Python : Objets et décorateurs

Rémi Oudin

Séminaire Technique du Cr@ns

15 Novembre 2016

Ce séminaire fait suite au séminaire Python avancé

1. Programmation Orientée objet
   - Introduction
   - Classes
   - Les Built-In
   - Héritage
2. Les décorateurs
   - Introduction
   - La syntaxe
   - Un vrai exemple
   - Décoration avancée
3. Une application : gérer des arguments en commandline
   - Introduction
4. Un dernier point : les conventions en Python
   - Pourquoi tant de conventions ?
   - La fameuse PEP 8
   - La PEP 257
5. Conclusion

1. Programmation Orientée objet
   - Introduction
   - Classes
   - Les Built-In
   - Héritage
2. Les décorateurs
   - Introduction
   - La syntaxe
   - Un vrai exemple
   - Décoration avancée
3. Une application : gérer des arguments en commandline
   - Introduction
4. Un dernier point : les conventions en Python
   - Pourquoi tant de conventions ?
   - La fameuse PEP 8
   - La PEP 257
5. Conclusion

La Programmation Orientée Objet : C'est quoi ?

- Une *classe* est une structure abstraite qui permet de créer des plusieurs objets différents, mais de même type. Ainsi, je peux créer un `compagnon` comme une instance de `Animal` : `compagnon = Animal(<parameters>)`
- Les objets ont des **attributs** : `pet.couleur`, `compagnon.poids`. Ils peuvent différer d'une instance à l'autre, et être modifiés (normalement, pas depuis l'extérieur de l'objet).
- Ils ont également des **méthodes**, fonctions internes à l'objet : `compagnon.crier()`, `pet.manger(qte_nourriture)`.

La Programmation Orientée Objet : pourquoi ?

Encapsulation : Chaque objet vient avec ses méthodes, son comportement propre. De l'extérieur, on n'a pas besoin de savoir comment il fonctionne pour l'utiliser, juste de connaître son interface (les noms de ses méthodes, et de ses attributs).

La Programmation Orientée Objet : pourquoi ?

Encapsulation : Chaque objet vient avec ses méthodes, son comportement propre. De l'extérieur, on n'a pas besoin de savoir comment il fonctionne pour l'utiliser, juste de connaître son interface (les noms de ses méthodes, et de ses attributs).

Réutilisation de code : Une classe est facilement réutilisable autre part. C'est ce qu'on fait en permanence quand on code en python.

La Programmation Orientée Objet : pourquoi ?

Encapsulation : Chaque objet vient avec ses méthodes, son comportement propre. De l'extérieur, on n'a pas besoin de savoir comment il fonctionne pour l'utiliser, juste de connaître son interface (les noms de ses méthodes, et de ses attributs).

Réutilisation de code : Une classe est facilement réutilisable autre part. C'est ce qu'on fait en permanence quand on code en python.

Clarté : Lorsqu'on crée un objet, il vient avec *son* comportement, *ses* méthodes

La Programmation Orientée Objet : pourquoi ?

Encapsulation : Chaque objet vient avec ses méthodes, son comportement propre. De l'extérieur, on n'a pas besoin de savoir comment il fonctionne pour l'utiliser, juste de connaître son interface (les noms de ses méthodes, et de ses attributs).

Réutilisation de code : Une classe est facilement réutilisable autre part. C'est ce qu'on fait en permanence quand on code en python.

Clarté : Lorsqu'on crée un objet, il vient avec *son* comportement, *ses* méthodes

Don't Repeat Yourself : On écrit une fois les méthodes et les classes, et après on n'a plus a réécrire des fonctions parfois différentes de quelques caractères (cf la notion d'héritage)

Définition d'une classe

```
class Animal:
        def __init__(self, poids, nom):
            """Initialiseur"""
            self.poids = poids
            self.nom = nom

        def crier(self):
            print("Grou")

        def manger(self, foodsize):
            self.poids += foodsize
>>> compagnon = Animal(3, "Neo")
>>> compagnon.poids
3
>>> compagnon.manger(5)
>>> compagnon.poids
8
>>> compagnon.crier()
Grou
```

Quelques conventions

- On nomme les classes en *CamlUpperCase*, les méthodes en *lower_case* (cf PEP 8, mais on en reparlera plus tard)
- Parlons de `self` : Il représente l'objet sur lequel on applique la méthode. Le nom est une convention (PEP 8 toujours), on pourrait le remplacer, contrairement à `this` de Java ou C++.
- En théorie, chaque méthode doit avoir une docstring (cf PEP 257)
- On ne devrait pas toucher directement (par exemple, faire `compagnon.poids = 4`), on va voir pourquoi...

