

Python

Занятие #11. Декораторы

 **О чем это занятие:**

- ☐ Виды функций, необходимых для создания декораторов:
 - ☐ first-class objects
 - ☐ внутренние функции
- ☐ Определение декораторов: без параметров и с параметрами
- ☐ Декораторы на классах

 **Теги:**

first-class objects, внутренние функции, простой декоратор, параметрический декоратор, wrapper, вложенные декораторы, декоратор-класс, встроенные декораторы

Краткий обзор темы

- ❑ `first-class objects` и внутренние функции являются строительными кирпичиками для декораторов
- ❑ функция без скобок (ссылка на объект) может передаваться как параметр в другую функцию и возвращаться как результат
- ❑ Декоратор – это специальная функция, которая принимает в качестве параметра другую функцию, модифицирует ее и возвращает как результат, при этом имя принимаемой функции не меняется
- ❑ Декораторы применяются и к функциям, и к классам
- ❑ Декораторы бывают простые и параметрические
- ❑ Декораторы можно накладывать друг на друга, в этом случае они действуют в обратном порядке
- ❑ Существует универсальный декоратор, который может быть вызван как с параметрами, так и без них
- ❑ Существуют декораторы-классы и встроенные в Python декораторы

First-class objects

👉 Первый тип объектов, необходимых для использования декораторов

Свойства

- Функция без скобок – это ссылка на объект функции (**не результат выполнения**)

- Функции подобны другим объектам в Python, например числам, строкам, спискам и т.д, а значит, их:
 - можно передавать как параметр в другие функции
 - можно возвращать как результат функции
- Функция не вызывается, используется ссылка на объект функции

Пример:

```
1 from typing import Callable
2
3 def hail_someone(name: str) -> str:
4     return 'Hail to you, {}'.format(name)
5
6 def say_hi(name: str) -> str:
7     return 'Hi, {}'.format(name)
8
9 def greetings(greet_func: Callable) -> str:
10     return greet_func('Master')
11
12 greetings(hail_someone)
13
14 Out: 'Hail to you, Master'
```

В функцию *greetings* передается по ссылке объект *hail_someone*. Внутренняя функция без скобок, поэтому передаётся не результат, а ссылка на объект.

Внутренние функции

👉 Второй тип объектов, необходимых для использования декораторов.

Свойства

Функция, определённая внутри другой функции:

- существует только внутри той функции, в которой определена
- имеет доступ к контексту внешней по отношению к ней функции
- существует как **локальный** объект-ссылка
- также является **first-class object**

👉 Внутреннюю функцию можно возвращать как результат внешней функции

💻 **Пример корректного вызова внутренней функции:**

```
1 def the_father():
2     print('I am the father')
3
4     def the_son():
5         print('I am the son')
6
7     def the_daughter():
8         print('I am the daughter')
9
10    the_daughter()
11    the_son()
12
13 the_father()
Out:
I am the father
I am the daughter
I am the son
```

За пределами функции
the_father функции
the_son и *the_daughter*
недоступны

 **Пример обращения к внутренней функции вне области видимости:**

```

1 def the_father():
2     print('I am the father')
3
4     def the_son():
5         print('I am the son')
6
7     def the_daughter():
8         print('I am the daughter')
9
10    the_daughter()
11    the_son()
12
13 the_daughter()

```

15 Out:

```

-----
NameError                                Traceback (most recent call last)
<ipython-input-3-2f2ab351d17d> in <module>
     14
     15
----> 16 the_daughter()

NameError: name 'the_daughter' is not defined

```

Если попытаться обратиться к функции *the_daughter* вне функции *the_father*, Python сообщит о том, что такой идентификатор не определен.

👉 Внутренние функции создаются и уничтожаются при выполнении тела внешней функции.

Пример использования функции в качестве результата:

```
1 from typing import Callable
2
3
4 def produce_child(name: str) -> Callable:
5     print('I am the children producer')
6
7     def the_son():
8         print('The son was born')
9
10    def the_daughter():
11        print('The daughter was born')
12
13    if name == 'Bob':
14        return the_son
15    else:
16        return the_daughter
17
18 child_func = produce_child('Alice')
19 child_func()
20
21 Out:
I am the children producer
The daughter was born
```

В функции *produce_child* создаются два объекта, в условии *if* эти объекты возвращаются во внешний код, на них появляется ссылка. А значит, объекты не удаляются даже после завершения работы внешней функции.

