MIPT
MOSCOW INSTITUTE OF PHYSICS AND TECHNOLOGY

АТП кафедра Алгоритмов и Технологий Программирования

Области видимости. Замыкания. Декораторы.

Емельянов А. А.
login-const@mail.ru

2018

https://github.com/king-menin/python-course

План лекции

1. Анонимные функции
2. Области видимости
3. Замыкания
4. Декораторы

Анонимные функции

- lambda arguments : expression

```
In [243]: from string import ascii_lowercase, ascii_uppercase

          upper = lambda x: x.upper()
          print(*map(upper, ascii_lowercase), sep="")
          print(*map(lambda x: x.lower(), ascii_uppercase), sep="")

          ABCDEFGHIJKLMNOPQRSTUVWXYZ
          abcdefghijklmnopqrstuvwxyz
```

```
In [241]: upper

Out[241]: <function __main__.<lambda>>
```

Области видимости

Область видимости: конструкции управления потоком выполнения

- Циклы и условные конструкции не имеют своей области видимости (scope).

In [1]:
```
for index in range(100):
    sub_index = index // 10
    pass
print(index, sub_index)
```

```
99 9
```

In [2]:
```
outer_int = 100
if outer_int > 10:
    inner_checker = True
else:
    inner_checker = False
print(outer_int, inner_checker)
```

```
100 True
```

In [3]:
```
outer_int = 100
if outer_int < 10:
    inner_checker = True
else:
    inner_checker = False
print(outer_int, inner_checker)
```

```
100 False
```

In [4]:
```
for i in range(10):
    for j in range(10):
        k = i * j
        if k > 50:
            z = k
print(i, j, k, z)
```

```
9 9 81 81
```

Область видимости: функции

- Функции имеют свою область видимости.

In [17]:
```
def mul(*args):
    if not len(args):
        print("Length of args is zero!")
    res = 1
    for arg in args:
        if isinstance(arg, int) or isinstance(arg, float):
            res *= arg
        else:
            print("Skip arg {} (not int or float)".format(arg))
    return res
```

In [21]:
```
print(res)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-21-2bc7c0cc4173> in <module>()
----> 1 print(res)

NameError: name 'res' is not defined
```

Область видимости: функции

- Функции имеют доступ ко внешним переменным

In [15]:
```
def print_outer_index():
    print("Outer index {}".format(index_))
for index_ in range(5):
    print_outer_index()
```

```
Outer index 0
Outer index 1
Outer index 2
Outer index 3
Outer index 4
```

Правило LEBG

- Поиска имен происходит во время исполнения, а не во время объявления.
- Поиск доступных переменных в python идет в порядке:

  -> Local (локальные переменные) **->**
  -> Enclosing (область памяти, откуда произошел вызов) **->**
  -> Global (переменные, объявленные в основной части программы) **->**
  -> Built-In (переменные, создаваемые автоматически при запуске интерпретатора).

Правило LEBG

```
# built-in
sum(range(5))
```

```
10
```

```
# global
sum = 9
```

```
def outer():
    # enclosing
    def inner():
        # enclosing
        def inner2():
            # local
            sum = 6
            print(sum)
        sum = 7
        try:
            print(sum)
        except NameError:
            print("NameError exeption!")
        inner2()
    inner()
    sum = 8
    print(sum)
```

```
outer()
```

```
7
6
8
```

globals and locals

```
In [87]: globals() is locals()
Out[87]: True

In [96]: a = 10
         def get_locals(x):
             a = 1
             print(locals())

In [97]: get_locals(5)
         {'a': 1, 'x': 5}
```

Присваивание

- Присваивание всегда происходит в локальной области видимости и не затрагивает более высокие.
- Менять это поведение можно с помощью global и nonlocal.

```
In [102]: a = 10
          def get_locals(x):
              a = 1
              print(locals())
              return a

In [103]: a, get_locals(5)
          {'a': 1, 'x': 5}
Out[103]: (10, 1)
```

global и local

WTF?

In [117]:
```
a = 10
def f():
    print(a)
f()
```

```
10
```

In [121]:
```
a = 10
def f():
    print(a)
    a = 5
f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-121-7421dde9c456> in <module>()
      3     print(a)
      4     a = 5
----> 5 f()

<ipython-input-121-7421dde9c456> in f()
      1 a = 10
      2 def f():
----> 3     print(a)
      4     a = 5
      5 f()

UnboundLocalError: local variable 'a' referenced before assignment
```

global и local

- Любая переменная, которая изменяется или создается внутри функции является локальной.

```
In [117]: a = 10
          def f():
              print(a)
          f()
```

```
10
```

```
In [121]: a = 10
          def f():
              print(a)
              a = 5
          f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-121-7421dde9c456> in <module>()
      3     print(a)
      4     a = 5
----> 5 f()

<ipython-input-121-7421dde9c456> in f()
      1 a = 10
      2 def f():
----> 3     print(a)
      4     a = 5
      5 f()

UnboundLocalError: local variable 'a' referenced before assignment
```

global

- Обращение к глобальным переменным.

