

La notion de décorateur en Python

Table des matières

I. Notion de base sur les décorateurs en python	3
A. Création de décorateurs en Python	4
B. Création de décorateurs multiples	4
II. Exercice : Quiz	6
III. Aller plus loin avec les décorateurs	7
A. Réutiliser les décorateurs	7
B. Passer des paramètres au décorateur	8
IV. Exercice : Quiz	9
V. Essentiel	11
VI. Auto-évaluation	12
A. Exercice	12
B. Test	12
Solutions des exercices	13

I. Notion de base sur les décorateurs en python

Durée : 1 h

Environnement de travail : PC

Contexte

En Python, le décorateur est une fonction utilisée pour modifier le comportement des fonctions ou des classes. Il est utile pour tester les performances des fonctions, en calculant sa durée d'exécution.

Une autre solution pour tester les performances des fonctions serait de modifier chacune des fonctions devant intégrer ce test. Or, ce n'est ni élégant ni pratique. En revanche, l'autre possibilité consiste à utiliser un décorateur qui est chargé d'exécuter la fonction en calculant son temps d'exécution. Pour indiquer à la fonction d'intégrer ce test, il suffit d'ajouter une ligne avant sa définition.

Cette application n'est qu'un exemple parmi d'autres, mais il résume l'intérêt du décorateur. Parfois, leur construction est plus complexe. Quand le décorateur prend en compte des arguments comme paramètres ou s'il contient des paramètres de la fonction, le code sera plus complexe.

Définition

S'il faut faire simple, un décorateur sert à ajouter des détails à des fonctions et classes afin de changer leur exécution sans pour autant faire de modifications majeures sur ces fonctions. Les fonctions en Python sont des citoyens de première classe. Cela signifie qu'elles supportent des opérations telles que le passage en argument, le retour d'une fonction, la modification et l'affectation à une variable. Il s'agit d'un concept fondamental à comprendre avant de se plonger dans la création de décorateurs Python. Le décorateur doit toutefois précéder la fonction à décorer.

Le principe de l'utilisation d'un décorateur est très simple. L'interpréteur de code utilisé compilera d'abord une fonction quelconque et appliquera le décorateur. Le décorateur ici prend la fonction et y ajoute les détails requis afin de changer le fonctionnement.

Exemple

```
1 @nom_decorateur
2 def ma_fonction():
3
4 # Le décorateur s'exécute à la définition de fonction et non pas à
5 # l'appel. Il prend en paramètre une fonction (celle qu'il modifie) et
6 # renvoie une fonction (qui peut être la même).
7
8 def mon_decorateur(fonction):
9     def fonction_modifiee():
10         print('Décorateur {0}'.format(fonction))
11         return fonction
12     return fonction_modifiee()
13
14 @mon_decorateur
15 def salut():
16     print('Salut')
```

A. Création de décorateurs en Python

On commence par créer le décorateur qui prend en paramètre la fonction qu'il modifie. Ici, il se contente d'afficher cette fonction puis de la renvoyer. Avant d'écrire la fonction, on marque la présence du décorateur avec la syntaxe `@mon_decorateur`. Cette ligne montre à Python que cette fonction doit être modifiée par le décorateur `@mon_decorateur`.

Créer un simple décorateur en Python

La structure du décorateur est simple, il prend un paramètre qui dans ce cas est la fonction à modifier (celle sous la ligne du `@`) et renvoie également la fonction. La fonction renvoyée remplace celle définie. Dans ce cas, il s'agit de la même. Vous pouvez demander à Python d'exécuter une autre fonction pour modifier son comportement.

```
1 # Décorateur
2 @decorateur
3 def fonction (...) :
4
5 # Sans décorateur
6 def fonction (...) :
7 fonction = decorateur ( fonction )
```

Ces deux morceaux de code font la même chose. Le second code sert juste à vous montrer la manière dont Python interprète le code lorsqu'il s'agit d'un ou de plusieurs décorateurs. Quand le « **salut** » est exécuté, il n'y a aucun changement, ce qui est normal puisque c'est la même fonction qui est renvoyée.

Le décorateur est appelé lors de la définition de la fonction. La fonction salut n'a pas été modifiée par le décorateur, elle est renvoyée telle quelle.

