PYTHON

Тема2: Функции.Анонимные функции, инструкция lambda.

Lambda-функция – это безымянная функция с произвольным числом аргументов и вычисляющая одно выражение.
Тело такой функции не может содержать более одной инструкции (или выражения). Данную функцию можно использовать в рамках каких-либо конвейерных вычислений (например внутри **filter()**, **map()** и **reduce()**) либо самостоятельно, в тех местах, где требуется произвести какие вычисление, которые
удобно “завернуть” в функцию.

```
func = lambda x, y: x + y
func(1, 2)
```

3

func('a', 'b')

'ab'

(lambda x, y: x + y)(1, 2)

3

(lambda x, y: x + y)('a', 'b')

'ab'

***lambda** функции*, в отличие от обычной, не требуется инструкция **return**, а в остальном, ведет себя точно так же:

func = lambda *args: args
func(1, 2, 3, 4)
(1, 2, 3, 4)

PYTHON

Тема2: Функции. Декораторы.

Применение декораторов

Декораторы используются
- для расширения возможностей функций из сторонних библиотек (код которых нельзя изменять)
- для упрощения отладки (мы не хотим изменять код, который ещё не устоялся)
- для расширения различных функций одним и тем же кодом, без повторного его переписывания каждый раз

AP2

PYTHON

Тема2: Функции. Декораторы.

```
def specify_decorator(function_to_decorate):
    # Внутри себя декоратор определяет функцию-"обёртку". Она будет обёрнута вокруг декорируемой,
    # получая возможность исполнять произвольный код до и после неё.
    def the_wrapper_around_the_original_function():
        print("Pre-function call")
        function_to_decorate()            # call base function
        print("Post-function call")
    # Вернём эту функцию
    return the_wrapper_around_the_original_function

# Функция контракт которой нельзя изменять
def stand_alone_function():
    print("Third Party Function")

stand_alone_function()

# Однако, чтобы изменить её поведение, мы можем декорировать её, то есть просто передать декоратору,
# который обернет исходную функцию в любой код, который нам потребуется, и вернёт новую,
# готовую к использованию функцию:
stand_alone_function_decorated = my_shiny_new_decorator(stand_alone_function)
stand_alone_function_decorated()
Pre-function call
Third Party Function
Post-function call
```

Декораторы — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код,
Декораторы получают декорируемую-функцию в качестве параметра

Slide 58

AP2 Andrii Poliakov, 1/21/2020

```
>>> stand_alone_function = my_shiny_new_decorator(stand_alone_function)
>>> stand_alone_function()

Pre-function call
Third Party Function
Post-function call
```

Decorator declaration through annotation

```
@specify_decorator
def specify_function():
    print("Another third-Party Function")

specify_function()
```

Decorator via annotation

Pre-function call
Another third-Party Function
Post-function call

Применение нескольких декораторов для одной функции

```
def bread(func):
    def wrapper():
        print()
        func()
        print("<\_____/>")

    return wrapper


def ingredients(func):
    def wrapper():
        print("#помидоры#")
        func()
        print("~салат~")

    return wrapper
```

```
def sandwich(food="--ветчина--"):
    print(food)

sandwich()
sandwich = bread(ingredients(sandwich))
sandwich()
```

```
--ветчина--

#-помидоры-#
--ветчина--
~салат~
<\_____/>
```

используя синтаксис декораторов:

```
@bread
@ingredients
def sandwich(food="- ветчина -"):
    print(food)

sandwich()

# помидоры #
- ветчина -
~ салат ~
< bread >
```

Также нужно помнить о том, что важен порядок декорирования

```
@ingredients
@bread
def sandwich(food="- ветчина -"):
    print(food)

sandwich()

# помидоры #

- ветчина -
~ салат ~
< bread >
```

