

Décorateurs

Complément - niveau (très) avancé

Le mécanisme des décorateurs – qui rappelle un peu, pour ceux qui connaissent, les macros Lisp – est un mécanisme très puissant. Sa portée va bien au delà de simplement rajouter du code avant et après une fonction, comme dans le cas de `NbAppels` que nous avons vu dans la vidéo.

Par exemple, les notions de méthodes de classe (`@classmethod`) et de méthodes statiques (`@staticmethod`) sont implémentées comme des décorateurs. Pour une liste plus représentative de ce qu'il est possible de faire avec les décorateurs, je vous invite à parcourir même rapidement ce recueil de décorateurs (<https://wiki.python.org/moin/PythonDecoratorLibrary>) qui propose du code (à titre indicatif car rien de ceci ne fait partie de la librairie standard) pour des thèmes qui sont propices à la décoration de code.

Nous allons voir en détails quelques-uns de ces exemples, nos décorateurs `caching` et `singleton` ont la même fonctionnalité que les `memoize` et `singleton` du recueil de décorateurs, mais implémentés différemment.

Un décorateur implémenté comme une fonction

Dans la vidéo on a vu `NbAppels` pour compter le nombre de fois qu'on appelle une fonction. Pour mémoire on avait écrit :

```
# un rappel du code montré dans la vidéo
class NbAppels(object):
    def __init__(self, f):
        self.f = f
        self.appels = 0
    def __call__(self, *args):
        self.appels += 1
        print "{}: {} appels".format(self.f.__name__, self.appels)
        return self.f(*args)

@NbAppels
def fibo_aux(n):
    "Fibonacci en log(n), credits Bernardi"
    if n < 1:
        return 0, 1
    u, v = fibo_aux(n/2)
    u, v = u * (2 * v - u), u*u + v*v
    if n & 1:
```

```

        return v, u + v
    else:
        return u, v

def fibo_log(n):
    return fibo_aux(n)[0]


fibo_log(300)

```

Ici nous allons implémenter **caching**, un décorateur qui permet de mémoriser les résultats d'une fonction, et de les cacher pour ne pas avoir à les recalculer la fois suivante.

Alors que `NbAppels` était **implémenté comme une classe**, pour varier un peu, nous allons implémenter cette fois **caching comme une vraie fonction**, pour vous montrer les deux alternatives que l'on a quand on veut implémenter un décorateur : une vraie fonction ou une classe de callables.

Le code du décorateur

 # une première implémentation de caching

```

# un decorateur de fonction
# implémenté comme une fonction
def caching(a_decorer):
    """
    Un décorateur pour conserver les résultats
    précédents et éviter de les recalculer
    """
    def decoree(*args):
        # si on a déjà calculé le résultat
        # on le renvoie
        try:
            return decoree.cache[args]
        # si les arguments ne sont pas hashables,
        # par exemple une liste, on ne peut pas cacher
        # et on reçoit TypeError
        except TypeError:
            return a_decorer(*args)
        # les arguments sont hashables mais on
        # n'a pas encore calculé cette valeur
        except KeyError:
            result = a_decorer(*args)
            decoree.cache[args] = result
            return result
    # on initialise l'attribut 'cache'
    decoree.cache = {}
    return decoree

```

Comment l'utiliser

Avant de rentrer dans le détail du code, voyons comment cela s'utiliserait; il n'y a pas de

changement de ce point de vue par rapport à l'option développée dans la vidéo :

```
> # créer une fonction décorée
@ caching
def fibo_cache(n):
    """
    Un fibonacci hyper-lent (exponentiel) se transforme
    en temps linéaire une fois que les résultats sont cachés
    """
    return n if n <= 1 else fibo_cache(n-1) + fibo_cache(n-2)
```

Bien que l'implémentation utilise un algorithme épouvantablement lent, le fait de lui rajouter du caching redonne à l'ensemble un caractère linéaire.

En effet, si vous y réfléchissez une minute, vous verrez qu'avec le cache, lorsqu'on calcule `fibo_cache(n)`, on calcule d'abord `fibo_cache(n-1)`, puis lorsqu'on évalue `fibo_cache(n-2)` le résultat **est déjà dans le cache** si bien qu'on peut considérer ce deuxième calcul comme, sinon instantané, du moins du même ordre de grandeur qu'une addition.

On peut calculer par exemple :

```
> fibo_cache(300)
```

qu'il serait hors de question de calculer sans le caching.

