Les décorateurs en Python

Décorateurs

Exercice 1. les fonctions en python sont des objets

Pour comprendre les décorateurs, il faut d'abord comprendre que les fonctions sont des objets en Python. Cela a d'importantes conséquences :
Examiner le code ci-dessous et enregistrer le sans les commentaires dans un fichier nommé `exo1.py` et l'exécuter

```
1  #!/usr/bin/env python3
2
3  def crier(mot="oui"):
4      return mot.capitalize() + "!"
5
6  print(type(crier))
7  # output :<class 'function'>
8
9
10 print(crier())
11 # output : 'Oui!'
12
13 # Puisque les fonctions sont des objets,
14 # on peut les assigner à des variables
15
16 hurler = crier
17
18 # Notez que l'on n'utilise pas les parenthèses :
19 # la fonction n'est pas appelée. Ici nous mettons la
       fonction "crier"
20 # dans la variable "hurler" afin de pouvoir appeler "crier"
       avec "hurler"
21
22 print(hurler())
23 # output : 'Oui!'
24
25 # Et vous pouvez même supprimer l'ancien nom "crier",
26 # la fonction restera accessible avec "hurler"
27
```

```
28 del (crier)
29 try:
30     print(crier())
31 except NameError as e:
32     print(e)
33     #output: "name 'crier' is not defined"
34
35 print(hurler())
36 # output: 'Oui!'
```

Exercice 2. fonction contenant des fonctions et fonction ayant pour arguments des fonctions

Une autre propriété intéressante des fonctions en Python est qu'on peut les définir à l'intérieur d'une autre fonction. On peut également passer à une fonction une ou plusieurs fonctions comme arguments. Testez le programme ci-dessous :

```
 1 #!/usr/bin/env python3
 2
 3 def crier(mot="oui"):
 4     return mot.capitalize() + "!"
 5
 6 def creerParler(type="crier"):
 7
 8     # On fabrique 2 fonctions à la volée
 9     def crier(mot="oui"):
10         return mot.capitalize() + "!"
11
12     def chuchoter(mot="oui") :
13         return mot.lower() + "...";
14
15     # Puis on retourne l'une ou l'autre
16     if type == "crier":
17         # on n'utilise pas "()", on n'appelle pas la
               fonction
18         # on retourne l'objet fonction
19         return crier
20     else:
21         return chuchoter
22
23 # On peut ensuite obtenir la fonction et l'assigner à une
      variable:
```

```
parler = creerParler()

# "parler" est une variable qui contient la fonction "crier":
print(parler)
#output : <function crier at 0xb7ea817c>

# On peut appeler "crier" depuis "parler":
print(parler())
#ouput : Oui!

# on peut même créer et appeler la fonction en une seule fois:
print(creerParler("chuchoter")())
#output : oui...

#Mais ce n'est pas fini. Si on peut retourner une fonction, on peut aussi en passer une en argument ...

def faireQuelqueChoseAvant(fonction):
    print("Je fais quelque chose avant d'appeler la fonction")
    print(fonction())

faireQuelqueChoseAvant(crier)
#output:
#Je fais quelque chose avant d'appeler la fonction
#Oui!
```

A présent, vous avez toutes les cartes en main pour comprendre les décorateurs. En effet, les décorateurs sont des wrappers, c'est à dire qu'ils permettent d'exécuter du code avant et après la fonction qu'ils décorent, sans modifier la fonction elle-même.

Exercice 3. créons notre premier décorateur sans utiliser pour l'instant l'implémentation qu'en fait python

