Flask - les décorateurs

Flask - les décorateurs

Ce document, qui n'est pas forcément nécessaire pour utiliser les décorateurs avec flask, explique la mise en œuvre de ce mécanisme dans le langage python.

I. Explication sur la mise en œuvre des décorateurs en python

D'après http://sametmax.com/comprendre-les-decorateurs-python-pas-a-pas-partie-1/

1. Les fonctions python sont des objets

- Soit le fonctionnement ligne par ligne :

```
def crier(mot="crier"):
    return mot.capitalize() + "!"

print(crier())
```

Affiche 'CRIER'

- Puisque les fonctions sont des objets, on peut les assigner à des variables.
- Nous mettons la fonction "crier" dans "hurler" afin de pouvoir appeler la fonction "crier" avec la fonction "hurler"
- Notez que l'on n'utilise pas les parenthèses :

```
hurler = crier
```

- Appelons la fonction Hurler

```
print(hurler())
```

L'affichage est 'Crier'

- Et vous pouvez même supprimer l'ancien nom "crier" :

```
del crier
```

- La fonction restera accessible avec "hurler"

```
print(hurler())
```

L'affichage est 'Crier'

- Mais inaccessible par la suite :

```
try:
    print(crier())
except NameError as e:
    print(e)
```

Affichage : "name 'crier' is not defined"

L'exemple complet est :

```
def crier(mot="crier"):
    return mot.capitalize() + "!"
```

```
print(crier())
# output : Crier!'

# Puisque les fonctions sont des objets,
# on peut les assigner à des variables

hurler = crier

# Notez que l'on n'utilise pas les parenthèses :
# la fonction n'est pas appelée. Ici nous mettons la fonction "crier"
# dans la variable "hurler" afin de pouvoir appeler "crier" avec "hurler"

print(hurler())
# output : Crier!'

# Et vous pouvez même supprimer l'ancien nom "crier",
# la fonction restera accessible avec "hurler"

del crier
print(hurler())
try:
    print(crier())
except NameError as e:
    print(e)
    #output: "name 'crier' is not defined"

print(hurler())
# output: Crier!'
```

2. Définition de fonction à l'intérieur … de fonctions

Par exemple :

- On peut définir une fonction à la volée la fonction "chuchoter" dans "parler" et l'utiliser immédiatement dans la fonction « parler »

```
def parler():
    def chuchoter(mot="PARLER"):
        return mot.lower()+"...";

    print(chuchoter())
```

- On appelle "parler", qui définit "chuchoter" A CHAQUE APPEL, puis "chuchoter" est appelé à l'intérieur de " parler"

```
parler()
```

L'affichage est parler

- Mais "chuchoter" N'EXISTE PAS en dehors de "parler"

```
try:
```

```
    print(chuchoter())
except NameError as erreur:
    print(erreur)
```

L'affichage : "name 'chuchoter' is not defined"

L'exemple complet est :

```
def parler():

    # On peut définir une fonction à la volée dans "parler" ...
    def chuchoter(mot="PARLER"):
        return mot.lower()+"...";

    # ... et l'utiliser immédiatement !

    print(chuchoter())

# On appelle "parler", qui définit "chuchoter" A CHAQUE APPEL,
# puis "chuchoter" est appelé à l'intérieur de "parler"

parler()
# output:
# "yes..."

# Mais "chuchoter" N'EXISTE PAS en dehors de "parler"

try:
    print(chuchoter())
except NameError as erreur:
    print(erreur)
    #output : "name 'chuchoter' is not defined"
```

3. Définition de plusieurs fonctions dans un autre fonction

- On fabrique 2 fonctions à la volée dans une seule fonction

```
def creerParler(type="FonctionCrier"):

    def crier(mot="crier"):
        return mot.capitalize() + "!"

    def chuchoter(mot="chuchoter") :
        return mot.lower() + "...";

    if type == "FonctionCrier":
        # on utilise pas "()", on n'appelle pas la fonction
        # on retourne l'objet fonction
        return crier
```

```
    else:
        return chuchoter
```

- "parler" est une variable qui contient la fonction "crier":

```
parler = creerParler()
```

Il n'y a aucun affichage

- Affichons-la :

```
print(parler)
```

L'affichage est <function crier at 0xb7ea817c>

Explication : on affiche le contenu de la variable *parler* qui est la fonction *creerParler* et qui se fait donc sans donner une valeur à l'argument type et cette variable vaut par défaut *FonctionCrier*. Ainsi c'est la fonction crier qui est retourné ou plutôt son adresse mémoire

- # On peut appeler "crier" depuis "parler" en ajoutant des parenthèses « () »

```
print(parler())
```

L'affichage : Crier!

- Il est possible de peut même créer et d'appeler la fonction en une seule fois mais cette fois avec une valeur pour l'argument *type*:

```
print(creerParler("chuchoter")())
```

L'affichage : chuchoter..

