Функции. Декораторы

Python 5.0

Потылицина Е.М.

Функциональный стиль

Благодаря функциональным возможностям языка Python, на нем многое можно написать в одну строчку.

Считываем числа, вводимые через пробел, и сортируем их по возрастанию:

```
print(sorted(map(int, input().split())))
```

**Ознакомьтесь с разделом «Функциональное программирование на opened.ru*

Функциональный стиль

заполняем таблицу N * M числами от 1 до M * N змейкой:

```
[ [ m * i + j * (i % 2 == 0) + (m - 1 - j) * (i % 2 == 1) for j in range(m)] for i in range(n)]
```

кодируем текст шифром Цезаря:

```
print("".join([chr(ord(c) + 1) for c in input()]))
```

Рекурсия

Функции могут вызывать и другие функции, и даже вызывать сами себя. Рассмотрим это на примере функции вычисления факториала.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Подобный прием (вызов функцией самой себя) называется рекурсией, а сама функция называется **рекурсивной.**

Рекурсия

При этом программирование рекурсии выглядит так. Функция должна сначала проверить, не является ли переданный набор параметров простым (крайним) случаем. В этом случае функция должна вернуть значение (или выполнить действия), соответствующие простому случаю.

Иначе функция должна вызвать себя рекурсивно для другого набора параметров, и на основе полученных значений вычислить значение, которое она должна вернуть.

Рекурсия

Как это работает? Допустим, мы вызвали функцию $factorial(4)$. Будет вызвана функция, у которой значение параметра $n == 4$. Она проверит условие $n == 0$, поскольку условие ложно, то будет выполнена инструкция $return\ n * factorial(n - 1)$. Но чтобы вычислить это значение, будет вызвана функция $factorial(3)$, т. к. параметр n имеет значение, равное 4. Теперь в памяти будет находиться две функции factorial – одна со значением параметра $n == 4$, а другая – $n == 3$. При этом активна будет последняя функция.

Эта функция в свою очередь вызовет функцию $factorial(2)$, та вызовет функцию $factorial(1)$, затем $factorial(0)$. В случае этой функции ничего более вызвано не будет, функция просто вернет значение 1, и управление вернется в функцию $factorial(1)$. Та умножит значение $n == 1$ на значение 1, которое вернула функция $factorial(0)$, и вернет полученное произведение, равное 1. Управление вернется в функцию $factorial(2)$, которая умножит $n == 2$ на значение 1, которое вернула функция $factorial(1)$ и вернет полученное произведение, равное 2. Функция $factorial(3)$ вернет $3 * 2 == 6$, а функция $factorial(4)$ вернет $4 * 6 == 24$.

Вычисление суммы натуральных чисел от 1 до n

```
def sum(n):
    if n == 1:
        return 1
    else:
        return n + sum(n — 1)
```

Проверка строки на палиндромность

Напишем функцию IsPalindrome, которая возвращает значение типа bool в зависимости от того, является ли строка палиндромом.

Крайнее значение — пустая строка или строка из одного символа всегда палиндром. Рекурсивный переход — строка является палиндромом, если у нее совпадают первый и последний символ, а также строка, полученная удалением первого и последнего символа является палиндромом.

```
def IsPalindrome(S):
    if len(S) <= 1:
        return True
    else:
        return S[0] == S[-1] and IsPalindrome(S[1:-1])
```

lambda

lambda a, b: a + b

Может быть записана как элемент любой конструкции python

a = [4, 5, lambda: print(‘lambda’), 7, 9]

a[2]

<function <lambda> at 0x0000025C677CF280>

список хранит ссылку на функцию

a[2]()

lambda

lambda

Обычно lambda функция возвращает какой-то результат и для этого и используется

```
lst = [5, 3, 0, -6, 8, 10, 1]
def get_filter(a, filter=None):
    if filter is None:
        return a
    res = []
    for x in a
        if filter(x):
            res.append()
    return res
```

