Основи на Програмирането

Лекция 7

- ✓ Функции. Аргументи. Области на действие. Полиморфизъм.

Какво ще научите

- ✓ Структуриране и декомпозиране
- ✓ Скриване на подробности
- ✓ Използване на функции
- ✓ Елементи от дефиницията на една функция
- ✓ Предаване на аргументи и връщане на резултат
- ✓ Области на действие на имената
- ✓ Използване на функция като аргумент
- ✓ Особености по подаване на аргументи

Как съставяме програми

- Изучихме основните конструкции в езика
- Знаем как да създаваме програма за даден алгоритъм и да я запазваме във файл
- Всеки файл включва множество от команди и изрази
- Всеки файл има собствено пространство от обекти и свойства

Програми на Python

- Програмите на Python са модули, създавани като текстови файлове с разширението .py
- Всеки модул в Python има свое пространство на имената
- Това пространство в един Python модул е глобално

Програми на Python

- Програмите и модулите в езика Python се различават само по начина на използване:
  - .py файловете, които се изпълняват от интерпретатора, са програми (често наричани скриптове)
  - .py файлове, към които има обръщение с командата **import** са модули
- Така един и същи файл може да бъде както програма (скрипт), така и модул

Проблеми с подхода файл - програма

- Лесен за използване при малки по обем и сложност програми
- Води до объркване и липса на яснота при големи програми
- Трудно се следят детайлите при големи програми
- Трудно се анализират връзките между отделните части на програмите

Добър стил на програмиране

- Наличието на повече код (команди) не винаги означава нещо добро
- Добрите програмисти се отличават по функционалните възможности на техните програми
- Изисква използването на **функции**
- Включва механизъм за постигане на **декомпозиция** и **абстрахиране (абстракция)**

Пример – прожекторът в клас

- Представлява черна кутия
- Не знаем как точно работи
- Знаем интерфейса: вход/изход и команди
- Можем да включим всякакво устройство което може да използва входа
- Черната кутия по някакъв начин прожектира на стената изображения в мащаб
- **Идея за абстракция**: не трябва да знаем как работи прожектора за да го използваме

Пример – спортни състезания

- Голямо изображение на видео стена се декомпозира на няколко по-малки, всяко излъчвано от отделен прожектор
- Всеки прожектор има свой вход и генерира отделен независим изход
- Всички прожектори заедно показват едно голямо изображение
- **Идея за декомпозиция**: една задача разбиваме на няколко по-малки

Създаване на структура чрез ДЕКОМПОЗИЦИЯ

- Разделяне на командите в **модули**
  - Всеки е **самостоятелен**
  - **По-малък** брой команди във файл
  - Цел за **многократно използване**
  - По-добра **организация** на командите
  - **Ясни съгласувани команди**
- Сега: декомпозиция чрез **функции**
- По-нататък, декомпозиция а **класове**

Скриване на детайли чрез АБСТРАКЦИЯ

- Множество от команди като **черна кутия**
  - Не можем да видим детайлите вътре
  - Не трябва да знаем детайлите вътре
  - Не искаме да знаем детайлите вътре
  - Скрива досадни подробности от програмите
- Постигаме абстракция чрез **дефиниране на функции** или **описание на функционалността им (docstrings)**

Функции

- Многократно изпълнимо множество команди
- Не се изпълняват в програмата докато не бъдат "**извикани**" (има "**обръщение**" към тях)
- Свойства:
  - Имат **име**
  - Имат **аргументи (параметри) -** 0 или повече
  - Имат **docstring** (документация)
  - Имат **тяло**
  - **Връщат** стойност

Общ синтаксис

Кл. дума

Име

Параметри
аргументи

Описание
docstring

```
def is_even (i):
    """
    Вход: Положително цяло число i
    Връща: True ако i е четно, иначе - False
    """
    print("inside is_even”)
    return i % 2 == 0
```

Тяло
Body

```
is_even(3)
```

Обръщение към функция с
име и стойности на параметри

Дефиниране на функции

Използва се командата **def**:

```
>>> def foo(bar):
        ...
        return bar
>>>
```

Това е елементарна функция с име **foo** и с един параметър **bar**

Този параметър се връща като резултат от изчислението.

Функции

Всяка дефиниция създава обект от тип function в текущото пространство на имена

```
>>> foo
<function foo at fac680>
>>>
```

Този обект може да бъде изпълнен (след обръщение към името и задаване на реални аргументи в скобите) и да върне резултат:

```
>>> foo(3)
3
>>>
```

Функции

- **Командата *def* създава функция с дадено име**
- **Командата return връща резултат там, където е направено обръщение към функцията**
- **Аргументите се предават чрез присвояване**
- **Аргументите и връщаната стойност не се декларират**

```
def <име>(arg1, arg2, ..., argN):
    <команди>
    return <израз-стойност>

def times(x,y): return x*y
```

Ако няма команда return

```
def is_even( i ):
    """
    Вход: i, положително цяло число
    Не връща никакъв резултат
    """
    i % 2 == 0
```

Без команда return

Python връща резултат **None, ако няма команда return**

Еквивалентно е на липса на стойност

return и print

- return има смисъл само **вътре** във функцията
- само **един** return се изпълнява във функцията
- командите в дефиницията на функция след изпълнена команда return не се изпълняват
- има стойност свързана с нея, **която се връща там където е обръщението**

- print може да бъде ползвана **във / извън** функцията
- Във функция може да се изпълнят **няколко** команди print
- командите във функцията се изпълняват и след print
- има странично действие: **извежда стойността на параметрите си** на екрана

Предаване на аргументи във функции

- *Аргументите се предават чрез присвояване*
- *Предават стойностите си на локални променливи*
- *Това по никакъв начин не променя стойността на оригиналните неизменяеми обекти и променливи*
- *Промяна в стойност на локална променлива свързана с изменяем реален аргумент може да промени стойността на оригиналния обект или променлива*

```
def changer (x,y):
    x = 2                  # променя само локалната x
    y[0] = 'hi'            # променя споделен mutable y
```

Област на действие на имената

- **Формален (локален) параметър** се свързва с обект - **реален аргумент** при обръщение към функция
- Създава се нова област на имена **(scope)** при изпълнение на функция
- **Тази област (scope)** е съпоставяне на имена с обекти

```
def f( x ):
    x = x + 1
    print('in f(x): x = ', x)
    return x

x = 3
z = f(x)
```

