

**Моя IT Школа**  
сертифицированные IT курсы

# Декораторы

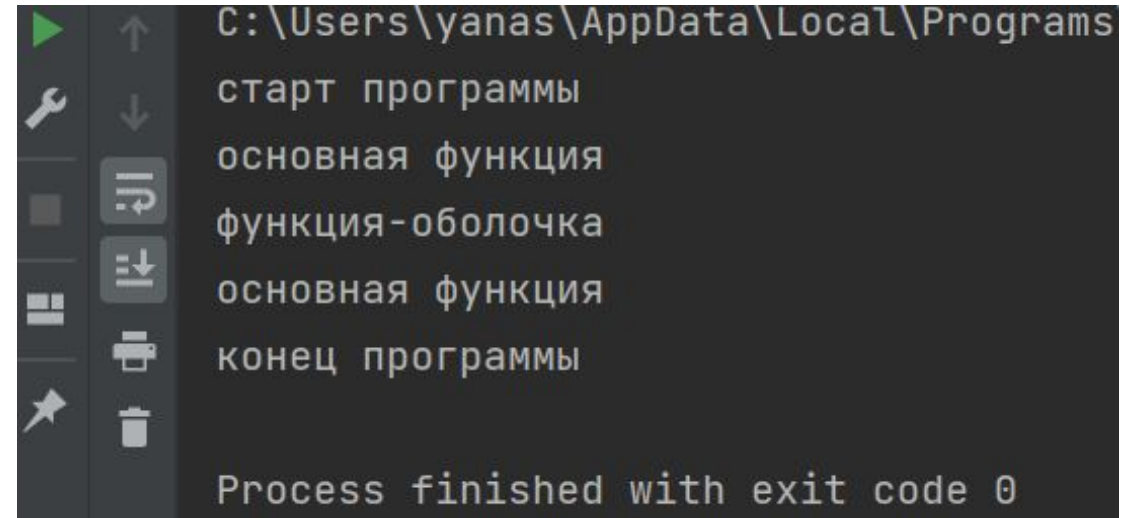


## **«Обёртка» функции**

Они предназначены для модификации функций, при помощи других функций

```
def decorator(func):  
    def wrapper():  
        print('функция-оболочка')  
        func()  
    return wrapper  
  
def basic():  
    print('основная функция')  
  
wrapped = decorator(basic)  
print('старт программы')  
basic()  
wrapped()  
print('конец программы')
```

Результат работы программы:



```
C:\Users\yanas\AppData\Local\Programs  
старт программы  
основная функция  
функция-оболочка  
основная функция  
конец программы  
  
Process finished with exit code 0
```

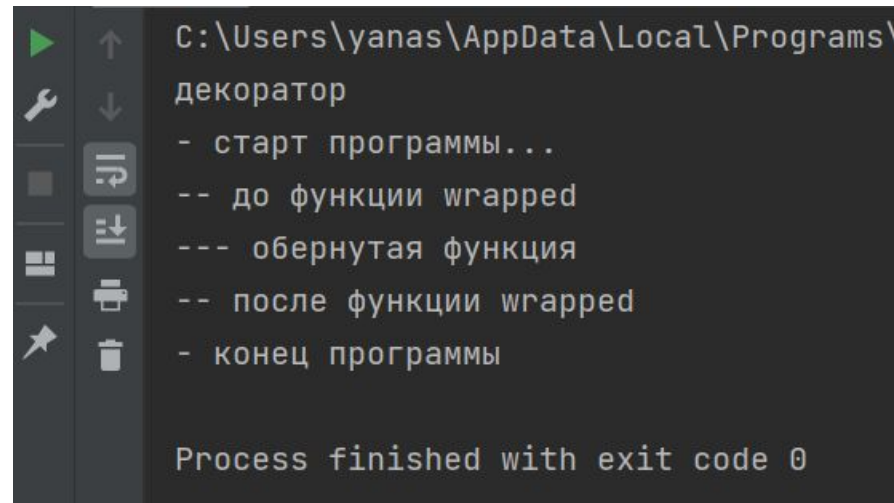
**Функция decorator** — это, как можно понять по названию, декоратор. Она принимает в качестве параметра функцию func. Внутри функции объявляется другая под названием wrapper. Объявлять ее внутри необязательно, но так проще работать.

В конце возвращается функция wrapper. Напомним, что нам все еще нужен вызываемый объект. Теперь результат можно вызывать с оригинальным набором возможностей, а также новым включенным кодом.

