Глава 7. *Декораторы и функции замыкания*

Декоратор - это вызываемый объект, принимающий в качестве аргумента другую функцию.

Главное, что нужно знать о декораторах:

- Тот факт, что они властны заменить декорируемую функцию другой;
- Выполняется сразу после загрузки модуля;

Паттерн Стратегия, дополненный декоратором

Список `promos` *заполняется декоратором* `promotion`

```
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product                      # Наименование
        self.quantity = quantity                    # Количество
        self.price = price                          # Цена

    def total(self):
        return self.price * self.quantity           # Общая стоимость позиции


class Order:                                        # Контекст

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = cart
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount
```

```
    def __repr__(self):
        return f'<Order total: {self.total():.2f} due: {self.due():.2f}>'


promos = []


def promotion(promo_func):
    promos.append(promo_func)
    return promo_func


@promotion
def fidelity(order):
    """
    5%-я скидка для заказчиков, имеющих не менее 1000 баллов лояльности
    """
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0


@promotion
def bulk_item(order):
    """
    10%-я скидка для каждой позиции LineItem, в которой заказано не мене 20 единиц
    """
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount


@promotion
def large_order(order):
    """
    7%-я скидка для заказов, включающих не менее 10 различных позици
    """
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0


def best_promo(order):
    """
    Выбрать максимально возможную скидку
    """
    return max(promo(order) for promo in promos)
```

По сравнению с другими решениями, у этого есть несколько преимуществ:

- Функции, реализующие стратегии вычисления скидки могут избавиться от суффикса `_promo`
- Декоратор `@promotion` явно описывает назначение декорируемой функции и без труда позволяет временно отменить предоставление ссылки: достаточно закомментировать декоратор
- Стратегии скидки можно определить в других модулях.

Правила видимости переменных

Видимость локальных переменных определяется при компилировании байт-кода и если одноименная переменная определена в теле функции, то она будет считаться локальной.

Замыкания

Замыкание вступает в игру только при наличии вложенной функции.

Замыкание — это функция с расширенной областью видимости, которая охватывает все не глобальные переменные, имеющие ссылки в теле функции, хотя они в нем не определены.

Эту идею довольно трудно переварить, поэтому пример.

Рассмотрим функцию `avg`, которая вычисляет среднее продолжающегося ряда чисел, например, среднюю цену закрытия биржевого товара за всю историю торгов. Каждый день ряд пополняется новой ценой, а при вычислении среднего учитываются все прежние цены.

Если начать с чистого листа, то функция `avg` можно было бы использовать следующим образом:

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Вопрос на подумать

Откуда берется `avg` и где она хранит предыдущие значения?

Реализация Average основанная на классах.

average_oo.py: класс для вычисления накопительного среднего значения

```
class Averager:

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

Класс `Averager` *создает вызываемые объекты*

```
>>> from source.average_oo import Averager
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Результат тестирования

```
>>> import doctest
>>> doctest.testfile('./source/doctest/avg_oo.txt')
TestResults(failed=0, attempted=5)
```

Функциональная реализация с использованием функции высшего порядка `make_averager`

```
def make_averager():
    """
    При обращении к make_averager возвращается объект-функция averager.
    При каждом вызове averager добавляет переданный аргумент в конец
    списка series и вычисляет текущее среднее.
    :return:
    """
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

Обратите внимание на сходство обоих примеров: мы обращаемся к `Averager` и к `make_averager` что бы получить вызываемый объект `avg`, который обновляет временной ряд и вычисляет среднее значение.

Совершенно ясно, где хранит историю объектов `avg` класса `Averager`: в аттрибуте экземпляра `self.series`. Но где находится `series` функции `avg` из второго примера?

Внутри `averager` переменная `series` является *свободной*. Этот технический термин обозначает, что переменная не связана в локальной области видимости.

Python хранит имена локальных и свободных переменных в аттрибуте `__code__`, который представляет собой откомпилированное тело функции.

Инспекция функции, созданной функцией `make_averager`

```
>>> from source.average import make_averager
>>> avg = make_averager()
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

Привязка переменной `series` хранится в аттрибуте `__closure__` возвращенной функцией `avg`.

Каждому элементу `avg.__closure__` соответствует имя в `avg.__code__.co_freevars`. Эти элементы называются *ячейками(cells)*, и у каждого из них есть атрибут `cell_contents`, где можно найти само значение.

