de méthode, la fonction `afficher_temps()` est un argument de l'instance `timer`. On demande à l'instance `timer` de faire la méthode `afficher_temps()` de lui-même (`self`). C'est pour ça que les parenthèses n'ont pas besoin de l'argument `self`. Il est déjà indiqué par l'instance.

Cependant, il arrive que des méthodes demande d'autre arguments. Ceux-ci doivent alors être inscrits dans les parenthèses. Si un second argument est une autre instance `Temps`, il faut l'appelé par convention `other`. Voici la méthode `est_apres()` qui vérifie si le premier argument `Temps` est plus grand que le second argument `Temps` (il faut avant cela ajouter la méthode `temps_vers_int()`):

In [32]:

```
class Temps:
    """Représente le moment de la journée.

    attributs : heure, minute, seconde
    """

    def afficher_temps(self):
        print('%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde))

    def temps_vers_int(self):
        minutes = self.heure * 60 + self.minute
        secondes = minutes * 60 + self.seconde
        return secondes

    def est_apres(self, other):
        return self.temps_vers_int() > other.temps_vers_int()
```

In [33]:

```
chrono = Temps()
chrono.heure = 9
chrono.minute = 12
chrono.seconde = 10

timer = Temps()
timer.heure = 3
timer.minute = 56
timer.seconde = 22

chrono.est_apres(timer)
```

Out[33]:

```
True
```

L'argument `other` (`timer` dans l'exemple) est le deuxième de la fonction et doit être indiqué dans les parenthèses. L'avantage de cette méthode est sa ressemblence avec le français.

La méthode `__init__()`.

La méthode `__init__()` est l'une des méthodes spéciales les plus importante. Elle permet d'initialiser chaque instance qu'on crée, fixant ainsi une valeur de base souhaitée pour chacun des attributs et en fonction des arguments donnés dans l'instanciation.

In [34]:

```
class Temps:
    """Représente le moment de la journée.

    attributs : heure, minute, seconde
    """

    def __init__(self, heure = 0, minute = 0, seconde = 0):
        self.heure = heure
        self.minute = minute
        self.seconde = seconde

    def afficher_temps(self):
        print('%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde))
```

Les arguments sont affectés aux attributs correspondants. Cela facilite grandement l'instanciation, car les attributs peuvent tous être inscrits dans l'ordre entre les parenthèses en tant qu'arguments de l'instanciation :

In [35]:

```
rdv = Temps(15, 30, 0)
rdv.afficher_temps()
```

```
15:30:00
```

La méthode `__str__()`.

La méthode `__str__()` permet de représenter une instance sous la forme d'un `str` lorsqu'elle est appelée par la fonction `print()`.

In [36]:

```
class Temps:
    """Représente le moment de la journée.

    attributs : heure, minute, seconde
    """

    def __init__(self, heure = 0, minute = 0, seconde = 0):
        self.heure = heure
        self.minute = minute
        self.seconde = seconde

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde)
```

La méthode `__str__()` renvoie toutes les informations souhaitée de l'instance sous forme de string. Lorsque la fonction print reçoit comme argument une instance, elle affichera automatiquement ce que renvoie la méthode `__str__()` :

In [37]:

```
moment = Temps(12, 45, 2)
print(moment)
```

```
12:45:02
```

Cette méthode est très utile pour vérifier les donnée d'une instance et permet un débogage plus facile.

Les méthodes d'opérateurs

Lorsqu'on utilise des opérateurs sur des instances de classe, généralement ceux-ci ne supportent pas les opérandes sous forme d'instance. Il existe des méthodes qui spécifie l'opération à faire pour chaque opérateur avec chaque type de classe :

In [38]:

```
class Temps:
    """Représente le moment de la journée.

    attributs : heure, minute, seconde
    """

    def __init__(self, heure = 0, minute = 0, seconde = 0):
        self.heure = heure
        self.minute = minute
        self.seconde = seconde

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde)

    def temps_vers_int(self):
        minutes = self.heure * 60 + self.minute
        secondes = minutes * 60 + self.seconde
        return secondes

    def int_vers_temps(secondes):
        temps = Temps()
        minutes, temps.seconde = divmod(secondes, 60)
        temps.heure, temps.minute = divmod(minutes, 60)
        return temps
```

```
    def __add__(self, other):
        secondes = self.temps_vers_int() + other.temps_vers_int()
        return int_vers_temps(secondes)
```