Но в Python есть синтаксис для упрощения такого объявления. Чтобы декорировать функции, используется символ @ рядом с именем декоратора. Он размещается над функцией, которую требуется декорировать.

```
def decorator(func):  
    '''Основная функция'''  
    print('декоратор')  
  
    def wrapper():  
        print('-- до функции', func.__name__)  
        func()  
        print('-- после функции', func.__name__)  
  
    return wrapper  
  
@decorator  
def wrapped():  
    print('--- обернутая функция')  
  
print('- старт программы...')  
wrapped()  
print('- конец программы')
```

Результат работы программы:



```
C:\Users\yanas\AppData\Local\Programs\Python\Python39-64\python.exe  
декоратор  
- старт программы...  
-- до функции wrapped  
--- обернутая функция  
-- после функции wrapped  
- конец программы  
  
Process finished with exit code 0
```



## Задание №1

Напишите декоратор, который будет считать, сколько раз была вызвана декорируемая функция

```
1  def counter(func):
2      """
3      Декоратор, считающий и выводящий количество вызовов
4      декорируемой функции.
5      """
6
7      def wrapper(*args, **kwargs):
8          wrapper.count += 1
9          res = func(*args, **kwargs)
10         print("{0} была вызвана: {1}x".format(func.__name__, wrapper.count))
11         return res
12
13     wrapper.count = 0
14     return wrapper
```

## Объединение декораторов в цепочки

Декораторы можно объединять в цепочки. Это значит, что можно использовать несколько декораторов одновременно.

```
def bold(func):  
    def wrapper():  
        return "" + func() + "  
  
    return wrapper  
  
def italic(func):  
    def wrapper():  
        return "" + func() + "  
  
    return wrapper  
  
@bold  
@italic  
def formatted_text():  
    return 'Python rocks!'
```

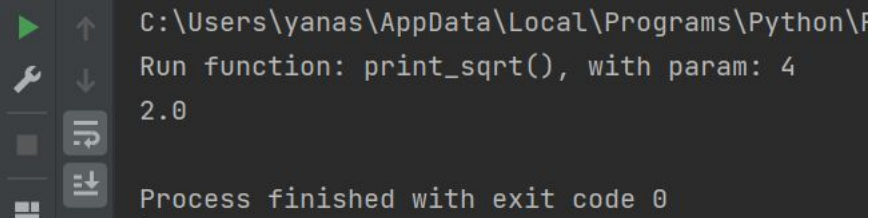


## Передача аргументов в функцию через декоратор

Если функция в своей работе требует наличие аргумента, то его можно передать через декоратор.

```
def param_transfer(fn):  
    def wrapper(arg):  
        print("Run function: " + str(fn.__name__) + "(), with param: " + str(arg))  
        fn(arg)  
    return wrapper  
  
@param_transfer  
def print_sqrt(num):  
    print(num ** 0.5)  
  
print_sqrt(4)
```

Результат работы программы:



```
C:\Users\yanas\AppData\Local\Programs\Python\F  
Run function: print_sqrt(), with param: 4  
2.0  
Process finished with exit code 0
```

Декораторы – очень мощный и полезный инструмент в Python, поскольку он позволяет программистам изменять поведение функции или класса. Декораторы позволяют нам обернуть другую функцию, чтобы расширить поведение обернутой функции, не изменяя ее навсегда.

Мы можем определить декоратор как класс для этого мы должны использовать метод `__call__` классов. Когда пользователю необходимо создать объект, который действует как функция, декоратор функции должен возвращать объект, который действует как функция, поэтому `__call__` может быть полезен.

```
class Functor:
    def __call__(self, a, b):
        print(a * b)

f = Functor()
# вызов как будто функция
f(10, 20)
```

## Таймеры

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

```
from datetime import datetime
import time

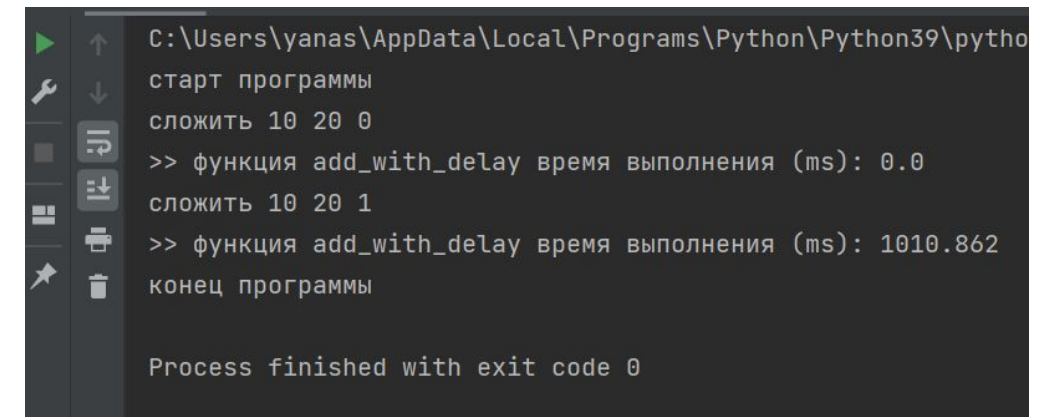
def elapsed(func):
    def wrapper(a, b, delay=0):
        start = datetime.now()
        func(a, b, delay)
        end = datetime.now()
        elapsed = (end - start).total_seconds() * 1000
        print(f'>> функция {func.__name__} время выполнения (ms): {elapsed}')

    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')
```