Формален параметър

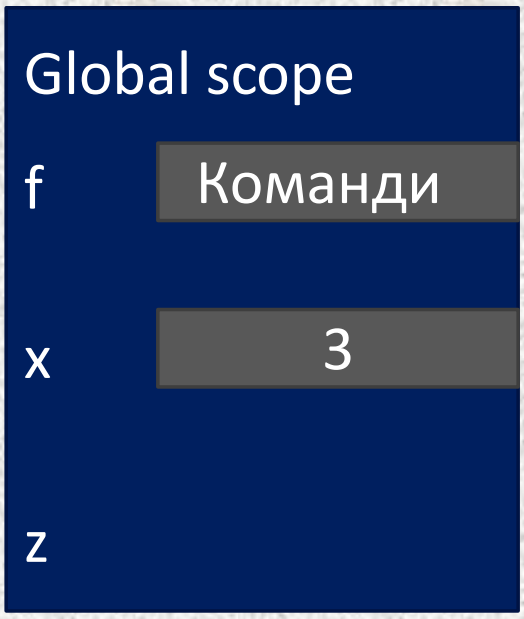
Дефиниция на функция

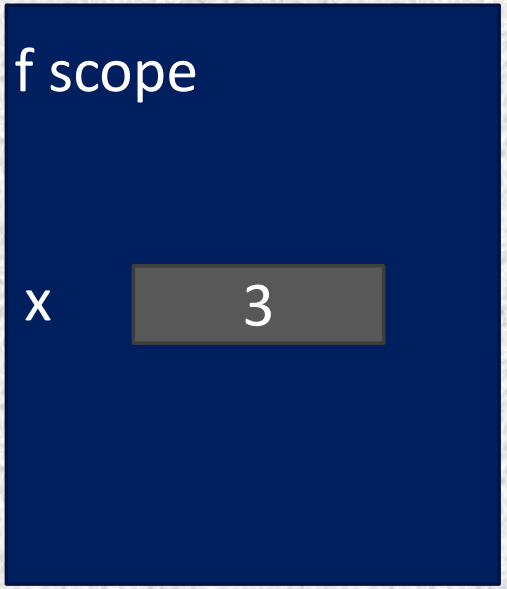
Реален параметър

Основна програма:
(1) инициализира x;
(2) обръщение към f(x);
(3) присвоява рез. на z

Пример

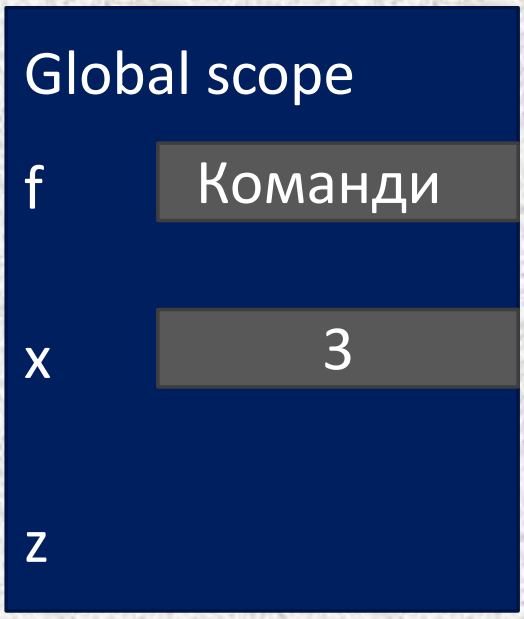
```
def f(x):
    x = x + 1
    print('in f(x): x= ', x)
    return x

x = 3
z = f(x)
```

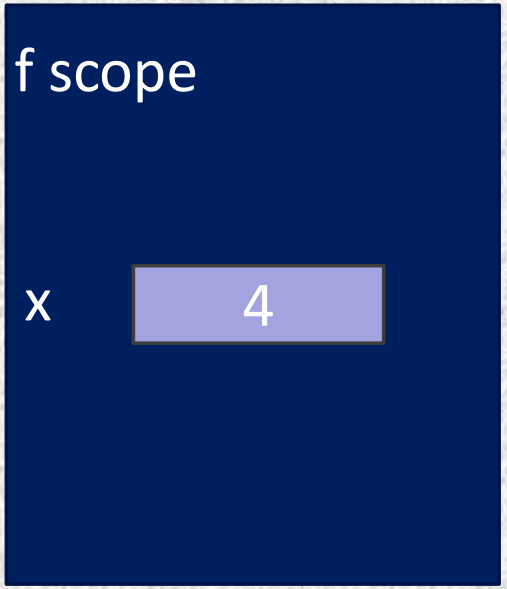




Пример

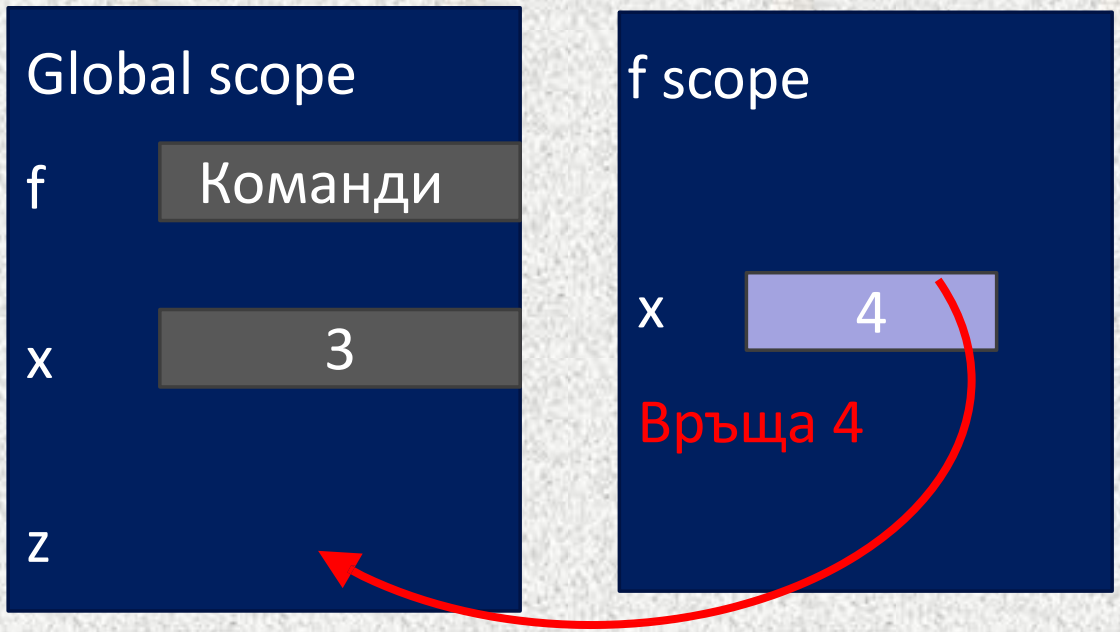
```
def f(x):
    x = x + 1
    print('in f(x): x= ', x)
    return x

x = 3
z = f(x)
```