On peut naturellement inspecter le cache, qui est rangé dans l'attribut `cache` de l'objet fonction lui-même :

```
> len(fibo_cache.cache)
```

et voir que, comme on aurait pu le prédire, on a calculé et mémorisé les 301 premiers résultats, pour `n` allant de 0 à 300.

Comment ça marche ?

On l'a vu dans la vidéo avec `NbAppels`, tout se passe exactement comme si on avait écrit :

```
> def fibo_cache(n):
    <le code>

    fibo_cache = caching(fibo_cache)
```

Donc `caching` est une fonction qui prend en argument une fonction `a_decorer` qui ici vaut `fibo_cache`, et retourne une autre fonction, `decoree`; on s'arrange naturellement pour que `decoree` retourne le même résultat que `a_decorer`, avec seulement des choses supplémentaires.

Les points clés de l'implémentation sont les suivants.

- On attache à l'objet fonction `decoree` un dictionnaire `cache` qui va nous permettre de retrouver les valeurs déjà calculées, à partir d'un hash des arguments.

- On ne peut pas cacher le résultat d'un objet qui ne serait pas globalement immuable; or si on essaie on reçoit l'exception `TypeError`, et dans ce cas on recalcule toujours le résultat. C'est de toute façons plus sûr.
- Si on ne trouve pas les arguments dans le cache, on reçoit l'exception `KeyError`, dans ce cas on calcule le résultat, et on le retourne après l'avoir rangé dans le cache.
- Vous remarquerez aussi qu'on initialise l'attribut `cache` dans l'objet `decoree` à l'appel du décorateur (une seule fois, juste après avoir défini la fonction), et non pas dans le code de `decoree` qui lui est évalué à chaque appel.

Cette implémentation, sans être parfaite, est tout à fait utilisable dans un environnement réel, modulo les remarques de bon sens suivantes :

- évidemment l'approche ne fonctionne que pour des fonctions déterministes; s'il y a de l'aléatoire dans la logique de la fonction, il ne faut pas utiliser ce décorateur ;
- tout aussi évidemment, la consommation mémoire peut être importante si on applique le caching sans discrimination ;
- enfin en l'état la fonction `decoree` ne peut pas être appelée avec des arguments nommés; en effet on utilise le tuple `args` comme clé pour retrouver dans le cache la valeur associée aux arguments.

Décorateurs, *docstring* et `help`

En fait, avec cette implémentation, il reste aussi un petit souci :

```
➤ help(fibo_cache)
```

Ce n'est pas exactement ce qu'on veut; ce qui se passe ici c'est que `help` utilise les attributs `__doc__` et `__name__` de l'objet qu'on lui passe. Et dans notre cas `fibo_cache` est une fonction qui a été créée par l'instruction

```
➤ def decoree(*args):
    # etc.
```

Pour arranger ça et faire en sorte que `help` nous affiche ce qu'on veut, il faut s'occuper de ces deux attributs. Et plutôt que de faire ça à la main, il existe un utilitaire dans le module `functools`, qui fait tout le travail nécessaire. Ce qui nous donne une deuxième version de ce décorateur, avec deux lignes supplémentaires signalées par des `+++` ;:

```
➤ # une deuxième implémentation de caching, avec la doc

import functools                                     # +++

# un decorateur de fonction
# implémenté comme une fonction
def caching(a_decorer):
    """
    Un décorateur pour conserver les résultats
```

```

précédents et éviter de les recalculer
"""
# on décore la fonction pour qu'elle ait les
# propriétés de a_decorer : __doc__ et __name__
@functools.wraps(a_decorer)          # +++
def decoree(*args):
    # si on a déjà calculé le résultat
    # on le renvoie
    try:
        return decoree.cache[args]
    # si les arguments ne sont pas hashables,
    # par exemple une liste, on ne peut pas cacher
    # et on reçoit TypeError
    except TypeError:
        return a_decorer(*args)
    # les arguments sont hashables mais on
    # n'a pas encore calculé cette valeur
    except KeyError:
        result = a_decorer(*args)
        decoree.cache[args] = result
        return result
# on initialise l'attribut 'cache'
decoree.cache = {}
return decoree

# créer une fonction décorée
@ caching
def fibo_cache2(n):
    """
    Un fibonacci hyper-lent (exponentiel) se transforme
    en temps linéaire une fois que les résultats sont cachés
    """
    return n if n <= 1 else fibo_cache2(n-1) + fibo_cache2(n-2)

```

Et on obtient à présent une aide en ligne cohérente :

```

❏ help(fibo_cache2)

```

On peut décorer les classes aussi

De la même façon qu'on peut décorer une fonction, on peut décorer une classe.