lambda

```
r = get_filter(lst, lambda x: x % 2 ==0)
print(r)
```

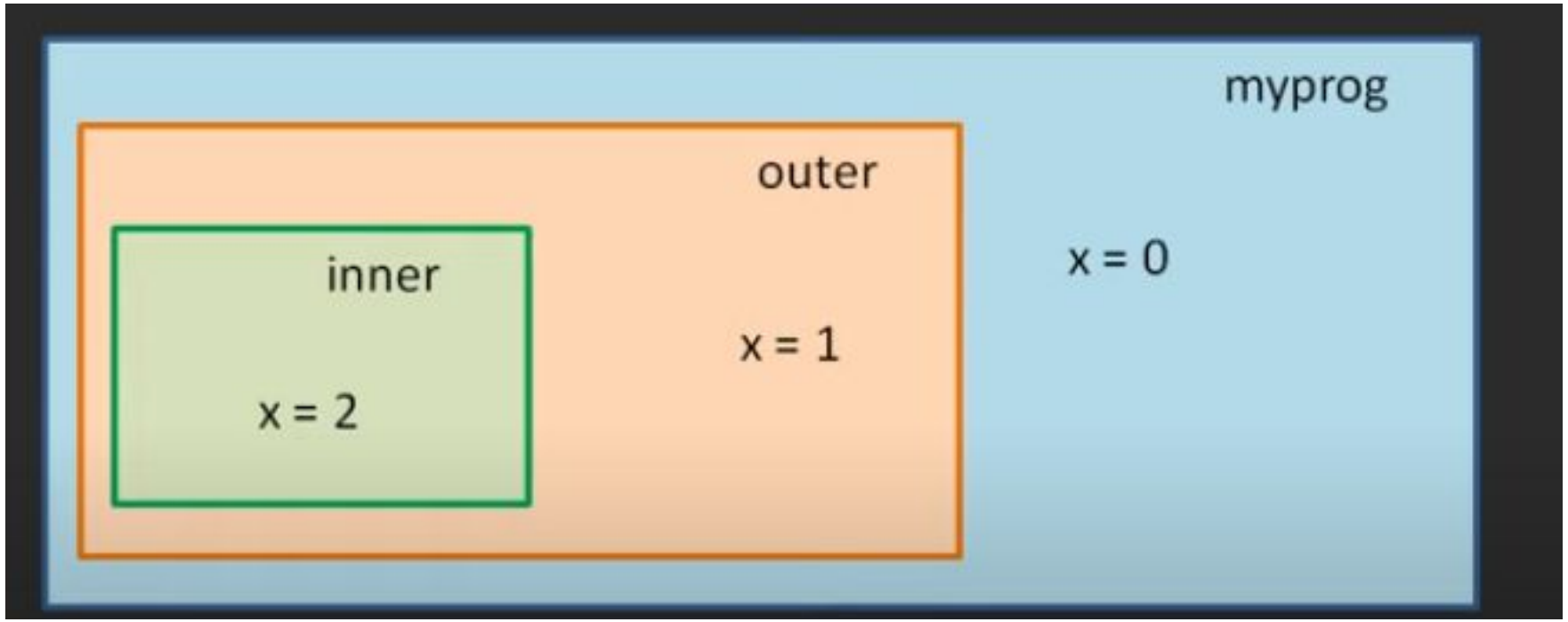
[0, -6, 8, 10]