B. Création de décorateurs multiples

Maintenant que vous savez comment décorer une fonction, allons à un cas plus complexe. Il s'agira ici d'utiliser plusieurs décorateurs sur une même fonction et de veiller à appliquer selon un ordre précis.

Vous savez déjà utiliser un décorateur et comment appliquer sa syntaxe (`@decorateur`) pour décorer une fonction. Vous devez savoir aussi qu'en Python, on a la possibilité d'augmenter le nombre de décorateurs sur une même fonction. Le premier détail que vous devez garder avant d'ajouter plusieurs décorateurs est que l'ordre des décorateurs est important. Prenons l'exemple du code suivant :

```
1 @decorateur1
2 @decorateur2
3 def ma_fonction():
```

Le décorateur 2 s'exécutera avant le décorateur 1.

Exemple

Prenons l'exemple d'un processus pour mieux comprendre ce code. On souhaite exécuter deux actions sur une fonction. On reprend le code précédent en changeant quelques détails :

```
1 @decorA
2 @decorB
3 def execute_action():
```

Le code est interprété comme suit :

```
1 execute_action = decorateur1(decorateur2(execute_action))
```

Vous remarquez, qu'ici, le décorateur 2 a été appliqué avant le décorateur 1. C'est ainsi pour les décorateurs multiples.

Continuons maintenant avec un code plus complexe. On appliquera deux décorateurs à une même fonction pour changer son comportement.

Voici un code Python :

```
1 def decorateur1(func):
2     def inner():
3         a = func()
4         return a * a
5     return inner
6
7 def decorateur(func):
8     def inner():
9         a = func()
10        return 2 * a
11    return inner
12
13 @decorateur1
14 @decorateur
15 def num():
16     return 20
17
18 print(num())
```

Avec ce genre de code, on aura en sortie : « 1 600 ».

L'explication ici est bien simple. Le décorateur **@decorateur** mis en deuxième position transforme le nombre 20 (ici $20 \times 2 = 40$), ensuite le décorateur **@decorateur1** vient transformer le résultat du décorateur 2 (ici $40 \times 40 = 1\,600$). C'est l'exemple typique de l'application de deux décorateurs sur une fonction.

Voici ci-dessous un autre exemple de deux décorateurs avec deux autres fonctions :

```
1 def inverser_decor(function):
2
3     def fonction_decor():
4         faire_inverse = "".join(reversed(function()))
5         return faire_inverse
6
7     return fonction_decor
8
9 def maj_decor(function):
10
11     def uppercase_wrapper():
12         maj_lettres = function().upper()
13         return maj_lettres
14
15     return uppercase_wrapper
16
17 @maj_decor
18 @inverser_decor
19 def say_hi():
20     return 'salut David'
21
22 def main():
23     print(say_hi())
24
25 if __name__ == "__main__":
26     main()
```

La sortie devrait donner : « DIVAD TULAS ».

Avec ce code, on a inversé les lettres de l'expression de départ (salut David) mais on a aussi mis toutes les lettres en majuscule, ce qui a donné un résultat pareil.

Exercice : Quiz

[solution n°1 p.15]

Question 1

Quel symbole utilise-t-on pour spécifier à Python qu'on insère un décorateur ?

- ☐ #
- ☐ \$
- ☐ @
- ☐ &

Question 2

Une fonction avec des paramètres ne peut pas être décorée.

- ☐ Vrai
- ☐ Faux

Question 3

Quelle sera la sortie du code suivant ?

```
1 class A:
2     @staticmethod
3     def a(x):
4         print(x)
5 A.a(100)
```

- ☐ Erreur
- ☐ 100
- ☐ Pas de sortie

Question 4

Quelle sera la sortie du code suivant ?

```
1 def d(f):
2     def n(*args):
3         return '$' + str(f(*args))
4     return n
5 @d
6 def p(a, t):
7     return a + a*t
8 print(p(100,0))
```

- ☐ 100
- ☐ \$100
- ☐ \$0
- ☐ 0

Question 5

Quelle sera la sortie du code suivant ?

```

1 def decor_code(f):
2     def fonc_interne(*args, **kwargs):
3         fonc_interne.co += 1
4         return f(*args, **kwargs)
5     fonc_interne.co = 0
6     return fonc_interne
7 @decor_code
8 def fonct_result():
9     pass
10 if __name__ == '__main__':
11     fonct_result()
12     fonct_result()
13     print(fonct_result.co)

```

- ☐ 4
- ☐ 2
- ☐ 0
- ☐ 1

III. Aller plus loin avec les décorateurs

A. Réutiliser les décorateurs

Rappelez-vous qu'un décorateur n'est qu'une fonction Python normale. Tous les outils usuels pour une réutilisation aisée sont disponibles.

