Декораторы

Сапунов Н.А.

1 Что такое декораторы

Все в python - объект

Чтобы понять, что такое декораторы, нужно помнить, что функции в python - объекты. Да и вообще, все в python - объект. Объект, напоминаю, это экземпляр класса. Например, если Robot - это класс:

```
In [2]: class Robot:
   ...:
   ...:     def __init__(self, name):
   ...:
   ...:         self.name = name
```

То экземпляром этого класса будет объект. Например, создадим робота Валли. Для этого переменной vally присвоим экземпляр класса Robot с именем Vally.

```
In [3]: vally = Robot(name='Vally')

In [4]: vally.name
Out[4]: 'Vally'
```

Любые объекты в python можно передавать в качестве аргументов функции. Создадим функцию **foo**, которая первым позиционным аргументом будет принимать какие-либо данные, а вторым функцию, которая будет с этими данными что-то делать. Функция, которая будет модифицировать данные, функция **bar** - принимает на вход обычный текст.

```
In [5]: def foo(data, func):
   ...:
   ...:     return func(data)
   ...:

In [6]: def bar(text):
   ...:
   ...:     return 'I\'m function `bar`. You entered: {0}'.format(text)
   ...:

In [7]: foo('How are you?', bar)
Out[7]: "I'm function `bar`. You entered: How are you?"
```

Также стоит помнить, что внутри функции можно легко объявить другую функцию. Создадим функцию, которая будет принимать на вход какой-либо текст. Внутри этой функции определим еще одну функцию, которая будет заменять только 3 символ во входном тексте на символ в нижнем регистре. Вернем из внешней функции результат применения внутренней функции.

```
In [10]: def one_function(text):
    ...:
    ...:     def third_elem_lower(text):
    ...:
    ...:         return text[:2] + text[2].lower() + text[3:]
    ...:
    ...:     return third_elem_lower(text)
    ...:

In [11]: one_function('RUSSIAN')
Out[11]: 'RUsSIAN'
```

Функция, которая возвращает другую функцию

Декоратор - это функция, которая возвращает другую функцию. Смысл применения декораторов в изменении поведения какой-либо функции без явного модифицирования её самой.

Например, у нас имеется функция, которая генерирует случайное число, основываясь на идентификаторе пользователя:

```
In [12]: import random

In [13]: def get_rand_digit(user_id):
    ...:
    ...:     return random.randint(1, user_id * 100)
    ...:

In [14]: get_rand_digit(user_id=13)
Out[14]: 344
```

Но нам хотелось бы знать это случайное число для каждого пользователя. Мы можем написать такой декоратор:

```
In [22]: def sniff_decorator(func):
    ...:
    ...:     def inner(user_id):
    ...:
    ...:         generated_digit = func(user_id=user_id)
    ...:
    ...:         print('Agent 007 says: user-{0}\'s random digit: {1}'.format(user_id, generated_digit))
    ...:
    ...:         return generated_digit
    ...:
    ...:     return inner
    ...:

In [23]: new_get_rand_digit = sniff_decorator(get_rand_digit)

In [24]: new_get_rand_digit(user_id=13)
Agent 007 says: user-13's random digit: 57
Out[24]: 57
```

Как видно, мы передаем функцию **get_rand_digit** в созданный декоратор **sniff_decorator**. Внутри sniff_decorator создается еще одна функция, **inner** [1] Функция inner вызывает функцию **func**, которой в данном случае является get_rand_digit, печатает сообщение со сгенерированным номером, и, как ни в чём не бывало, возвращает ответ наружу. Сам же декоратор возвращает эту внутреннюю функцию, и пользователь ничего не замечает[2] при следующей генерации случайного числа.

Приведенный выше синтаксис декораторов мягко говоря не слишком привлекательный и в python есть сладкий синтаксический сахар на этот счет.

[1] Вообще не имеет значения, как назвать эту внутреннюю функцию, название inner(от англ.: внутренний) просто очень хорошо подходит для этих целей, и это является хорошей практикой.

[2] В данном случае конечно же пользователь заметит, что ему печатается что-то еще, но в настоящем шпионском декораторе лучше применять не print, а что-то вроде вывода в сторонний файл

```
In [25]: def sniff_decorator(func):
    ...:
    ...:     def inner(user_id):
    ...:
    ...:         generated_digit = func(user_id=user_id)
    ...:
    ...:         print('Agent 007 says: user-{0}\'s random digit: {1}'.format(user_id, generated_digit))
    ...:
    ...:         return generated_digit
    ...:
    ...:     return inner
    ...:

In [26]: @sniff_decorator
    ...: def get_rand_digit(user_id):
    ...:
    ...:     return random.randint(1, user_id * 100)
    ...:

In [27]: get_rand_digit(user_id=13)
Agent 007 says: user-13's random digit: 720
Out[27]: 720
```

В данном случае прямо при определении функции **get_rand_digit** ей навешивается декоратор **sniff_decorator** через собачку **@**. Выглядит намного приятнее, чем то, что мы писали до этого.

