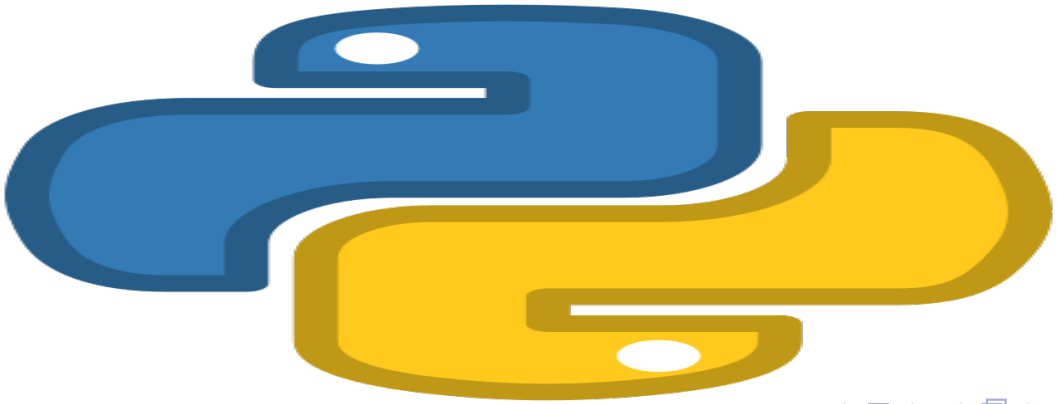
Python : décorateurs

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Plan

Python

Décorateur

- L'équivalent d'annotations en **Java** et **PHP**
- Méta-programmation : ajout des informations ou modification du comportement d'une fonction, méthode, attribut, classe
- Utilisé avec le préfixe `@`

Python

Considérons la fonction `dire_bonjour` suivante

```
def dire_bonjour():
    print("bonjour")
```

Python

Considérons la fonction `dire_bonjour` suivante

```
def dire_bonjour():
    print("bonjour")
```

La fonction `dire_bonjour` peut être appelée ainsi

```
dire_bonjour()
# affiche bonjour
```

Python

Ajoutons le décorateur `@premier_decorateur()` **à la fonction** `dire_bonjour()`

```
@premier_decorateur
def dire_bonjour():
    print("bonjour")
```

Python

Ajoutons le décorateur `@premier_decorateur()` **à la fonction** `dire_bonjour()`

```
@premier_decorateur
def dire_bonjour():
    print("bonjour")
```


Constat

`@premier_decorateur` souligné en rouge

Python

Ajoutons le décorateur `@premier_decorateur()` à la fonction `dire_bonjour()`

```
@premier_decorateur
def dire_bonjour():
    print("bonjour")
```


Constat

`@premier_decorateur` souligné en rouge

Solution

Définir une fonction `premier_decorateur` qui précise le traitement à faire avant d'exécuter les fonctions décorées par `@premier_decorateur`.

Python

La fonction associée au décorateur `@premier_decorateur`) prend en paramètre la fonction que le décorateur modifie. Elle retourne également cette fonction reçu comme paramètre (modifiée).

```
def premier_decorateur(function):
    print("premier_decorateur")
    return function
```

Python

La fonction associée au décorateur `@premier_decorateur`) prend en paramètre la fonction que le décorateur modifie. Elle retourne également cette fonction reçu comme paramètre (modifiée).

```
def premier_decorateur(function):
    print("premier_decorateur")
    return function
```


En appelant la fonction `dire_bonjour`, le résultat est :

```
dire_bonjour()
# affiche premier_decorateur
# bonjour
```

Python

Un décorateur peut décorer plusieurs fonctions. On peut donc récupérer le nom de la fonction décorée.

```
def premier_decorateur(function):
    print("premier_decorateur appelée pour la
       fonction", function.__name__)
    return function
```

