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).

Основы декораторов Python

Итак, что же такое декораторы на самом деле? Они «декорируют» или «оборачивают» другую функцию и позволяют выполнять код до и после выполнения обернутой функции.

Декораторы позволяют определять повторно используемые модули, которые могут изменять или расширять поведение других функций. При этом они позволяют делать это без постоянного изменения самой обернутой функции. Поведение функции меняется только тогда, когда она *декорируется*.

Итак, как выглядит реализация простого декоратора? В общих чертах декоратор — это вызываемый объект, который принимает на вход вызываемый объект и возвращает другой вызываемый объект.

Следующая функция обладает этим свойством и может считаться самым простым декоратором, который только можно написать:

```
def null_decorator(func):
    return func
```

Как видите, null_decorator является вызываемым объектом, он принимает на вход другой вызываемый объект и возвращает тот же самый входной объект, не изменяя его.

Давайте используем его для декорирования (или обертывания) другой функции:

```
def greet():
    return 'Hello!'

greet = null_decorator(greet)

>>> greet()
'Hello!'
```

В этом примере определена функция greet, а затем сразу же декорирована, прогнав ее через функцию null_decorator. Пока это не выглядит чем-то очень полезным (мы ведь специально разработали декоратор null, чтобы он был бесполезным, верно?), но через некоторое время это прояснит, как работает синтаксис декораторов в Python.

Вместо того, чтобы явно вызывать null_decorator для greet, а затем переназначать переменную greet, можно использовать синтаксис Python @ для декорирования функции за один шаг:

```
@null_decorator
def greet():
    return 'Hello!'

>>> greet()
'Hello!'
```

Разместить строки @null_decorator перед определением функции — это то же самое, что сначала определить функцию, а затем применить к ней декоратор.

Использование синтаксиса @ — это просто синтаксический сахар и сокращение для этого часто используемого шаблона.

Обратите внимание, что использование синтаксиса @ декорирует функцию непосредственно во время определения. Это затрудняет доступ к недекорированному оригиналу без хрупких хаков. Поэтому вы можете декорировать некоторые функции вручную, чтобы сохранить возможность вызова недекорированной функции.

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.

Декораторы могут изменять поведение

Теперь, когда вы немного познакомились с синтаксисом декораторов, давайте напишем еще один декоратор, который *действительно* что-то делает и изменяет поведение декорируемой функции.

Вот немного более сложный декоратор, который преобразует результат декорированной функции в заглавные буквы:

```
def uppercase(func):
    def wrapper():
        original_result = func()
        modified_result = original_result.upper()
        return modified_result
    return wrapper
```

Вместо того, чтобы просто возвращать входную функцию, как это делал декоратор null, декоратор uppercase определяет новую функцию на лету (замыкание) и использует ее для обертывания входной функции, чтобы изменить ее поведение во время вызова.

Замыкание wrapper имеет доступ к недекорированной входной функции и может свободно выполнять дополнительный код до и после вызова входной функции. (Технически, ей вообще не нужно вызывать входную функцию).

Обратите внимание, что до сих пор декорированная функция никогда не выполнялась. На самом деле вызов входной функции в этот момент не имеет никакого смысла — декоратор должен иметь возможность изменять поведение своей входной функции, когда она будет вызвана.

Пришло время увидеть декоратор uppercase в действии. Что произойдет, если декорировать им исходную функцию greet?

```
@uppercase
def greet():
    return 'Hello!'

>>> greet()
'HELLO!'
```

Надеюсь, это был тот результат, которого вы ожидали. Давайте рассмотрим подробнее, что здесь произошло. В отличие от null_decorator, декоратор uppercase возвращает *другой объект функции*, когда он декорирует функцию:

```
>>> greet
<function greet at 0x10e9f0950>

>>> null_decorator(greet)
<function greet at 0x10e9f0950>

>>> uppercase(greet)
<function uppercase.<locals>.wrapper at 0x10da02f28>
```

Как вы видели ранее, это необходимо для того, чтобы изменить поведение декорированной функции, когда она будет вызвана. Декоратор uppercase сам является функцией. И единственный способ повлиять на «будущее поведение» входной функции, которую он декорирует, — это заменить (или обернуть) входную функцию замыканием.

Вот почему uppercase определяет и возвращает другую функцию (замыкание), которую можно вызвать позднее, запустить исходную входную функцию и изменить ее результат.

Декораторы изменяют поведение вызываемого объекта с помощью обертки, поэтому вам не нужно постоянно изменять оригинал. Вызываемый объект не подвергается постоянным изменениям — ее поведение меняется только при декорировании.

Давайте взглянем на другие, более полезные:

```
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.

Декорирование функций, принимающих аргументы

Все примеры до сих пор декорировали только простую нульарную функцию greet, которая не принимала никаких аргументов. Поэтому декораторы, которые вы видели здесь до сих пор, не имели дела с передачей аргументов во входную функцию.

Если вы попытаетесь применить один из этих декораторов к функции, принимающей аргументы, он будет работать неправильно. Как декорировать функцию, принимающую произвольные аргументы?

Здесь на помощь приходит функция Python *args *и* **kwargs для работы с переменным количеством аргументов. Декоратор proxy использует эту возможность:

```
def proxy(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

В этом декораторе есть два примечательных момента:

- Он использует операторы * и ** в определении замыкания wrapper для сбора всех позиционных и ключевых аргументов и хранения их в переменных (args и kwargs).

- Затем замыкание wrapper передает собранные аргументы исходной входной функции с помощью операторов «распаковки аргументов» * и **.

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)
```

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.