Ограничения: сокращенный функционал

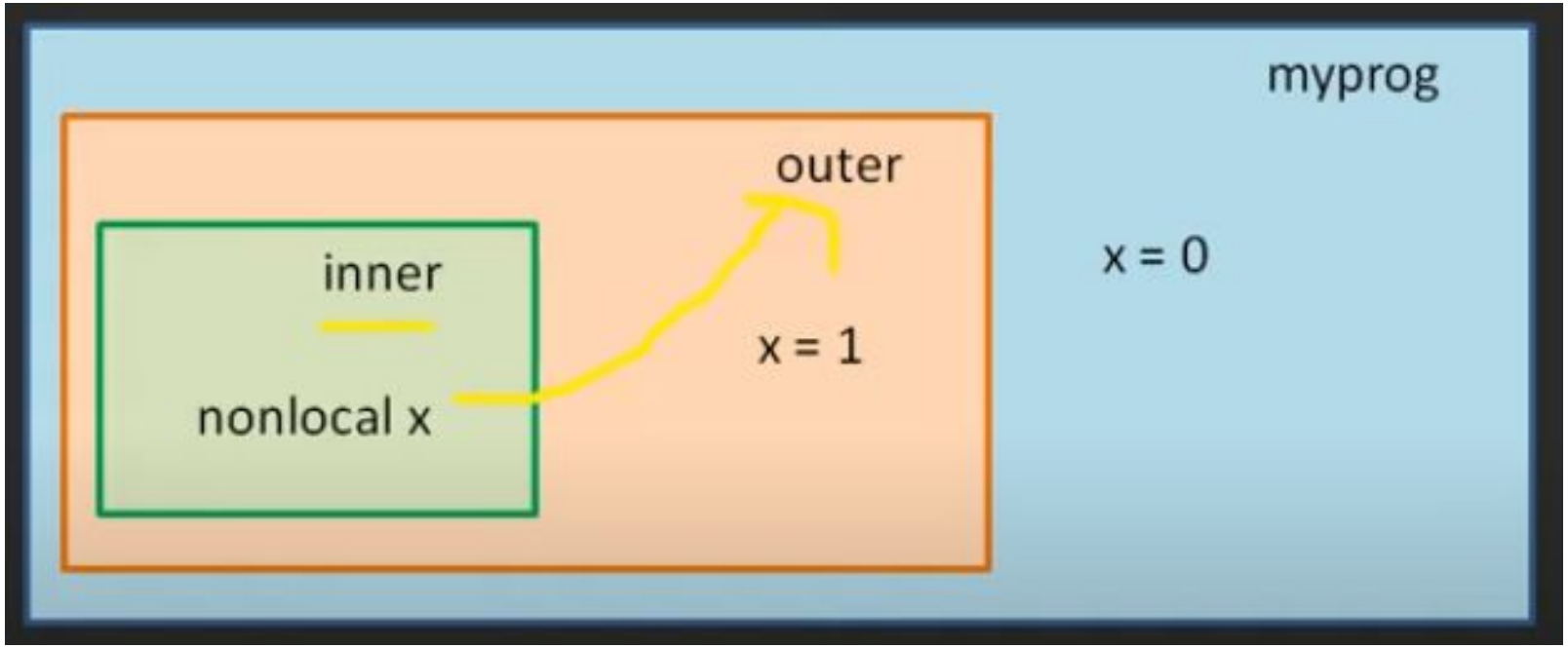
Создают новый объект на основе входных параметров

Не используют оператор присваивания

nonlocal



nonlocal



nonlocal

```
def outer():
    x = 1
    def inner():
        x = 2
        print(“inner: “, x)

    inner()
    print(‘outer: ‘, x)
outer()
print(‘global: ‘, x)
```

nonlocal

```
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print(“inner: “, x)

     inner()
    print(‘outer: ‘, x)
outer()
print(‘global: ‘, x)
```

Замыкания

```
def say_name(name):
    def say_goodbye():
        print (‘Goodbye, ‘ + name + ‘!’)

    say_goodbye()

say_name(‘Ivan’)
```

Goodbye, Ivan!

```
def say_name(name):
    def say_goodbye():
        print (‘Goodbye, ‘ + name + ‘!’)

    return say_goodbye

say_name(‘Ivan’)
```