Результат работы программы:



```
C:\Users\yanas\AppData\Local\Programs\Python\Python39\python.exe
старт программы
сложить 10 20 0
>> функция add_with_delay время выполнения (ms): 0.0
сложить 10 20 1
>> функция add_with_delay время выполнения (ms): 1010.862
конец программы

Process finished with exit code 0
```

В Python есть модуль time, который используется для решения задач, связанных со временем. Для использования определенных в нем функций необходимо сначала его импортировать

- `time.altzone` - смещение DST часового пояса в секундах к западу от нулевого меридиана. Если часовой пояс находится восточнее, смещение отрицательно.
- `time.asctime([t])` - преобразовывает кортеж или `struct_time` в строку вида "Thu Sep 27 16:42:37 2012". Если аргумент не указан, используется текущее время.
- `time.clock()` - в Unix, возвращает текущее время. В Windows, возвращает время, прошедшее с момента первого вызова данной функции.
- `time.ctime([сек])` - преобразует время, выраженное в секундах с начала эпохи в строку вида "Thu Sep 27 16:42:37 2012".
- `time.daylight` - не 0, если определено, зимнее время или летнее (DST).
- `time.gmtime([сек])` - преобразует время, выраженное в секундах с начала эпохи в `struct_time`, где DST флаг всегда равен нулю.
- `time.sleep(сек)` - приостановить выполнение программы на заданное количество секунд

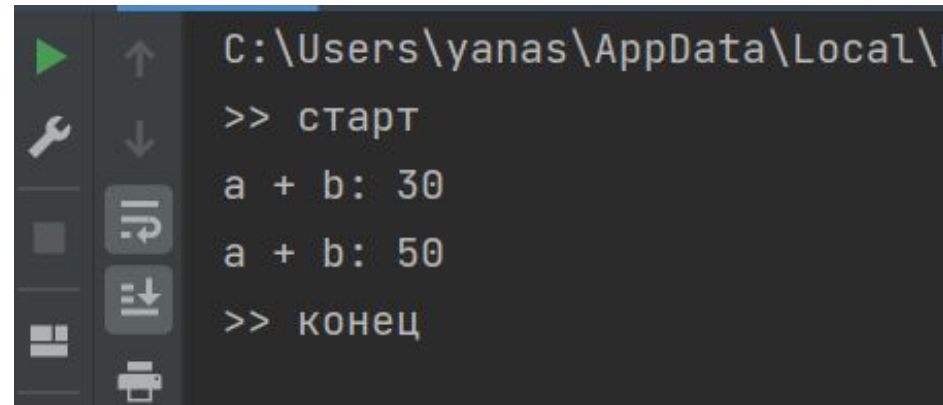
Модуль datetime предоставляет классы для обработки времени и даты разными способами. Поддерживается и стандартный способ представления времени, однако больший упор сделан на простоту манипулирования датой, временем и их частями

- Класс `datetime.date(year, month, day)` - стандартная дата. Атрибуты: `year`, `month`, `day`. Неизменяемый объект.
- Класс `datetime.time(hour=0, minute=0, second=0, microsecond=0, tzinfo=None)` - стандартное время, не зависит от даты. Атрибуты: `hour`, `minute`, `second`, `microsecond`, `tzinfo`.
- Класс `datetime.timedelta` - разница между двумя моментами времени, с точностью до микросекунд.
- `datetime.today()` - объект `datetime` из текущей даты и времени. Работает также, как и `datetime.now()` со значением `tz=None`.
- `datetime.now(tz=None)` - объект `datetime` из текущей даты и времени.
- `datetime.combine(date, time)` - объект `datetime` из комбинации объектов `date` и `time`.
- `datetime.strptime(date_string, format)` - преобразует строку в `datetime` (так же, как и функция `strptime` из модуля `time`).
- `datetime.date()` - объект даты (с отсечением времени).
- `datetime.time()` - объект времени (с отсечением даты).
- `datetime.weekday()` - день недели в виде числа, понедельник - 0, воскресенье - 6.