Déplaçons le décorateur vers son propre module qui peut être utilisé dans de nombreuses autres fonctions. Créez un fichier appelé « **decorators.py** » avec le contenu suivant :

```

1 def do_twice(func):
2     def wrapper_do_twice():
3         func()
4         func()
5     return wrapper_do_twice

```

Remarque

Remarquez que vous pouvez nommer votre fonction interne comme vous le souhaitez, et un nom générique comme `wrapper()` convient généralement.

Vous verrez beaucoup de décorateurs dans ce cours. Pour les séparer, nous nommerons la fonction interne avec le même nom que le décorateur mais avec un `wrapper_` préfixe.

Vous pouvez maintenant utiliser ce nouveau décorateur dans d'autres fichiers en faisant un import régulier :

```

1 from decorators import do_twice
2 @do_twice
3 def say_whee():
4     print("Whee!")

```

Lorsque vous exécutez cet exemple, vous devriez voir que l'original `say_whee()` est exécuté deux fois :

```

1 >>> say_whee()
2 Whee!
3 Whee!

```

Supposons que vous ayez du code qui vérifie si le nom de quelqu'un est Alice. Toutes les instructions de contrôle de flux se terminent par deux points et sont suivies d'un nouveau bloc de code (la clause). Cette clause if est le bloc qui imprime ('Bonjour, Alice').

B. Passer des paramètres au décorateur

Ici, le décorateur est chargé d'exécuter une fonction qui contrôle le temps d'exécution. Si la fonction met un temps supérieur à la durée définie dans le paramètre, une alerte s'affiche. Le décorateur au-dessus de la définition de la fonction ressemblera à ceci :

```
1 @controler_temps ( 2 . 5 )
```

Jusqu'à présent, les décorateurs n'utilisaient pas de parenthèses. La fonction de décorateur prend en paramètres ses paramètres, elle renverra ainsi un décorateur. On remplace la fonction définie au-dessus par la fonction que renvoie le décorateur. C'est le mécanisme du décorateur. Le décorateur attend les paramètres.

```
1 @decorateur (parametre)
2 def fonction (...):
```

Remarque

On utilise comme décorateur une fonction dont les arguments sont les paramètres du décorateur, le temps prédéfini et le renvoi d'un nouveau décorateur à l'issue de l'opération.

Continuons avec un cas plus simple et facile à comprendre :

```
1 def decorateur(fonction):
2     def autre_fonction(*args, **kwargs):
3         print('Exécute une action')
4         fonction(*args, **kwargs)
5         print("Exécute un une action complémentaire")
6     return autre_fonction
```

Les arguments * args et ** kwargs interviennent ici pour effectuer un changement sur la fonction d'origine. C'est le mode opératoire pour passer des arguments à une fonction.

Si nous voulons que la nouvelle fonction (autre_fonction) renvoie la même valeur que l'ancienne (fonction), il suffit de l'appliquer comme suit :

```
1 def decorateur(fonction):
2     def autre_fonction(*args, **kwargs):
3         print('Exécute une action.')
4         ma_valeur = fonction(*args, **kwargs)
5         print("Exécute un une action complémentaire.")
6         return ma_valeur
7     return autre_fonction
```

Avec le code ci-dessus, la nouvelle fonction (autre_fonction) renverra la même valeur que l'ancienne (fonction). Le plus ici est que si l'ancienne fonction ne renvoie aucune valeur en sortie, « **None** » sera la valeur par défaut.