Декораторы

Основы

Декоратор – это специальная функция, структура которой следует некоторым правилам:

- ❑ принимает другую функцию и дополняет и/или меняет её поведение
- ❑ возвращает модифицированную версию полученной функции

💬 Принятая функция может быть полностью изменена

👉 Декоратор можно применить, используя обозначение `@decorator` (тот самый синтаксический сахар, которым отличается Python)

👉 Для работы с декорируемыми функциями хорошо использовать [functools.wraps](https://docs.python.org/3/library/functools.html#functools.wraps)

💻 **Пример простого декоратора**


```

1 from typing import Callable
2
3
4 def oops_decorator(func: Callable) -> Callable:
5     def wrapper():
6         print('I say oops')
7         func()
8         print('Oops I did it again')
9
10    return wrapper
11
12
13 def say_hello_world() -> str:
14     print('Hello world')
15
16 say_hello_world = oops_decorator(say_hello_world)
17
18 say_hello_world()
19
20 Out:
    I say oops
    Hello world
    Oops I did it again

```

Есть функция *oops_decorator*, которая принимает некоторую функцию *func*. В теле основной программы реализация исходной функции *say_hello_world* меняется с помощью декоратора, но остаётся с тем же именем

💬 Внутренняя функция, которая будет декорировать, называется оберткой (в данном примере это *wrapper*)

Пример простого декоратора “с сахаром”

```
1  from typing import Callable
2
3
4  def oops_decorator(func: Callable) -> Callable:
5      def wrapper():
6          print('I say oops')
7          func()
8          print('Oops I did it again')
9
10     return wrapper
11
12  @oops_decorator
13  def say_hello_world() -> str:
14      print('Hello world')
15
16  say_hello_world()
17
18  Out:
19  I say oops
20  Hello world
21  Oops I did it again
```


Реализация идентична предыдущей, но здесь вызов декоратора записан как `@oops_decorator` перед определением декорируемой функции

Пример. Параметры и результат декорируемой функции

```
1 from typing import Callable
2
3
4 def enclose_with_tags(func: Callable) -> Callable:
5     def wrapper(*args, **kwargs):
6         result = func(*args, **kwargs)
7         return '<Entity>{}</Entity>'.format(result)
8
9     return wrapper
10
11
12 @enclose_with_tags
13 def say_hi(name: str) -> str:
14     return 'Hi, {}'.format(name)
15
16
17 say_hi('Bob')
18
19 Out:
    '<Entity>Hi, Bob</Entity>'
```

Wrapper – это функция, которая будет “притворяться” исходной функцией.

После применения декоратора все вызовы функции `say_hi` будут оборачиваться в теги

 Чтобы декоратор был универсальным, в качестве параметров функции лучше всего использовать `*args` и `**kwargs`, а не конкретную сигнатуру функции.

 Шаблон простого декоратора, где есть стандартная обертка и параметры:

```
1 def deco_name(func):  
2     def wrapper(*args,**kwargs):  
3         return func(*args,**kwargs)  
4     return wrapper
```

Особенности декораторов

- 📌 Декораторы вызываются во время импорта

💬 Это значит, что если в декораторе есть помимо обертки другие операторы, они выполнятся только один раз, во время импорта декоратора.

- Применимы и к функциям, и к классам (альтернатива [метаклассам](#))

💬 Декорируется сам процесс создания класса


- Могут быть наложены друг на друга (**порядок наложения крайне важен**)

💻 Пример наложения декораторов (с использованием `@имя_декоратора` и без):

```
1 @oops
2 @no_mercy
3 def say_hello(name:str,*,fold_spaces:bool=True):
4     print('Hello')
5
6 say_hello=oops(no_mercy(say_hello))
```

Верхняя и нижняя записи и
6)

- Бывают простые и параметрические

 Параметрические декораторы – это те, которые кроме функции принимают еще какие-то параметры

- Могут быть в виде класса (т.е. не обязательно декоратор должен быть функцией)

 **Пример наложения декораторов:**

```

1 from typing import Callable
2
3
4 def enclose_with_entity_tag(func: Callable) -> Callable:
5     def wrapper(*args, **kwargs):
6         result = func(*args, **kwargs)
7         return '<Entity>{}</Entity>'.format(result)
8     return wrapper
9
10
11 def enclose_with_name_tag(func: Callable) -> Callable:
12     def wrapper(*args, **kwargs):
13         result = func(*args, **kwargs)
14         return '<Name>{}</Name>'.format(result)
15     return wrapper
16
17
18 @enclose_with_entity_tag
19 @enclose_with_name_tag
20 def say_hi(name: str) -> str:
21     return 'Hi, {}'.format(name)
22
23 say_hi('Alice')
24
25 Out:
    '<Entity><Name>Hi, Alice</Name></Entity>'

```

Еще один пример последовательного вызова декораторов. Здесь видно, что сначала был вызван тот, который записан последним, затем предыдущий и т.д.