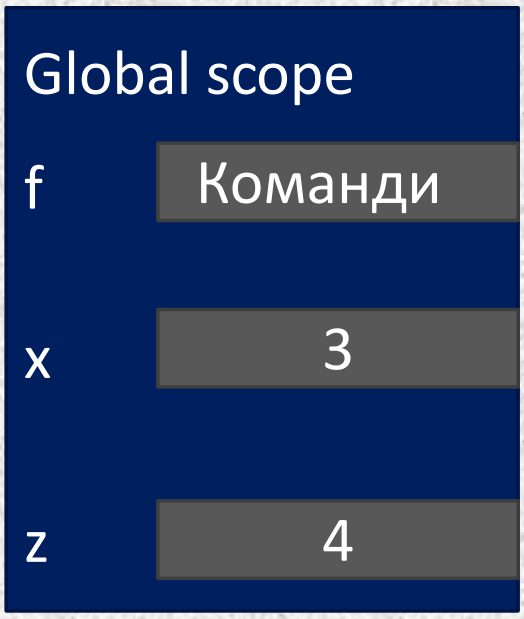
Пример

```
def f(x):
    x = x + 1
    print('in f(x): x= ', x)
    return x

x = 3
z = f(x)
```



Пример

```
def f(x):
    x = x + 1
    print('in f(x): x= ', x)
    return x

x = 3
z = f(x)
```



Глобални и нелокални променливи

- global – с тази ключова дума се декларират променливи, които са извън областта на функцията, и които могат да бъдат използвани и променяни вътре в тялото на функцията
- nonlocal - с тази ключова дума се декларират променливи, които са в йерархичната система от области на имена на функцията, и които могат да бъдат използвани и променяни вътре в тялото на функцията

Глобални и нелокални променливи

```
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 1
# global: 0
```

Глобални и нелокални променливи

```
x = 0
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 2
# global: 0
```

Глобални и нелокални променливи

```
x = 0
def outer():
    x = 1
    def inner():
        global x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 1
# global: 2
```

Видове аргументи

- Позиционни (задължителни) – съпоставят се с формални параметри (както позиционни така и ключови, но могат да бъдат и само позиционни) в зависимост от позицията си
- Ключови – съпоставят се с формален параметър със същото име. Могат и да се изпускат (ако се съпоставят с ключов формален параметър – тогава взимат стойността по подразбиране). Задължително са след позиционните.
- Произволни / контейнери – съпоставят се със съответните формални параметри с * / ** пред името.

Ограничения за аргументите

Указват се в списъка с формални параметри със символите ‘/’ и ‘*’

Символът ‘/’ указва, че параметрите преди него са позиционни (задължителни) и не могат да се указват с ключ, а символа ‘*’ указва, че формалните параметри след него задължително трябва да се съпоставят с ключови реални аргументи.

В примера параметрите a и b са задължително позиционни, c и d могат да са както позиционни така и ключови, а e и f са задължително ключови:

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)
```

Незадължителни аргументи

- Някои възможни аргументи не се подават при обръщение към функция (незадължителни)
- За тях се дефинира стойност по премълчаване
- Те са след задължителните в дефиницията

```
def func(a, b, c=10, d=100):
    print (a, b, c, d)
>>> func(1,2)
1 2 10 100
>>> func(1,2,3,4)
1,2,3,4
```

По-важни свойства и факти

- **Всички функции в Python връщат стойност**
- **При липса на команда return се връща обект *None***
- **Всяка функция трябва да има уникално име – не се допускат функции с еднакви имена**
- **Функциите са обекти и могат:**
  - Да бъдат аргументи към други функции
  - Да връщат като резултат обект - функция
  - Да бъдат присвоявани на променливи
  - Да бъдат елементи в редици, списъци и други

Видове аргументи

```
# Позиционни
def sum(a, b):
    return a + b

# Ключови
def shout(vik="Ураааааааа!!!"):
    print(vik)

# Позиционни и ключови
def echo(niz, prefix=" "):
    print(prefix, niz)
```

Име на функция

- ✓ Всяка функция се определя еднозначно от своето име
- ✓ Броят, редът, имената и видът на аргументите не могат да се използват за разграничаване на две функции с едно и също име
- ✓ Две различни функции не могат да имат едно и също име
- ✓ Единственото изключение е при функция като метод в клас – всеки клас (тип данни) може да си има свои методи (функции), но те имат значение само в този клас (тип данни)

Ключови аргументи и параметри

- ✓ Използват се за задаване на стойност по подразбиране (зададени като формални параметри)
- ✓ Използват се за промяна в реда на задаването на аргументите (в сравнение с реда на имената на формалните параметри)

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

Всички обръщения към функцията връщат резултат 8

Ключови аргументи

- Пример за промяна в реда на аргументите спрямо реда на формалните параметри

```
>>> def foo(x,y,z): return(2*x,4*y,8*z)
>>> foo(2,3,4)
(4, 12, 32)
>>> foo(z=4, y=2, x=3)
(6, 8, 32)
>>> foo(-2, z=-4, y=-3)
(-4, -12, -32)
```

- Комбиниране със стойности по подразбиране

```
>>> def foo(x=1,y=2,z=3): return(2*x,4*y,8*z)
>>> foo()
(2, 8, 24)
>>> foo(z=100)
(2, 8, 800)
```

Произволни аргументи

- Когато формален параметър се съпоставя с произволен брой аргументи, пред името му се слага *. Тогава той се съпоставя с редица от произволен брой подадени аргументи
- Ако се съпоставя с ключови аргументи, тогава пред името на формалния параметър се поставя **. Тогава той се съпоставя с речник.

