Функции

Часть II

Python Lambda

```
Lambda a : a*n
```

Argument Expression

Лямбда-функции, анонимные функции

Раньше мы использовали функции, обязательно связывая их с каким-то именем. В Python есть возможность создания однострочных анонимных функций

Конструкция:

```
lambda [param1, param2, ..]: [выражение]
```

lambda – функция, возвращает свое значение в том месте, в котором вы его объявляете.

Функция тождества (identity function)

функция, которая возвращает свой параметр

```
def identity(x):
    return x
```

#identity() принимает передаваемый аргумент в x и возвращает его при вызове.

Лямбда-функция :

```
lambda x: x
```

Ключевое слово: lambda

Параметр: x

Выражение(тело): x

Вызов lambda

Для вызова lambda обернем функцию и ее аргумент в круглые скобки. Передадим функции аргумент.

```
>>> (lambda x: x + 1)(2)
3
```

Именование lambda

Поскольку лямбда-функция является выражением, оно может быть именовано. Поэтому вы можете написать предыдущий код следующим образом:

```
>>> add_one = lambda x: x + 1
>>> add_one(2)
3
```

Аргументы

Как и обычный объект функции, определенный с помощью def, лямбда поддерживают все различные способы передачи аргументов. Это включает:

- Позиционные аргументы
- Именованные аргументы (иногда называемые ключевыми аргументами)
- Переменный список аргументов (часто называемый *args)
- Переменный список аргументов ключевых слов *kwargs

Аргументы функции

Функции с несколькими аргументами (функции, которые принимают более одного аргумента) выражаются в лямбда-выражениях Python, перечисляя аргументы и разделяя их запятой (,), но не заключая их в круглые скобки:

```
>>> full_name = lambda first, last: f'Full name: {first.title()} {last.title()}'


>>> full_name('guido', 'van rossum')
'Full name: Guido Van Rossum'
```

Именованные параметры

Как и в случае **def**, для аргуменов **lambda** можно указывать стандартные значения.

```
>>>str = ( lambda a='He', b='ll' , c='o': a+b+c)
str(a='Ze')
'Zello'
```

Пример

```
>>> (lambda x, y, z: x + y + z)(1, 2, 3)
6
>>> (lambda x, y, z=3: x + y + z)(1, 2)
6
>>> (lambda x, y, z=3: x + y + z)(1, y=2)
6
>>> (lambda *args: sum(args))(1,2,3)
6
>>> (lambda **kwargs: sum(kwargs.values()))(one=1,
two=2, three=3)
6
>>> (lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3)
```

Функции высокого порядка

Лямбда-функция может быть функцией более высокого порядка, принимая функцию (нормальную или лямбда-функцию) в качестве аргумента, как в следующем надуманном примере:

```
>>> high_ord_func = lambda x, func: x + func(x)
>>> high_ord_func(2, lambda x: x * x)
6
>>> high_ord_func(2, lambda x: x + 3)
7
```

Для чего используется lambda ?

Использование лямбда-выражения как литерала списка.

```
import random
# Словарь, в котором формируются три случайные числа
# с помощью лямбда-выражения
L = [ lambda : random.random(),
      lambda : random.random(),
      lambda : random.random() ]
# Вывести результат
for l in L:
    print(l())
```

Лямбда-выражения как литералы кортежей

Формируется кортеж, в котором элементы умножаются на разные числа.

```
import random
# Кортеж, в котором формируются три литерала-строки
# с помощью лямбда-выражения
T = ( lambda x: x*2,
      lambda x: x*3,
      lambda x: x*4 )
# Вывести результат для строки 'abc'
for t in T:
    print(t('abc'))
Результат:
abcabc
abcabcabc
abcabcabcabc
```

Использование лямбда-выражения для формирования таблиц переходов.

```
# Словарь, который есть таблицей переходов
Dict = {
     1 : (lambda: print('Monday')),
     2 : (lambda: print('Tuesday')),
     3 : (lambda: print('Wednesday')),
     4 : (lambda: print('Thursday')),
     5 : (lambda: print('Friday')),
     6 : (lambda: print('Saturday')),
     7 : (lambda: print('Sunday'))
     }

# Вызвать лямбда-выражение, выводящее название
вторника
Dict[2]() # Tuesday
```

Таблица переходов , в которой вычисляется площадь известных фигур.

```
import math

Area = {
    'Circle' : (lambda r: math.pi*r*r), # окружность
    'Rectangle' : (lambda a, b: a*b),   # прямоугольник
    'Trapezoid' : (lambda a, b, h: (a+b)*h/2.0) # трапеция
    }

# Вызвать лямбда-выражение, которое выводит площадь окружности
радиуса 2
print('Area of circle = ', Area['Circle'](2))

# Вывести площадь прямоугольника размером 10*13
print('Area of rectangle = ', Area['Rectangle'](10, 13))

# Вывести площадь трапеции для a=7, b=5, h=3
areaTrap = Area['Trapezoid'](7, 5, 3)
print('Area of trapezoid = ', areaTrap)
```

