Лабораторная работа 2.12 Декораторы функций в языке Python

Цель работы: *приобретение навыков по работе с декораторами функций при написании программ с помощью языка программирования Python версии 3.x.*

Ход работы

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.

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).

Примеры

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`.

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
```

Просто добавив `@decorator_function` перед определением функции `hello_world()`, мы модифицировали её поведение. Однако 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.

Давайте взглянем на другие, более полезные декораторы:

```
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.

Аппаратура и материалы

1. Компьютерный класс общего назначения с конфигурацией ПК не хуже рекомендованной для ОС Windows 10 с подключением к глобальной сети Интернет.
2. Операционная система Windows 10.
3. Система контроля версий Git.
4. Браузер для доступа к web-сервису GitHub, рекомендован к использованию Google Chrome.
5. Дистрибутив языка программирования Python, включающий набор популярных библиотек Anaconda.
6. Интегрированная среда разработки PyCharm Community Edition.

Указания по технике безопасности

При работе на ЭВМ без разрешения руководителя занятия запрещается:

- подавать (снимать) напряжение на ПЭВМ и электрические розетки с распределительного щита;
- включать и выключать блоки питания ПЭВМ и мониторы;

- извлекать ПЭВМ из защитного кожуха;
- устранять неисправности, возникшие в ходе выполнения лабораторной работы.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.
3. Выполните клонирование созданного репозитория.
4. Дополните файл `.gitignore` необходимыми правилами для работы с IDE PyCharm.
5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создайте проект PyCharm в папке репозитория.
7. Проработать примеры лабораторной работы.
8. Выполнить индивидуальное задание.
9. Зафиксируйте изменения в репозитории.
10. Добавьте отчет по лабораторной работе в *формате PDF* в папку *doc* репозитория. Зафиксируйте изменения.
11. Выполните слияние ветки для разработки с веткой *master/main*.
12. Отправьте сделанные изменения на сервер GitHub.
13. Отправьте адрес репозитория GitHub на электронный адрес преподавателя.

Индивидуальное задание

Составить программу с использованием замыканий для решения задачи. Номер варианта определяется по согласованию с преподавателем.

1. Объявите функцию с именем `get_sq`, которая вычисляет площадь прямоугольника по двум параметрам: `width` и `height` – ширина и высота прямоугольника и возвращает результат. Определите декоратор для этой функции с именем (внешней функции) `func_show`, который отображает результат на экране в виде строки (без кавычек): "Площадь прямоугольника: <значение>". Вызовите декорированную функцию `get_sq`.
2. На вход программы поступает строка из целых чисел, записанных через пробел. Напишите функцию `get_list`, которая преобразовывает эту строку в список из целых чисел и возвращает его. Определите декоратор для этой функции, который сортирует список чисел, полученный из вызываемой в нем функции. Результат сортировки должен возвращаться при вызове декоратора. Вызовите декорированную функцию get_list и отобразите полученный отсортированный список на экране.
3. Вводятся два списка (каждый с новой строки) из слов, записанных через пробел. Имеется функция, которая преобразовывает эти две строки в два списка слов и возвращает эти списки. Определите декоратор для этой функции, который из этих двух списков формирует словарь, в котором ключами являются слова из первого списка, а значениями – соответствующие элементы из второго списка. Полученный словарь должен возвращаться при вызове декоратора. Примените декоратор к первой функции и вызовите ее. Результат (словарь) отобразите на экране.
4. Объявите функцию с именем `to_lat`, которая принимает строку на кириллице и преобразовывает ее в латиницу, используя следующий словарь для замены русских букв на соответствующее латинское написание:

```
t = {'ё': 'yo', 'а': 'a', 'б': 'b', 'в': 'v', 'г': 'g', 'д': 'd', 'е': 'e',
'ж': 'zh',
'з': 'z', 'и': 'i', 'й': 'y', 'к': 'k', 'л': 'l', 'м': 'm', 'н': 'n', 'о':
'o', 'п': 'p',
'р': 'r', 'с': 's', 'т': 't', 'у': 'u', 'ф': 'f', 'х': 'h', 'ц': 'c', 'ч':
'ch', 'ш': 'sh',
'щ': 'shch', 'ъ': '', 'ы': 'y', 'ь': '', 'э': 'e', 'ю': 'yu', 'я': 'ya'}
```

Функция должна возвращать преобразованную строку. Замены делать без учета регистра (исходную строку перевести в нижний регистр – малые буквы). Все небуквенные символы `"! ?:;.,_"` превращать в символ '-' (дефиса). Определите декоратор для этой функции, который несколько подряд идущих дефисов, превращает в один дефис. Полученная строка должна возвращаться при вызове декоратора. Примените декоратор к функции `to_lat` и вызовите ее Результат работы декорированной функции отобразите на экране.