Le programme complet :

```
def creerParler(type="FonctionCrier"):

    # On fabrique 2 fonctions à la volée
    def crier(mot="crier"):
        return mot.capitalize() + "!"

    def chuchoter(mot="chuchoter") :
        return mot.lower() + "...";

    # Puis on retourne l'une ou l'autre
    if type == "FonctionCrier":
        # on utilise pas "()", on n'appelle pas la fonction
        # on retourne l'objet fonction
        return crier
    else:
        return chuchoter

# Comment ce truc bizarre s'utilise ?

# 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 : Crier!

# Et si on se sent chaud, on peut même créer et appeler la
# fonction en une seule fois:
print(creerParler("chuchoter")())
#output : chuchoter..
```

4. Passer une fonction en argument

Tout comme il est possible de renvoyer une fonction, il est également possible d'en passer une en argument

```
def crier(mot="crier"):
    return mot.capitalize() + "!"

def dire(uneFonction):
    print("Avant l'appel de la fonction")
    print(uneFonction()) #remarquez les ()
    print("Après l'appel de la fonction")

dire(crier) # pas besoin de print c'est dans la fonction
```

Ou alors avec un print :

```
def crier(mot="crier"):
    return mot.capitalize() + "!"

def dire(uneFonction):
    return ("Avant l'appel de la fonction\n" +
    uneFonction() + "\n" +
    "Après l'appel de la fonction")

print(dire(crier))
```

5. Des décorateurs artisanaux

On va utiliser des *wrapper* qui font entourer le code d'une autre fonction.

a) Fonction « décorateur »

- Un décorateur est une fonction qui attend une autre fonction en paramètre :

```
def decorateur(fonction_a_decorer):
```

- En interne, le décorateur définit une fonction à la volée: le wrapper. Le wrapper va enrober la fonction origina le de telle sorte qu'il puisse exécuter du code avant et après celle-ci

```
    def wrapper_autour_de_la_fonction_originale():
```

- Le code à exécuter AVANT que la fonction s'exécute

```
        print("Avant que la fonction ne s'exécute")
```

- Appel de la fonction (en utilisant donc les parenthèses)

```
        fonction_a_decorer()
```

- Le code APRES l'exécution de la fonction

```
        print("Après que la fonction se soit exécutée")
```

- Arrivé ici, la "fonction_a_decorer" n'a JAMAIS ETE EXECUTEE.
  On retourne le wrapper que l'on vient de créer. Le wrapper contient la fonction originale et le code à exécuter avant et après, prêt à être utilisé.

```
    return wrapper_autour_de_la_fonction_originale
```

Le code de la fonction décorateur :

```
def decorateur(fonction_a_decorer):
     def wrapper_autour_de_la_fonction_originale():
        print("Avant que la fonction ne s'exécute")
        fonction_a_decorer()
        print("Après que la fonction se soit exécutée")
    return wrapper_autour_de_la_fonction_originale
```

b) La fonction à décorer

- Maintenant imaginez une fonction que l'on ne souhaite pas modifier.

```
def une_fonction():
    print("Je suis une_fonction !")

une_fonction()
```

- L'affichage : Je suis une_fonction !

c) Etendre son comportement :

- On peut malgré tout étendre son comportement,
  il suffit de la passer au décorateur, qui va alors l'enrober dans le code que l'on souhaite, pour ensuite retourner une nouvelle fonction

```
 une_fonction_decoree = decorateur(une_fonction)
```

- Appel de la fonction

```
une_fonction_decoree()
```

- L'affichage est :

Avant que la fonction ne s'exécute
Je suis une_fonction !
Après que la fonction se soit exécutée

d) Le code complet :

```
# Un décorateur est une fonction qui attend une autre fonction en paramètre
def decorateur(fonction_a_decorer):

    # En interne, le décorateur définit une fonction à la volée: le wrapper.
    # Le wrapper va enrober la fonction originale de telle sorte qu'il
    # puisse exécuter du code avant et après celle-ci
    def wrapper_autour_de_la_fonction_originale():

        # Mettre ici le code que l'on souhaite exécuter AVANT que la
        # fonction s'exécute
        print("Avant que la fonction ne s'exécute")
```

```
        # Apperler la fonction (en utilisant donc les parenthèses)
        fonction_a_decorer()

        # Mettre ici le code que l'on souhaite exécuter APRES que la
        # fonction s'exécute
        print("Après que la fonction se soit exécutée")

    # Arrivé ici, la "fonction_a_decorer" n'a JAMAIS ETE EXECUTEE
    # On retourne le wrapper que l'on vient de créer.
    # Le wrapper contient la fonction originale et le code à exécuter
    # avant et après, prêt à être utilisé.
    return wrapper_autour_de_la_fonction_originale

# Maintenant imaginez une fonction que l'on ne souhaite pas modifier.
def une_fonction():
    print("Je suis une_fonction !")

une_fonction()
#output: Je suis une_fonction !

# On peut malgré tout étendre son comportement
# Il suffit de la passer au décorateur, qui va alors l'enrober dans
# le code que l'on souhaite, pour ensuite retourner une nouvelle fonction

une_fonction_decoree = decorateur(une_fonction)

#Appel de la fonction
une_fonction_decoree()

#output:
#Avant que la fonction ne s'exécute
#Je suis une_fonction !
#Après que la fonction se soit exécutée
```

