

Декораторы

Декораторы — один из самых полезных инструментов в Python, однако новичкам они могут показаться непонятными. Возможно, вы уже встречались с ними, например, при работе с Flask, но не хотели особо вникать в суть их работы. Эта статья поможет вам понять, чем являются декораторы и как они работают.

Что такое декоратор?

Новичкам декораторы могут показаться неудобными и непонятными, потому что они выходят за рамки «обычного» процедурного программирования как в Си, где вы объявляете функции, содержащие блоки кода, и вызываете их. То же касается и объектно-ориентированного программирования, где вы определяете классы и создаёте на их основе объекты. Декораторы не принадлежат ни одной из этих парадигм и исходят из области функционального программирования. Однако не будем забегать вперёд, разберёмся со всем по порядку.

Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода. Вот почему декораторы можно рассматривать как практику метапрограммирования, когда программы могут работать с другими программами как со своими данными. Чтобы понять, как это работает, сначала разберёмся в работе функций в Python.

Как работают функции

Все мы знаем, что такое функции, не так ли? Не будьте столь уверены в этом. У функций Python есть определённые аспекты, с которыми мы нечасто имеем дело, и, как следствие, они забываются. Давайте проясним, что такое функции и как они представлены в Python.

Функции как процедуры

С этим аспектом функций мы знакомы лучше всего. Процедура — это именованная последовательность вычислительных шагов. Любую процедуру можно вызвать в любом месте программы, в том числе внутри другой процедуры или даже самой себя. По этой части больше нечего сказать, поэтому переходим к следующему аспекту функций в Python.

Функции как объекты первого класса

В Python всё является объектом, а не только объекты, которые вы создаёте из классов. В этом смысле он (Python) полностью соответствует идеям объектно-ориентированного программирования. Это значит, что в Python всё это — объекты:

- числа;
- строки;
- классы (да, даже классы!);
- функции (то, что нас интересует).

Тот факт, что всё является объектами, открывает перед нами множество возможностей. Мы можем сохранять функции в переменные, передавать их в качестве аргументов и возвращать из других функций. Можно даже определить одну функцию внутри другой. Иными словами, функции — это объекты первого класса. Из определения в [Википедии](#):



Объектами первого класса в контексте конкретного языка программирования называются элементы, с которыми можно делать всё то же, что и с любым другим объектом: передавать как параметр, возвращать из функции и присваивать переменной.

И тут в дело вступает функциональное программирование, а вместе с ним — декораторы.

Функциональное программирование — функции высших порядков

В Python используются некоторые концепции из функциональных языков вроде Haskell и OCaml. Пропустим формальное определение функционального языка и перейдём к двум его характеристикам, свойственным Python:

- функции являются объектами первого класса;
- следовательно, язык поддерживает функции высших порядков.

Функциональному программированию присущи и другие свойства вроде отсутствия побочных эффектов, но мы здесь не за этим. Лучше сконцентрируемся на другом — функциях высших порядков. Что есть функция высшего порядка? Снова обратимся к [Википедии](#):

“Функции высших порядков — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.

Если вы знакомы с основами высшей математики, то вы уже знаете некоторые математические функции высших порядков порядка вроде дифференциального оператора d/dx . Он принимает на входе функцию и возвращает другую функцию, производную от исходной. Функции высших порядков в программировании работают точно так же — они либо принимают функцию(и) на входе и/или возвращают функцию(и).

Пара примеров

Раз уж мы ознакомились со всеми аспектами функций в Python, давайте продемонстрируем их в коде:

```
def hello_world():  
    print('Hello world!')
```

Здесь мы определили простую функцию. Из фрагмента кода далее вы увидите, что эта функция, как и классы с числами, является объектом в Python:

```
>>> def hello_world():  
...     print('Hello world!')  
...  
>>> type(hello_world)  
<class 'function'>  
>>> class Hello:  
...     pass  
...  
>>> type(Hello)  
<class 'type'>  
>>> type(10)  
<class 'int'>
```

Как вы заметили, функция `hello_world` принадлежит типу `<class 'function'>`. Это означает, что она является объектом класса `function`. Кроме того, класс, который мы определили, принадлежит классу `type`. От этого всего голова может пойти кругом, но чуть поигравшись с функцией `type` вы со всем разберётесь.

Теперь давайте посмотрим на функции в качестве объектов первого класса.