```
def some_fun(*a, **b):
    print(a)
    print(b)
some_fun(1, 2, 3, name="Digits")
(1, 2, 3)
{'name': 'Digits'}
```

Ред на аргументите

1. Първо се съпоставят позиционните аргументи (с формалните параметри на съответна позиция)
2. След това се съпоставят ключовите аргументи с формални параметри със същото име
3. След това се съпоставят произволни аргументи с формални параметри със * пред името им
4. След това се съпоставят произволни аргументи с формални параметри с ** пред името им
5. Последни се съпоставят незадължителните аргументи не подадени в обръщението (с ключови формални параметри – взимат стойността им по подразбиране)

Пример с редица на Фибоначи

```
def fib(n)
# Връща като списък първите n числа на Фибоначи
    result = []
    a, b = 0, 1
    while a<n:
        result.append(a)
        a, b = b, a+b
    return result
```

Същата функция но като генератор

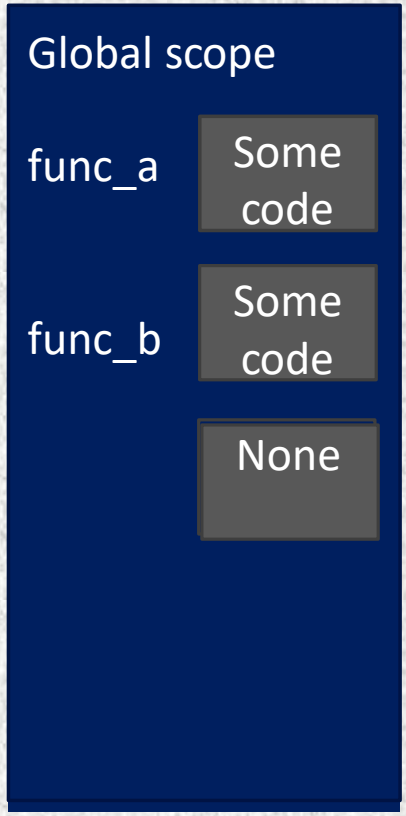
```
def fib()
# Връща при поредно извикване поредното число на
# Фибоначи
    a, b = 0, 1
    while true:
        yield(a)
        a, b = b, a+b
# Всяка нова стойност се инициира с обръщението
# next()
f = fib()
next(f)
…
```

Функции като аргументи

```
def func_a():
    print("От тялото на func_a")
def func_b(x):
    print("От тялото на func_b")
    return x

print(func_a)
print(func_a())
print(5 + func_b(3))
print(func_b(func_a))
print(func_b(func_a()))
```

Функции като аргументи

```
def func_a():
    print("От func_a")
def func_b(x):
    print("От func_b")
    return x
```

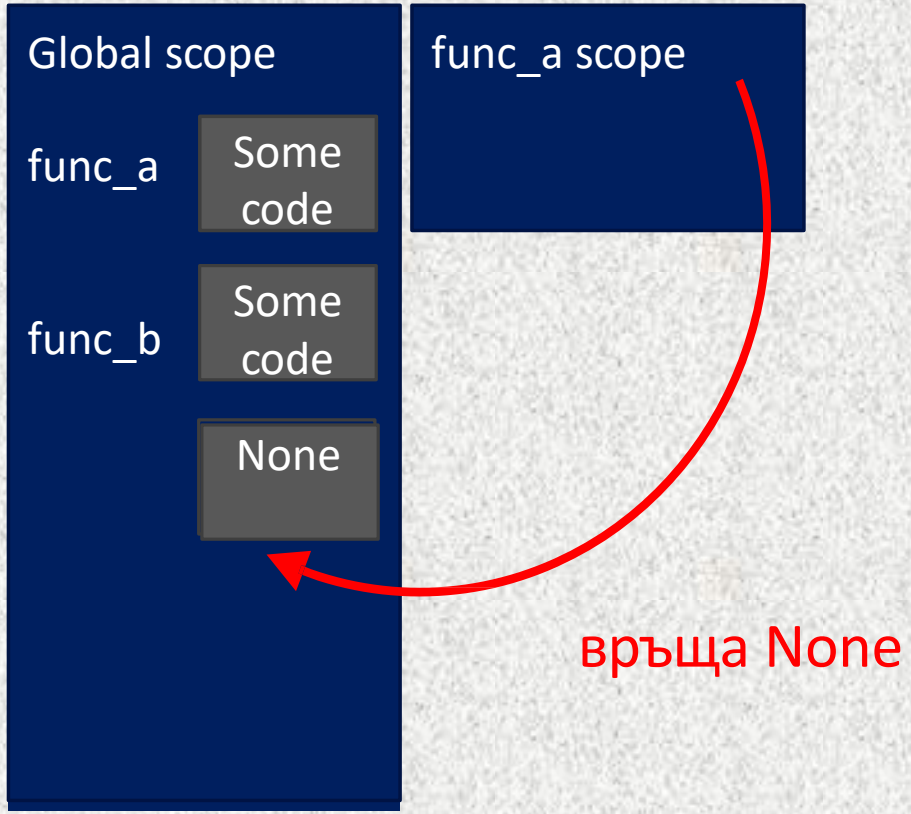