5. Вводится строка целых чисел через пробел. Напишите функцию, которая преобразовывает эту строку в список чисел и возвращает их сумму. Определите декоратор для этой функции, который имеет один параметр `start` – начальное значение суммы. Примените декоратор со значением `start=5` к функции и вызовите декорированную функцию. Результат отобразите на экране.
6. Объявите функцию, которая возвращает переданную ей строку в нижнем регистре (с малыми буквами). Определите декоратор для этой функции, который имеет один параметр `tag`, определяющий строку с названием тега (начальное значение параметра `tag` равно `h1`). Этот декоратор должен заключать возвращенную функцией строку в тег `tag` и возвращать результат. Пример заключения строки "python" в тег h1: `<h1>python</h1>`. Примените декоратор со значением `tag="div"` к функции и вызовите декорированную функцию. Результат отобразите на экране.
7. Объявите функцию, которая вычисляет периметр многоугольника и возвращает вычисленное значение. Длины сторон многоугольника передаются в виде коллекции (списка или кортежа). Определите декоратор для этой функции, который выводит на экран сообщение: «Периметр фигуры равен = <число>». Примените декоратор к функции и вызовите декорированную функцию.
8. Объявите функцию, которая вычисляет площадь круга и возвращает вычисленное значение. В качестве аргумента ей передается значение
радиуса. Определите декоратор для этой функции, который выводит на экран сообщение: «Площадь круга равна = <число>». В строке выведите числовое значение с точностью до сотых. Примените декоратор к функции и вызовите декорированную функцию.
9. Объявите функцию, которая принимает строку на кириллице и преобразовывает ее в латиницу, используя следующий словарь для замены русских букв на соответствующее латинское написание:

```
t = {'ё': 'yo', 'а': 'a', 'б': 'b', 'в': 'v', 'г': 'g', 'д': 'd', 'е': 'e',
'ж': 'zh',
'з': 'z', 'и': 'i', 'й': 'y', 'к': 'k', 'л': 'l', 'м': 'm', 'н': 'n', 'о':
'o', 'п': 'p',
'р': 'r', 'с': 's', 'т': 't', 'у': 'u', 'ф': 'f', 'х': 'h', 'ц': 'c', 'ч':
'ch', 'ш': 'sh',
'щ': 'shch', 'ъ': '', 'ы': 'y', 'ь': '', 'э': 'e', 'ю': 'yu', 'я': 'ya'}
```

Функция должна возвращать преобразованную строку. Замены делать без учета регистра (исходную строку перевести в нижний регистр – малые буквы). Определите декоратор с параметром `chars` и начальным значением `" !?"`, который данные символы преобразует в символ `"-"` и, кроме того, все подряд идущие дефисы (например, `"--"` или `"---"`) приводит к одному дефису. Полученный результат должен возвращаться в виде строки. Примените декоратор со значением `chars="?!:;,. "` к функции и вызовите декорированную функцию. Результат отобразите на экране.

10. Объявите функцию, которая принимает строку, удаляет из нее все подряд идущие пробелы и переводит ее в нижний регистр – малые буквы. Результат (строка) возвращается функцией. Определите декоратор, который строку, возвращенную функцией, переводит в азбуку Морзе, используя следующий словарь для замены русских букв и символа пробела на соответствующие последовательности из точек и тире:

```
morze = {'а': '.-', 'б': '-...', 'в': '.--', 'г': '--.', 'д': '-..', 'е':
'.', 'ё': '.', 'ж': '...-', 'з': '--..', 'и': '..', 'й':'.---', 'к': '-.-',
'л': '.-..', 'м': '--', 'н': '-.', 'о': '---', 'п': '.--.', 'р': '.-.', 'с':
'...', 'т': '-', 'у': '..-', 'ф': '..-.', 'х': '....', 'ц': '-.-.', 'ч': '--
-.', 'ш': '----', 'щ': '--.-', 'ъ': '--.--', 'ы': '-.--', 'ь': '-..-', 'э':
'..-..', 'ю': '..--', 'я': '.-.-', ' ': '-...-'}
```

Преобразованная строка возвращается декоратором. Примените декоратор к функции и вызовите декорированную функцию. Результат работы отобразите на экране.

Содержание отчета и его форма

Отчет по лабораторной работе оформляется электронно в формате PDF, должен содержать ответы на контрольные вопросы, ссылку на репозиторий с которым выполнялась работа, скриншоты IDE PyCharm, скриншоты результатов работы программ.

Вопросы для защиты работы

1. Что такое декоратор?
2. Почему функции являются объектами первого класса?
3. Каково назначение функций высших порядков?
4. Как работают декораторы?
5. Какова структура декоратора функций?
6. Самостоятельно изучить как можно передать параметры декоратору, а не декорируемой функции?