



# Функции, расширенные возможности функций

Python - Лекция #3

# Концепции проектирования функций

- Для передачи значений функции используйте аргументы, для возврата результатов - инструкцию `return`
- Используйте глобальные переменные, только если это действительно необходимо
- Не воздействуйте на изменяемые аргументы, если вызывающая программа не предполагает этого
- Связность: каждая функция должна иметь единственное назначение
- Размер: каждая функция должна иметь относительно небольшой размер

# Косвенный вызов функций

```
>>> def echo(message): # Имени echo присваивается объект функции
...     print(message)
...
>>> echo("Direct call") # Вызов объекта по оригинальному имени
Direct call

>>> x = echo # Теперь на эту функцию ссылается еще и имя x

>>> x("Indirect call!") # Вызов объекта по другому имени с добавлением ()
Indirect call
```

# Атрибуты функций

```
>>> foo
<function foo at 0x0257C738>
>>> foo.count = 0
>>> foo.count += 1
>>> foo.count
1
>>> foo.handles = "Button-Press"
>>> foo.handles
"Button-Press"
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...__str__', '__subclasshook__', 'count', 'handles']
```

# Рекурсивные функции

Рекурсивные функции - функции, которые могут вызывать сами себя, прямо или косвенно, образуя цикл

```
>>> def mysum(L):  
...     if not L:  
...         return 0  
...     else:  
...         return L[0] + mysum(L[1:]) # Вызывает саму себя  
>>> mysum([1, 2, 3, 4, 5])  
15
```

# Фибоначчи

```
>>> def fib(n):  
...     if n <= 2:  
...         return 1  
...     else:  
...         return fib(n - 1) + fib(n - 2)  
...  
>>> def fib2(n): return 1 if n <= 2 else fib2(n - 1) + fib2(n - 2)  
...  
>>> print(fib(20))  
6765
```

# Анонимные функции: lambda

Python поддерживает синтаксис, позволяющий определять небольшие однострочные функции на лету. Позаимствованные из Lisp, так называемые lambda-функции могут быть использованы везде, где требуется функция.

`lambda argument1, argument2, ... argumentN` : выражение, использующее аргументы

- `lambda` - это выражение, а не инструкция
- Тело `lambda` - это не блок инструкций, а единственное выражение

<https://docs.python.org/3/reference/expressions.html#lambda>

# Пример

```
>>> def foo(x):  
...     return x*2  
...  
>>> foo(3)  
6  
>>> g = lambda x: x * 2  
>>> g(3)  
6  
>>> (lambda x: x * 2)(3)  
6
```



# Lambda-функции: захваты

```
>>> x = 10
>>> foo = lambda: x
>>> foo()
10
>>> x = 11
>>> foo()
11
```

```
>>> foo = lambda x=x: x
>>> foo()
11
>>> x = 12
>>> foo()
11
```

# Присваивание функции переменной

```
process = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

process теперь ссылается на функцию, но на какую — это зависит от значения переменной collapse. Если collapse является истиной, process(string) будет сворачивать символы пропуска, в противном случае process (string) будет возвращать аргумент без изменений.

# Вложенные lambda-выражения и области видимости

```
>>> def action(x):  
...     return (lambda y: x + y) # Создать и вернуть ф-цию, запомнить x  
...
```

# Декораторы

# Синтаксис использования декораторов

*Декоратор* — функция, которая принимает другую функцию и что-то возвращает.  
Синтаксический сахар декоратора:

```
>>> @trace
... def foo(x):
...     return 42
... 
```

Аналогичная по смыслу версия без синтаксического сахара

```
>>> def foo(x):
...     return 42
...
>>> foo = trace(foo)
```

## Пример: @trace

Декоратор trace выводит на экран сообщение с информацией о вызове декорируемой функции.

```
>>> def trace(func):
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     return inner
>>> @trace
... def useful(x):
...     print("I do nothing useful.")
...
>>> useful(42)
useful (42, ) {}
I do nothing useful.
```

# Декораторы: подводные камни

- Проблема с help и атрибутами декорируемой функции.

