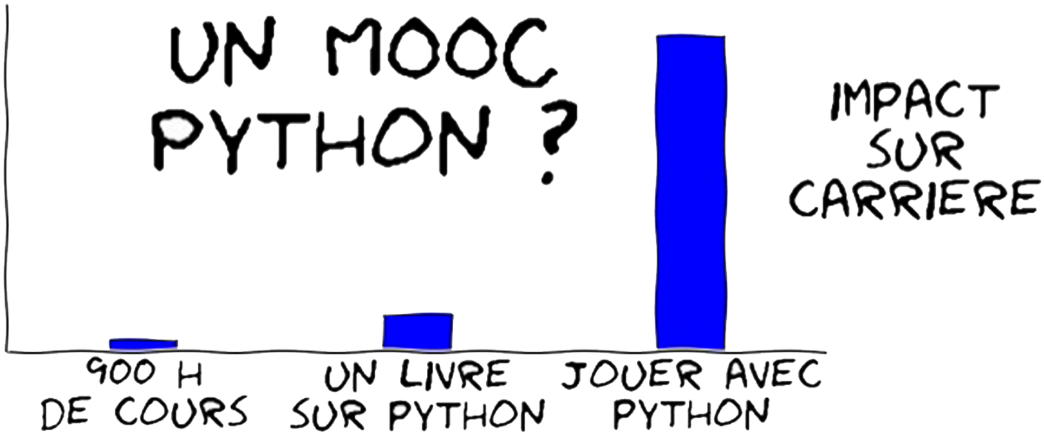
python™ 3

Des fondamentaux au concepts avancés du langage

Session 2 - 17 septembre 2018

Thierry Parmentelat Arnaud Legout



https ://www.fun-mooc.fr

Table des matières

Chapitre 9

Sujets avancés

9.1 w9-s2-c1-decorateurs

Décorateurs

9.1.1 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 qui propose du code (à titre indicatif, car rien de ceci ne fait partie de la bibliothèque standard) pour des thèmes qui sont propices à la décoration de code.

Nous allons voir en détail quelques-uns de ces exemples.

Un décorateur implémenté comme une classe

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

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

[2]:
```
# nous utilisons ici une implémentation en log(n)
# de la fonction de fibonacci

@NbAppels
def fibo_aux(n):
    "Fibonacci en log(n)"
    if n < 1:
        return 0, 1
```

```
    u, v = fibo_aux(n//2)
    u, v = u * (2 * v - u), u*u + v*v
    if n % 2 == 1:
        return v, u + v
    else:
        return u, v

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

[3]:
```
# pour se convaincre que nous sommes bien en log2(n)
from math import log
```

[4]:
```
n1 = 100

log(n1)/log(2)
```

[4]: 6.643856189774725

[5]:
```
fibo_log(n1)
```

```
1-ème appel à fibo_aux
2-ème appel à fibo_aux
3-ème appel à fibo_aux
4-ème appel à fibo_aux
5-ème appel à fibo_aux
6-ème appel à fibo_aux
7-ème appel à fibo_aux
8-ème appel à fibo_aux
```

[5]: 354224848179261915075

[6]:
```
# on multiplie par 2**4 = 16,
# donc on doit voir 4 appels de plus
n2 = 1600

log(n2)/log(2)
```

[6]: 10.643856189774725

[7]:
```
fibo_log(n2)
```

```
9-ème appel à fibo_aux
10-ème appel à fibo_aux
11-ème appel à fibo_aux
12-ème appel à fibo_aux
13-ème appel à fibo_aux
14-ème appel à fibo_aux
15-ème appel à fibo_aux
16-ème appel à fibo_aux
17-ème appel à fibo_aux
18-ème appel à fibo_aux
19-ème appel à fibo_aux
20-ème appel à fibo_aux
```

[7]:
```
1073345148918961110312160904303871047716692524192564541342409937035560545685216973603399187601476280834086584844747617342611516217281889032383713813678295186505453841749403522978597100258793263890231141601890415617026935472046089636355816812900423113841522520473858255072079106158146393409272610745834929857729298437527621023258243807 5
```

memoize implémenté comme une fonction

Ici nous allons implémenter **memoize**, 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 **memoize** 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

[8]:
```
# une première implémentation de memoize