In [105]:
```
a = 10
def sum_a(x):
    global a
    a += x
    return a
```

In [108]:
```
a, sum_a(1)
```

Out[108]:
```
(12, 13)
```

In [137]:
```
x = 40
def f():
    x = 42
    def g():
        global x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

f()
print("x in main: " + str(x))
```

```
Before calling g: 42
Calling g now:
After calling g: 42
x in main: 43
```

In [140]:
```
x = 10
def f():
    x = 40
    print(x)
    global x
    x = 20
```

```
  File "<ipython-input-140-310ee235a553>", line 5
    global x
SyntaxError: name 'x' is used prior to global declaration
```

nonlocal

- Обращение к переменным следующего уровня.

In [142]:
```
def f():
    x = 42
    def g():
        nonlocal x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

x = 3
f()
print("x in main: " + str(x))
```

```
Before calling g: 42
Calling g now:
After calling g: 43
x in main: 3
```

In [146]:
```
def f():
    def g():
        nonlocal x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    # x = 42
    print("After calling g: " + str(x))

x = 3
f()
print("x in main: " + str(x))
```

```
  File "<ipython-input-146-349d08298903>", line 3
    nonlocal x
SyntaxError: no binding for nonlocal 'x' found
```

In [143]:
```
x = 10
def f():
    x = 40
    print(x)
    nonlocal x
    x = 20
```

```
  File "<ipython-input-143-7336c4019f7d>", line 5
    nonlocal x
SyntaxError: name 'x' is used prior to nonlocal declaration
```

nonlocal – еще пример

```
def build_functions(value=[]):
    a = 1
    def get():
        return value
    def put(new_value):
        nonlocal value
        if isinstance(value, list):
            value.append(new_value)
        else:
            value += new_value
    return get, put
```

```
for _ in range(3):
    get1, put1 = build_functions()
    get2, put2 = build_functions([])
    get3, put3 = build_functions(0)
    put1(10)
    put2(20)
    put3(1)
    print(get1(), get2(), get3())
```

```
[10] [20] 1
[10, 10] [20] 1
[10, 10, 10] [20] 1
```

```
value = []
value2 = 0
for _ in range(3):
    get1, put1 = build_functions(value)
    get2, put2 = build_functions(value)
    get3, put3 = build_functions(value2)
    put1(10)
    put2(20)
    put3(1)
    print(get1(), get2(), get3())
```

```
[10, 20] [10, 20] 1
[10, 20, 10, 20] [10, 20, 10, 20] 1
[10, 20, 10, 20, 10, 20] [10, 20, 10, 20, 10, 20] 1
```

Замыкания

Замыкания

- То что происходит называется замыканием (или closure) – функция, которая ссылается на переменные в своём контексте.

```
In [173]: def outer_func(x):
              def inner_func(y):
                  # inner_func замкнуло в себе x
                  return y + x
              return inner_func

In [174]: inner_func1 = outer_func(10)
          inner_func2 = outer_func(20)
          inner_func1(10), inner_func2(10)

Out[174]: (20, 30)

In [175]: print(*map(lambda x: x.cell_contents, inner_func1.__closure__))

          10

In [176]: print(*map(lambda x: x.cell_contents, inner_func2.__closure__))

          20
```

Замыкания и атрибуты

In [205]:
```
def get_adder(x):
    def adder(y):
        return adder.x + y

    def update(x):
        adder.x = x

    adder.x = x
    adder.update = update

    return adder
```

In [206]:
```
inc_adder = get_adder(1)
```

In [207]:
```
inc_adder(2)
```

Out[207]: 3

In [209]:
```
inc_adder.update(3)
inc_adder(4)
```

Out[209]: 7

Замыкания и атрибуты: пример

In [233]:
```
def connect_manager(connector_id, prev_connections=[]):
    def connector(guid):
        connector.user_guids.append(guid)

    def clear():
        connector.user_guids = []

    connector.user_guids = prev_connections
    connector.clear = clear
    connector.id = connector_id

    return connector
```

In [234]:
```
connector1 = connect_manager(1)
connector2 = connect_manager(2)
```

In [235]:
```
connector1("guid1")
connector2("guid2")
```

In [236]:
```
connector1.user_guids, connector2.user_guids
```

Out[236]:
```
(['guid1', 'guid2'], ['guid1', 'guid2'])
```

In [237]:
```
connector1.id, connector2.id
```

Out[237]:
```
(1, 2)
```

Декораторы

Декораторы

In [247]:
```
def deprecated(func):
    def wrapper(*args, **kwargs):
        print("Function {} is deprecated!".format(func.__name__))
        return func(*args, **kwargs)
    return wrapper

def f(x):
    return x
# WTF
f = deprecated(f)
f(10)
```

```
Function f is deprecated!
```