La méthode `__add__()` permet ainsi d'additionner deux instances de type Temps en utilisant l'opérateur `+` entre les deux opérandes et d'afficher le résultat avec `print()` :

In [39]:

```
chrono = Temps(10, 11, 34)
time = Temps(2, 23, 12)
print(chrono + time)
```

```
12:34:46
```

La fonction print prend automatiquement le retour de la méthode `__str__()` pour afficher le total.

Voici une liste des opérateurs et leur méthode :

`+` : `__add__(self, other)`

`-` : `__sub__(self, other)`

`*` : `__mul__(self, other)`

`//` : `__floordiv__(self, other)`

`/` : `__truediv__(self, other)`

`%` : `__mod__(self, other)`

`**` : `__pow__(self, other, [modulo])`

`<<` : `__lshift__(self, other)`

`>>` : `__rshift__(self, other)`

`&` : `__and__(self, other)`

`^` : `__xor__(self, other)`

`|` : `__or__(self, other)`

Héritage

L'héritage est une fonctionnalité qui permet de créer, à partir de classes existantes, des classes semblables modifiées. Cela permet à des classes d'intéragire facilement car elles se ressemblent et partagent les mêmes attributs de classes.

Par exemple, avec une classe `Carte` :

In [40]:

```
class Carte:
    """Représente une carte à jouer standard."""

    noms_couleurs = ['trèfle', 'carreau', 'coeur', 'pique']
    noms_valeurs = [None, 'as', '2', '3', '4', '5', '6', '7',
                    '8', '9', '10', 'valet', 'dame', 'roi']

    def __init__(self, couleur = 0, valeur = 2):
        self.couleur = couleur
        self.valeur = valeur

    def __str__(self):
        return '%s de %s' % (Carte.noms_valeurs[self.valeur],
                             Carte.noms_couleurs[self.couleur])
```

Pour créer un objet Carte on indique sa couleur et sa valeur. Les variables `noms_couleurs` et `noms_valeurs` sont des attributs de classe. Ils sont propre à la classe et ne change jamais. Ces attribut de classe nous permette de traduire les valeurs numérique des instances afin de les comprendres.

In [41]:

```
carte1 = Carte(1, 12)
```

La `carte1` à comme attributs `couleur=1` et `valeur=12`. Grâce aux attributs de classe nous pouvons afficher la description exacte de la carte (en français):

In [42]:

```
print(carte1)
```

```
dame de carreau
```

Comparaison

Tout comme les opérateurs et les méthodes comme `__add__()`, il existe aussi des méthodes de comparaison qui corresponde aux signes `<`, `>`, `==`. Pour comparer deux cartes savoir laquelle est plus petite que l'autre il y a la méthode `__lt__()` qui correspond à la comparaison `<` :

In [43]:

```
class Carte:
    """Représente une carte à jouer standard."""

    noms_couleurs = ['trèfle', 'carreau', 'coeur', 'pique']
    noms_valeurs = [None, 'as', '2', '3', '4', '5', '6', '7',
                    '8', '9', '10', 'valet', 'dame', 'roi']

    def __init__(self, couleur = 0, valeur = 2):
        self.couleur = couleur
        self.valeur = valeur

    def __str__(self):
        return '%s de %s' % (Carte.noms_valeurs[self.valeur],
                             Carte.noms_couleurs[self.couleur])

    def __lt__(self, other):
        t1 = self.couleur, self.valeur
        t2 = other.couleur, other.valeur
        return t1 < t2
```

In [44]:

```
carte1 = Carte(1, 12)
carte2 = Carte(0, 13)
carte1 < carte2
```

Out[44]:

```
False
```

In [45]:

```
carte2 < carte1
```

Out[45]:

```
True
```

Voici une liste des opérateurs de comparaison et leur méthode:

`<` : `__lt__(self, other)`

`<` : `__lt__(self, other)`

`<=` : `__le__(self, other)`

`==` : `__eq__(self, other)`

`!=` : `__ne__(self, other)`

`>=` : `__ge__(self, other)`

`>` : `__gt__(self, other)`

Voici maintenant une classe `Paquet` avec l'attribut `cartes` initialisé :

In [46]:

```
class Paquet:
    """Représente un paquet de 52 cartes."""

    def __init__(self):
        self.cartes = []
        for couleur in range(4):
            for valeur in range(1, 14):
                carte = Carte(couleur, valeur)
                self.cartes.append(carte)

    def __str__(self):
        res = []
        for carte in self.cartes:
            res.append(str(carte))
        return '\n'.join(res)
```

Avec la méthode `join` la méthode `__str__` retourne chaque carte sous forme de `str` avec devant '\n' qui provoque un retour à la ligne :

In [47]:

```
paquet = Paquet()
print(paquet)
```

```
as de trèfle
2 de trèfle
3 de trèfle
4 de trèfle
5 de trèfle
6 de trèfle
7 de trèfle
8 de trèfle
9 de trèfle
10 de trèfle
valet de trèfle
dame de trèfle
roi de trèfle
as de carreau
2 de carreau
3 de carreau
4 de carreau
5 de carreau
6 de carreau
7 de carreau
8 de carreau
9 de carreau
10 de carreau
valet de carreau
dame de carreau
roi de carreau
as de coeur
2 de coeur
3 de coeur
4 de coeur
5 de coeur
6 de coeur
7 de coeur
```

```
8 de coeur
9 de coeur
10 de coeur
valet de coeur
dame de coeur
roi de coeur
as de pique
2 de pique
3 de pique
4 de pique
5 de pique
6 de pique
7 de pique
8 de pique
9 de pique
10 de pique
valet de pique
dame de pique
roi de pique
```

Pour manipuler les cartes d'un paquet, il faut utiliser les méthodes de liste comme `pop` et `append`.

In [48]:

```
def pop_carte(self):
    return self.cartes.pop()

def ajouter_carte(self, carte):
    self.cartes.append(carte)
```

Pour mélanger les cartes, la fonction `shuffle` du module `random` s'y prête bien :

In [49]:

```
def battre(self):
    random.shuffle(self.cartes)
```

La technique d'héritage est la création d'une nouvelle classe modifiée, issue d'une classe déjà existante. Les classes déjà existantes sont appelée classe mère et les classes qui hérite de celles-ci sont appelées classe fille. Cela illustre le lien de parentées des classes.

Pour créer une classe qui hérite d'une autre, la classe déjà existante doit être mise en parenthèse :

In [50]:

```
class Main(Paquet):
    """Représente une main au jeu de cartes."""
```

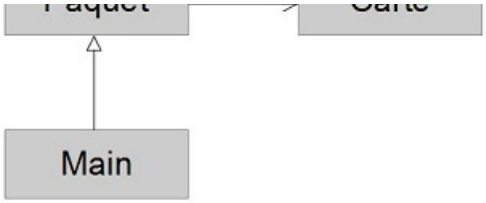
la classe `Main` possède les mêmes méthodes que la classe `Paquet`. Il est possible d'en modifier une en la redéfinissant :

In [51]:

```
class Main(Paquet):
    """Représente une main au jeu de cartes."""

    def __init__(self, etiquette = ''):
        self.cartes = []
        self.etiquette = etiquette
```

L'objet main est initialisé vide, mais elle possède les méthodes `pop` et `append` qui lui permettent d'ajouter et retirer des cartes comme pour le paquet.

Voici à quoi ressemble leur diagramme de classes :



- La flèche à pointe triangulaire creuse représente une relation IS-A. C'est un héritage.
- La flèche à pointe normale représente une relation HAS-A. Ce sont les références.
- L'astérisque indique un nombre quelconque de cartes.

Sources

- Pensez en Python, par Allen B. Downey (traduit par Mishulyna & Laurent Rosenfeld)
- Cours de M. Raphael Holzer
- https://www.python-course.eu/python3_magic_methods.php

In [ ]:

```
```