# un décorateur de fonction
# implémenté comme une fonction
def memoize(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 s'ils contiennent 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:
            # on fait vraiment le calcul
            result = a_decorer(*args)
            # on le range dans le cache
            decoree.cache[args] = result
            # on le retourne
            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 :

[9]:
```
# créer une fonction décorée
@memoize
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 :

[10]:
```
fibo_cache(300)
```

[10]:
```
222232244629420445529739893461909967206666939096499764990979600
```

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 :

[11]:
```
len(fibo_cache.cache)
```

[11]:
```
301
```

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 = memoize(fibo_cache)
```

Donc `memoize` 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`, sous la forme d'un attribut`cache`, un dictionnaire 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 façon 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 décorée 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 :

[12]:
```
help(fibo_cache)
```

```
Help on function decoree in module __main__:

decoree(*args)
```

Et 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 `functools.wraps`, qui fait tout le travail nécessaire. Ce qui nous donne une deuxième version de ce décorateur, avec deux lignes supplémentaires signalées par des `+++` :

[13]:
```
# une deuxième implémentation de memoize, avec la doc

import functools                                    # +++

# un décorateur de fonction
# implémenté comme une fonction
def memoize(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:
            # on fait vraiment le calcul
            result = a_decorer(*args)
            # on le range dans le cache
            decoree.cache[args] = result
            # on le retourne
            return result
    # on initialise l'attribut 'cache'
    decoree.cache = {}
    return decoree
```

[14]:
```
# créer une fonction décorée
@memoize
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 :

[15]:
```
help(fibo_cache2)
```

```
Help on function fibo_cache2 in module __main__:

fibo_cache2(n)
    Un fibonacci hyper-lent (exponentiel) se transforme
    en temps linéaire une fois que les résultats sont cachés
```

On peut décorer les classes aussi

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

Pour ne pas alourdir le complément, et aussi parce que le mécanisme de métaclasse offre une autre alternative qui est souvent plus pertinente, nous ne donnons pas d'exemple ici, cela vous est laissé à titre d'exercice si vous êtes intéressé.

Un décorateur peut lui-même avoir des arguments

Reprenons l'exemple de `memoize`, 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 :

```
@memoize_expire(600)
def resolve_host(hostname):
    …

@memoize_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, `memoize_expire`, est ce qu'on appelle une factory à décorateurs, c'est-à-dire que l'interpréteur va d'abord appeler `memoize_expire(600)` qui doit retourner un décorateur ;
— le deuxième objet est ce décorateur retourné par `memoize_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 = (memoize_expire(600))(resolve_host)
```

Ou encore si vous préférez :

```
memoize = memoize_expire(600)
resolve_host = memoize(resolve_host)
```

Ce qui nous mène au code suivant :

[16]:
```
import time

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

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

    # memoize_expire va retourner un décorateur sans argument
    # c'est-à-dire un objet qui se comporte
    # comme notre tout premier `memoize`
    def memoize(a_decorer):
        # à partir d'ici on fait un peu comme dans
        # la première version de memoize
        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 memoize_expire, c'est memoize
    return memoize
```

[17]:
```
@memoize_expire(0.5)
def fibo_cache_expire(n):
    return n if n<=1 else fibo_cache_expire(n-2)+fibo_cache_expire(n-1)
```

[18]:
```
fibo_cache_expire(300)
```

[18]:
```
222232244629420445529739893461909967206666939096499764990979600
```

[19]:
```
fibo_cache_expire.cache[(200,)]
```

[19]:
```
(28057117299251014003761193241303867718952 5, 1689089437.158507)
```

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 `memoize_expire` a fini son travail. En effet, `memoize_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, 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 `memoize` sur une - encore nouvelle - version de fibonacci, le code obtenu ne converge pas ; en fait les techniques 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 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 mais plus en détails.