Мы можем хранить функции в переменных:

```
>>> hello = hello_world
>>> hello()
Hello world!
```

Определять функции внутри других функций:

```
>>> def wrapper_function():
...     def hello_world():
...         print('Hello world!')
...     hello_world()
...
>>> wrapper_function()
Hello world!
```

Передавать функции в качестве аргументов и возвращать их из других функций:

```
>>> def higher_order(func):
...     print('Получена функция {} в качестве аргумента'.format(func))
...     func()
...     return func
...
>>> higher_order(hello_world)
Получена функция <function hello_world at 0x032C7FA8> в качестве аргумента
Hello world!
<function hello_world at 0x032C7FA8>
```

Из этих примеров должно стать понятно, насколько функции в Python гибкие. С учётом этого можно переходить к обсуждению декораторов.

Как работают декораторы

Повторим определение декоратора:

“Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.

Раз мы знаем, как работают функции высших порядков, теперь мы можем понять как работают декораторы. Сначала посмотрим на пример декоратора:

```
def decorator_function(func):  
    def wrapper():  
        print('Функция-обёртка!')  
        print('Оборачиваемая функция: {}'.format(func))  
        print('Выполняем обёрнутую функцию...')  
        func()  
        print('Выходим из обёртки')  
    return wrapper
```

Здесь `decorator_function()` является функцией-декоратором. Как вы могли заметить, она является функцией высшего порядка, так как принимает функцию в качестве аргумента, а также возвращает функцию. Внутри `decorator_function()` мы определили другую функцию, обёртку, так сказать, которая обёртывает функцию-аргумент и затем изменяет её поведение. Декоратор возвращает эту обёртку. Теперь посмотрим на декоратор в действии:

```
>>> @decorator_function  
... def hello_world():  
...     print('Hello world!')  
...  
>>> hello_world()  
Оборачиваемая функция: <function hello_world at 0x032B26A8>  
Выполняем обёрнутую функцию...  
Hello world!  
Выходим из обёртки
```

Магия, не иначе! Просто добавив `@decorator_function` перед определением функции `hello_world()`, мы модифицировали её поведение. Однако как вы уже могли догадаться, выражение с `@` является всего лишь синтаксическим сахаром для `hello_world = decorator_function(hello_world)`.

Иными словами, выражение `@decorator_function` вызывает `decorator_function()` с `hello_world` в качестве аргумента и присваивает имени `hello_world` возвращаемую функцию.

И хотя этот декоратор мог вызвать вау-эффект, он не очень полезный. Давайте взглянем на другие, более полезные (наверное):

```
def benchmark(func):
    import time

    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
    return wrapper

@benchmark
def fetch_webpage():
    import requests
    webpage = requests.get('https://google.com')

fetch_webpage()
```

Здесь мы создаём декоратор, замеряющий время выполнения функции. Далее мы используем его на функции, которая делает GET-запрос к главной странице Google. Чтобы измерить скорость, мы сначала сохраняем время перед выполнением обёрнутой функции, выполняем её, снова сохраняем текущее время и вычитаем из него начальное.

После выполнения кода получаем примерно такой результат:

```
[*] Время выполнения: 1.4475083351135254 секунд.
```

К этому моменту вы, наверное, начали осознавать, насколько полезными могут быть декораторы. Они расширяют возможности функции без редактирования её кода и являются гибким инструментом для изменения чего угодно.

Используем аргументы и возвращаем значения

В приведённых выше примерах декораторы ничего не принимали и не возвращали. Модифицируем наш декоратор для измерения времени выполнения:

```
def benchmark(func):
    import time

    def wrapper(*args, **kwargs):
        start = time.time()
        return_value = func(*args, **kwargs)
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
        return return_value
    return wrapper

@benchmark
def fetch_webpage(url):
    import requests
    webpage = requests.get(url)
    return webpage.text

webpage = fetch_webpage('https://google.com')
print(webpage)
```

Вывод после выполнения:

```
[*] Время выполнения: 1.4475083351135254 секунд.
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage".....
```


Как вы видите, аргументы декорируемой функции передаются функции-обёртке, после чего с ними можно делать что угодно. Можно изменять аргументы и затем передавать их декорируемой функции, а можно оставить их как есть или вовсе забыть про них и передать что-нибудь совсем другое. То же касается возвращаемого из декорируемой функции значения, с ним тоже можно делать что угодно.