Python

Un décorateur peut décorer plusieurs fonctions. On peut donc récupérer le nom de la fonction décorée.

```
def premier_decorateur(function):
    print("premier_decorateur appelée pour la
       fonction", function.__name__)
    return function
```


En appelant la fonction `dire_bonjour`, le résultat est :

```
dire_bonjour()
# affiche premier_decorateur appelée pour la
   fonction dire_bonjour
# bonjour
```

Python

La fonction associée au décorateur peut être redéfinie avec une fonction interne

```
def premier_decorateur(function):
    print("premier_decorateur appelée pour la fonction",
       function.__name__)

    def new_function():
        function()

    return new_function
```

Python

La fonction associée au décorateur peut être redéfinie avec une fonction interne

```
def premier_decorateur(function):
    print("premier_decorateur appelée pour la fonction",
       function.__name__)

    def new_function():
        function()

    return new_function
```


En appelant la fonction `dire_bonjour`, le résultat est :

```
dire_bonjour()
# affiche premier_decorateur appelée pour la fonction
   dire_bonjour
# bonjour
```

Python

Pour faire un traitement avant et après l'exécution de la fonction décorée

```
def premier_decorateur(function):
    print("premier_decorateur appelée pour la fonction", function.
       __name__)
    def new_function():
        print("avant exécution de la fonction")
        function()
        print("après exécution de la fonction")
    return new_function
```

Python

Pour faire un traitement avant et après l'exécution de la fonction décorée

```
def premier_decorateur(function):
    print("premier_decorateur appelée pour la fonction", function.
       __name__)
    def new_function():
        print("avant exécution de la fonction")
        function()
        print("après exécution de la fonction")
    return new_function
```


En appelant la fonction `dire_bonjour`, le résultat est :

```
dire_bonjour()
# affiche premier_decorateur appelée pour la fonction dire_bonjour
# avant exécution de la fonction
# bonjour
# après exécution de la fonction
```

Python

Question

Et si notre fonction prenait des paramètres et retournait une valeur, comment faire pour les récupérer ?

Python

Considérons la fonction `somme` suivante

```
def somme(a, b):
    return a + b
```

Python

Considérons la fonction `somme` suivante

```
def somme(a, b):
    return a + b
```

La fonction `somme` peut être appelée ainsi

```
print(somme(2, 3))
# affiche 5
```

Python

Ajoutons le décorateur `@premier_decorateur()` à la fonction `somme()`

```
@premier_decorateur
def somme(a, b):
    return a + b
```

Python

Pour récupérer les paramètres de la fonction, dans le décorateur, utilisés au moment de l'appel.

```
def premier_decorateur(function):
    print("premier_decorateur")
    def new_function(*args, **kwargs):
        print("avant exécution de la fonction", args, kwargs)
        function(*args, **kwargs)
        print("après exécution de la fonction")
    return new_function
```

Python

Pour récupérer les paramètres de la fonction, dans le décorateur, utilisés au moment de l'appel.

```
def premier_decorateur(function):
    print("premier_decorateur")
    def new_function(*args, **kwargs):
        print("avant exécution de la fonction", args, kwargs)
        function(*args, **kwargs)
        print("après exécution de la fonction")
    return new_function
```


En appelant la fonction `somme`, le résultat est :

```
print(somme(2, 3))
# affiche premier_decorateur
# avant exécution de la fonction (2, 3) {}
# après exécution de la fonction
# None
```

Python

Pour récupérer les paramètres de la fonction, dans le décorateur, utilisés au moment de l'appel.

```
def premier_decorateur(function):
    print("premier_decorateur")
    def new_function(*args, **kwargs):
        print("avant exécution de la fonction", args, kwargs)
        function(*args, **kwargs)
        print("après exécution de la fonction")
    return new_function
```


En appelant la fonction `somme`, le résultat est :

```
print(somme(2, 3))
# affiche premier_decorateur
# avant exécution de la fonction (2, 3) {}
# après exécution de la fonction
# None
```

`None` est affiché à la place du résultat de la somme.