Инспекция функции, созданной функцией `make_averager` *(продолжение)*

```
>>> avg.__closure__
(<cell at 0x040DC358: list object at 0x039FA368>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

Резюмируем:

Замыкание — это функция, которая запоминает привязку свободных переменных, существовавшие на момент определения функции. Так что их можно использовать впоследствии при вызове функции, когда область видимости, в которой она была определена уже не существует.

Единственная ситуация, когда функции может понадобиться доступ к внешней не глобальной переменной,- это когда она вложена в другую функцию.

Объявление `nonlocal`

Приведенная ранее реализация `make_averager` не эффективна. Мы храним в переменной все значения и каждый раз вычисляем их сумму при каждом вызове `averager`. Лучше было бы хранить предыдущую сумму и количество элементов, тогда зная два числа мы можем вычислить среднее.

В Pyton 3 было добавлено nonlocal

nonlocal позволяет пометить переменную как свободную, даже если ей присваивается значение внутри функции. В таком случае изменяется привязка, хранящаяся в замыкании.

Правильная реализация идеи

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal total, count
        count += 1
        total += new_value
        return total/count

    return averager
```

Реализация простого декоратора

Простой декоратор для вывода времени выполнения функции

```
import functools
import time


def clock(func):
    """
        Декоратор functools.wraps копирует аргументы
        декорируемой функции.
    """
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_list = []
        if args:
            arg_list.append(', '.join(repr(arg) for arg in args))
        if kwargs:
            pairs = [f'{k}={w}' for k, w in sorted(kwargs.items())]
            arg_list.append(', '.join(pairs))
        arg_string = ', '.join(arg_list)
        print(f'[{elapsed:0.8f}] {name}({arg_string} -> {result})')     # [0.00000120]
factorial(1 -> 1)
        return result
    return clocked
```

Использование декоратора clock

```
# clock_decorator_demo.py
import time
from clock_decorator import clock


@clock
def snooze(seconds):
    time.sleep(seconds)


@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)


if __name__ == '__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(5)')
    print('5! =', factorial(5))
```

Результат выполнения

```
PS C:\Users\User\PycharmProjects\Python. Design patterns and other\source> py
.\clock_decorator_demo.py
**************************************** Calling snooze(.123)
[0.13673660] snooze(0.123 -> None)
**************************************** Calling factorial(5)
[0.00000120] factorial(1 -> 1)
[0.00017130] factorial(2 -> 2)
[0.00031230] factorial(3 -> 6)
[0.00047800] factorial(4 -> 24)
[0.00066200] factorial(5 -> 120)
5! = 120
```

Декораторы в стандартной библиотеке

Два самых любопытных декоратора в стандартной библиотеке - `lru_cache` и совсем новый `singledispatch` *(Python >= 3.4)*, оба определены в `functools`.

Кэширование с помощью `functools.lru_cache`

Декоратор `functools.lru_cache` очень полезен на практике

Он реализует "запоминание" (memorization): прием оптимизации, смысл которого заключается в запоминании дорогостоящих вычислений, позволяет избежать повторных вычислений с теми же аргументами, что и раньше.

Пример использования кэширования

```
import functools
from contextlib import redirect_stdout
from clock_decorator import clock


#@functools.lru_cache()
@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)


if __name__ == '__main__':
    with open('./doctest/fibo_demo_out.txt', 'a') as f:
        with redirect_stdout(f):
            print(fibonacci(10))
```

На примере функция уже декорирована. Для сравнения, вот выводы декорированной и не декорированной функции `fibonacci`:

Вывод скрипта без использования `lru_cache`. Очевидны лишние вычисления.

```
[0.00000050] fibonacci(0 -> 0)
[0.00000040] fibonacci(1 -> 1)
[0.00003930] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000870] fibonacci(2 -> 1)
[0.00001650] fibonacci(3 -> 2)
[0.00006430] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000720] fibonacci(2 -> 1)
[0.00001430] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
```

```
[0.00000240] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00001180] fibonacci(2 -> 1)
[0.00001890] fibonacci(3 -> 2)
[0.00003320] fibonacci(4 -> 3)
[0.00005450] fibonacci(5 -> 5)
[0.00012600] fibonacci(6 -> 8)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00001410] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001380] fibonacci(3 -> 2)
[0.00002810] fibonacci(4 -> 3)
[0.00004980] fibonacci(5 -> 5)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00001460] fibonacci(3 -> 2)
[0.00002830] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000720] fibonacci(2 -> 1)
[0.00001410] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000740] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000050] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000810] fibonacci(2 -> 1)
[0.00001680] fibonacci(3 -> 2)
[0.00003080] fibonacci(4 -> 3)
[0.00005170] fibonacci(5 -> 5)
[0.00008660] fibonacci(6 -> 8)
[0.00014380] fibonacci(7 -> 13)
[0.00027750] fibonacci(8 -> 21)
[0.00000030] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
```

```
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001400] fibonacci(3 -> 2)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000890] fibonacci(2 -> 1)
[0.00001620] fibonacci(3 -> 2)
[0.00003080] fibonacci(4 -> 3)
[0.00005150] fibonacci(5 -> 5)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000720] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00002470] fibonacci(2 -> 1)
[0.00003160] fibonacci(3 -> 2)
[0.00004530] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001380] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00001540] fibonacci(2 -> 1)
[0.00002220] fibonacci(3 -> 2)
[0.00003610] fibonacci(4 -> 3)
[0.00005660] fibonacci(5 -> 5)
[0.00010850] fibonacci(6 -> 8)
[0.00016680] fibonacci(7 -> 13)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00001430] fibonacci(3 -> 2)
[0.00002820] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
```

```
[0.00000700] fibonacci(2 -> 1)
[0.00001410] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001390] fibonacci(3 -> 2)
[0.00002730] fibonacci(4 -> 3)
[0.00004810] fibonacci(5 -> 5)
[0.00008580] fibonacci(6 -> 8)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000830] fibonacci(2 -> 1)
[0.00001530] fibonacci(3 -> 2)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000030] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00001580] fibonacci(3 -> 2)
[0.00003140] fibonacci(4 -> 3)
[0.00005330] fibonacci(5 -> 5)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000710] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000030] fibonacci(1 -> 1)
[0.00000700] fibonacci(2 -> 1)
[0.00001440] fibonacci(3 -> 2)
[0.00002800] fibonacci(4 -> 3)
[0.00000020] fibonacci(1 -> 1)
[0.00000020] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000730] fibonacci(2 -> 1)
[0.00001400] fibonacci(3 -> 2)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00000930] fibonacci(2 -> 1)
[0.00000020] fibonacci(1 -> 1)
[0.00000030] fibonacci(0 -> 0)
[0.00000020] fibonacci(1 -> 1)
[0.00003070] fibonacci(2 -> 1)
[0.00003770] fibonacci(3 -> 2)
[0.00005370] fibonacci(4 -> 3)
```

```
[0.00007440] fibonacci(5 -> 5)
[0.00010920] fibonacci(6 -> 8)
[0.00016910] fibonacci(7 -> 13)
[0.00026170] fibonacci(8 -> 21)
[0.00043520] fibonacci(9 -> 34)
[0.00072080] fibonacci(10 -> 55)
55
```

