***Замыкания и Декораторы***

**Замыкания**

Замыкание *(****closure****)*, это такая функция, которая ссылается на локальные переменные (использует их в своём теле) в области видимости, в которой она была создана. Этим замыкание отличается от обычной функции, которая может использовать только свои аргументы и глобальные переменные.

Функция, определенная внутри другой функции, называется вложенной функцией. Вложенные функции могут получать доступ к переменным из локальной области видимости объемлющих функций (**enclosing scope**).

В Python нелокальные переменные по умолчанию доступны только для чтения. Если нам необходимо их модифицировать, то мы должны объявить их явно как нелокальные (используя ключевое слово **nonlocal**).

**Пример 1**

***def print\_msg(msg):***

***# объемлющая функция***

***def printer():***

***# вложенная функция***

***print(msg)***

***printer()***

***print\_msg("Hello")***

Мы видим, что вложенная функция printer() смогла получить доступ к нелокальной переменной **msg** в функции **print\_msg(msg).**

Что произойдет в приведенном выше примере, если последняя строка функции print\_msg() вернет функцию printer() вместо ее вызова?

**Пример 2**

***def print\_msg(msg):***

***def printer():***

***print(msg)***

***return printer # возвращаем вложенную функцию***

***another = print\_msg("Hello")***

***another()***

Функция print\_msg() вызывалась со строкой «Hello», а возвращаемая функция была присвоена переменной another. При вызове another() сообщение все еще сохранялось в памяти, хотя мы уже закончили выполнение функции print\_msg().

**Пример 3**

***def print\_msg(name):***

***def printer():***

***print("hello ",name)***

***return printer***

***another = print\_msg("Kirill")***

***another()***

Как видно из приведенного выше примера, мы пользуемся замыканием в Python, когда вложенная функция ссылается на значение из локальной области видимости функции в которой она находится.

Критерии, которые должны быть выполнены для создания замыкания в Python, изложены в следующих пунктах:

-У нас должна быть вложенная функция (функция внутри функции).

-Вложенная функция должна ссылаться на значение, определенное в объемлющей функции.

-Объемлющая функция должна возвращать вложенную функцию.

Замыкания позволяют избежать использования глобальных (global) значений и обеспечивают некоторую форму сокрытия данных. Для этого также может использоваться объектно-ориентированный подход.

Вот простой пример, где замыкание может быть более предпочтительным, чем определение класса и создание объектов. Но выбор остается за вами.

**Пример 4**

***def make\_multiplier\_of(n):***

***def multiplier(x):***

***return x \* n***

***return multiplier***

***times3 = make\_multiplier\_of(3)***

***times5 = make\_multiplier\_of(5)***

***time3(4)# 12***

***time5(4)# 20***

Ещё один пример решение задачи с помощью замыкания.

**Пример 5**

***def print\_count(x):***

***def count(a):***

***nonlocal x***

***x+=1***

***print(a+x)***

***return count # возвращаем вложенную функцию***

***f1 = print\_count(7)***

***f1(2)# 10***

***f1(2)# 11***

***f1(2)# 12***

В области видимости вложенной функции **count()** мы не можем изменить переменную х которая видна в функции **print\_count()** просто так. Однако с помощью ключевого слова **nonlocal** мы сможем это сделать, тем самым реализуем замыкание в области видимости функции **print\_count().**

**Декораторы функций**

В Python все является объектом. И функции — не исключение. Поскольку они тоже являются объектами, их можно передавать в качестве аргументов другим функциям и возвращать в качестве результата.

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

**Пример 6**

***def my\_decorator(func):***

***def wrapper():***

***print("До вызова функции.")***

***func()***

***print("После вызова функции.")***

***return wrapper***

***def say\_whee():***

***print("Ура!")***

***say\_whee = my\_decorator(say\_whee) # До вызова функции. Ура! После вызова функции.***

Итак декорирование происходит в последней строчке:

***say\_whee = my\_decorator(say\_whee)***

Мы передаем в функцию **my\_decorator()** ссылку на функцию **say\_whee.** В **my\_decorator()** есть внутренняя функция **wrapper(),** ссылка на которую возвращается в инструкции return внешней функции. В результате мы передали в **my\_decorator()** в качестве аргумента ссылку на одну функцию, а назад получили ссылку на её функцию-обёртку и записали её в функцию **say\_whee.**

Теперь имя say\_whee указывает на внутреннюю функцию wrapper:

Рассмотрим второй пример, иллюстрирующий динамическое поведение декораторов. Сделаем так, чтобы наша функция кричала "Ура!" только в дневное время.

**Пример 7**

***from datetime import datetime***

***def not\_during\_the\_night(func):***

***def wrapper():***

***if 8 <= datetime.now().hour < 22:***

***func()***

***else:***

***pass # Тише, соседи спят!***

***return wrapper***

***def say\_whee():***

***print("Ура!")***

***say\_whee = not\_during\_the\_night(say\_whee)***

То, как мы декорировали **say\_whee(),** прямо скажем, выглядит неуклюже. В последнем примере мы три раза использовали имя say\_whee: при определении функции-оригинала, при передаче ссылку в функцию **not\_during\_the\_night()** и при переопределении имени для создания ссылки на декоратор.