Voyons comment tirer profit des décorateurs pour implémenter le pattern de singleton ; une classe implémentée comme un singleton (http://en.wikipedia.org/wiki/Singleton_pattern) est une classe qui n'existe qu'en un seul exemplaire (une seule instance); tous les appels au constructeur retournent une référence vers le même objet.

Le décorateur singleton va nous permettre de transformer une classe en un singleton.

```

❏ # le décorateur qui transforme une classe en singleton

```

```

# le code gère correctement *args et **kwds, même si généralement
# dans le cas d'un singleton on n'a pas d'argument à passer
# au constructeur

# vous pouvez réactiver les lignes de print si vous voulez
# mieux voir ce qui se passe

class singleton(object):
    '''make the class a singleton'''

    def __init__(self, classe):
        """construit l'instance de la classe décorée"""

        # on garde une référence vers la classe à décorer
        self.classe = classe
        # l'instance
        self.instance = None
        # on met de cote la methode __new__
        self.new_original = classe.__new__
        # on redéfinit la méthode __new__
        @staticmethod
        def singleton_new(classe, *args, **kwds):
            # on retourne l'instance si elle existe
            if self.instance is not None:
                return self.instance
            # sinon on la cree en utilisant le __new__ original
            print 'in singleton_new', classe, 'args', args, 'kwds', kwds
            self.instance = self.new_original(classe, *args, **kwds)
            # on l'initialise
            self.instance.__init__(*args, **kwds)
            # on la retourne
            return self.instance

        # on 'installe' singleton_new comme le __new__ de la nouvelle classe
        self.classe.__new__ = singleton_new
        # pour la doc, quoi que ce soit apparemment insuffisant pour help()
        self.__name__ = classe.__name__
        self.__doc__ = classe.__doc__

        # l'objet doit être un callable, et si on l'appelle
        # il faut construire un objet de la classe a decorer
        def __call__(self, *args, **kwds):
            print 'in __call__', self, 'args', args, 'kwds', kwds
            return self.classe(*args, **kwds)

```

Le décorateur en action

```

# avec une classe 'normale'
class Eggs(object):
    "La classe Eggs n'est pas un singleton"

# si on crée deux objets ils sont différents

```

```

obj1 = Eggs()
obj2 = Eggs()
obj1 is obj2

# avec une classe décorée
@singleton
class Spam(object):
    "La classe Spam est un singleton"

# deux instances créés successivement sont identiques
obj1 = Spam()
obj2 = Spam()
obj1 is obj2

```

Comment ça marche

La subtilité ici réside dans le fait que

La méthode `__new__` fait partie de la même famille que `__init__` et autres `__repr__` dont on a parlé dans la Semaine 5, Séquence 'surcharge des opérateurs'. C'est une méthode spéciale également, et comme `__init__` elle est mise en jeu au moment de la création d'un objet.

Par contre, elle est vraiment particulière dans la ménagerie des méthodes spéciales, car c'est la seule qui soit une **méthode statique**, et ça se comprend quand on sait qu'elle est appelée **pour** créer l'objet instance ; on ne peut donc pas lui passer en paramètre l'instance en cours de création, puisqu'elle n'est pas encore là.

Rappelez-vous que lorsqu'on définit `Spam` décorée avec `singleton`, on fait essentiellement

```
Spam = singleton(Spam)
```

C'est donc que l'instance de la classe `singleton` est considéré comme une classe ; cette instance doit être un callable, qui est utilisé lorsqu'on crée une instance de `Spam` en faisant

```
obj1 = Spam()
```

Notre décorateur, pour l'essentiel, fait les opérations suivantes.