```
>>> help(identity)
Help on function inner in module __main__:
inner(*args, **kwargs)
```

# Декораторы и help: проблема

```
>>> @trace
... def useful(x):
...     """Useful function"""
...     print("I do nothing useful.")
...
>>> useful.__name__, useful.__doc__
('useful', 'Useful function')
>>> useful = trace(useful)
>>> useful.__name__, useful.__doc__
('inner', None)
```



# Декораторы и help: решение в лоб

Давайте просто возьмём и установим "правильные" значения в атрибуты декорируемой функции:

```
>>> def trace(func):
...     def inner(*args,**kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args,**kwargs)
...     inner.__module__ = func.__module__
...     inner.__name__ = func.__name__
...     inner.__doc__ = func.__doc__
...     return inner
```

# Декораторы и help: решение в лоб

```
>>> @trace
... def useful(x):
...     """Useful function"""
...     print("I do nothing useful.")
...
>>> useful.__name__, useful.__doc__
('useful', 'Useful function')
```

# Декораторы и help: модуль functools

В модуле functools из стандартной библиотеки Python есть функция, реализующая логику копирования внутренних атрибутов

```
>>> import functools
>>> def trace(func):
...     def inner(*args, **kwargs):
...         print(func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     functools.update_wrapper(inner, func)
...     return inner
...
```

# Декораторы и help: модуль functools

То же самое можно сделать с помощью декоратора wraps

```
>>> def trace(func):  
...     @functools.wraps(func)  
...     def inner(*args,**kwargs):  
...         print(func.__name__, args, kwargs)  
...         return func(*args,**kwargs)  
...     return inner  
...
```

# Декораторы с аргументами: синтаксис

Напоминание:

```
>>> @trace
... def useful(x):
...     return x
```

```
>>> def useful(x):
...     return x
...
>>> useful = trace(useful)
```

Для декораторов с аргументами эквивалентность сохраняется

```
>>> @trace(sys.stderr)
... def useful(x):
...     return x
```

```
>>> def useful(x):
...     return x
...
>>> deco = trace(sys.stderr)
>>> useful= deco(useful)
```

# Декораторы с аргументами: реализация



# Декораторы с аргументами: реализация

```
>>> def trace(handle):  
...     def decorator(func):  
...         @functools.wraps(func)  
...         def inner(*args, **kwargs):  
...             print(func.__name__, args, kwargs, file=handle)  
...             return func(*args,**kwargs)  
...         return inner  
...     return decorator  
...
```

# Task #1

Написать декоратор, который будет считать время работы функции и выводить на экран. Для текущего времени можно использовать модуль `time`.

```
>>> import time  
>>> time.time()  
1522321308.174503
```



# Цепочки декораторов

Синтаксис Python разрешает одновременное применение нескольких декораторов.

Порядок декораторов имеет значение:

```
def first(func):  
    def wrapper():  
        print('first')  
        return func()  
    return wrapper
```

```
def second(func):  
    def wrapper():  
        print('second')  
        return func()  
    return wrapper
```

```
@first  
@second  
def foo():  
    print('func')
```

# Декораторы: резюме

---

- Декораторы в мире Python вездесущи и полезны.
- Больше декораторов по ссылке

<https://wiki.python.org/moin/PythonDecoratorLibrary>

## Task #3

Написать декоратор `validate`, который будет валидировать входящие аргументы функции на предмет выхода за заданные границы

```
>>> @validate(low_bound=0, upper_bound=256)
... def set_pixel(red, green, blue):
...     return "Pixel created!"
...
>>> set_pixel(0, 127, 300)
Function call is not valid!
>>> set_pixel(0, 127, 250)
Pixel created!
```

# Homework

Необходимо написать фабрику декораторов(также декоратор).  
Фабрика принимает на вход функцию (lambda) возвращает декоратор,  
который вернет результат выполнения функции в которую первым аргументом  
передается результат выполнения декорируемой функции. Пример:

```
@apply(lambda user_id: user_id + 1)
def return_user_id():
    return 42
```

```
>> return_user_id()
43
```