```
print(func_a)
print(func_a())
print(5+func_b(2))
print(func_b(func_a))
```

<function func_a at 0x000001B27FA1A430>

Функции като аргументи

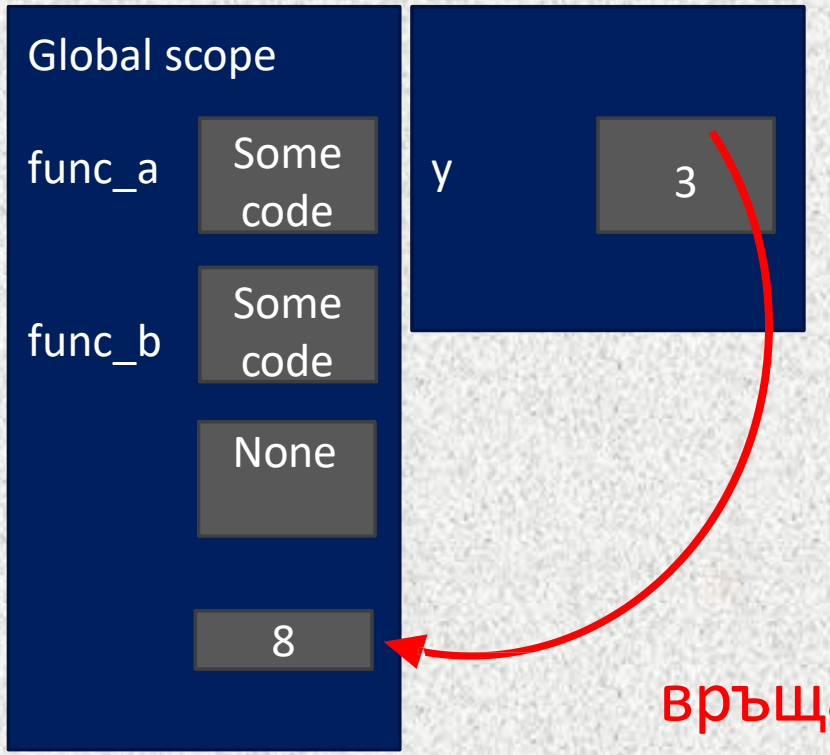
```
def func_a():
    print("От func_a")
def func_b(x):
    print("От func_b")
    return x
```




```
print func_a()
print 5 + func_b(2)
print func_b(func_a)
```

Функции като аргументи

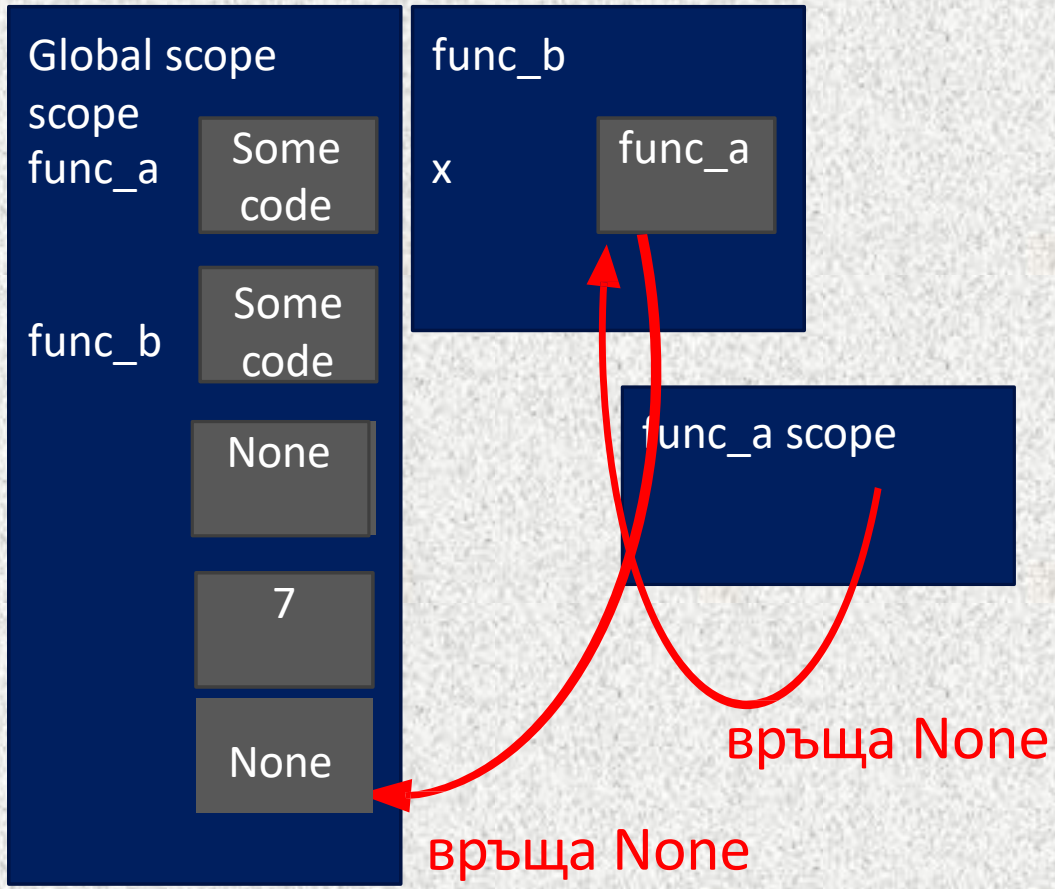
```
def func_a():
    print("От func_a")
def func_b(x):
    print("От func_b")
    return x
```




```
print(func_a())
print(5+func_b(3))
print(func_b(func_a))
```

Функции като аргументи

```
def func_a():
    print("От func_a")
def func_b(x):
    print("От func_b")
    return x
```




```
print(func_a())
print(5+func_b(3))
print(func_b(func_a()))
```

Външни променливи във функции

- Във функция, **има достъп** към външна променлива
- Но **не може да се изменя** външна променлива (може чрез **глобални променливи** – приема се за лош стил)

```
def f(y):
    x = 1
    x += 1

x = 5
f(x)
print(x)
```

x is re-defined in scope of f

different x objects

```
def g(y):
    print(x)
    print(x + 1)

x = 5
g(x)
print(x)
```

x from outside g

x inside g is picked up from scope that called function g

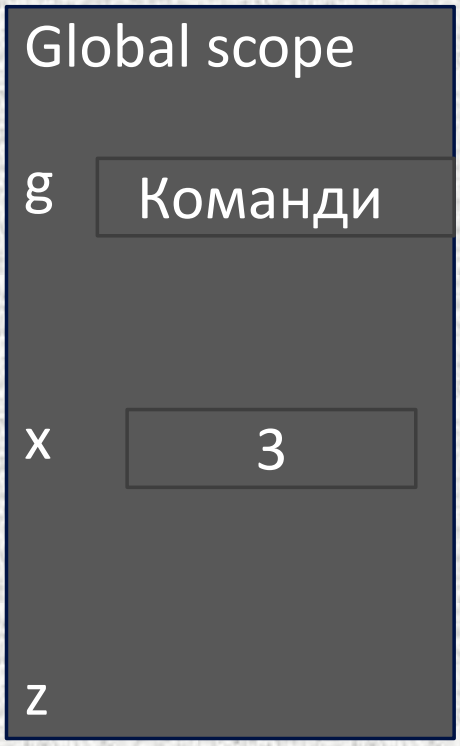
```
def h(y):
    x += 1
print(x)