2 Проблемы с описанием

Всеми любимая функция help

В python есть очень полезная и часто используемая функция **help**. С помощью этой функции можно вывести описание функции, которое оставил разработчик. Добавим небольшое описание в нашу функцию **get_rand_digit** и попробуем его получить через help.

```
In [28]: def get_rand_digit(user_id):
    ...:     """Return random number according to user_id.
    ...:
    ...:     Arguments:
    ...:         user_id(int) - user idenificator
    ...:     Returns:
    ...:         random number(int)"""
    ...:
    ...:     return random.randint(1, user_id * 100)
    ...:

In [29]: help(get_rand_digit)
Help on function get_rand_digit in module __main__:

get_rand_digit(user_id)
    Return random number according to user_id.

    Arguments:
        user_id(int) - user idenificator
    Returns:
        random number(int)
```

Все прекрасно, оно работает. Но что будет, если мы добавим декоратор. Ведь как мы знаем, декоратор возвращает уже не нашу функцию, а внутреннюю функцию. Так что же будет в help.

```
In [30]: @sniff_decorator
    ...: def get_rand_digit(user_id):
    ...:     """Return random number according to user_id.
    ...:
    ...:     Arguments:
    ...:         user_id(int) - user idenificator
    ...:     Returns:
    ...:         random number(int)"""
    ...:
    ...:     return random.randint(1, user_id * 100)
    ...:

In [31]: help(get_rand_digit)
Help on function inner in module __main__:

inner(user_id)
```

Как уже наверное стало понятно из название этого раздела, будет проблема. Проблема очень понятная. Ведь декоратор возвращает уже не нашу функцию, а свою внутреннюю. Вот наш агент 007 и спалился. А жаль. Хороший был агент.

Ладно, делать то что? Есть 2 пути, первый - это пойти напролом и ручками в каждом декораторы заменять соответствующие свойства. Ведь функция - это объект, а у любого объекта есть свойста и одно из них - **__doc__**, отвечающее за хранения пояснительной информации. Перепишем наш декоратор в новых реалиях.

```
In [34]: def sniff_decorator(func):
    ...:
    ...:     def inner(user_id):
    ...:
    ...:         generated_digit = func(user_id=user_id)
    ...:
    ...:         print('Agent 007 says: user-{0}\'s random digit: {1}'.format(user_id, generated_digit))
    ...:
    ...:         return generated_digit
    ...:
    ...:     inner.__doc__ = func.__doc__
    ...:
    ...:     return inner
    ...:

In [35]: @sniff_decorator
    ...: def get_rand_digit(user_id):
    ...:     """Return random number according to user_id.
    ...:
    ...:     Arguments:
    ...:         user_id(int) - user idenificator
    ...:     Returns:
    ...:         random number(int)"""
    ...:
    ...:     return random.randint(1, user_id * 100)
    ...:

In [36]: help(get_rand_digit)
Help on function inner in module __main__:

inner(user_id)
    Return random number according to user_id.

    Arguments:
        user_id(int) - user idenificator
    Returns:
        random number(int)
```

Мы исправили ситуацию, но так ли это. Кроме свойства __doc__ есть еще свойство **__module__**, отвечающее за хранение названия модуля, в котором определена функция, и конечно же **__name__**. Эти свойства тоже бывают полезны. Да и вообще, кому нужно писать лишний код. Итак, метод №2. Чтобы облегчить эту задачу переписывания свойств, в python есть специальный модуль **functools**, а в нем спициальный декоратор **wraps**, который все делает за нас.

```
In [37]: import functools

In [38]: def sniff_decorator(func):
    ...:
    ...:     @functools.wraps(func)
    ...:     def inner(user_id):
    ...:
    ...:         generated_digit = func(user_id=user_id)
    ...:
    ...:         print('Agent 007 says: user-{0}\'s random digit: {1}'.format(user_id, generated_digit))
    ...:
    ...:         return generated_digit
    ...:
    ...:     return inner
    ...:

In [39]: @sniff_decorator
    ...: def get_rand_digit(user_id):
    ...:     """Return random number according to user_id.
    ...:
    ...:     Arguments:
    ...:         user_id(int) - user idenificator
    ...:     Returns:
    ...:         random number(int)"""
    ...:
    ...:     return random.randint(1, user_id * 100)
    ...:

In [40]: help(get_rand_digit)
Help on function get_rand_digit in module __main__:

get_rand_digit(user_id)
    Return random number according to user_id.

    Arguments:
        user_id(int) - user idenificator
    Returns:
        random number(int)
```

3 Тренировка

Репозитарий с задачами - https://github.com/Sapunov/progtasks.

Задача 004

Сейчас там 2 файла:

- data.txt - файл с данными
- grabber.py - скрипт с функцией

При запуске grabber.py скрипт выведет приведенный в текстовый вид список с кортежами. Список имеет следующий вид:

```
[
        (
                ('Вася', 'name'),
                ('Пупкин', 'surname'),
                (40, 'age'),
                ('Москва', 'city')
        ),
        ...
]
```

То есть каждый элемент списка - это кортеж, в каждом из которых находится 4 кортежа с каким-то значением, и названием этого значения. То есть значение - Вася, а название этого значения - name.

Формат так себе, именно поэтому нужно написать декоратор, который будет принимать на вход то, что выдает функция и приводить это к виду json. Напоминаю, в python есть встроенный модуль **json**. Его надо использовать. Вывод в виде json должен быть красивым, с отступами в 4 пробела. Нужно использовать всю теорию, которая есть выше.

Все изменения необходимо сделать в файле grabber.py и прислать Pull Request.

Задача 005

Задание откроется когда, будет сделана задача 004. Эта задача более интересная.