пусто, потому что внутренняя функция не вызывается

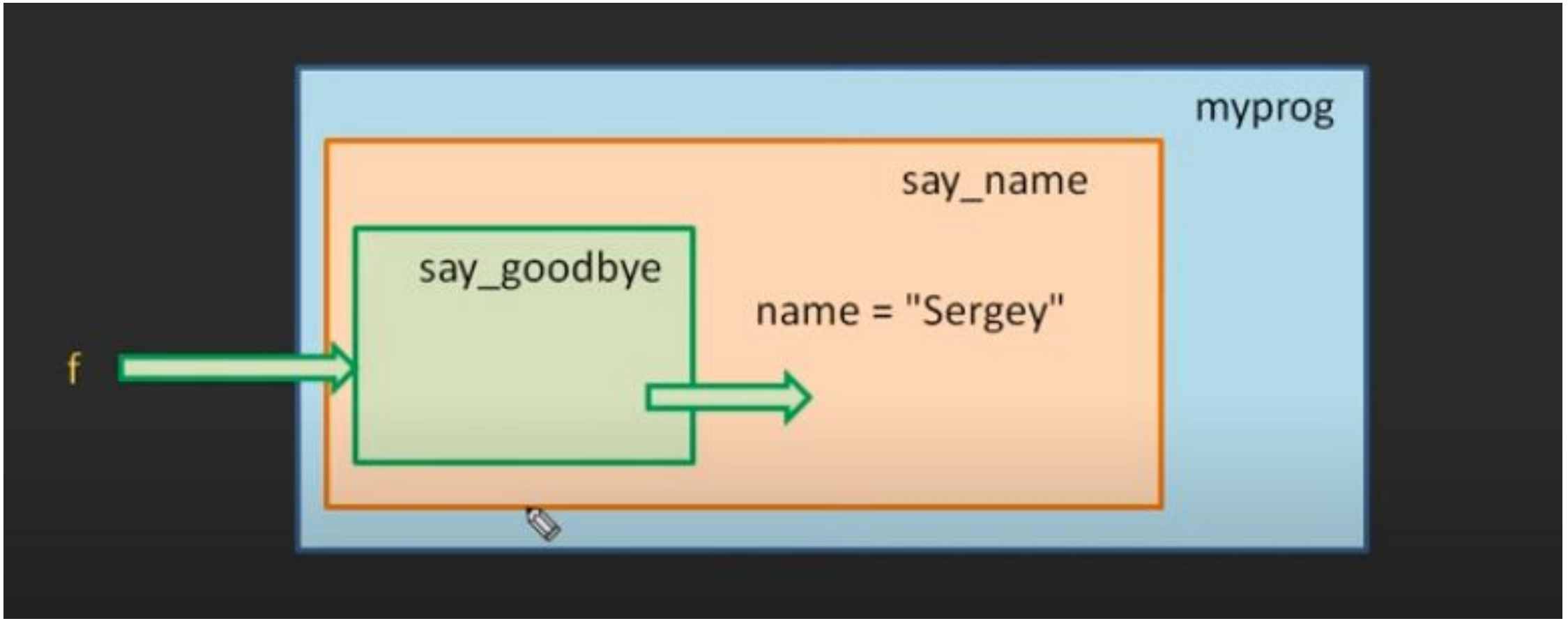
```
def say_name(name):
    def say_goodbye():
        print (‘Goodbye, ‘ + name + ‘!’)

    return say_goodbye

f = say_name(‘Ivan’)
f()
```

Goodbye, Ivan!