x = 5
h(x)
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

Области на имена във функции

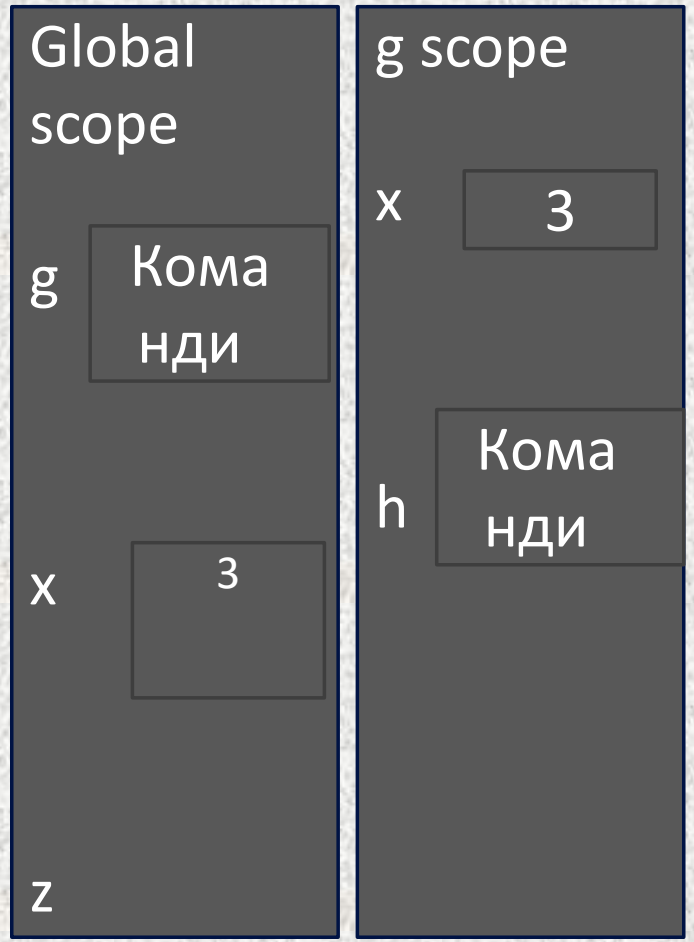
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g:x= ', x)
    h()
    return x

x = 3
z = g(x)
```

Команди



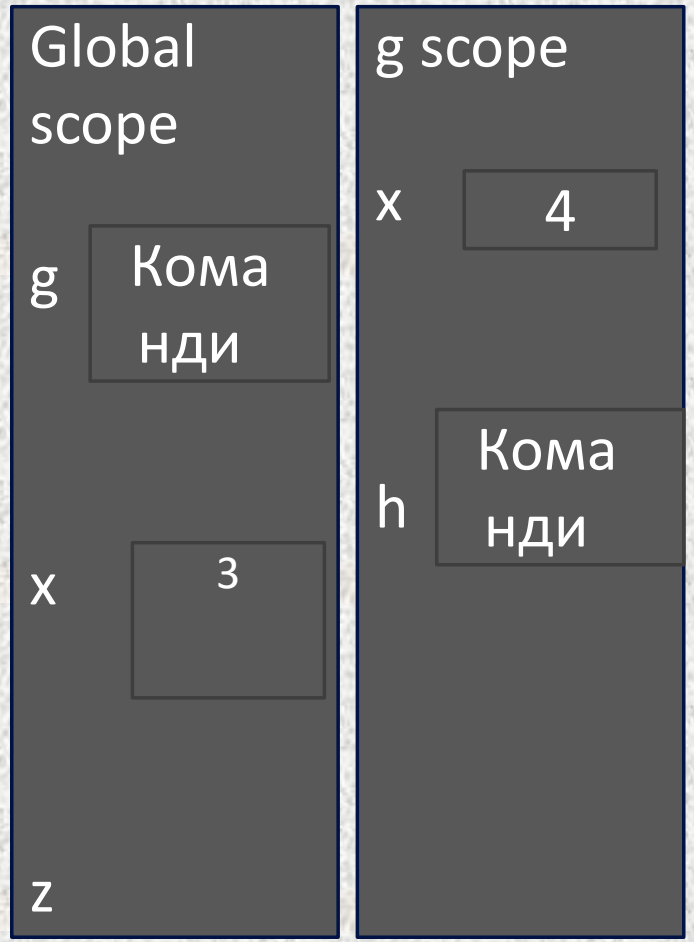
Пример за области на имена във функции

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x= ', x)
    h()
    return x

x = 3
z = g(x)
```