Еще один распространенный сценарий применения для декоратора — логирование функций

```
1 import logging
2
3 def logger(func):
4     log = logging.getLogger(__name__)
5
6     def wrapper(a, b):
7         log.info("Вызов функции ", func.__name__)
8         ret = func(a, b)
9         log.info("Вызвана функция ", func.__name__)
10        return ret
11
12    return wrapper
13
14
15 @logger
16 def add(a, b):
17     print('a + b:', a + b)
18     return a + b
19
20
21 print('>> старт')
22 add(10, 20)
23 add(20, 30)
24 print('>> конец')
```

Результат работы программы:



```
C:\Users\yanas\AppData\Local\F
>> старт
a + b: 30
a + b: 50
>> конец
```

По мере того, как приложения меняются и усложняются, наличие лог-журнала будет полезным при отладке и для понимания проблем, анализа производительности приложений.

Стандартная библиотека логгирования в Python поставляется модулем `logging`, который предлагает большинство главных функций для ведения лога. При правильной настройке сообщения лога, мы получим много полезной информации. О том, когда и где запускается логгирование, о контексте лог-журнала, например: запущенном процессе или потоке

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 выполнялась после определенного события.

```
1  app = {}
2
3
4  def callback(route):
5      def wrapper(func):
6          app[route] = func
7
8          def wrapped():
9              ret = func()
10             return ret
11
12             return wrapped
13
14     return wrapper
15
16
17  @callback('/')
18  def index():
19      print('index')
20      return 'OK'
21
```

```
21
22
23  print('> старт')
24  route = app['/']
25  if route:
26      resp = route()
27      print('ответ:', resp)
28
29  print('> конец')
30
```

Результат работы программы:

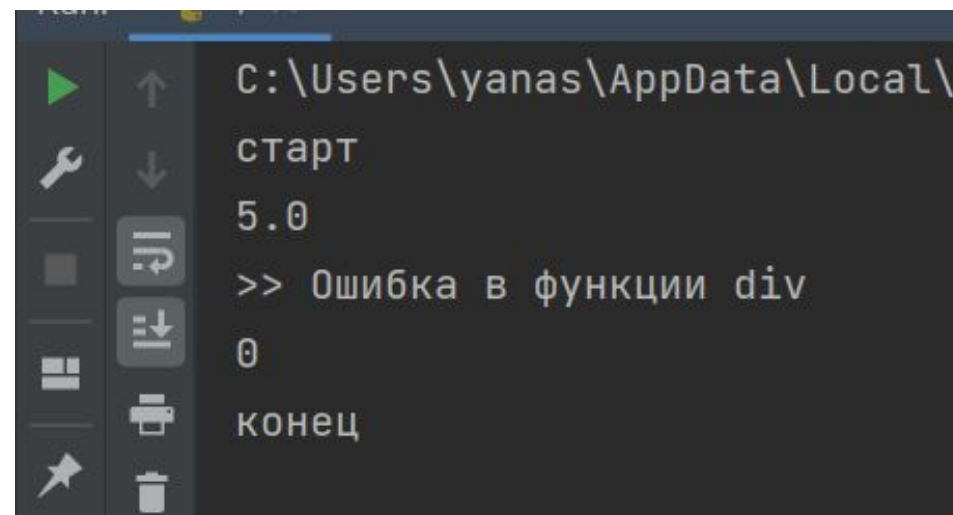
```
C:\Users\yanas\AppData\Local\Programs\
> старт
index
ответ: OK
> конец
Process finished with exit code 0
```



Можно убедиться, что обрабатываются определенные типы ошибок без использования блока try для каждой функции

```
1 def error_handler(func):
2     def wrapper(*args, **kwargs):
3         ret = 0
4         try:
5             ret = func(*args, **kwargs)
6         except:
7             print('>> Ошибка в функции', func.__name__)
8         return ret
9     return wrapper
10
11 @error_handler
12 def div(a, b):
13     return a / b
14
15 print('старт')
16 print(div(10, 2))
17 print(div(10, 0))
18 print('конец')
```

Результат работы программы:



```
C:\Users\yanas\AppData\Local\Python\Python38-64\Scripts\python.exe
старт
5.0
>> Ошибка в функции div
0
конец
```



## Домашнее задание

Напишите декоратор `debug`, который при каждом вызове декорируемой функции выводит её имя (вместе со всеми передаваемыми аргументами), а затем — какое значение она возвращает. После этого выводится результат её выполнения