можно сохранить ссылку на внутреннюю функцию и вызвать через ссылку f

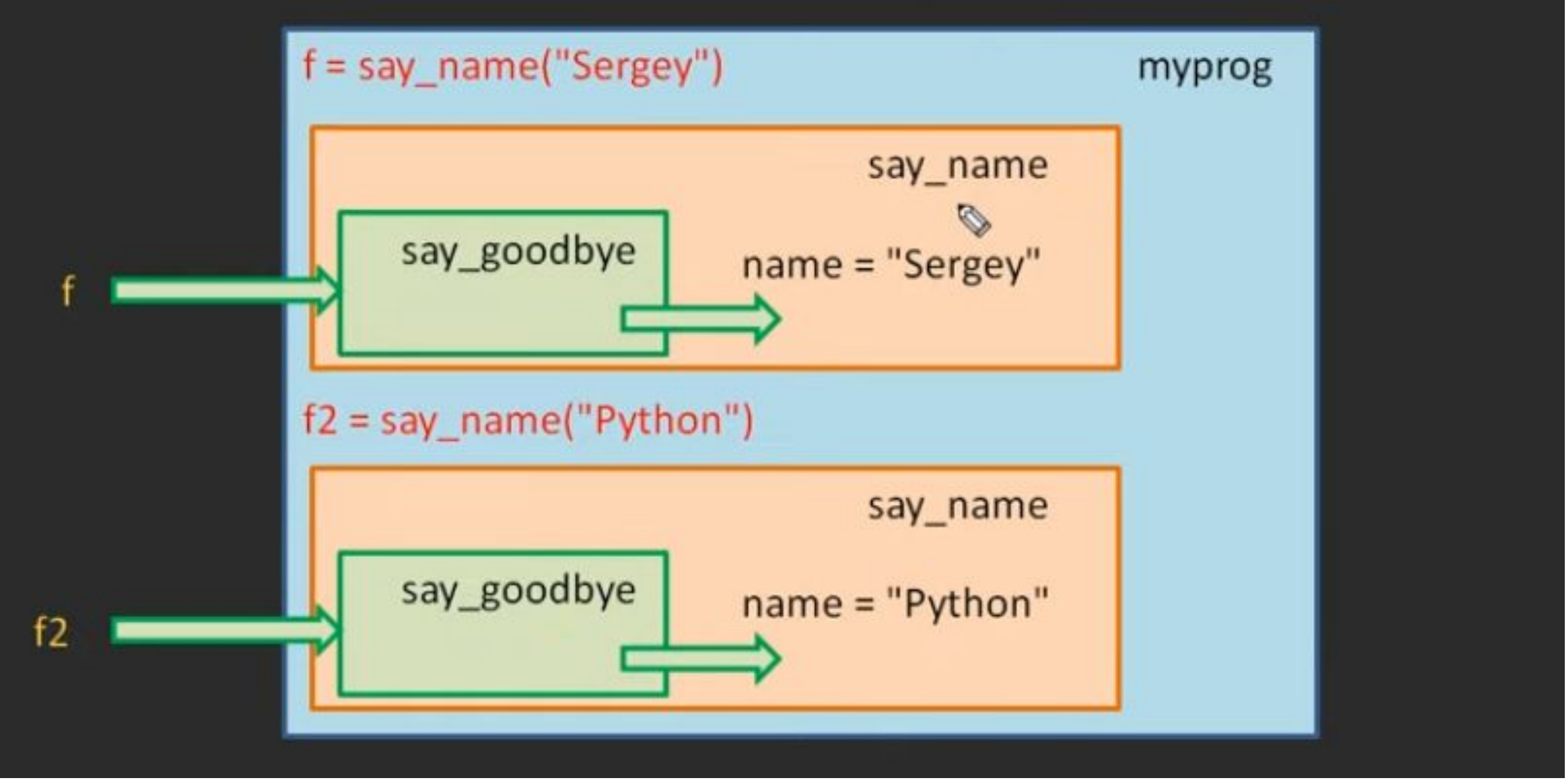


Если имеется глобальная ссылка f на внутреннее локальное окружение say_goodbye, то оно продолжает существовать и не удаляется сборщиком мусора.
Вместе с локальным существуют все остальные с ним связанные say_name---myprog
все не пропадают пока существует f

Замыкание

Эффект существования локальных окружений и продолжение использования внешних окружений - замыкания

f (принадлежит глобальной области видимости) ссылается на локальное окружение say_goodbye, say_goodbye ссылается на внешнее say_name, say_name -> myprog



при каждом вызове say_name будет создаваться свое новое независимое локальное окружение

```
def say_name(name):
    def say_goodbye():
        print ('Goodbye, ' + name + '!')

    return say_goodbye

f = say_name('Ivan')
f2 = say_name('Masha')
f()
f2()
```

Goodbye, Ivan!

Goodbye, Masha!

при каждом вызове внешней функции say_name создается свое независимое локальное окружение

Например можно создать функцию счетчик, которая при каждом новом запуске увеличивает свое значение на 1

```
def counter (start=0):
    def step():
        nonlocal start
        start +=1
        return start
    return step   # возвращает ссылку на функцию step
```

работают независимо друг от друга
с помощью замыкания можно создавать много независимых счетчиков

```
c1 = counter(10) #первый счетчик
c2 = counter()
print(c1(), c2())
print(c1(), c2())
print(c1(), c2())
```

11 1
12 2
13 3

```
def strip_string(strip_chars=‘ ‘):
    def do_strip(string):
        return string.strip(strip_chars)
    return do_strip

strip1 = strip_string()
strip2 = strip_string(‘ !?,.’)

print(strip(‘ hello python!.. ‘))
print(strip2(‘ hello python!..’))
```

функция, которая удаляет ненужные символы в начале и конце строки
внутри определена функция (строка, из которой удаляем символы)

в первой удалили только пробелы
с помощью замыкания описали алгоритм для обработки строк,
вызвали с двумя параметрами (для внешней функции с удаляемыми символами и для внутренней функции со строкой)

внутренняя функция возвращает преобразованную строку

Decorators

Декораторы позволяют «декорировать» функцию.