Python

Pour afficher la valeur de retour de la fonction `somme`

```
def premier_decorateur(function):
    print("premier_decorateur")
    def new_function(*args, **kwargs):
        print("avant exécution de la fonction", args, kwargs)
        returned_value = function(*args, **kwargs)
        print("après exécution de la fonction")
        return returned_value
    return new_function
```

Python

Pour afficher la valeur de retour de la fonction `somme`

```
def premier_decorateur(function):
    print("premier_decorateur")
    def new_function(*args, **kwargs):
        print("avant exécution de la fonction", args, kwargs)
        returned_value = function(*args, **kwargs)
        print("après exécution de la fonction")
        return returned_value
    return new_function
```


En appelant la fonction `somme`, le résultat est :

```
print(somme(2, 3))
# affiche premier_decorateur
# avant exécution de la fonction (2, 3) {}
# après exécution de la fonction
# 5
```

Python

Question

Et comment récupérer le paramètre d'un décorateur ?

Python

Question

Et comment récupérer le paramètre d'un décorateur ?

Réponse

- La fonction associée au décorateur récupère le paramètre du décorateur
- Définir une deuxième une fonction interne pour récupérer la fonction décorée
- La fonction associée au décorateur retourne la nouvelle fonction interne

Python

Ajoutons la nouvelle fonction interne

```
def premier_decorateur(param_decorator):
    print("paramètre du décorateur :", param_decorator)

    def decorateur(function):
        def new_function(*args, **kwargs):
            print("avant exécution de la fonction", args, kwargs)
            returned_value = function(*args, **kwargs)
            print("après exécution de la fonction")
            return int(returned_value) if param_decorator == "int" else
                float(returned_value)

        return new_function

    return decorateur
```

Python

En appelant la fonction `somme`, le résultat est :

```
@premier_decorateur("float")
def somme(a, b):
    return a + b

print(somme(2, 3))
# affiche paramètre du décorateur : float
# avant exécution de la fonction (2, 3) {}
# après exécution de la fonction
# 5.0
```

Python

Remarque

Une fonction peut être décorée par plusieurs décorateurs

Python

Remarque

Une fonction peut être décorée par plusieurs décorateurs

Décorons la fonction `@premier_decorateur()` par les deux décorateurs suivants

```
@a
@b
def dire_bonjour():
    print("bonjour")
```

Python

Définissons les fonctions associées aux deux décorateurs `@a` et `@b`

```
def a(function):
    print("@a", function.__name__)
    return function


def b(function):
    print("@b", function.__name__)
    return function
```

Python

Définissons les fonctions associées aux deux décorateurs `@a` et `@b`

```
def a(function):
    print("@a", function.__name__)
    return function


def b(function):
    print("@b", function.__name__)
    return function
```


En appelant la fonction `dire_bonjour`, le résultat est :

```
dire_bonjour()

# affiche
# @b dire_bonjour
# @a dire_bonjour
# bonjour
```

Python

Considérons les deux fonctions suivantes décorées par `@second_decorateur`

```
@second_decorateur
def dire_bonjour():
    """DocString de dire_bonjour"""
    print("dire_bonjour")

@second_decorateur
def say_hello():
    """DocString de say_hello"""
    print("say_hello")
```

Python

Considérons les deux fonctions suivantes décorées par `@second_decorateur`

```
@second_decorateur
def dire_bonjour():
    """DocString de dire_bonjour"""
    print("dire_bonjour")

@second_decorateur
def say_hello():
    """DocString de say_hello"""
    print("say_hello")
```


Le code de la fonction associée au `@second_decorateur`

```
def second_decorateur(function):
    def new_function(*args, **kwargs):
        """DocString de new_function"""
        function()

    return new_function
```