Respect du privé

```
>>> import math
>>> class Vecteur:
        def __init__(self, x, y):
            self.x, self.y = x, y
            self._update_norme()

        def __calculer_norme(self):
            return math.sqrt(self.x**2 + self.y**2)

        def _update_norme(self):
            self.norme = self.__calculer_norme()

        def setcoords(self, newx, newy):
            self.x, self.y = newx, newy
            self._update_norme()

>>> v = Vecteur(3, 4)
>>> v.norme
5.0
>>> v.x = 10
>>> v.norme
5.0
>>> v.setcoords(6, 8)
>>> v.norme
10.0
```

Plus ou moins privé

- Par convention (PEP 8, oui vous l'avez deviné) les méthodes commençant par _ sont privées. On peut quand même y accéder (en fait, on peut presque toujours accéder à tout en Python.

```
>>> v.x, v.y = 9, 12
>>> v._update_norme()
>>> v.norme
15.0
```

- Les méthodes commençant par _ _ mais qui ne finissent pas par _ _ sont réellement privées et donc inaccessibles (mais pas trop malgré tout...) :

```
>>> v.__calculer_norme()
AttributeError: Vecteur instance has no attribute '__calculer_norme'
>>> v._Vecteur__calculer_norme()
15.0
```

La méthode peut être retrouvée quand même. Python la renomme en `_<ClassName>_ _<method>`.

Les opérateurs Built-in des classes

- `__init__`
- `__repr__`(self)
- `__str__`(self)
- `__add__`(self, other) → +
- `__sub__`(self, other) → −
- `__mul__`(self, other) → ∗
- `__pow__`(self, other) → ∗∗
- `__truediv__`(self, other) → /
- `__floordiv__`(self, other) → /
- `__mod__`(self, other) → %
- `__lt__`(self, other) →<=
- `__gt__`(self, other) →>=
- `__eq__`(self, other) →==
- `__ne__`(self, other) →! =
- `__and__`(self, other) → &
- `__or__`(self, other) → |

Un exemple

```
class Vecteur:

    [...]

    def __repr__(self):
        return "(%d, %d)" % (self.x, self.y)

    def __str__(self):
        return "Vecteur de coordonnées (%d, %d)" % (self.x, self.y)

    def __add__(self, other):
        return Vecteur(self.x + other.x, self.y + other.y)
```

```
>>> v1 = Vecteru(3,4)
>>> v2 = Vecteru(4,3)
>>> v1
(3, 4)
>>> str(v1)
'Vecteur de coordonnées (3, 4)'
>>> print(v1)
Vecteur de coordonnées (3, 4)
>>> v1 + v2
(7,7)
```

Héritage

Reprenons l'exemple des animaux : Il y a plein de types d'animaux, et chacun a des comportements, des caractéristiques différentes. On pourrait créer une classe par animal, mais ça ressemblerait à ça :

Héritage

Reprenons l'exemple des animaux : Il y a plein de types d'animaux, et chacun a des comportements, des caractéristiques différentes. On pourrait créer une classe par animal, mais ça ressemblerait à ça :

```
class Chat:
    def __init__(self, poids):
        self.poids = poids.
        self.cri = "Miaou"

    def crier(self):
        print(self.cri)
```

```
class Chien:
    def __init__(self, poids):
        self.poids = poids.
        self.cri = "Ouaf"

    def crier(self):
        print(self.cri)
```

```
class Loutre:
    def __init__(self, poids):
        self.poids = poids.
        self.cri = "Grouuu"

    def crier(self):
        print(self.cri)
```