Совместное использование lambda-функции со встроенными функциями

Полезные встроенные функции (built-in)

print, len, str, int, float, list, tuple, dict, set, range

bool, enumerate, **zip**, reversed, sum, max, min, sorted, any, all

type, **filter**, **map**, round, divmod, bin, oct, hex, abs, ord, chr, pow

Если не знаете или забыли, что за функция, то набираете

```
help(имя функции)
```

```
пример: help(zip)
```

Функция **zip()**

Функция **zip()** принимает итерируемый объект, например, список, кортеж, множество или словарь в качестве аргумента. Затем она генерирует список кортежей, которые содержат элементы из каждого объекта, переданного в функцию.

```
# Пример
 A = [1, 2, 3, 4, 5]
 b = [2, 2, 9, 0, 9]
>>> zip(a, b)
[(1, 2),(2, 2),(3, 9),(4, 0),(5, 9)]
map( lambda pair: max(pair),
         zip(a, b)  # create a list of tuples
    )
[2, 2, 9, 4, 9]
```

Функция **unzip()**

Если есть список кортежей (упакованных), которые нужно разделить, можно использовать специальный оператор функции zip().Это оператор-звездочка (*)

```
# Пример

employees_zipped = [('Дима', 2), ('Марина', 9), ('Андрей', 18), ('Никита', 28)]

employee_names, employee_numbers = zip(*employees_zipped)


print(employee_names)

print(employee_numbers)
```

```
("Дима", "Марина", "Андрей", "Никита")

(2, 9, 18, 28)
```

Функция **filter()**

Функция **filter()** принимает два параметра — функцию и список для обработки. В примере мы применим функцию list(), чтобы преобразовать объект filter в список.

```
# Пример 1

numbers=[0,1,2,3,4,5,6,7,8,9,10]

list(filter(lambda x:x%3==0,numbers))

[0, 3, 6, 9]

#  Код берет список numbers, и отфильтровывает все элементы из него, которые не делятся нацело на 3. При этом фильтрация никак не изменяет изначальный список.
```

filter()

```
# Пример 2
even = lambda x: x%2 == 0
list(filter(even, range(11)))
[0, 2, 4, 6, 8, 10]
```

```
# Обратите внимание, что filter() возвращает итератор,
поэтому необходимо вызывать list, который создает
список с заданным итератором.
```

```
Реализация, использующая конструкцию генератора
списка, дает одинаковый результат:
>>> [x for x in range(11) if x%2 == 0]
[0, 2, 4, 6, 8, 10]
```

Функция **map()**

Функция **map()** в отличие от функции **filter()** возвращает значение выражения для каждого элемента в списке.

```
#Пример 1

numbers=[0,1,2,3,4,5,6,7,8,9,10]

list(map(lambda x:x%3==0,numbers))

[True, False, False, True, False, False, True, False,
False, True, False]


#Пример 2

list(map(lambda x: x.capitalize(), ['cat', 'dog',
'cow']))

['Cat', 'Dog', 'Cow']
```

reduce()

функция reduce() принимает два параметра — функцию и список. Сперва она применяет стоящую первым аргументом функцию для двух начальных элементов списка, а затем использует в качестве аргументов этой функции полученное значение вместе со следующим элементом списка и так до тех пор, пока весь список не будет пройден, а итоговое значение не будет возвращено. Для того, чтобы использовать reduce(), вы должны сначала импортировать ее из модуля functools.

reduce()

Пример

```
from functools import reduce
numbers=[0,1,2,3,4,5,6,7,8,9,10]
reduce(lambda x,y:y-x,numbers)
5


1-0=1
2-1=1
3-1=2
4-2=2
5-2=3
6-3=3
7-3=4
8-4=4
9-4=5
10-5=5
```

Можно ли в лямбда-выражениях использовать стандартные операторы управления if, for, while?

НЕТ

НО !