Передача декоратором аргументов в функцию

```
>>>
>>> def a_decorator_passing_arguments(function_to_decorate):
...     def a_wrapper_accepting_arguments(arg1, arg2):
...         print("Смотри, что я получил:", arg1, arg2)
...         function_to_decorate(arg1, arg2)
...     return a_wrapper_accepting_arguments
...
>>> # Теперь, когда мы вызываем функцию, которую возвращает декоратор, мы вызываем её "обёртку",
>>> # передаём ей аргументы и уже в свою очередь она передаёт их декорируемой функции
>>> @a_decorator_passing_arguments
... def print_full_name(first_name, last_name):
...     print("Меня зовут", first_name, last_name)
...
>>> print_full_name("Vasya", "Pupkin")
Смотри, что я получил: Vasya Pupkin
Меня зовут Vasya Pupkin
```

Если мы создаём максимально общий декоратор и хотим, чтобы его можно было применить к любой функции или методу, то можно воспользоваться распаковкой аргументов:

```
def decorator_args(function_to_decorate):
    # Данная "обёртка" принимает любые аргументы
    def wrapper_arguments(*args, **kwargs):
        print("Передали ли мне что-нибудь?:")
        print(args)
        print(kwargs)
        function_to_decorate(*args, **kwargs)
    return wrapper_arguments

@decorator_args
def fun_with_no_argument():
    print("Python is cool, no argument here.")

fun_with_no_argument()

@decorator_args
def fun_with_arguments(a, b, c):
    print(a, b, c)

fun_with_arguments(1, 2, 3)
```

```
Output:

Передали ли мне что-нибудь?:
()
{}
Python is cool, no argument here.

Передали ли мне что-нибудь?:
(1, 2, 3)
{}
1 2 3
```

PYTHON

Тема2: Функции. Декораторы.

Декораторы с аргументами

```
>>>
>>> def decorator_maker():
...     print("Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать декоратор.")
...     def my_decorator(func):
...         print("Я - декоратор! Я буду вызван только раз: в момент декорирования функции.")
...         def wrapped():
...             print ("Я - обёртка вокруг декорируемой функции.\n"
...                    "Я буду вызвана каждый раз, когда ты вызываешь декорируемую функцию.\n"
...                    "Я возвращаю результат работы декорируемой функции.")
...             return func()
...         print("Я возвращаю обёрнутую функцию.")
...         return wrapped
...     print("Я возвращаю декоратор.")
...     return my_decorator
...
```

Декораторы с аргументами

```
>>> # Давайте теперь создадим декоратор. Это всего лишь ещё один вызов функции
>>> new_decorator = decorator_maker()
Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать декоратор.
Я возвращаю декоратор.
>>>
>>> # Теперь декорируем функцию
>>> def decorated_function():
...     print("Я - декорируемая функция.")
...
>>> decorated_function = new_decorator(decorated_function)
Я - декоратор! Я буду вызван только раз: в момент декорирования функции.
Я возвращаю обёрнутую функцию.
>>> # Теперь наконец вызовем функцию:
>>> decorated_function()
Я - обёртка вокруг декорируемой функции.
Я буду вызвана каждый раз, когда ты вызываешь декорируемую функцию.
Я возвращаю результат работы декорируемой функции.
Я - декорируемая функция.
```

Теперь перепишем данный код с помощью декораторов:

```
>>>
>>> @decorator_maker()
... def decorated_function():
...     print("Я - декорируемая функция.")
...
Я создаю декораторы! Я буду вызван только раз: когда ты попросишь меня создать декоратор.
Я возвращаю декоратор.
Я - декоратор! Я буду вызван только раз: в момент декорирования функции.
Я возвращаю обёрнутую функцию.
>>> decorated_function()
Я - обёртка вокруг декорируемой функции.
Я буду вызвана каждый раз когда ты вызываешь декорируемую функцию.
Я возвращаю результат работы декорируемой функции.
Я - декорируемая функция.
```

PYTHON

Тема2: Функции. Декораторы.