DRY++

- On peut factoriser le code en définissant la même méthode pour plusieurs classes.

```
>>> class Animal:
        def __init__(self, poids):
            """Initialiseur"""
            self.poids = poids
        def crier(self):
            print "%s !" % (self.cri)
        def manger(self, foodsize):
            self.poids += foodsize

>>> class Chien(Animal):
        def __init__(poids):
            super().__init__(poids)
            self.cri = "Ouaf"

>>> class Chat(Animal):
        def __init__(self, poids):
            super().__init__(poids)
            self.cri = "Miaou"

>>> snoopy, felix = Chien(5), Chat(4)
>>> snoopy.manger(2)
>>> felix.manger(2)
>>> snoopy.poids, felix.poids
(7, 6)
>>> snoopy.crier(); felix.crier()
Ouaf !
Miaou !
```

DRY++

- On peut factoriser le code en définissant la même méthode pour plusieurs classes.
- Appeler l'initialiseur du parent n'est pas automatique (et donc pas obligatoire).

```
>>> class Animal:
        def __init__(self, poids):
            """Initialiseur"""
            self.poids = poids
        def crier(self):
            print "%s !" % (self.cri)
        def manger(self, foodsize):
            self.poids += foodsize

>>> class Chien(Animal):
        def __init__(poids):
            super().__init__(poids)
            self.cri = "Ouaf"

>>> class Chat(Animal):
        def __init__(self, poids):
            super().__init__(poids)
            self.cri = "Miaou"

>>> snoopy, felix = Chien(5), Chat(4)
>>> snoopy.manger(2)
>>> felix.manger(2)
>>> snoopy.poids, felix.poids
(7, 6)
>>> snoopy.crier(); felix.crier()
Ouaf !
Miaou !
```

DRY++

- On peut factoriser le code en définissant la même méthode pour plusieurs classes.
- Appeler l'initialiseur du parent n'est pas automatique (et donc pas obligatoire).
- On peut écraser une méthode :

```
>>> class Lapin(Animal):
      [...]
      def crier(self):
        raise NotImplementedError
```

-
```
>>> class Animal:
        def __init__(self, poids):
            """Initialiseur"""
            self.poids = poids
        def crier(self):
            print "%s !" % (self.cri)
        def manger(self, foodsize):
            self.poids += foodsize

>>> class Chien(Animal):
        def __init__(poids):
            super().__init__(poids)
            self.cri = "Ouaf"

>>> class Chat(Animal):
        def __init__(self, poids):
            super().__init__(poids)
            self.cri = "Miaou"

>>> snoopy, felix = Chien(5), Chat(4)
>>> snoopy.manger(2)
>>> felix.manger(2)
>>> snoopy.poids, felix.poids
(7, 6)
>>> snoopy.crier(); felix.crier()
Ouaf !
Miaou !
```

Comment ça fonctionne ? Le late binding

- Lorsqu'on appelle une méthode, l'interpréteur regarde au niveau le plus profond (la sous-classe la plus spécifique) si elle existe.
- Si elle y est, elle l'exécute.
- Sinon, elle parcourt séquentiellement toutes les classes dont hérite l'objet, et effectue cette recherche.
- Si à la fin, elle n'a toujours pas été trouvée, on soulève une erreur.

Les new-style classes

En Python3 (ce dont je vous parle depuis le début), toutes les classes sont des *new-style* classes, elles ont de nombreuses fonctionnalités :

- Unifie types et classes → Une classe est juste un type défini par l'utilisateur
- Permet le support des méta-classes (Oui, des classes pour créer des classes...)
- Les properties : Gérer facilement les setters et les getters :

```
class C(object):

    def __init__(self):
        self.__x = 0

    def getx(self):
        return self.__x

    def setx(self, x):
        if x < 0: x = 0
        self.__x = x

    x = property(getx, setx)
```

```
>>> a = C()
>>> a.x = 10
>>> print(a.x)
10
>>> a.x = -10
>>> print(a.x)
0
>>> a.setx(12)
>>> print(a.getx())
12
```

- La signature : `property(fget=None, fset=None, fdel=None, doc=None)`

Les décorateursoù Comment se protéger...