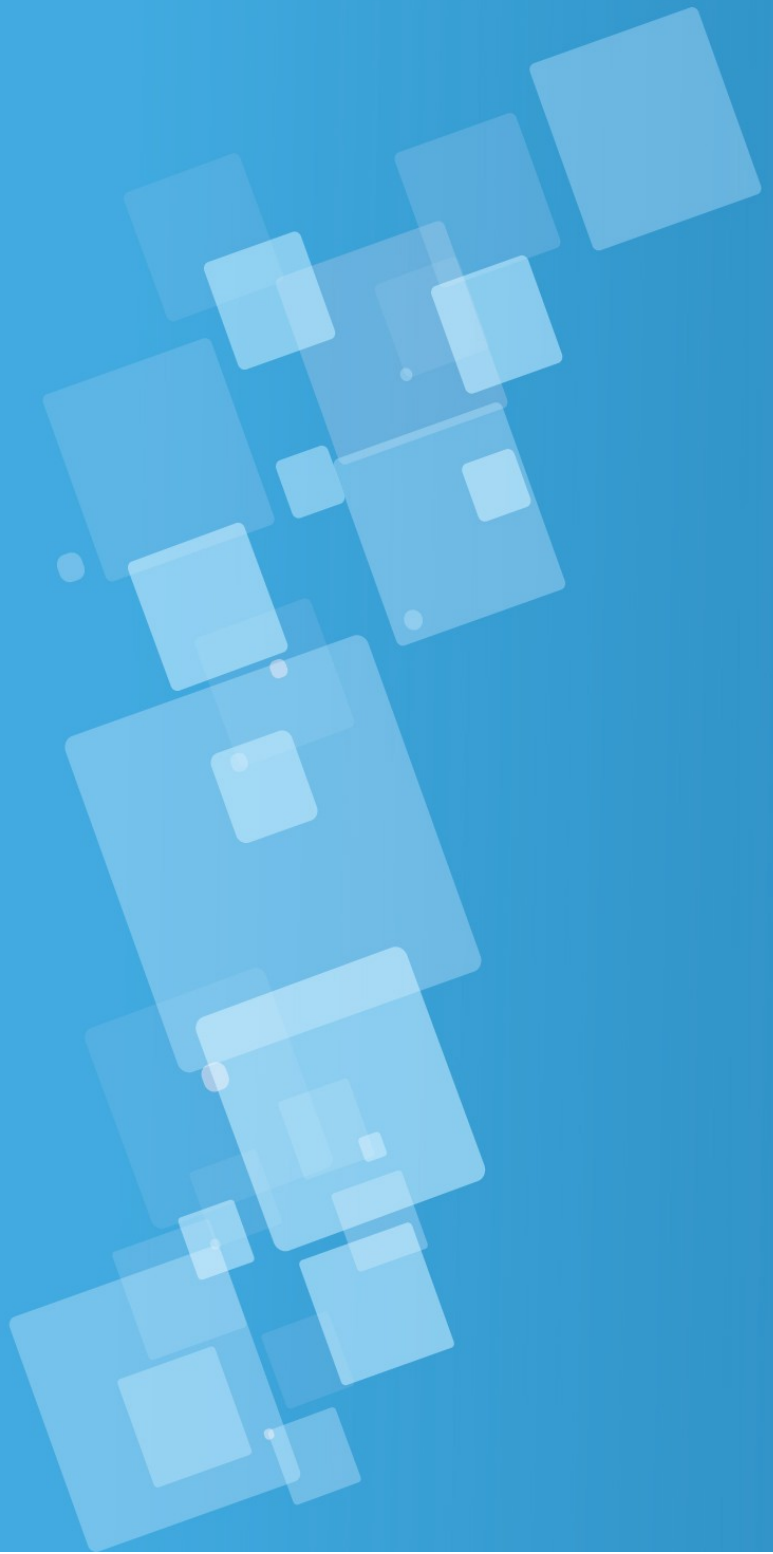
Можно использовать тернарный оператор.

```
lower = (lambda x, y: x if x < y else y)
# Вызов 1 способ
(lambda x, y: x if x < y else y)(10,3)

# Вызов 2 способ
lower(10,3)
3
```

Заключение

- Избегать чрезмерного использования лямбд
- Использовать лямбды с функциями высшего порядка или ключевыми функциями Python
- Функции более высокого порядка, такие как map(), filter() и functools.reduce(), могут быть преобразованы в более элегантные формы с небольшими изменениями, в частности, со списком или генератором выражений.



Функции

Часть III
Декораторы

Функции – это объекты

Функции Python относятся к объектам. Их можно **присваивать переменным,** хранить в структурах данных (коллекциях), **передавать их в качестве аргументов** другим функциям и даже **возвращать** их в качестве значений из других функций.

Почему функция является в языке Python объектом ?

Пример присвоения переменной

```
# Поскольку функция render является объектом. Ее можно присвоить еще одной переменной, точно также как это происходит с любым другим объектом.

def render(text):
    print(text.upper() + '!')

render('Hello')
#Hello!

show = render
show('Wellcome')
#Wellcome!

print(show is render)              # True
del render
print(show.__name__)
#render
```

Передача функций в качестве аргумента

Поскольку функции являются объектами, их можно передавать в качестве аргументов другим функциям.

```
def render(text):
    return text.upper() + '!'


def call_hello(func):    ← фукнция более выс. пор.
    result = func('Hello')
    print(result)


>>> call_hello(render)
```

Функция более высокого порядка

Функции которые принимают в качестве аргументов другие функции называют функциями более высокого порядка.