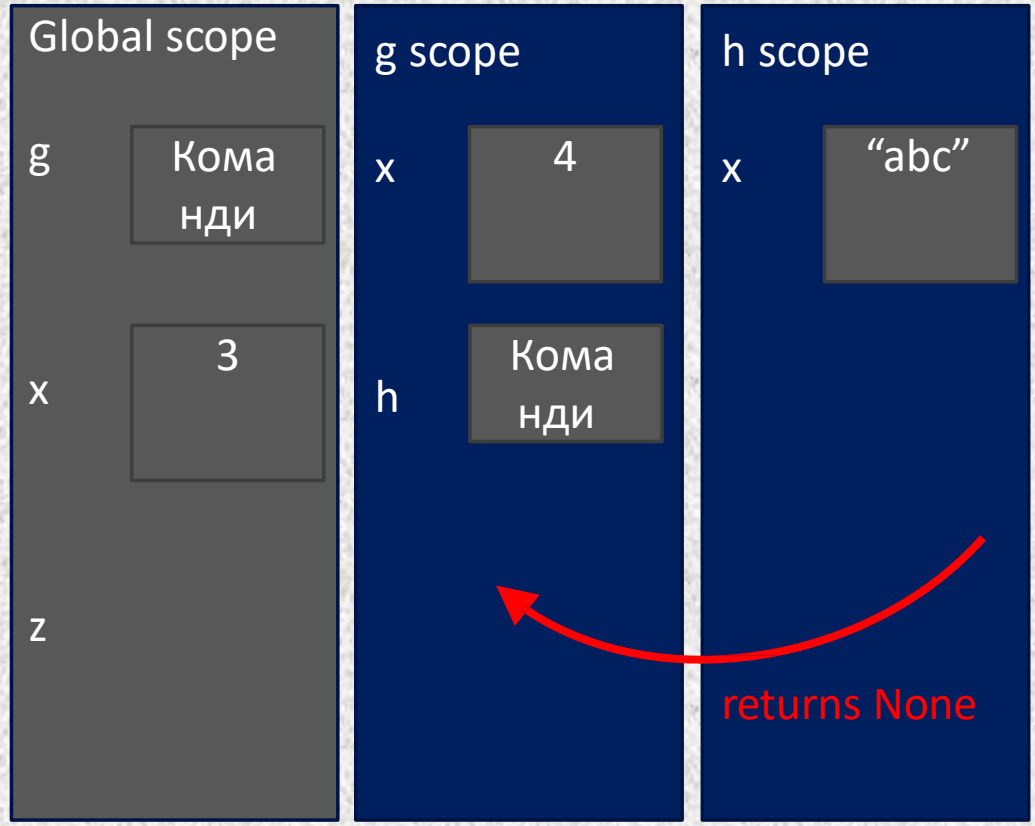
Пример за области на имена във функции

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



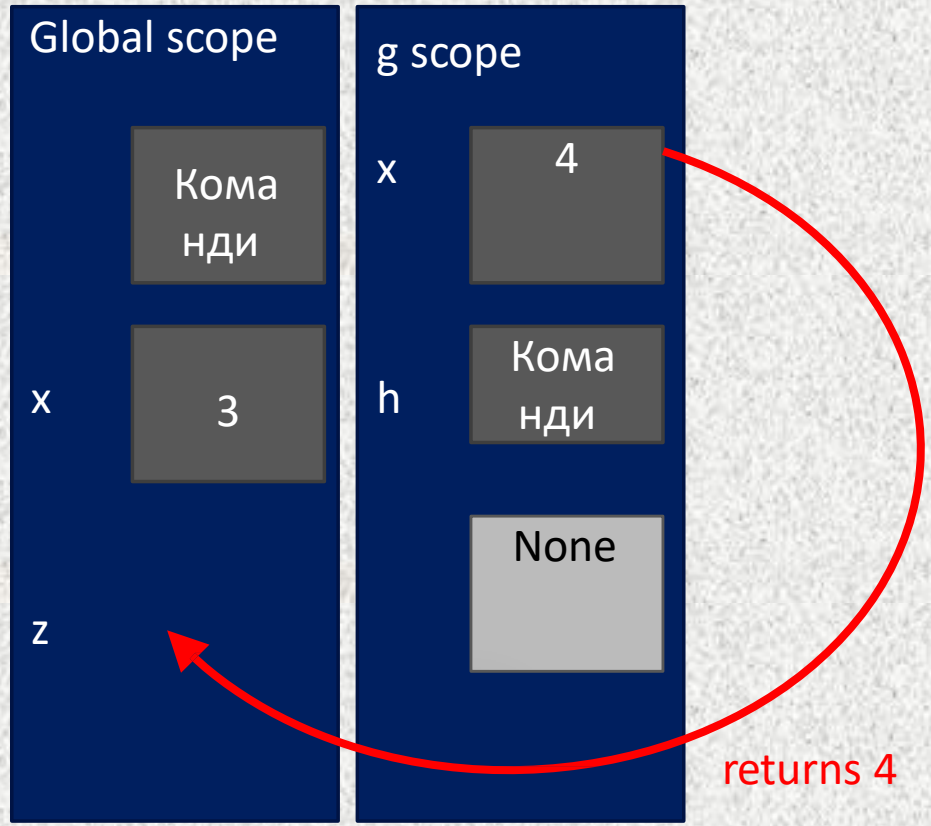
Пример за области на имена във функции

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



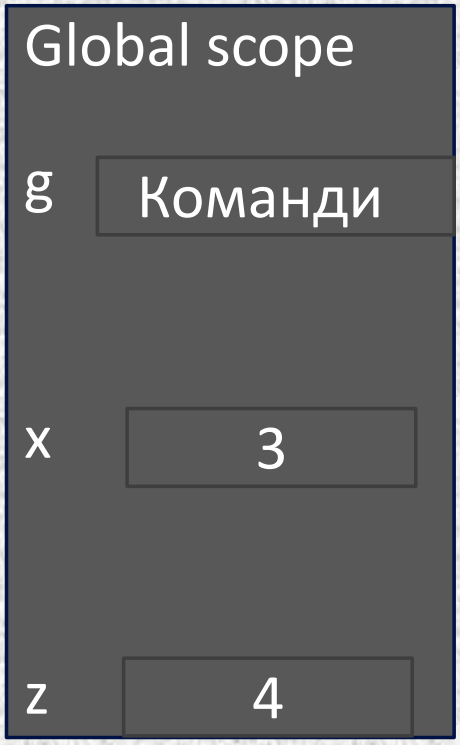
Пример за области на имена във функции

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Пример за области на имена във функции

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Предаване на стойности

- ✓ Формалните параметри в Python получават стойност от реално подадените аргументи
- ✓ Формалните параметри нямат достъп до обектите, чиято стойност получават
- ✓ Този начин в информатиката се нарича: pass parameters by value
- ✓ В някои езици за програмиране (C++, Ada и други) се предават самите обекти с техния адрес (pass parameters by reference)
- ✓ В Python този механизъм може да се симулира чрез подаване на изменяеми обекти като аргументи

Функциите като обекти

- ✓ Функциите в Python представляват обекти
- ✓ За всяка дефиниция на функция интерпретаторът създава нов обект от тип функция
- ✓ Всяка променлива можем да свържем с обект от тип функция (променливата има стойност функция)
- ✓ В дефиницията на функция можем да създадем друга дефиниция на функция (вложена функция)
- ✓ Можем да подадем обект от тип функция като аргумент при обръщение към друга функция
- ✓ Можем да върнем като резултат от изпълнение на функция обект от тип функция
- ✓ Обект от тип функция може да е елемент от контейнер

Пример: closure

```
>>> def counter(start=0, step=1):
        x = [start]
        def _inc():
            x[0] += step
            return x[0]
        return _inc
>>> c1 = counter()
>>> c2 = counter(100, -10)
>>> c1()
1
>>> c2()
90
```