Декораторы можно представить себе как функции, которые меняют *функциональность* другой функции. Они помогают сделать Ваш код короче, а также по стилю более похожим на стиль Python.

Создадим простую функцию

```
def simple_func():
    #выполняем действия
    return <результат>
```

А теперь мы хотим добавить в функцию дополнительные возможности:

```
def simple_func():
#дополнительные действия
#выполняем действия
    return <результат>
```

Существует два варианта, как это сделать:

- добавить новую функциональность в старую функцию
- создать новую функцию, скопировать в нее старый код и добавить новый код

Но что, если вы затем захотите убрать эту новую функциональность? Можно ли включать/выключать функциональность?

Декораторы позволяют добавить функциональность в уже существующую функцию. Они используют оператор @ и помещаются поверх исходной функции

Когда дополнительный код уже больше не нужен вы просто удаляете декоратор

Декоратор

Декораторы — это, по сути, просто своеобразные «обёртки», которые дают нам возможность делать что-либо до и после того, что сделает декорируемая функция, не изменяя её.

Для того, чтобы понять, как работают декораторы, в первую очередь следует вспомнить, что функции в python являются объектами, соответственно, их можно возвращать из другой функции или передавать в качестве аргумента. Также следует помнить, что функция в python может быть определена и внутри другой функции.

Вы можете легко добавить новую функциональность с помощью декоратора:

```
@some_decorator
def simple_func():
  #выполняем действия
  return <результат>
```

Подробнее

Создадим простую функцию

```
def func():
    return 1
```

Поскольку функции являются объектами, мы можем сохранить эту функцию в переменной, затем выполнить ее с помощью этой переменной.

Пример 1

```
def hello():
    return "Привет!"


hello()
```

Привет!

```
greet = hello       # связываем функцию hello с переменной greet
greet()       #теперь мы  можем вызывать "hello" через "greet"
```

Привет!

```
del hello  #если удалить hello, то greet все еще будет работать
```

Пример 2

```
def hello (name = ‘Мария’): #функция со значением по умолчанию
    print (‘Мы запустили функцию hello’)

hello()          #вызываем функцию
```

Мы запустили функцию hello #получаем фразу

определим функцию внутри этой функции

```
def hello (name = ‘Мария’): #функция со значением по умолчанию
    print (‘Мы запустили функцию hello’)
    def greet ():
        return ‘\t Это функция greet внутри hello’
    print(greet())

hello()
```

Мы запустили функцию hello()

Это функция greet внутри функции hello

добавим еще одну вложенную функцию:

```
def hello(name='Name'):
    print('Запущена функция hello()')

    def greet():
        return '\t Мы находимся внутри функции greet()'

    def welcome():
        return "\t Мы находимся внутри функции welcome()"

    print(greet())
    print(welcome())
    print("Теперь мы вернулись в функцию hello()")
```

```
hello()
Запущена функция hello()
    Мы находимся внутри функции greet()
    Мы находимся внутри функции welcome()
Теперь мы вернулись в функцию hello()
welcome()
---------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-efaf77b113fd> in <module>()
----> 1 welcome()

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

из-за области видимости функция welcome() не определена вне функции hello(). Теперь посмотрим как можно возвращать функции изнутри функций:

```
def hello(name='Name'):

    def greet():
        return '\t Мы находимся внутри функции greet()'

    def welcome():
        return "\t Мы находимся внутри функции welcome()"

    if name == 'Name':
        return greet
    else:
        return welcome
```

Теперь посмотрим какая функция будет возвращена, если мы установим x = hello(), обратите внимание что пустые скобки означают, что имя name определено как ‘Name’.

```
x = hello()
x
<function __main__.hello.<locals>.greet>
```

Мы видим что x указывает на функцию greet внутри функции hello.