Классический пример таких функций это встроенная функция **map**, которая принимает в качестве аргументов: объект функцию и итерируемый объект. А затем вызывает эту функцию с каждым элементом итерируемого объекта, выдавая результат по мере прохождения итерируемого объекта.

```
def up_cap(text):
    return text.capitalize()


lst = list(map(up_cap, ['cat', 'dog', 'cow']))
print(lst)
```

Функции могут быть вложенными

Python допускает определение функций внутри других функций. Такие функции называются вложенным функциями (nested function) или внутренними функциями(inner function)

```
def speak(text):
    def whisper(t):
        return t.lower() +  '...'
    return  whisper(text)
>>>print(speak('Hello, World'))
```

Всякий раз, когда вы вызываете функцию speak, она определяет новую функцию whisper и затем после этого ее вызывает

Вложенная функция

Внимание! Вложенная функция whisper не существует за пределами функции speak

```
def speak(text):
    def whisper(t):
        return t.lower() +  '...'
    return  whisper(text)


# Попытка вызвать функцию whisper
>>> whisper('Hello, World')
NameError: name 'whisper' is not defined
```

Вопрос! Как получить доступ к вложенной функции whisper за пределами функции speak?

Возврат функции в качестве значения

Не забывайте функции являются объектами – и вы можете вернуть вложенную функцию в качестве значения.

```
def speak():
    def whisper(t):
        return t.lower() +  '...'
    return  whisper
# Получаем из функции speak объект функции whisper
wr = speak()
# Вызываем функцию с одним аргументом
print(wr('Hello, World'))
```

Функции могут захватывать локальные состояния

Перепишем функцию speek следующим образом

```
def speak(text):
    def whisper():
          print(text.lower() +  '...')
    return  whisper

foo = speak('Hello World')
bar = speak('Wellcome')
foo()  → hello, world...
bar()  → wellcome…
```

Внутренние функции получают доступ к родительскому параметру text, определенному в родительской функции. Такой доступ называется лексическим замыканием или для краткости замыканием. Замыкание помнит значения из своего лексического контекста, даже когда поток управления программы больше не находится в этом контексте.

Ключевые выводы

- В Python абсолютно все является объектом, включая функции. Их можно присваивать переменным, передавать в функции более высокого порядка а также возвращать из них.
- Функции могуть быть вложенными , и они могут захватывать и уносить с собой часть состояния родительской функции. Функции которые это делают, называются замыканиями.

Python

Function Decorator

Декаратор функции.

Alisa: *Декораторы – они что украсят нашу функцию ?*

Bob: *Нет , декоратор обертывает другую функцию и позволяет исполнять программный код до и после того, как обернутая функция выполниться.*

Декаратор функции.

Alisa: *А что нужно чтобы написать декоратор ?*

Bob: *Объяви функцию декоратор , в нее передай функцию которую будешь обертывать. Реализуй в декораторе вложенную функцию - обертку (wrapper) в котором будет содержатся логика до или после выполнения передаваемой функции. Вызови во вложенной функции , функцию из параметра. Верни вложенную функцию из декоратора.*

Декоратор

```
def decorator_render(func):  ← функция декоратор
    print("Call decorator function")
    def wrapper(text):   ← вложенная функция
        print(f"Логика перед вызовом функции")
        func(text)
        print("Логика после вызова функции")
    return wrapper


def render(text):
    print(f"Это функция render выводит text.upper()}")

# Вызываем декоратор и передаем туда функцию

>>> wrap_func = decorator_render(render)

#Получаем из декоратора вложенную функ. и вызываем её

>>> wrap_func('аргумент1')
```

Аргументы для вложенной функции

```
def decorator_render(func):
    print("Call decorator function")
    def wrapper(*args, **kwargs):
        print(f"Логика перед вызовом функции")
        func(*args, **kwargs)
        print("Логика после вызова функции")
    return wrapper


def render(text):
    print(f"Это функция render выводит text.upper()}")


wrap_func = decorator_render(render)
wrap_func('аргумент1')
```

Оператор @

Вызов декоратора можно переписать так:

```
def decorator_render(func):
    print("Call decorator function")
    def wrapper(text):
        print(f"Логика перед вызовом функции")
        res = func(text)
        print("Логика после вызова функции")
        return res
    return wrapper


@decorator_render

def render(text):
    return f"Это функция render выводит text.upper()}"

render('аргумент1')
```

Для чего использовать декораторы?

- Ведение протокола операции (журналирование)
- Обеспечение контроля за доступом и аутентификацией
- Функции хронометража
- Ограничение частоты вызова API
- Кеширование и др.