Области на действие на имената

- На ниво система глобалното пространство е с име __main__
- Всяко име се търси за разрешаване на 4 нива
  - Локално
  - Обхващаща функция (ако има)
  - Глобално (напр. дефинирани като global)
  - Системно (вградените в системата)
- Всяко име от обхващащо ниво е достъпно, но не може да се променя

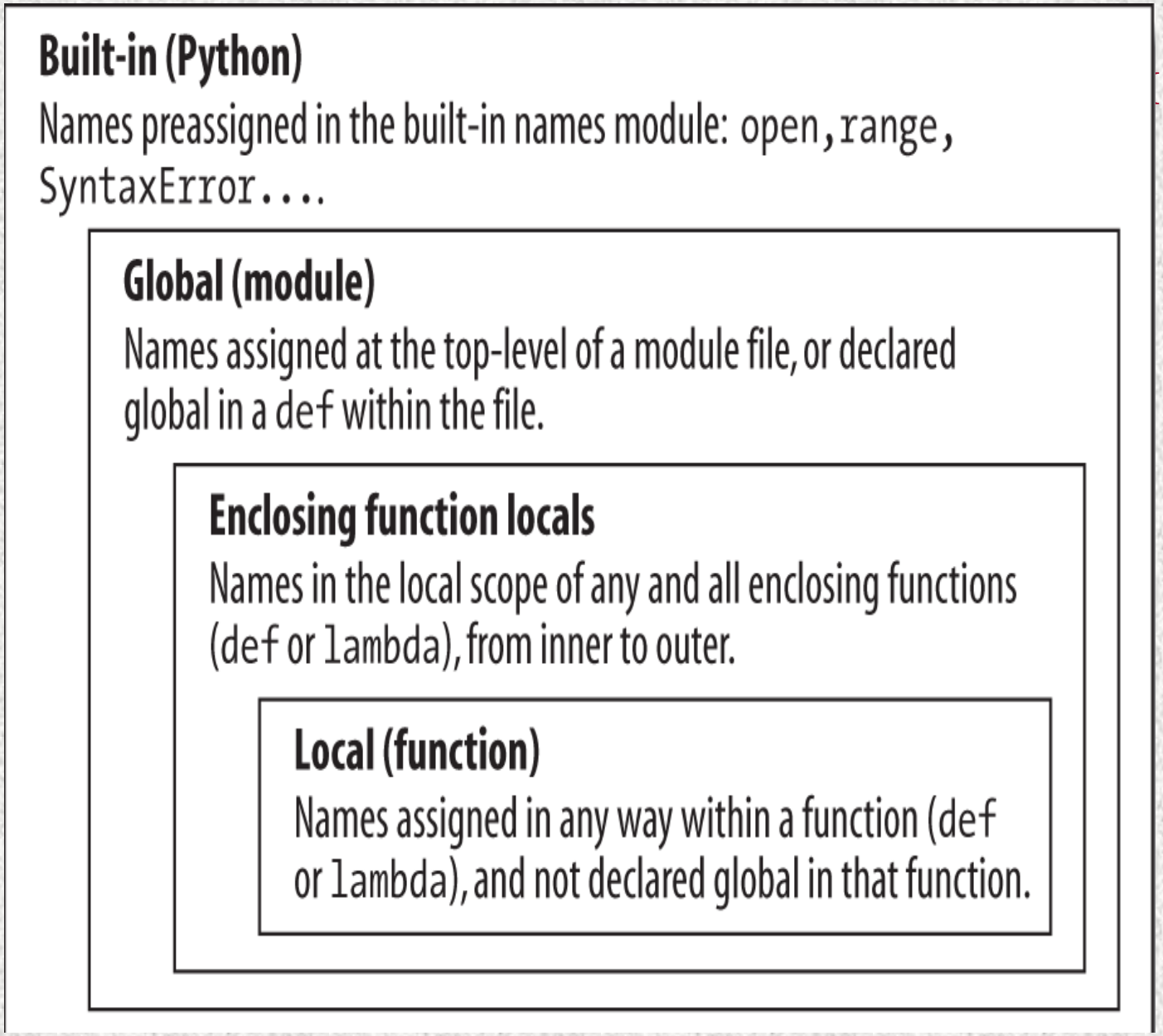
Области на имена

- ✓ scope = област от програма на Python където областта на имената е директно достъпна (без “.”)
  - ➢ вътрешна (първо) = локални имена
  - ➢ средна = глобалните имена в текущия модул
  - ➢ външна (последно) = вградени имена
- ✓ присвояванията винаги се отнасят към вътрешната област (inner scope)
  - ➢ Не се копират обекти, само се създава връзка от име към обект
- ✓ Глобалните имена на обекти са в глобалната област

Особености с имена на променливи

- Дефинираните с global могат да се променят от вътрешни области

- Дефинираните с nonlocal могат да се достъпват и променят само в текущата йерархия от области


Built-in (Python)
Names preassigned in the built-in names module: open, range, SyntaxError, ...
Global (module)
Names assigned at the top-level of a module file, or declared global in a def within the file.
Enclosing function locals
Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.
Local (function)
Names assigned in any way within a function (def or lambda), and not declared global in that function.

Декомпозиция & Абстракция

- мощни концепции когато са приложени едновременно

- командите се използват многократно но се проверяват само един път!

Полиморфизъм

- Всеки оператор е полиморфичен – действието му зависи от типа на аргументите (print, *, +, indexing, …)
- Полиморфизмът е директно следствие от динамичния характер на типовете данни
- За сметка на малък брой команди за проверка на тип ние печелим с намаляване на общия брой команди, голяма гъвкавост и лесно разширяване на програмите

Пример

```
""" Функция която по два зададени контейнера, намира сечението от елементи общи и за двата контейнера """

def intersect(seq1, seq2):
    res = [] # В началото е празен
    for x in seq1: # Претърсваме seq1
        if x in seq2: # Общ елемент?
        res.append(x) # Добавяме го в края
return res
```

Заключение

- ✓ Структуриране и декомпозиране
- ✓ Скриване на подробности
- ✓ Използване на функции
- ✓ Елементи от дефиницията на една функция
- ✓ Предаване на аргументи и връщане на резултат
- ✓ Области на действие на имената
- ✓ Използване на функция като аргумент
- ✓ Особености по подаване на аргументи