- Dans le constructeur `__init__` de la classe `singleton`, l'objet décoré mémorise la classe à décorer (ici `Spam`) dans son attribut `classe`.
- On crée l'attribut `instance` qui contiendra une référence vers l'unique objet créé pour cette classe.
- On mémorise également la méthode `__new__` de la classe à décorer dont on aura besoin à l'intérieur de notre propre version de `__new__`.
- On définit une méthode statique (parce que `__new__` doit être statique) `singleton_new` qui est destinée à remplacer la méthode `__new__` dans la classe décorée.
- Cette nouvelle méthode `__new__`
 - renvoie l'attribut `instance` sur la classe décorée lorsqu'il existe (c'est qu'on déjà

- crée l'unique objet de la classe à décorer);
- et sinon utilise les méthode `__new__` et `__init__` originales pour instancier un objet, le mémoriser, et le renvoyer.
- `singleton_new` est *installée* dans la classe décorée, dans l'attribut de nom `__new__`.
- On met à jour `__name__` et `__doc__` dans la classe décorée à partir de la classe originale.

Toutes les combinaisons (fonction, classe) x (fonction, classe)

Nous avons choisi d'implémenter `singleton` comme une classe ; vous trouverez ici une version implémentée comme une fonction

(<https://wiki.python.org/moin/PythonDecoratorLibrary#Singleton>) .

Si bien qu'avec ces 4 exemples

- `NbAppels` de la vidéo,
- notre version de `cached`,
- notre version de `singleton`,
- et le `singleton` de la référence ci- dessus

(<https://wiki.python.org/moin/PythonDecoratorLibrary#Singleton>) ,

vous avez toutes les combinaisons de décorateurs, de classes ou de fonctions, implémentés comme une classe ou comme une fonction.

Un décorateur peut lui-même avoir des arguments

Reprenons l'exemple de `cached`, mais imaginons qu'on veuille ajouter un trait de "durée de validité du cache". Le code du décorateur a besoin de connaître la durée pendant laquelle on doit garder les résultats dans le cache.

On veut pouvoir préciser ce paramètre, appelons le `cache_timeout`, pour chaque fonction ; par exemple on voudrait écrire quelque chose comme

```
@cached_expire(600)
def resolve_host(hostname):
    ...

@cached_expire(3600*24)
def network_neighbours(hostname):
    ...
```

Ceci est possible également avec les décorateurs, avec cette syntaxe précisément. Le modèle qu'il faut avoir à l'esprit pour bien comprendre le code qui suit est le suivant et se base sur deux objets :

- le premier objet, `cached_expire`, est ce qu'on appelle une *factory* à décorateurs, c'est-à-dire que l'interpréteur va d'abord appeler `cached_expire(600)` qui doit retourner un décorateur ;
- le deuxième objet est ce décorateur retourné par `cached_expire(600)` qui lui même doit

se comporter comme les décorateurs sans argument que l'on a vus jusqu'ici.

Pour faire court, cela signifie que l'interpréteur fera

```
> resolve_host = (caching_expire(600))(resolve_host)
```

Ou encore si vous préférez

```
> caching = caching_expire(600)
| resolve_host = caching(resolve_host)
```

Ce qui nous mène au code suivant :

```
> import time

# comme pour caching, on est limité ici et on ne peut pas
# supporter les appels à la **kwargs, voir plus haut
# la discussion sur l'implémentation de caching

# caching_expire est une factory à décorateur
def caching_expire(timeout):

    # caching_expire va retourner un décorateur sans argument
    # c'est à dire un objet qui se comporte
    # comme notre tout premier 'caching'
    def caching(a_decorer):
        # à partir d'ici on fait un peu comme dans
        # la première version de caching
        def decoree(*args):
            try:
                # sauf que disons qu'on met dans le cache un tuple
                # (valeur, timestamp)
                valeur, timestamp = decoree.cache[args]
                # et là on peut accéder à timeout
                # parce que la liaison en python est lexicale
                if (time.time()-timestamp) <= timeout:
                    return valeur
            else:
                # on fait comme si on ne connaissait pas,
                raise KeyError

        # si les arguments ne sont pas hashables,
        # par exemple une liste, on ne peut pas cacher
        # et on reçoit TypeError
        except TypeError:
            return a_decorer(*args)

        # les arguments sont hashables mais on
        # n'a pas encore calculé cette valeur
        except KeyError:
            result = a_decorer(*args)
            decoree.cache[args] = (result, time.time())
            return result
```

```

        decoree.cache = {}
        return decoree
# le retour de caching_expire, c'est caching
return caching

@ caching_expire(0.5)
def fibo_cache_expire(n):
    return n if n<=1 else fibo_cache_expire(n-2)+fibo_cache_expire(n-1)

fibo_cache_expire(300)

fibo_cache_expire.cache[(200,)]

```

Remarquez la clôture

Pour conclure sur cet exemple, vous remarquez que dans le code de `decoree` on accède à la variable `timeout`. Ça peut paraître un peu étonnant, si vous pensez que `decoree` est appelée **bien après** que la fonction `caching_expire` ait fini son travail. En effet, `caching_expire` est évaluée **une fois** juste après **la définition** de `fibo_cache`. Et donc on pourrait penser que la valeur de `timeout` ne serait plus disponible dans le contexte de `decoree`.