Декораторы с аргументами

Вернёмся к аргументам декораторов, ведь, если мы используем функцию, чтобы создавать декораторы "на лету", мы можем передавать ей любые аргументы

```
>>>
>>> def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):
...     print("Я создаю декораторы! И я получил следующие аргументы:",
...           decorator_arg1, decorator_arg2)
...     def my_decorator(func):
...         print("Я - декоратор. И ты всё же смог передать мне эти аргументы:",
...               decorator_arg1, decorator_arg2)
...         # Не перепутайте аргументы декораторов с аргументами функций!
...         def wrapped(function_arg1, function_arg2):
...             print ("Я - обёртка вокруг декорируемой функции.\n"
...                    "И я имею доступ ко всем аргументам\n"
...                    "\t- и декоратора: {0} {1}\n"
...                    "\t- и функции: {2} {3}\n"
...                    "Теперь я могу передать нужные аргументы дальше"
...                    .format(decorator_arg1, decorator_arg2,
...                            function_arg1, function_arg2))
...             return func(function_arg1, function_arg2)
...         return wrapped
...     return my_decorator
...

>>> @decorator_maker_with_arguments("Леонард", "Шелдон")
... def decorated_function_with_arguments(function_arg1, function_arg2):
...     print ("Я - декорируемая функция и я знаю только о своих аргументах: {0}"
...            " {1}".format(function_arg1, function_arg2))
...
Я создаю декораторы! И я получил следующие аргументы: Леонард Шелдон
Я - декоратор. И ты всё же смог передать мне эти аргументы: Леонард Шелдон
>>> decorated_function_with_arguments("Раджеш", "Говард")
Я - обёртка вокруг декорируемой функции.
И я имею доступ ко всем аргументам
   - и декоратора: Леонард Шелдон
   - и функции: Раджеш Говард
Теперь я могу передать нужные аргументы дальше
Я - декорируемая функция и я знаю только о своих аргументах: Раджеш Говард
```

Некоторые особенности работы с декораторами

- декораторы несколько замедляют вызов функции
- не взможно "раздекорировать" функцию

безусловно, существуют трюки, позволяющие создать декоратор, который можно отсоединить от функции, но это плохая практика. Правильнее будет запомнить, что если функция декорирована — это не отменить.

- декораторы оборачивают функции

Последняя проблема частично решена добавлением в модуле functools функции functools.wraps, копирующей всю информацию об оборачиваемой функции (её имя, из какого она модуля, её документацию и т.п.) в функцию-обёртку.

Забавным фактом является то, что functools.wraps тоже является декоратором.

PYTHON

Тема2: Функции. Декораторы.

```
def foo():
    print("foo")

print(foo.__name__)


# Однако, декораторы мешают нормальному ходу дел:
def bar(func):
    def wrapper():
        print("bar")
        return func()
    return wrapper

@bar
def foo():
    print("foo")

print(foo.__name__)
```

```
import functools
def bar(func):
    # Объявляем "wrapper" оборачивающим "func"
    # и запускаем магию:
    @functools.wraps(func)
    def wrapper():
        print("bar")
        return func()
    return wrapper

@bar
def foo():
    print("foo")

print(foo.__name__)
```

Получение базового доступа к атрибутам функции:

```
print(foo.__name__)

print(dir(foo))
```

```
foo
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__wrapped__']
```

Похожий механизм, называется интроспекцией - механизм исследования деталей реализации функции.

```
print(bar.__code__)
# <code object foo at 0x0257C9B0, file “<stdin>”, line 1>
print(dir(bar.__code__))
# [‘__class__’, ‘__delattr__’, ‘__doc__’, ‘__eq__’, ‘__format__’, ‘__ge__’,
# ...остальные имена опущены...
# ‘co_argcount’, ‘co_cellvars’, ‘co_code’, ‘co_consts’, ‘co_filename’,
# ‘co_firstlineno’, ‘co_flags’, ‘co_freevars’, ‘co_kwonlyargcount’, ‘co_lnotab’,
# ‘co_name’, ‘co_names’, ‘co_nlocals’, ‘co_stacksize’, ‘co_varnames’]
#
print(bar.__code__.co_varnames)
# (‘a’, ‘b’)
print(bar.__code__.co_argcount)
```

К функциям можно присоединять свои атрибуты:

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

Такие атрибуты можно использовать для хранения информации о состоянии непосредственно в объекте функции и отказаться от использования других приемов, таких как применение глобальных или нелокальных переменных и классов

PYTHON

Тема2: Функции и другие их возможности

Аннотации функций

Краткое описание (аннотация) – произвольные данные об аргументах функции и о возвращаемом значении. Аннотации необязательны, но если они есть, тогда они просто сохраняются в атрибутах __annotations__ объектов функций и могут использоваться другими инструментами.