La surprise en affichant les constantes de fonction ou en appelant la fonction `help`

```
print(dire_bonjour.__name__)
# new_function

print(dire_bonjour.__doc__)
# DocString de new_function

help(dire_bonjour)
# Help on function new_function in module __main__:
#
# new_function(*args, **kwargs)
#     DocString de new_function

print(say_hello.__name__)
# new_function

print(say_hello.__doc__)
# DocString de new_function

help(say_hello)
# Help on function new_function in module __main__:
#
# new_function(*args, **kwargs)
#     DocString de new_function
```

Python

Pour résoudre le problème précédent, une première solution consiste à modifier la fonction associée au décorateur ainsi

```
def second_decorateur(function):
    def new_function(*args, **kwargs):
        """DocString de new_function"""
        function()

    new_function.__name__ = function.__name__
    new_function.__doc__ = function.__doc__
    return new_function
```

Les constantes de fonction affichent des valeurs correctes mais pas la fonction `help`

```
print(dire_bonjour.__name__)
# dire_bonjour

print(dire_bonjour.__doc__)
# DocString de dire_bonjour

help(dire_bonjour)
# Help on function dire_bonjour in module __main__:
#
# dire_bonjour(*args, **kwargs)
#     DocString de dire_bonjour

print(say_hello.__name__)
# say_hello

print(say_hello.__doc__)
# DocString de say_hello

help(say_hello)
# Help on function say_hello in module __main__:
#
# say_hello(*args, **kwargs)
#     DocString de say_hello
```

Python

Remarque

Les changements précédents n'ont pas permis d'avoir les signatures exactes pour les méthodes `dire_bonjour` et `say_hello`.

Python

Pour résoudre le problème précédent, une deuxième solution consiste à utiliser le décorateur `@wraps` du module `functools`

```
from functools import wraps

def second_decorateur(function):
    @wraps(function)
    def new_function(*args, **kwargs):
        """DocString de new_function"""
        function()

    return new_function
```

Les problèmes précédents sont résolus

```
print(dire_bonjour.__name__)
# dire_bonjour

print(dire_bonjour.__doc__)
# DocString de dire_bonjour

help(dire_bonjour)
# Help on function dire_bonjour in module __main__:
#
# dire_bonjour()
#     DocString de dire_bonjour

print(say_hello.__name__)
# say_hello

print(say_hello.__doc__)
# DocString de say_hello

help(say_hello)
# Help on function say_hello in module __main__:
#
# say_hello()
#     DocString de say_hello
```

Python

Décorateurs prédéfinis

- `@staticmethod` : Utilisé pour définir une méthode de classe qui n'a pas accès à l'instance elle-même.
- `@classmethod` : Utilisé pour définir une méthode de classe qui prend la classe elle-même en tant que premier argument au lieu de l'instance.
- `@property` : Utilisé pour définir une méthode qui peut être accédée comme un attribut, sans utiliser de parenthèses pour l'appeler.
- `@abstractmethod` : Utilisé dans les classes abstraites pour définir une méthode qui doit être implémentée par les sous-classes.
- `@lru_cache` : Utilisé pour mémoriser les résultats de manière efficace pour les fonctions coûteuses en temps de calcul, en utilisant une mémoire cache de taille limitée.
- ...

Python

Exemple avec `@lru_cache` (Least Recently Used se traduit littéralement par le moins récemment utilisé). `maxsize` spécifie la taille maximale de la mémoire cache utilisée pour stocker les résultats des appels précédents.

```
from functools import lru_cache


@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(40))
# affiche 102334155
```

Python

Exemple avec `@lru_cache` (Least Recently Used se traduit littéralement par le moins récemment utilisé). `maxsize` spécifie la taille maximale de la mémoire cache utilisée pour stocker les résultats des appels précédents.

```
from functools import lru_cache


@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(40))
# affiche 102334155
```

Commentez le décorateur, relancez et vérifiez que le programme met beaucoup plus de temps pour répondre.