6. En se passant d'un décorateur ?

Au lieu d'appeler une fonction « relais » comme celle-ci :

```
une_fonction_decoree()
```

Je voudrais directement faire un appel de ce type :

```
une_fonction()
```

La solution est d'écraser la fonction originale comme ceci dans le code :

```
une_fonction = decorateur(une_fonction)
```

Le code complet est :

```
def decorateur(fonction_a_decorer):
    def wrapper_autour_de_la_fonction_originale():
        print("Avant que la fonction ne s'exécute")
```

```
        fonction_a_decorer()
        print("Après que la fonction se soit exécutée")
    return wrapper_autour_de_la_fonction_originale

def une_fonction():
    print("Je suis une_fonction !")

une_fonction = decorateur(une_fonction)
une_fonction()
```

Et l'affichage est :

```
Avant que la fonction ne s'exécute
Je suis une_fonction !
Après que la fonction se soit exécutée
```

II. Les décorateurs en python

1. Comment ont été implémenté les décorateurs en Python ?

Avec les exemples précédents :

Et bien tout simplement :

Sans mécanisme :	Avec les décorateurs :

`une_fonction = decorateur(une_fonction)`	`@decorateur`
	`def une_fonction():`
	`    print("Je suis une_fonction !")`

Le code complet est donc :

```
def decorateur(fonction_a_decorer):
    def wrapper_autour_de_la_fonction_originale():
        print("Avant que la fonction ne s'exécute")
        fonction_a_decorer()
        print("Après que la fonction se soit exécutée")
    return wrapper_autour_de_la_fonction_originale


@decorateur
def une_fonction():
    print("Je suis une_fonction !")

une_fonction()
```

Et l'affichage est exactement le même que sans les décorateurs commençant par @ :

```
Avant que la fonction ne s'exécute
Je suis une_fonction !
Après que la fonction se soit exécutée
```

2. Cumuler les décorateurs :

Il est également possible de les cumuler :

```
def decorateur_1(fonction_a_decorer):
    def wrapper_autour_de_la_fonction_originale():
        print("--- Début Décorateur 1---")
        fonction_a_decorer()
        print("--- Fin Décorateur 1---")
    return wrapper_autour_de_la_fonction_originale

def decorateur_2(fonction_a_decorer):
    def wrapper_autour_de_la_fonction_originale():
        print("--- Début Décorateur 2---")
        fonction_a_decorer()
        print("--- Fin Décorateur 2---")
    return wrapper_autour_de_la_fonction_originale

@decorateur_1 # L'ordre est important
@decorateur_2
def une_fonction():
    print("Je suis une_fonction !")

une_fonction()
```

Et l'affichage est :
--- Début Décorateur 1---
--- Début Décorateur 2---
Je suis une_fonction !
--- Fin Décorateur 2---
--- Fin Décorateur 1---

3. Passer des arguments :

Si on fait cela :

```
def decorateur(fonction_a_decorer):
    def wrapper_autour_de_la_fonction_originale():
        print("--- Début Décorateur 1---")
        fonction_a_decorer()
        print("--- Fin Décorateur 1---")
    return wrapper_autour_de_la_fonction_originale

@decorateur
def bonjour(nom1, nom2):
    print("Bonjour",nom1," et également à ",nom2)

bonjour("toto","tata")
```

L'affichage avec une erreur est :

```
Traceback (most recent call last):
  File "d:\Documents\Hubic\Programmation\Flask\Décorateur\fonctionsObjet.py", line 12, in <module>
```

```
   bonjour("toto","tata")
TypeError: wrapper_autour_de_la_fonction_originale() takes 0 positional arguments but 2 were given
PS D:\Documents\Hubic\Programmation\Flask\Décorateur>
```

Il manque des arguments à la fonction *wrapper :*

```
def wrapper_autour_de_la_fonction_originale(arg1,arg2):
```

Le code complet est :

```
def decorateur(fonction_a_decorer):
    def wrapper_autour_de_la_fonction_originale(arg1,arg2):
        print("--- Début Décorateur 1---")
        fonction_a_decorer(arg1,arg2)
        print("--- Fin Décorateur 1---")
    return wrapper_autour_de_la_fonction_originale

@decorateur
def bonjour(nom1, nom2):
    print("Bonjour",nom1,"et également à",nom2)

bonjour("toto","tata")
```