Exemple

Voici ci-dessous un exemple de processus qui résume tout ceci :

```
1 def mon_decorateur(fonction):
2     def x_fonction(*args):
3
4         # Une nouvelle fonction
5         print(message 1)
6         valeur = fonction(*args)
7         print(message 2)
```



```

8         return valeur
9     return x_fonction
10
11 @mon_decorateur
12 def fonction(args...)
13     instructions

```

Maintenant que vous comprenez mieux comment fonctionnent les décorateurs avec paramètres, on peut passer à un cas plus complexe. Dans l'exemple ci-dessous, on essaie de compléter la fonction.

```

1 import time
2 def controler_temps(nb_secs):
3     def decorateur(fonction_a_executer):
4         def fonction_modifiee():
5             temps_avant = time.time()
6             valeur_renvoyee = fonction_a_executer()
7             temps_apres = time.time()
8             temps_exec = temps_apres - temps_avant
9
10            if temps_exec >= nb_secs:
11                print("La fonction {0} a mis {1} pour s'executer".format(fonction_a_executer,
temps_exec))
12            return valeur_renvoyee
13        return fonction_modifiee
14    return decorateur

```

Ici, il y a trois niveaux dans la fonction, controler_temps, qui définit dans le décorateur, définissant lui-même dans son corps, notre fonction fonction_modifiee. Nous pouvons maintenant utiliser notre décorateur avec une fonction pour le tester.

```

1 @controler_temps (4)
2 def attendre () :
3     input (" Appuyez sur Entrée ... ")
4 attendre ()

```

Exercice : Quiz

[solution n°2 p.16]

Question 1

Quelle sera la sortie du code suivant ?

```

1 def f(x):
2     def f1(a, b):
3         print("hello")
4         if b==0:
5             print("NO")
6             return
7         return f(a, b)
8     return f1
9 @f
10 def f(a, b):
11     return a%b
12 f(4,0)

```

- ☐ hello
NO
- ☐ hello
Zero Division Error

- ☐ NO
- ☐ hello

Question 2

Quelle sera la sortie du code suivant ?

```

1 def f(x):
2     def f1(*args, **kwargs):
3         print("*"* 5)
4         x(*args, **kwargs)
5         print("*"* 5)
6     return f1
7 def a(x):
8     def f1(*args, **kwargs):
9         print("%"* 5)
10        x(*args, **kwargs)
11        print("%"* 5)
12    return f1
13 @f
14 @a
15 def p(m):
16     print(m)
17 p("hello")

```

- ☐ *****
%%%%%%%%
hello
%%%%%%%%

- ☐ Error
- ☐ *****%%%%%%%%hello%%%%%%%%*****
- ☐ hello

Question 3

Identifiez la ligne du décorateur dans le code suivant :

```

1 def mk(func):
2     def sf():
3         pass
4 @mk
5 def sf():
6     return

```

- ☐ @mk
- ☐ @funct
- ☐ sf()
- ☐ mk

Question 4

Quelle sera la sortie du code Python suivant ?

```
1 def mk(x):  
2     def mk1():  
3         print("Decorated")  
4         x()  
5     return mk1  
6 def mk2():  
7     print("Ordinary")  
8 p = mk(mk2)  
9 p()
```

- ☐ Decorated
Decorated
- ☐ Ordinary
Ordinary
- ☐ Ordinary
Decorated
- ☐ Decorated
Ordinary

Question 5

Identifier le décorateur dans le code Python suivant.

```
1 def mk(x):  
2     def mk1():  
3         print("Decorated")  
4         x()  
5     return mk1  
6 def mk2():  
7     print("Ordinary")  
8 p = mk(mk2)  
9 p()
```

- ☐ p()
- ☐ mk()
- ☐ mk1()
- ☐ mk2()

V. Essentiel

En Python, la possibilité d'insérer des paramètres dans le code est facile grâce aux décorateurs. Les décorateurs interviennent en Python comme une fonction de rajout de paramètre à une autre fonction. Ils modifient dynamiquement la fonctionnalité d'une fonction, d'une méthode ou d'une classe sans avoir à utiliser directement les sous-classes ou modifier la fonction décorée elle-même.

L'utilisation de décorateurs en Python garantit également que votre code est DRY (*Don't Repeat Yourself*). Les décorateurs ont plusieurs cas d'utilisation tels que :

- Autorisation dans les frameworks Python tels que Flask et Django
- Journalisation
- Mesure du temps d'exécution
- Synchronisation

VI. Auto-évaluation

A. Exercice

En Python, les décorateurs servent à changer la manière d'exécuter une fonction. C'est un atout de ce langage et un décorateur dans un code Python peut être très utile. Il peut étendre une fonction ou une classe sans toucher le code de ces dernières. Ainsi, pour un simple programme qui calcule un solde, une entreprise souhaite afficher un message agréable à ses clients.