- On veut pouvoir modifier une fonction, ou alors effectuer des actions lors de l'appel d'une fonction en particulier.
- On n'a pas forcément envie de l'écrire dans chaque fonction [1], ou que l'utilisateur le sache.
- Ils sont définis dans la PEP 318
- Un décorateur emballe (*wrap*) la fonction qu'il décore par des opérations avant et/ou après.

1. rappelez-vous, DRY, mais je me répète...

La syntaxe

Soit `decorator` une fonction qui prend en argument une fonction `func`. Si on veut décorer la fonction `foobar` par `decorate`, on adopte la syntaxe suivante :

```
def decorator(func):
    [...] #Définition de decorator

@decorator
def foobar():
    print("I am ordinary")
```

```
def decorator(func):
    [...] #Définition de decorator

def foobar():
    print("I am ordinary")

foobar = decorator(foobar)
```

Ne pas se faire avoir

Attention

C'est au moment de définir les fonctions que les décorateurs sont appliqués.

```
>>> def decorate(func):
        print("I decorate")
        return func

>>> @decorate
    def ordinary():
        print("I am ordinary")

I decorate

>>> ordinary()
I am ordinary
```

```
>>> def decorate(func):
        def inner():
            print("I decorate")
            func()
            print("I have decorated")
        return inner

>>> @decorate
    def ordinary():
        print("I am ordinary")

>>> ordinary()
I decorate
I am ordinary
I have decorated
```

Un vrai exemple

```
import logging
import math

logging.basicConfig(filename='calls.log', level = logging.DEBUG)

def logger(level=logging.DEBUG):
    def decorated(func):
        def wrapper(*args, **kwargs):
            logging.log(level, "Called function %s on %s"
                        % (func.__name__, args[1:]))
            return func(*args, **kwargs)
        return wrapper
    return decorated

class Vecteur:
    [...]

    @logger()
    def __calculer_norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    @logger()
    def _update_norme(self):
        self.norme = self.__calculer_norme()

    @logger(logging.CRITICAL)
    def setcoords(self, newx, newy):
        self.x, self.y = newx, newy
        self._update_norme()
```

L'exécution

On exécute le code suivant, et on observe :

```
>>> v = Vecteur(3,4)
>>> v.setcoords(4,3)
>>> from subprocess import call
>>> call(["cat","calls.log"])
DEBUG:root:Called function _update_norme on ()
DEBUG:root:Called function __calculer_norme on ()
CRITICAL:root:Called function setcoords on (4, 3)
DEBUG:root:Called function _update_norme on ()
DEBUG:root:Called function __calculer_norme on ()
```

- On a appelé un décorateur
- On sait gérer des décorateurs avec des arguments
- On peut logguer tous les appels de fonctions, ou faire d'autres choses (envoyer un mail ?)
- Les décorateurs sont très utiles par exemple en Django, mais vous verrez ça dans le séminaire consacré à Django.

Encore plus loin

- On peut chaîner les décorateurs
- Leur ordre compte :

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner

@star
@percent
def printer(msg):
    print(msg)
```

```
>>> printer("Hello")
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

Le décorateur @property

Vous vous rappelez de property ? Il y a un décorateur spécial, built-in qui permet de le gérer plus facilement.

```
class Celsius:
    def __init__(self, temperature = 0):
        self._temperature = temperature

    @property
    def temperature(self):
        print("Getting value")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value
```

```
>>> c = Celsius()
>>> c.temperature
Getting value
0
>>> c.temperature = 10
Setting value
```

Introduction

- On veut pouvoir gérer facilement les arguments donnés en ligne de commande
- On veut utiliser la programmation objet pour rendre ça agréable à écrire.
- Il existe une bibliothèque qui permet de faire ça : argparse

Parser les paramètres commandline

- Code :

```
#!/usr/bin/python3
# -*- coding : utf-8 -*-
import argparse

# Déclaration du parser
PARSER = argparse.ArgumentParser(description="Help message")
PARSER.add_argument("-v", "--verbose", help="Be more verbose",
                    action="store_true")
PARSER.add_argument("-f", "--file", help="File to handle",
                    action="store", type=str, metavar="FILE")
PARSER.add_argument("-n", dest="num", help="Line number",
                    action="store", type=int)

if __name__ == "__main__":
    # Parsing
    print("Before parsing")
    args = PARSER.parse_args()
    print("After parsing")

    # Utilisation des arguments parsés
    print("Valeurs : %r, %r, %r" % (args.verbose, args.file, args.num))
```