A nouveau examiner le code ci-dessous et l'exécuter :

```
#!/usr/bin/env python3
# Un décorateur est une fonction qui attend une autre fonction en paramètre
```

```
3  def decorateur_tout_neuf(fonction_a_decorer):
4
5      # En interne, le décorateur définit une fonction à la
           volée: le wrapper.
6      # Le wrapper va enrober la fonction originale de telle
           sorte qu'il
7      # puisse exécuter du code avant et après celle-ci
8      def wrapper_autour_de_la_fonction_originale():
9
10         # Mettre ici le code que l'on souhaite exécuter
               AVANT que la
11         # fonction s'exécute
12         print("Avant que la fonction ne s'exécute")
13
14         # Apperler la fonction (en utilisant donc les
               parenthèses)
15         fonction_a_decorer()
16
17         # Mettre ici le code que l'on souhaite exécuter
               APRES que la
18         # fonction s'exécute
19         print("Après que la fonction se soit exécutée")
20
21     # Arrivé ici, la "fonction_a_decorer" n'a JAMAIS ETE
           EXECUTEE
22     # On retourne le wrapper que l'on vient de créer.
23     # Le wrapper contient la fonction originale et le code à
           exécuter
24     # avant et après, prêt à être utilisé.
25     return wrapper_autour_de_la_fonction_originale
26
27 # Maintenant imaginez une fonction que l'on ne souhaite pas
       modifier.
28 def une_fonction_intouchable():
29     print("Je suis une fonction intouchable, on ne me
           modifie pas !")
30
31 une_fonction_intouchable()
32 #output: Je suis une fonction intouchable, on ne me modifie
       pas !
33
34 # On peut malgré tout étendre son comportement
35 # Il suffit de la passer au décorateur, qui va alors l'
```

```
      enrober dans
36 # le code que l'on souhaite, pour ensuite retourner une
      nouvelle fonction
37
38 une_fonction_intouchable_decoree = decorateur_tout_neuf(
      une_fonction_intouchable)
39 une_fonction_intouchable_decoree()
40 #output:
41 #Avant que la fonction ne s'exécute
42 #Je suis une fonction intouchable, on ne me modifie pas !
43 #Après que la fonction se soit exécutée
44
45 #Puisqu'on y est, autant faire en sorte qu'à chaque fois qu'
      on appelle une_fonction_intouchable, c'est
      une_fonction_intouchable_decoree qui est appelée à la
      place. C'est facile, il suffit d'écraser la fonction
      originale par celle retournée par le décorateur :
46
47 une_fonction_intouchable = decorateur_tout_neuf(
      une_fonction_intouchable)
48 une_fonction_intouchable()
49 #output:
50 #Avant que la fonction ne s'exécute
51 #Je suis une fonction intouchable, on ne me modifie pas !
52 #Après que la fonction se soit exécutée
53 # et maintenant en syntaxe python
54 @decorateur_tout_neuf
55 def fonction_intouchable():
56     print("Me touche pas !")
57
58 fonction_intouchable()
59 #output:
60 #Avant que la fonction ne s'exécute
61 #Me touche pas !
62 #Après que la fonction se soit exécutée
```

Vous remarquerez que :

`@decorateur_tout_neuf est juste un raccourci pour`

`fonction_intouchable = decorateur_tout_neuf(fonction_intouchable)`

Les décorateurs sont juste une variante de python du classique design pattern “décorateur”.

Exercice 4. Et bien sûr, on peut cumuler les décorateurs :

```python
def pain(func):
    def wrapper():
        print("</''''''\>")
        func()
        print("<\______/>")
    return wrapper
def ingredients(func):
    def wrapper():
        print("#tomates#")
        func()
        print("~salade~")
    return wrapper

def sandwich(food="--jambon--"):
    print(food)

sandwich()
#output: --jambon--
sandwich = pain(ingredients(sandwich))
sandwich()
#output:
#</''''''\>
# #tomates#
# --jambon--
# ~salade~
#<\______/>
# en syntaxe python cela donne
@pain
@ingredients
def sandwich(nourriture="--jambon--"):
    print(nourriture)
 sandwich()
#output:
#</''''''\>
# #tomates#
# --jambon--
# ~salade~
#<\______/>

#attention l'ordre des décorateurs est important

```

```
@ingredients
@pain
def sandwich_zarb(nourriture="--jambon--"):
    print(nourriture)
sandwich_zarb()
#output:
##tomates#
#</''''''\>
# --jambon--
#<\______/>
# ~salade~
```

Exercice 5. Ecrire deux décorateurs de fonctions

Le premier nommé `timer` calculera le temps d'exécution de la fonction décorée. Le second nommé `counter` comptera le nombre de fois où la fonction a été appelée.

Nb : Vous pourrez faire appel à l'introspection avec la méthode `__name__`dans vos wrappers et vous utiliserez `functools` et le décorateur `wraps` pour que cette méthode affiche effectivement le nom de la fonction décorée et pas celle du wrapper.