Вывод скрипта с `lru_cache`.

```
[0.00000140] fibonacci(0 -> 0)
[0.00000040] fibonacci(1 -> 1)
[0.00005920] fibonacci(2 -> 1)
[0.00000070] fibonacci(3 -> 2)
[0.00007080] fibonacci(4 -> 3)
[0.00000060] fibonacci(5 -> 5)
[0.00008260] fibonacci(6 -> 8)
[0.00000060] fibonacci(7 -> 13)
[0.00009310] fibonacci(8 -> 21)
[0.00000070] fibonacci(9 -> 34)
[0.00010440] fibonacci(10 -> 55)
55
```

`lru_cache` необходимо вызывать как функцию со скобками. `functools.lru_cache()`. Причина в том, что декоратор принимает конфигурационные параметры.

Полная сигнатура

`functools.lru_cache(max_size=128, typed=False)`

- `maxsize` — сколько результатов хранить (*для достижения результата maxsize* = n^2).
- `type` - если стоит True, то результаты разных типов будут храниться порознь.

Одиночная диспетчеризация и обобщенные функции

`functools.singledispatch` - (Python >= 3.4) позволяет каждому модулю вносить свой вклад в общее решение, так, что пользователь может легко добавить специализированную функцию, даже не имея возможности изменить класс.

Обычная функция, декорированная `singledispatch` становится *обобщенной функцией*: группой функций, выполняющих одну и ту же логическую операцию по-разному в зависимости от типа первого аргумента.

Декоратор `functools.singledispatch` создает функцию `htmlize.register` для объединения нескольких функций в одну обобщенную.

```
from collections import abc
import html
import numbers
from functools import singledispatch


@singledispatch             # Помечает базовую функцию, которая обрабатывает obj
def htmlize(obj):
    content = html.escape(repr(obj))
    return f'<pre>{content}</pre>'


@htmlize.register(str)      # Каждая специальная функция снабжается декоратором
def _(text):                # Имена функций не существенны
    content = html.escape(text).replace('\n', '<br>\n')
    return f'<pre>{content}</pre>'


@htmlize.regiter(numbers.Integral)
def _(n):
    return f'<pre>{n} (0x{0:x})</pre>'


@htmlize.register(tuple)
@htmlize.register(abc.MutableSequence)
def _(seq):
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return f'<ul>\n<li>{inner}</li>\n<li>'
```

Замечательное свойство данного декоратора в том, что специализированные функции можно зарегистрировать в любом месте системы, в любом модуле. Если в последствии, вы добавите модуль, содержащий новый пользовательский тип, то без труда сможете новую специализированную функцию для обработки данного типа.

Возможности этого декоратора шире, подробнее можно почитать тут:

- PEP-0443 Single-dispatch generic function

Композиция декораторов

Когда два декоратора `@d1` и `@d2` применяются к одной и той же функции `f` в указанном порядке, получается то же самое, что и в результате композиции `f=d1(d2(f))`

Параметризованные декораторы

Для реализации параметризованного декоратора, необходимо создать *фабрику декораторов*. Т.е. создать функцию, которая возвращает декоратор.

Ссылочки

- A Python module for decorators, wrappers and monkey patching.
- pip install decorator
- Python Decorator Library
- PEP 443 — Single-dispatch generic functions
- PEP 3104 — Access to Names in Outer Scopes