Question 1

[solution n°3 p.19]

Comment peut-on écrire le code d'un décorateur qui prend en paramètre la fonction qui affiche le solde et lui renvoie : « *Votre solde restant est de 50 % pour ce mois* ».

Question 2

[solution n°4 p.19]

Il peut également être utile pour les décorateurs de prendre eux-mêmes des arguments. Nous passons un argument entier au décorateur. On souhaite maintenant afficher la somme du solde et du paramètre passé au décorateur.

B. Test

Exercice 1 : Quiz

[solution n°5 p.19]

Question 1

Comme son nom l'indique, un décorateur est juste là pour la forme.

- ☐ Vrai
- ☐ Faux

Question 2

Le décorateur est précédé de « @ ».

- ☐ Vrai
- ☐ Faux

Question 3

Il est possible d'appliquer plusieurs décorateurs à une fonction.

- ☐ Vrai
- ☐ Faux

Question 4

Le décorateur permet d'exécuter du code avant ou après l'appel d'une fonction.

- ☐ Vrai
- ☐ Faux

Question 5

Ci-dessous un code Python :

```
1 @decorateur1
2 @decorateur2
3 def statut():
4     pass
```

Avec ce code, le décorateur 1 s'applique avant le décorateur 2

- ☐ Vrai
- ☐ Faux


Solutions des exercices

Exercice p. 6 Solution n°1

Question 1

Quel symbole utilise-t-on pour spécifier à Python qu'on insère un décorateur ?


- ☐ #
- ☐ \$
- ☒ @
- ☐ &

 Le symbole @ est mis pour permettre l'utilisation d'un décorateur.

Question 2

Une fonction avec des paramètres ne peut pas être décorée.

- ☐ Vrai
- ☒ Faux


 Peu importe la manière dont la fonction est déclarée, elle pourrait-être décorée.

Question 3

Quelle sera la sortie du code suivant ?

```
1 class A:
2     @staticmethod
3     def a(x):
4         print(x)
5 A.a(100)
```

- ☐ Erreur
- ☒ 100
- ☐ Pas de sortie


 Ici, la méthode a été décorée et utilisée de manière **static**.

Question 4

Quelle sera la sortie du code suivant ?

```
1 def d(f):
2     def n(*args):
3         return '$' + str(f(*args))
4     return n
5 @d
6 def p(a, t):
7     return a + a*t
8 print(p(100,0))
```

- ☐ 100
- ☒ \$100
- ☐ \$0
- ☐ 0


 Ici, le décorateur est utilisé pour mettre le symbole \$ à une fonction qui renvoie un prix.

Question 5

Quelle sera la sortie du code suivant ?

```
1 def decor_code(f):
2     def fonc_interne(*args, **kargs):
3         fonc_interne.co += 1
4         return f(*args, **kargs)
5     fonc_interne.co = 0
6     return fonc_interne
7 @decor_code
8 def fonct_result():
9     pass
10 if __name__ == '__main__':
11     fonct_result()
12     fonct_result()
13     print(fonct_result.co)
```

- ☐ 4
- ☒ 2
- ☐ 0
- ☐ 1

 Le décorateur est utilisé ici pour implémenter des bons de 1 chaque fois qu'on appelle la fonction.

Exercice p. 9 Solution n°2

Question 1

Quelle sera la sortie du code suivant ?


```
1 def f(x):
2     def f1(a, b):
3         print("hello")
4         if b==0:
5             print("NO")
6             return
7         return f(a, b)
8     return f1
9 @f
10 def f(a, b):
11     return a%b
12 f(4,0)
```


☒ hello
NO

☐ hello
Zero Division Error

☐ NO

☐ hello

 Le décorateur vérifie ici que le deuxième paramètre de la fonction est égal à **0**. Si c'est le cas, il complète **NO** à la chaîne de caractère **HELLO**.