Exercice 6. Decorateur lru_cache

On peut utiliser le décorateur lru_cache pour procéder à une *memoization* de valeurs d'une fonction. Par exemple pour la fonction *fibonacci*

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fibo(n):
    if n < 2:
        return n
    return fibo(n-1) + fibo(n-2)
```

Exercice 7. Packing, unpacking et autres **kwargs en Python

Voici ce qu'on appelle Packing et Unpacking en Python3 :

```
# Packing
a = 3
b = 4
c = 5
```

```
5 l=[a,b,c]
6 # Unpacking
7 d,e,f = l
```

Exercice 8. Les décorateurs avec arguments

Pour implémenter un décorateur dynamique capable d' accepter des arguments (ou options), nous devons ajouter un niveau supplémentaire : un décorateur qui prend pour arguments les arguments du décorateur de la fonction à décorer. Pour un décorateur simple, nous avons deux niveaux de fonctions (le décorateur et le wrapper). Pour un décorateur avec arguments, nous en avons trois (le décorateur du décorateur, le décorateur et le wrapper). Voici un exemple que vous devez tester :

```
1  def decorate(arg1, arg2, arg3):
2      print('Je suis dans la fonction "decorate".')
3      def decorated(func):
4          print ('Je suis dans la fonction "decorated".')
5          def wrapper(*args, **kwargs):
6              print('Je suis dans la fonction "wrapper".')
7              print ("Les arguments du décorateurs sont : %s,
                   %s, %s." % (arg1, arg2, arg3))
8              print ("Pré-traitement.")
9              print ("Exécution de la fonction %s." % func.
                   __name__)
10             response = func(*args, **kwargs)
11             print ("Post-traitement.")
12             return response
13         return wrapper
14     return decorated
15
16 @decorate("Arg 1", "Arg 2", "Arg 3")
17 def foobar():
18     print("Je suis foobar, je vous reçois 5 sur 5.")
19
20 foobar()
```

Exercice 9. Décorateurs de classe Il est possible de décorer une classe. Voici un exemple qui assure qu'une classe ne sera instanciée qu'une seule fois : c'est le décorateur singleton que vous devez également tester.

```
1 #!/usr/bin/python3
```

```
2
3  def singleton(classe_definie):
4    instances = {} # Dictionnaire de nos instances singletons
5    def get_instance(*args, **kwargs):
6      if classe_definie not in instances:
7        # On crée notre premier objet de classe_definie
8        instances[classe_definie] = classe_definie(*args, **
           kwargs)
9      return instances[classe_definie]
10   return get_instance
11
12 @singleton
13 class Test:
14   def __init__(self,val):
15     self.val=val
16
17 a=Test("truc")
18 b=Test("machin")
19 print (a.val,id(a))
20 print(b.val,id(b))
21 print ( a is b)
```

Exercice 10. le décorateur property, getter et setter des attributs d'une classe avec attributs privés

Examinez le code ci-dessous et testez le

```
1  #!/usr/bin/python3
2  class P:
3
4    """ une classe pour tester les décorateurs property et
       setter
5    d'une classe n'ayant qu'un seul attribut _x
6    """
7    def __init__(self,x):
8      self._x = x
9
10   @property
11   def x(self):
12     return self._x
13
14   @x.setter
```

```
15    def x(self, x):
16      if x < 0:
17        self._x = 0
18      elif x > 1000:
19        self._x = 1000
20      else:
21        self._x = x
22
23
24  class P2:
25    """ une classe pour tester la fonction property
26      cette fois on l'utilise directement sans passer par les
            décorateurs
27      d'une classe n'ayant qu'un seul attribut _x
28    """
29
30    def __init__(self,x):
31      self.setX(x)
32
33    def getX(self):
34      return self._x
35    def setX(self, x):
36      if x < 0:
37        self._x = 0
38      elif x > 1000:
39        self._x = 1000
40      else:
41        self._x = x
42    x = property(getX, setX)
43
44
45  print (P.__doc__)
46  p1=P(2000)
47  print(p1.x)
48  p1.x=500
49  print(p1.x)
50  p1.x=-12
51  print(p1.x)
52
53  print (P2.__doc__)
54  p1=P2(2000)
55  print(p1.x)
56  p1.x=500
```

```
57 print(p1.x)
58 p1.x=-12
59 print(p1.x)
60 # protection illusoire car il est possible d'accéder a l'
      attribut _x
61 print (p1.__dict__['_P2_x'])
62 print(p1._P2_x)
63 # cependant l'accès direct provoque un exception
      AttributeError
64 print(p1._x)
```