Чтобы не заниматься такими глупостями, в Python можно создать декоратор с помощью символа **@.** Следующий код эквивалентен первому рассмотренному примеру:

**Пример 8**

***def my\_decorator(func):***

***def wrapper():***

***print("До вызова функции.")***

***func()***

***print("После вызова функции.")***

***return wrapper***

***@my\_decorator***

***def say\_whee():***

***print("Ура!")***

То есть инструкция **@my\_decorator**, идущая перед определением функции say\_whee() эквивалентна инструкции

**say\_whee =my\_decorator(say\_whee).**

Можно использовать тот же декоратор с любым количеством функций, а также декорировать функцию любым декоратором. В том числе можно декорировать функции любым числом декораторов.

**Пример 9**

***def decorator\_1(func):***

***print('декоратор 1')***

***def wrapper():***

***print('перед функцией')***

***func()***

***return wrapper***

***def decorator\_2(func):***

***print('декоратор 2')***

***def wrapper():***

***print('перед функцией')***

***func()***

***return wrapper***

***@decorator\_1***

***@decorator\_2***

***def basic\_1():***

***print('basic\_1')***

***@decorator\_1***

***def basic\_2():***

***print('basic\_2')***

Когда у функции несколько декораторов, они вызываются в обратном порядке относительно того, как были вызваны. То есть, такой вызов:

***@decorator\_1***

***@decorator\_2***

***def wrapped():***

***Равен следующему:***

***a = decorator\_1(decorator\_2(wrapped))***

Если декорированная функция возвращает значение, и его нужно сохранить, то нужно сделать так, чтобы его возвращала и функция-обертка.

**Декоратор функции с аргументами**

А что если функция, которую требуется декорировать, должна получать аргументы? Для этого нужно вернуть функцию с той же сигнатурой, что и у декорируемой.

**Пример 10**

***def decorator\_with\_args(func):***

***print('> декоратор с аргументами...')***

***def decorated(a, b):***

***print('до вызова функции', func.\_\_name\_\_)***

***ret = func(a, b)***

***print('после вызова функции', func.\_\_name\_\_)***

***return ret***

***return decorated***

***@decorator\_with\_args***

***def add(a, b):***

***print('функция 1')***

***return a + b***

***@decorator\_with\_args***

***def sub(a, b):***

***print('функция 2')***

***return a - b***

***r = add(10, 5)***

***print('r:', r)***

***g = sub(10, 5)***

***print('g:', g)***

Можно использовать **\*args** и **\*\*kwargs** и для функции **wrapper**, если сигнатура заранее неизвестна, или будут приниматься разные типы функций.

**Пример 11**

***def decor\_with\_return(fn):***

***def wrapper(\*args, \*\*kwargs):***

***print("Run method: " + str(fn.\_\_name\_\_))***

***return fn(\*args, \*\*kwargs)***

***return wrapper***

**Декораторы с аргументами**

Мы также можем создавать декораторы, которые принимают аргументы.

**Пример 12**

***def info(arg1, arg2):***

***print('Decorator arg1 = ' + str(arg1))***

***print('Decorator arg2 = ' + str(arg2))***

***def the\_real\_decorator(function):***

***def wrapper(\*args, \*\*kwargs):***

***print('Function {} args: {} kwargs: {}'.format(function.\_\_name\_\_, str(args), str(kwargs)))***

***return function(\*args, \*\*kwargs)***

***return wrapper***

***return the\_real\_decorator***

***def doubler(number):***

***return number \* 2***

***decorator = info(3, 'Python')(doubler)***

***print(decorator(5))***

***decorator\_function = info(3, 'Python')***

***actual\_decorator = decorator\_function(doubler)***

***# Вызываем декорированную функцию***

***print(actual\_decorator(5))***

Сначала мы получаем объект функции декоратора. После этого мы получаем объект декоратора, который первым вложен в **info()** и называется **the\_real\_decorator().** Ему мы передаем функцию, которую нужно декорировать. Теперь у нас есть декорированная функция, и в последней строке мы просто вызываем ее.

Но как вы уже знаете мы можем задекорировать функцию намного быстрее и проще используя синтаксис языка.

**Пример 13**

***@info(3, 'Python')***

***def doubler(number):***

***return number \* 2***

***print(doubler(5))***

***Примеры использования Декораторов***

**Обработка ошибок**

Можно убедиться, что обрабатываются определенные типы ошибок без использования блока try для каждой функции. Результат после этого добавляется в лог или останавливает выполнение программы.