Out[247]:
```
10
```

Декораторы

- Декоратор – функция которая принимает другую функцию и что-то возвращает.
- То есть, декораторы в python — это просто синтаксический сахар для конструкций вида:

```
In [249]: def deprecated(func):
              def wrapper(*args, **kwargs):
                  print("Function {} is deprecated!".format(func.__name__))
                  return func(*args, **kwargs)
              return wrapper

          @deprecated
          def f(x):
              return x

          f(10)

          Function f is deprecated!

Out[249]: 10
```

Декораторы: проблемы

- Декораторы несколько замедляют вызов функции, не забывайте об этом.

- Вы не можете "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, что может затруднить отладку.

```
In [254]: @deprecated
          def f(x):
              return x
          f.__name__

          Decorator created!

Out[254]: 'wrapper'
```

Декораторы: решение проблемы

- Модуль functools

```
In [271]: import functools

          def deprecated(func):
              @functools.wraps(func)
              def wrapper(*args, **kwargs):
                  print("Function {} is deprecated!".format(func.__name__))
                  return func(*args, **kwargs)
              return wrapper

          @deprecated
          def f(x):
              return x

In [272]: f.__name__

Out[272]: 'f'
```


WE NEED TO GO
DEEPER
troll-face.ru

Декораторы с аргументами

In [277]:
```
import sys
import functools

def decorator_maker(dest=sys.stdout):
    print("Decorator creater!")
    def deprecated(func):
        print("Decorator created!")
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print("Function {} is deprecated!".format(func.__name__), file=dest)
            return func(*args, **kwargs)
        return wrapper
    return deprecated

@decorator_maker(sys.stderr)
def f(x):
    return x
```

```
Decorator creater!
Decorator created!
```

In [278]:
```
f(1)
```

```
Function f is deprecated!
```

Out[278]: 1

Классы декораторы

```
In [289]:
class Logger(object):
    def __init__(self, func):
        self.func = func
        self.log = []
    def __call__(self, *args, **kwargs):
        self.log.append((args, kwargs))
        return self.func(*args, **kwargs)
    pass

@Logger
def f(x, y=1):
    return x * y
```

```
In [291]:
f(1, 2)
```

```
Out[291]:
2
```

```
In [292]:
f.log
```

```
Out[292]:
[((1, 2), {})]
```

Несколько декораторов

In [296]:
```
import sys
import functools

def decorator_maker(dest=sys.stdout):
    def deprecated(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print("Function {} is deprecated!".format(func.__name__), file=dest)
            return func(*args, **kwargs)
        return wrapper
    return deprecated

class Logger(object):
    def __init__(self, func):
        self.func = func
        self.log = []
    def __call__(self, *args, **kwargs):
        self.log.append((args, kwargs))
        return self.func(*args, **kwargs)

@Logger
@decorator_maker(sys.stderr)
def f(x):
    print(x)
```

In [297]:
```
f(1)
```

```
1
Function f is deprecated!
```

In [298]:
```
f.log
```

Out[298]:
```
[((1,), {})]
```

Декораторы в классах

- staticmethod
  - Применяется к методу класса.
  - Делает метод статическим.
  - Позволяет игнорировать экземпляр (self).

- classmethod
  - В метод класса первым аргументом неявным образом передаётся класс (аналогично метод экземпляра получает в первом аргументе сам экземпляр).

Декорирование классов

In [336]:
```
def singleton(cls):
    instance = None
    @functools.wraps(cls)
    def inner(*args, **kwargs):
        nonlocal instance
        if instance is None:
            instance = cls(*args, **kwargs)
        return instance
    return inner

@singleton
class A:
    "nothing"
A() is A()
```

Out[336]: True

In [340]:
```
def class_dec(cls):
    class ClassWrapper(cls):
        def add(self, x, y):
            return x + y
    return ClassWrapper

@class_dec
class BinaryOperation(object):
    def mul(self, x, y):
        return x * y
    pass
```

In [341]:
```
b = BinaryOperation()
```

In [343]:
```
b.__class__
```

Out[343]: __main__.class_dec.<locals>.ClassWrapper

Домашнее задание 4

- **Целью этого задания является знакомство с областями видимости и декораторами в python.**
- **Deadline (получение полных баллов):** 28.03.2018

- **Адрес:** login-const@mail.ru

- Задание состоит из трех частей (разных декораторов):
  - cached,
  - checked,
  - Logger.

- **Текс условия доступен по** ссылке.

СПАСИБО ЗА ВНИМАНИЕ