```
print(x())
```

Мы находимся внутри функции greet()

Пояснение

В операторе **if/else** мы возвращаем greet и welcome, а не **greet()** и **welcome().**

Это потому что, когда мы пишем скобки после названия функции, то запускаем эту функцию. Однако, когда мы не пишем скобки, то мы можем передавать эту функцию, не запуская её.

Когда мы пишем **x = hello()**, то запускается функция hello(), и поскольку по умолчанию name равно Name, то возвращается функция greet.

Если мы поменяем команду на **x = hello(name = "Sam")**, то вернется функция welcome.

Функции как параметры

```
def hello():
    return 'Hi, Name!'


def other(func):
    print('Здесь будет указан другой код')
    print(func())
hello()       # запускаем hello()
```

Hi, Name!

```
other(hello)  # запускаем other, в качестве параметры используем hello
```

Здесь будет указан другой код

Hi, Name!

Создаем декоратор

```
def new_decorator(func):      #в качестве параметра другая функция

    def wrap_func():              #дополнительная функциональность
        print("Здесь находится код, до запуска функции")
        func()    #вызываем исходную функцию, которую принимаем в качестве параметра

        print("Этот код запустится после функции func()")

    return wrap_func

def func_needs_decorator():
    print("Для этой функции нужен декоратор")
```

пояснение

Функция **new_decorator** принимает на вход функцию и возвращает тоже функцию

На входе любая исходная функция, на выходе другая функция, в которую мы добавили дополнительный код до и после исходной функции, т.е. **декорировали** исходную функцию.

продолжение

Теперь создадим функцию, которую будем декорировать

```
def func_needs_decorator():
    print(‘Эта функция нуждается в декораторе’)
```

если мы просто запустим эту функцию, то получим одну фразу

```
func_needs_decorator()
```

Эта функция нуждается в декораторе

продолжение

теперь запустим new_decorator и сохраним результат в переменную

```
decorated_func = new_decorator(func_needs_decorator)
```

запускаем

```
decorated_func()
```

Здесь находится код, до запуска функции
Для этой функции нужен декоратор
Этот код запустится после функции func

Декоратор здесь служит оберткой функции, поменяв её поведение.

Теперь посмотрим, как можно переписать этот код с помощью символа @, который используется в Python для декораторов:

```
@new_decorator
def func_needs_decorator():
    print("Для этой функции нужен декоратор")
```

теперь при вызове функции:

```
func_needs_decorator()
```

Здесь находится код, до запуска функции
Для этой функции нужен декоратор
Этот код запустится после функции func

ИТОГ

Теперь, если мы захотим отключить дополнительную функциональность, то просто сделаем так

```
#@new_decorator
def func_needs_decorator():
    print("Для этой функции нужен декоратор")
```

символ @ используется для автоматизации, чтобы сделать код более чистым. Вы будете часто встречаться с декораторами, если начнете использовать Python для веб-разработки, например во Flask или Django

Структура

```
# Декораторы функций
def func_decorator(func):
    def wrapper():
        print("------ что-то делаем перед вызовом функции ------")
        func()
        print("------ что-то делаем после вызова функции ------")

    return wrapper


def some_func():
    print("Вызов функции some_func")
```

Практика рекурсия

1) Дан список чисел, необходимо просуммировать его.

Написать функцию Sum(A)

2) Дан список чисел, необходимо найти наибольшее значение в нем.

Практика декораторы

1. Написать декоратор, который оборачивает строку в теги

   Написать декоратор, который оборачивает строку в теги

   Применить оба декоратора.

2. Написать декоратор, который в любую функцию, в __doc__ дописывает имя и фамилию автора.

@createdbyme

def a():

Справочная информация для практики

Строки документации Python

Строки документации в Python — это строки, которые пишутся сразу после определения функции, метода, класса или модуля. Они используются для документирования нашего кода.

Мы можем получить доступ к этим строкам документации, используя атрибут __doc__.

Атрибут __doc__

Всякий раз, когда строковые литералы присутствуют сразу после определения функции, модуля, класса или метода, они становятся специальным атрибутом __doc__ этого объекта. Позже мы можем использовать этот атрибут для получения этой строки документации.

пример

```
def square(n):
    '''Принимает число n, возвращает квадрат числа n'''
    return n**2

print(square.__doc__)
```

Принимает число n, возвращает квадрат числа n

Здесь мы получили доступ к документации нашей функции square() с помощью атрибута __doc__.