 Пример параметрического декоратора:

```

1 from typing import Callable
2
3
4 def enclose_with(tag: str) -> Callable:
5     def outer_wrapper(func: Callable) -> Callable:
6         def inner_wrapper(*args, **kwargs):
7             result = func(*args, **kwargs)
8             return '<{tag}>{result}</tag>'.format(
9                 tag=tag, result=result
10            )
11
12        return inner_wrapper
13
14    return outer_wrapper
15 @enclose_with('Entity')
16 @enclose_with('Name')
17 def say_hi(name: str) -> str:
18     return 'Hi, {}'.format(name)
19
20
21 say_hi('Master')
22
23 Out:
    '<Entity><Name>Hi, Master</tag></tag>'

```

 **Шаблон параметрического декоратора:**

 Если в функцию с параметрами вложить обычный декоратор – это будет параметрический декоратор. В примере *enclose_with* – это та самая “надстроенная” функция с параметром, *outer_wrapper* – простой декоратор, *inner_wrapper* – обертка.

```
1 def deco_name(deco_param1, deco_param2):  
2     def wrapper(func):  
3         def inner_wrapper(*args, **kwargs):  
4             return func(*args, **kwargs)  
5         return inner_wrapper  
6     return wrapper
```

Универсальный декоратор

- ☐ Определён как параметрический
- ☐ Вызывается как с параметрами, так и без них
- ☐ Декорируемая функция (первый параметр) по умолчанию имеет значение **None**
- ☐ Все параметры передаются по имени принудительно

 подробнее можно посмотреть в документации [здесь](#)

Если декоратор вызывается без скобок, то в качестве первого параметра подставляется декорируемая функция, если декоратор с параметрами, то подставляется **None**.

Пример универсального декоратора


```

1 import functools
2
3
4 def enclose_with(_func=None, *, tag: str = 'Name'):
5     def outer_wrapper(func):
6         @functools.wraps(func)
7         def inner_wrapper(*args, **kwargs):
8             result = func(*args, **kwargs)
9             return '<{tag}>{result}</tag>'.format(
10                 tag=tag, result=result
11             )
12
13         return inner_wrapper
14
15     return outer_wrapper if _func is None else outer_wrapper(_func)
16
17
18 @enclose_with
19 def say_hi(name: str) -> str:
20     return 'Hi, {}'.format(name)
21
22
23 @enclose_with(tag='Entity')
24 def say_hello(name: str) -> str:
25     return 'Hello {}'.format(name)
26 print(say_hello('Master'))
27 print(say_hi('Margarita'))

```

28

29 **Out:**

```
<Entity>Hello Master</tag>
```

```
<Name>Hi, Margarita</tag>
```

[Альтернативная реализация с помощью functools.partial](#)

Декоратор-класс

Иногда необходимо, чтобы декоратор, кроме того что дополняет своим поведением исходную функцию, ещё и хранил состояние. Например, это бывает нужно для накопления знаний о декорируемой функции или о вызовах функции. Удобнее это накопление реализовать в классе, где есть атрибуты, к которым можно обратиться.

Декоратор-класс:

- ❑ принимает в качестве параметра `__init__` декорируемую функцию

- ❑ Должен быть вызываемым, т.е. реализовывать метод `__call__`
- ❑ Может хранить состояние (данные)

Пример декоратора-класса

```
1 import functools
2
3
4 class call_counter:
5     def __init__(self, func):
6         functools.update_wrapper(self, func)
7         self.func = func
8         self.num_calls = 0
9
10    def __call__(self, *args, **kwargs):
11        self.num_calls += 1
12        print('{} was called {} times'.format(self.func.__name__, self.num_calls))
13        return self.func(*args, **kwargs)
14
15
16 @call_counter
17 def say_hello(name: str) -> str:
18     return 'Hi, {}'.format(name)
19
```

```
20
21 say_hello('Alice')
22 say_hello('Bob')
23 say_hello('Master')
24
25 Out:
    say_hello was called 1 times
    say_hello was called 2 times
    say_hello was called 3 times

    'Hi, Master'
```





Встроенные декораторы

[cached_property](#) – аналог `property`, но кэширует результаты (доступен начиная с версии Python 3.8)

[lru_cache](#) – кэширует результаты вызова функции/метода, более универсален, т.к. может быть вызван для любой функции. Параметр декоратора – это количество хранимых результатов

[singledispatch](#) – позволяет задать несколько реализаций функции/метода в зависимости от типа параметров

Полезные ссылки

-  <https://www.geeksforgeeks.org/decorators-in-python/>
-  <https://www.datacamp.com/community/tutorials/decorators-python>
-  <https://realpython.com/primer-on-python-decorators/>
-  <https://blog.miguelgrinberg.com/post/the-ultimate-guide-to-python-decorators-part-i-function-registration>