Не аннотированная функция:

```
>>> def func(a, b, c):
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Аннотированная функция:

```
>>> def func(a: 'spam', b: (1, 10), c: float) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Если в объявлении функции присутствуют аннотации, интерпретатор
соберет их в словарь и присоединит его к объекту функции. Имена аргументов станут ключами, аннотация возвращаемого значения будет сохранена в ключе «return», а значениям ключей этого словаря будут присвоены результаты выражений в аннотациях:

```
>>> func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Обработка аннотаций:

```
>>> def func(a: 'spam', b, c: 99):
...     return a + b + c
...
>>> func(1, 2, 3)
6
>>> func.__annotations__
{‘a’: ‘spam’, ‘c’: 99}
>>> for arg in func.__annotations__:
...     print(arg, ‘=>’, func.__annotations__[arg])
...
a => spam
c => 99
```

В данном примере выполнен обход аннотаций.

Аннотации и значения по умолчанию: **Аннотации функций**

Также можно указывать значения по умолчанию – аннотация (и символ :) находится перед значением по умолчанию (и перед символом =). В следующем примере фрагмент
a: ‘spam’ = 4 означает, что аргумент a по умолчанию получает значение 4 и аннотирован строкой ‘spam’:

```
>>> def func(a: ‘spam’ = 4, b: (1, 10) = 5, c: float = 6) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
>>> func()        # 4 + 5 + 6 (все аргументы получают значения по умолчанию)
15
>>> func(1, c=10)  # 1 + 5 + 10 (именованные аргументы действуют как обычно)
16
>>> func.__annotations__
{‘a’: ‘spam’, ‘c’: <class ‘float’>, ‘b’: (1, 10), ‘return’: <class ‘int’>}
```

вы можете использовать или не использовать пробелы между компонентами в заголовках функций, однако отказ от использования пробелов может ухудшить удобочитаемость программного кода.

PYTHON

Тема2: Функции и другие их возможности

Кэширование функций

Кэширование — сохранение результата некоторого набора операций для быстрой его отдачи по запросу. Если какой-либо набор вычислений необходимо выполнять множество раз при одинаковых условиях, то сохранение результата в памяти и последующая его отдача может существенно ускорить вычислительный процесс в целом. Кэширование используется в компьютерных технологиях на различных уровнях его организации — начиная от микропроцессоров до сложных программных комплексов.

Кэширование функций позволяет кэшировать возвращаемые значения функций в зависимости от аргументов. Это может помочь сэкономить время при работе с вводом/выводом на повторяющихся данных. До Python 3.2 мы должны были бы написать собственную реализацию. В Python 3.2+ появился декоратор lpu_cache, который позволяет быстро кэшировать возвращаемые функцией значения.

Реализуем функцию расчета n-ого числа Фибоначчи с использованием lru_cache:

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

print([fib(n) for n in range(10)])
# Вывод: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Аргумент maxsize сообщает lru_cache сколько последних значений запоминать.
Мы также можем легко очистить кэш:

```
fib.cache_clear()
```

PYTHON

Тема2: Функции и другие их возможности

Расширенные возможности функций

Когда начинают использоваться функции, возникает проблема выбора, как лучше связать элементы между собой, например как разложить задачу на функции (связность), как должны взаимодействовать функции (взаимодействие) и так далее. Вы должны учитывать такие особенности, как размер функций, потому что от них напрямую зависит удобство сопровождения программного кода.

- Взаимодействие: для передачи значений функции используйте аргументы, для возврата результатов – инструкцию **return**.
- Взаимодействие: используйте глобальные переменные, только если это действительно необходимо.
- Взаимодействие: 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.
- Взаимодействие: избегайте непосредственного изменения переменных в другом модуле.

PYTHON

Тема2: Функции и другие их возможности