***def error\_handler(func):***

***def wrapper(\*args, \*\*kwargs):***

***ret = 0***

***try:***

***ret = func(\*args, \*\*kwargs)***

***except:***

***print('>> Ошибка в функции', func.\_\_name\_\_)***

***return ret***

***return wrapper***

***@error\_handler***

***def div(a, b):***

***return a / b***

***print('старт')***

***print(div(10, 2))***

***print(div(10, 0))***

***print('конец')***

**Таймеры**

Базовая функциональность — время работы функции. Есть возможность получить время до и после вызова функции, использовав полученный результат (для записи в лог, базу данных, для отладки и так далее).

***from datetime import datetime***

***import time***

***def elapsed(func):***

***def wrapper(a, b, delay=0):***

***start = datetime.now()***

***func(a, b, delay)***

***end = datetime.now()***

***elapsed = (end - start).total\_seconds() \* 1000***

***print(f'>> функция {func.\_\_name\_\_} время выполнения (ms): {elapsed}')***

***return wrapper***

***@elapsed***

***def add\_with\_delay(a, b, delay=0):***

***print('сложить', a, b, delay)***

***time.sleep(delay)***

***return a + b***

***print('старт программы')***

***add\_with\_delay(10, 20)***

***add\_with\_delay(10, 20, 1)***

***print('конец программы')***

**Логи и авторизация**

Еще один распространенный сценарий применения для декоратора — логирование функций.

***import logging***

***def logger(func):***

***log = logging.getLogger(\_\_name\_\_)***

***def wrapper(a, b):***

***log.info("Вызов функции ", func.\_\_name\_\_)***

***ret = func(a, b)***

***log.info("Вызвана функция ", func.\_\_name\_\_)***

***return ret***

***return wrapper***

***@logger***

***def add(a, b):***

***print('a + b:', a + b)***

***return a + b***

***print('>> старт')***

***add(10, 20)***

***add(20, 30)***

***print('>> конец')***

Декораторы могут использоваться в веб-приложениях для проверки авторизации пользователя перед тем, как открывать ему доступ к функционалу. Они активно используются в веб-фреймворках Flask и Django. Вот пример проверки авторизации на декораторах:

***from functools import wraps***

***def requires\_auth(f):***

***@wraps(f)***

***def decorated(\*args, \*\*kwargs):***

***auth = request.authorization***

***if not auth or not check\_auth(auth.username, auth.password):***

***authenticate()***

***return f(\*args, \*\*kwargs)***

***return decorated***

**Проверка состояний**

Декораторы можно использовать для проверки состояния перед выполнение функции: например, зарегистрирован ли пользователь, есть ли у него достаточное количество прав или валидны ли аргументы (типы, значения и так далее).

***user\_permissions = ["user"]***

***def check\_permission(permission):***

***def wrapper\_permission(func):***

***def wrapped\_check():***

***if permission not in user\_permissions:***

***raise ValueError("Недостаточно прав")***

***return func()***

***return wrapped\_check***

***return wrapper\_permission***

***@check\_permission("user")***

***def check\_value():***

***return "значение"***

***@check\_permission("admin")***

***def do\_something():***

***return "только админ"***

***print('старт программы')***

***check\_value()***

***do\_something()***

***print('конец программы')***

***Вывод:***

***Traceback (most recent call last):***

***File "C:\Programs\Python\Python38-32\test.py", line 22, in <module>***

***do\_something()***

***File "C:\Programs\Python\Python38-32\test.py", line 7, in wrapped\_check***

***raise ValueError("Недостаточно прав")***

***ValueError: Недостаточно прав***

**Функция обратного вызова**

Функция обратного вызова — это функция, которая вызывается при срабатывании определенного события (переходе на страницу, получении сообщения или окончании обработки процессором).

Можно передать функцию, чтобы она выполнилась после определенного события. Это используется, например, в HTTP-серверах в ответ на URL-запросы.

***app = {}***

***def callback(route):***

***def wrapper(func):***

***app[route] = func***

***def wrapped():***

***ret = func()***

***return ret***

***return wrapped***

***return wrapper***

***@callback('/')***

***def index():***

***print('index')***

***return 'OK'***

***print('> старт')***

***route = app.get('/')***

***if route:***

***resp = route()***

***print('ответ:', resp)***

***print('> конец')***

**Задачи**

1. Дана функция, которая выводит все простые числа в промежутке от 1 до 100. Написать декоратор который будет проверять время работы этой функции. Задекорировать функцию. Вывести время работы этой функции, а так же сами простые числа.
2. Дана функция, которая проверяет введённый пользователем пароль. Задекорировать её так, чтобы при правильно введённом пароле она приветствовала админа и открывала файл с документацией 2.txt.
3. Дана функция, которая получает имя пользователя. Задекорировать эту функцию так, чтобы в зависимости от времени суток. Программа отправляла пользователя на отдых или на работу.
4. Дана функция, которая конкатенирует 2 строки. С помощью декоратора проверить эту функцию на возможные ошибки.
5. Дана функция которая запрашивает у пользователя определённые данные для регистрации на портале и запоминает их. Напишите декоратор, который будет засекать время проведённое пользователем на портале при регистрации.