Parser les paramètres commandline

- Exécution :

```
remi@olaf $ ./parse.py -h
Before parsing
usage: parse.py [-h] [-v] [-f Fichier] [-n NUM]

Message d'aide

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Be more verbose
  -f Fichier, --file FILE
                        File to handle
  -n NUM                Line number

remi@olaf $ ./parse.py -f Pif
Before parsing
After parsing
Valeurs : False, 'pif', None

remi@olaf $ ./parse.py -f Pif -n
Before parsing
usage: parse.py [-h] [-v] [-f Fichier] [-n NUM]
parse.py: error: argument -n: expected one argument
```

Parser les paramètres commandline

- *argparse* est bien plus puissant que cela
- Regarder la doc et les docstrings d'argparse pour voir toutes les possibilités
- Notamment, `parser.add_mutually_exclusive_group`, qui permet, comme son nom l'indique, de créer des groupes d'options mutuellement exclusives

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

Pourquoi tant de conventions ?

- Python a été créé pour être facile à lire et à comprendre
- Respecter des conventions simples permet de rendre le code « presque » compréhensible tout le temps
- Un code est plus souvent lu qu'écrit.
- Ceci peut aider en plus les outils de génération automatique (documentation, coverage graph...)
- Si on ne respecte pas les conventions, il est plus difficile d'avoir des question à ses réponses, car les autres n'arriveront pas à lire.
- Python a mis en place pour cela les PEPs : *Python Enhancement Proposals*

La fameuse PEP 8

Elle décrit les conventions d'écriture de code en Python. Les points essentiels sont :

- Indenter à 4 espaces
- Les commandes sur plusieurs lignes doivent s'aligner verticalement
- La longueur maximale d'une ligne est de 79 caractères pour le code, et 82 caractères pour les commentaires et les docstrings
- Une ligne est coupée avant un opérateur binaire
- Encoder les fichiers en UTF-8
- Un import de module par ligne.
- Les imports wildcards `from <module> import *` sont à éviter
- Mettre un espace blanc de chaque côté d'un opérateur binaire, sauf pour distinguer la priorité sur les opérateurs.
- Mettre un espace blanc après une virgule, un point-virgule, un double-point
- Ne pas mettre d'espace blanc après une parenthèse ouvrante, ou avant une parenthèse fermante

La PEP 257

Elle décrit le format des docstrings :

- Toujours utiliser des """triple double quotes""" pour écrire des docstrings.
- Les one-liners sont à réserver pour les docstrings évidentes
- Une docstring est un ensemble de phrases, avec des majuscules et des points.
- Pour les docstrings sur plusieurs lignes, on adopte un format tel quel :

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Un outil bien pratique : pylint3

L'outil `pylint3` permet de vérifier de manière statique votre code. Entre autres, il marque les non-respects des conventions, et donne une note générale au code.

```
************* Module vecteur
C:  1, 0: Missing module docstring (missing-docstring)
C:  6, 8: Invalid attribute name "x" (invalid-name)
C:  6,16: Invalid attribute name "y" (invalid-name)
C:  4, 0: Missing class docstring (missing-docstring)
C: 15, 4: Missing method docstring (missing-docstring)
R:  4, 0: Too few public methods (1/2) (too-few-public-methods)
W:  1, 0: Unused import geometry (unused-import)
C:  2, 0: standard import "import math" comes before "import geometry"
          (wrong-import-order)


Report
======
19 statements analysed.

[......]

Global evaluation
-----------------
Your code has been rated at 5.79/10 (previous run: 5.79/10, +0.00)
```

Conclusion

Il reste encore énormément de choses qu'on a pas vu :

- Les méta-classes
- Les décorateurs de classes
- Les itérateurs, les générateurs
- Tant d'autres choses encore...