Pour comprendre ce qui se passe, il faut se souvenir que python est un langage à liaison lexicale. Cela signifie que la **résolution** de la variable `timeout` se fait au moment de la compilation (de la production du byte-code), et non au moment où est appelé `decoree`.

Ce type de construction s'appelle une **clôture**

(http://fr.wikipedia.org/wiki/Fermeture_%28informatique%29), en référence au lambda calcul : on parle de terme clos lorsqu'il n'y a plus de référence non résolue dans une expression. C'est une technique de programmation très répandue notamment dans les applications réactives, où on programme beaucoup avec des *callbacks* ; par exemple il est presque impossible de programmer en JavaScript sans écrire une clôture.

On peut chaîner les décorateurs

Pour revenir à notre sujet, signalons enfin que l'on peut aussi "chaîner les décorateurs" ; imaginons par exemple qu'on dispose d'un décorateur `add_field` qui ajoute dans une classe un *getter* et un *setter* basés sur un nom d'attribut.

C'est-à-dire que

```

@add_field('name')
class Foo:
    pass

```

donnerait pour `Foo` une classe qui dispose des méthodes `get_name` et `set_name` (exercice pour les courageux: écrire `add_field`).

Alors la syntaxe des décorateurs vous permet de faire quelque chose comme :

```
@add_field('name')
@add_field('address')
class Foo:
    pass
```

Ce qui revient à faire :

```
class Foo: pass
Foo = (add_field('address'))(Foo)
Foo = (add_field('name'))(Foo)
```

Discussion

Dans la pratique, écrire un décorateur est un exercice assez délicat. Le vrai problème est bien souvent la création d'objets supplémentaires : on n'appelle plus la fonction de départ mais un wrapper autour de la fonction de départ.

Ceci a tout un tas de conséquences, et le lecteur attentif aura par exemple remarqué :

- que dans l'état du code de `singleton`, bien que l'on ait correctement mis à jour `__doc__` et `__name__` sur la classe décorée, `help(Spam)` ne renvoie pas le texte attendu, il semble que `help` sur une instance de classe ne se comporte pas exactement comme attendu ;
- que si on essaie de combiner les décorateurs `NbAppels` et `caching` sur une
 - encore nouvelle – version de `fibonacci`, le code obtenu ne converge pas ; en fait les technique que nous avons utilisées dans les deux cas ne sont pas compatibles entre elles.

De manière plus générale, il y a des gens pour trouver des défauts à ce système de décorateurs ; je vous renvoie notamment à ce blog (<http://blog.dscpl.com.au/2014/01/how-you-implemented-your-python.html>) qui, pour résumer, insiste sur le fait que les objets décorés n'ont **pas exactement** les mêmes propriétés que les objets originaux. L'auteur y explique que lorsqu'on fait de l'introspection profonde – c'est-à-dire lorsqu'on écrit du code qui "fouille" dans les objets qui représentent le code lui-même – les objets décorés ont parfois du mal à se *faire passer* pour les objets qu'ils remplacent.

À chacun de voir les avantages et les inconvénients de cette technique. C'est là encore beaucoup une question de goût. Dans certains cas simples, comme par exemple pour `NbAppels`, la décoration revient à simplement ajouter du code avant et après l'appel à la fonction à décorer. Et dans ce cas, vous remarquerez qu'on peut aussi faire le même genre de choses avec un *context manager* (je laisse ça en exercice aux étudiants intéressés).

Ce qui est clair toutefois est que la technique des décorateurs est quelque chose qui peut être très utile, mais dont il ne faut pas abuser. En particulier de notre point de vue, la possibilité de combiner les décorateurs, si elle existe bien dans le langage d'un point de vue syntaxique, est dans la pratique à utiliser avec la plus extrême prudence.

Pour en savoir plus

Maintenant que vous savez presque tout sur les décorateurs, vous pouvez retourner lire ce recueil de décorateurs (<https://wiki.python.org/moin/PythonDecoratorLibrary>) mais plus en détails.