Question 2

Quelle sera la sortie du code suivant ?


```
1 def f(x):
2     def f1(*args, **kwargs):
3         print("*"* 5)
4         x(*args, **kwargs)
5         print("*"* 5)
6     return f1
7 def a(x):
8     def f1(*args, **kwargs):
9         print("%"* 5)
10        x(*args, **kwargs)
11        print("%"* 5)
12    return f1
13 @f
14 @a
15 def p(m):
16     print(m)
17 p("hello")
```

☒ *****
%%%%%%%%
hello
%%%%%%%%

☐ Error

☐ *****%%%%%%%%hello%%%%%%%%*****

☐ hello

 Ici, le décorateur nous permet de vraiment décorer l'affichage du hello. Les symboles ******* et **%%%** encadrent la valeur passée en paramètre de la fonction **p**.

Question 3

Identifiez la ligne du décorateur dans le code suivant :


```
1 def mk(func):
2     def sf():
3         pass
4 @mk
5 def sf():
6     return
```

☒ @mk

☐ @funct

☐ sf()

☐ mk

 Le décorateur utilisé est @mk. On reconnaît les décorateurs par le signe @.

Question 4

Quelle sera la sortie du code Python suivant ?


```
1 def mk(x):
2     def mk1():
3         print("Decorated")
4         x()
5     return mk1
6 def mk2():
7     print("Ordinary")
8 p = mk(mk2)
9 p()
```

☐ Decorated
Decorated

☐ Ordinary
Ordinary

☐ Ordinary
Decorated

☒ Decorated
Ordinary

 Comme la règle avec deux décorateurs l'indique, le mot décoré est d'abord affiché en premier et le mot ordinaire en deuxième position.

Question 5

Identifier le décorateur dans le code Python suivant.


```
1 def mk(x):
2     def mk1():
3         print("Decorated")
4         x()
5     return mk1
6 def mk2():
7     print("Ordinary")
8 p = mk(mk2)
9 p()
```

☐ p()

☒ mk()

☐ mk1()

☐ mk2()

-  Dans le code ci-dessus, la fonction `mk()` est le décorateur. La fonction qui est décorée est `mk2()`. La fonction de retour porte le nom de `p()`.

p. 12 Solution n°3

Tout d'abord, nous donnons au décorateur un nom approprié qui indique l'objectif visé. Ici, le décorateur, appelé **decorateur_solde** est simplement une fonction qui prend une autre comme un argument. Cette fonction a été nommée 'fonct' comme argument de **decorateur_solde**.

On définit ensuite une fonction locale appelée **inner** à l'intérieur de la fonction décorée. La fonction **inner** interne renvoie alors une première chaîne de caractères « **Votre solde restant est de** », concaténé avec le résultat de la fonction mit en paramètre du décorateur et enfin concaténé avec la chaîne de caractères « **pour ce mois** ».

p. 12 Solution n°4

Pour utiliser des arguments dans les décorateurs, nous avons simplement besoin de définir un autre décorateur qui prendra non pas la fonction d'origine, mais un paramètre appelé **param** (entier). Dans ce décorateur, sera défini notre fameux **decorateur_solde**.

Ensuite, le processus sera le même que précédemment à la différence que, lors de la concaténation des chaînes de caractère avec la fonction **func**, nous ferons d'abord une addition entre **func** et **param**.


Exercice p. 12 Solution n°5

Question 1

Comme son nom l'indique, un décorateur est juste là pour la forme.

☐ Vrai

☒ Faux


-  Un décorateur en Python permet d'ajouter d'autres détails à une fonction sans changer le code de cette dernière.

Question 2

Le décorateur est précédé de « @ ».

☒ Vrai

☐ Faux


-  Les décorateurs en Python sont définis par la présence de « @ » au début.

Question 3

Il est possible d'appliquer plusieurs décorateurs à une fonction.

☒ Vrai

☐ Faux


-  En Python, en fonction de l'objectif, plusieurs décorateurs peuvent être appliqués à une fonction.

Question 4

Le décorateur permet d'exécuter du code avant ou après l'appel d'une fonction.

☒ Vrai

☐ Faux

 Un décorateur peut être utilisé avant ou après la définition d'une fonction selon l'objectif du code.

Question 5


Ci-dessous un code Python :

```
1 @decorateur1
2 @decorateur2
3 def statut():
4     pass
```

Avec ce code, le décorateur 1 s'applique avant le décorateur 2

☐ Vrai

☒ Faux

 Lorsque plusieurs décorateurs sont appliqués, le premier décorateur s'applique en dernier.