Декораторы с аргументами

Мы также можем создавать декораторы, которые принимают аргументы. Посмотрим на пример:

```
def benchmark(iters):
    def actual_decorator(func):
        import time

        def wrapper(*args, **kwargs):
            total = 0
            for i in range(iters):
                start = time.time()
                return_value = func(*args, **kwargs)
                end = time.time()
                total = total + (end-start)
            print('[*] Среднее время выполнения: {} секунд.'.format(total/iters))
            return return_value

        return wrapper
    return actual_decorator

@benchmark(iters=10)
def fetch_webpage(url):
    import requests
    webpage = requests.get(url)
    return webpage.text

webpage = fetch_webpage('https://google.com')
print(webpage)
```

Здесь мы модифицировали наш старый декоратор таким образом, чтобы он выполнял декорируемую функцию `iters` раз, а затем выводил среднее время выполнения. Однако чтобы добиться этого, пришлось воспользоваться природой функций в Python.

Функция `benchmark()` на первый взгляд может показаться декоратором, но на самом деле таковым не является. Это обычная функция, которая принимает аргумент `iters`, а затем возвращает декоратор. В свою очередь, он декорирует функцию `fetch_webpage()`. Поэтому мы использовали не выражение `@benchmark`, а `@benchmark(iters=10)` — это означает, что тут вызывается функция `benchmark()` (функция со скобками после неё обозначает вызов функции), после чего она возвращает сам декоратор.

Да, это может быть действительно сложно уместить в голове, поэтому держите правило:

“Декоратор принимает функцию в качестве аргумента и возвращает функцию.

В нашем примере `benchmark()` не удовлетворяет этому условию, так как она не принимает функцию в качестве аргумента. В то время как функция `actual_decorator()`, которая возвращается `benchmark()`, является декоратором.

Объекты-декораторы

Напоследок стоит упомянуть, что не только функции, а любые вызываемые объекты могут быть декоратором. Экземпляры классов/объекты с методом `__call__()` тоже можно вызывать, поэтому их можно использовать в качестве декораторов. Эту функциональность можно использовать для создания декораторов, хранящих какое-то состояние. Например, вот декоратор для мемоизации:

```
from collections import deque
```

```
class Memoized:
```

```

def __init__(self, cache_size=100):
    self.cache_size = cache_size
    self.call_args_queue = deque()
    self.call_args_to_result = {}

def __call__(self, fn):
    def new_func(*args, **kwargs):
        memoization_key = self._convert_call_arguments_to_hash(args, kwargs)
        if memoization_key not in self.call_args_to_result:
            result = fn(*args, **kwargs)
            self._update_cache_key_with_value(memoization_key, result)
            self._evict_cache_if_necessary()
        return self.call_args_to_result[memoization_key]
    return new_func

def _update_cache_key_with_value(self, key, value):
    self.call_args_to_result[key] = value
    self.call_args_queue.append(key)

def _evict_cache_if_necessary(self):
    if len(self.call_args_queue) > self.cache_size:
        oldest_key = self.call_args_queue.popleft()
        del self.call_args_to_result[oldest_key]

    @staticmethod
    def _convert_call_arguments_to_hash(args, kwargs):
        return hash(str(args) + str(kwargs))

    @Memoized(cache_size=5)
    def get_not_so_random_number_with_max(max_value):
        import random
        return random.random() * max_value

```

Само собой, этот декоратор нужен в основном в демонстрационных целях, в реальном приложении для подобного кеширования стоит использовать [functools.lru_cache](#).

Заключение

Тут будут перечислены некоторые важные вещи, которые не были затронуты в статье или были затронуты вскользь. Вам может показаться, что они расходятся с тем, что было написано в статье до этого, но на самом деле это не так.

- Декораторы не обязательно должны быть функциями, это может быть любой вызываемый объект.
- Декораторы не обязаны возвращать функции, они могут возвращать что угодно. Но обычно мы хотим, чтобы декоратор вернул объект того же типа, что и декорируемый объект. Пример:

```
>>> def decorator(func):  
...     return 'sumit'  
...  
>>> @decorator  
... def hello_world():  
...     print('hello world')  
...  
>>> hello_world  
'sumit'
```

- Также декораторы могут принимать в качестве аргументов не только функции. [Здесь](#) можно почитать об этом подробнее.
- Необходимость в декораторах может быть не очевидной до написания библиотеки. Поэтому, если декораторы кажутся вам бесполезными, посмотрите на них с точки зрения разработчика библиотеки. Хорошим примером является декоратор представления в Flask.
- Также стоит обратить внимание на `functools.wraps()` — функцию, которая помогает сделать декорируемую функцию похожей на исходную, делая такие вещи, как сохранение docstring исходной функции.

Источники

1. [Декораторы в Python: понять и полюбить](#)
-

Revision #6

Created 30 May 2023 07:17:50 by Антон Сергеевич Абраменко

Updated 31 May 2023 04:01:26 by Антон Сергеевич Абраменко