



# Introduction to Programming

## Lesson 8

# Outline

- encapsulation
- namespaces
- exceptional control flow
- default arguments
- \*args, \*\*kwargs
- Strings revisited
- Tuples and zip function

# Lecture Links

[goo.gl/hd4qQf](https://goo.gl/hd4qQf)

# Why functions

Կոդի փաթաթումը function-ների մեջ ունի մի քանի  
կարևոր պատճառ

# Why functions

Կոդի փաթաթումը function-ների մեջ ունի մի քանի  
կարևոր պատճառ

- modularity:
  - complex program -> break into smaller ones
  - each tested/debugged separately

# Why functions

Կոդի փաթաթումը function-ների մեջ ունի մի քանի կարևոր պատճառ

- modularity:
  - complex program -> break into smaller ones
  - each tested/debugged separately
- code reuse
  - fragment of code used multiple times

# Why functions

Կոդի փաթաթումը function-ների մեջ ունի մի քանի կարևոր պատճառ

- modularity:
  - complex program -> break into smaller ones
  - each tested/debugged separately
- code reuse
  - fragment of code used multiple times
- encapsulation (of functions)
  - ֆունկցիան թաքցնում է իր իրականացման մանրամասները օգտագործողից
  - `sum(range(10))`, how sum and range work? don't care!

# Encapsulation & local variables

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in
<module>
    y
NameError: name 'y' is not defined
>>>
```

սիմուլյացիա function **double()** կանչելը  
variables **x** and **y** գոյություն չունեն

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```



# Encapsulation & local variables

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in
<module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>>
```

սիմուլյու function **double()** կանչելը  
variables **x** and **y** գոյություն չունեն

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

# Encapsulation & local variables

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in
<module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in
<module>
    y
NameError: name 'y' is not defined
```

մինչև function **double()** կանչելը  
variables **x** and **y** գոյություն չունեն

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

function **double()** կանչելուց հետո  
variables **x** and **y** գոյություն չունեն

# Encapsulation & local variables

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in
<module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in
<module>
    y
NameError: name 'y' is not defined
```

մինչև function **double()** կանչելը  
variables **x** and **y** գոյություն չունեն

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

function **double()** կանչելուց հետո  
variables **x** and **y** գոյություն չունեն

**x** and **y** գոյություն ունեն միայն  
**double(5)** կանչելու ժամանակ

# Encapsulation & local variables

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in
<module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in
<module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in
<module>
    y
NameError: name 'y' is not defined
```

մինչև function **double()** կանչելը  
variables **x** and **y** գոյություն չունեն

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

function **double()** կանչելուց հետո  
variables **x** and **y** գոյություն չունեն

**x** and **y** գոյություն ունեն միայն  
**double(5)** կանչելու ժամանակ

**x** and **y** –ը կոչվում են **double**  
function-ի **local variables**  
(լոկալ փոփոխականներ)

# Function call namespace

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

# Function call namespace

Ինչու values of x and y իրար չեն խանգարում

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

# Function call namespace

Ինչու values of x and y իրար չեն խանգարում

```
>>> x
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    y
NameError: name 'y' is not defined
>>> res = double(5)
x = 2, y = 5
>>> x
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    x
NameError: name 'x' is not defined
>>> y
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

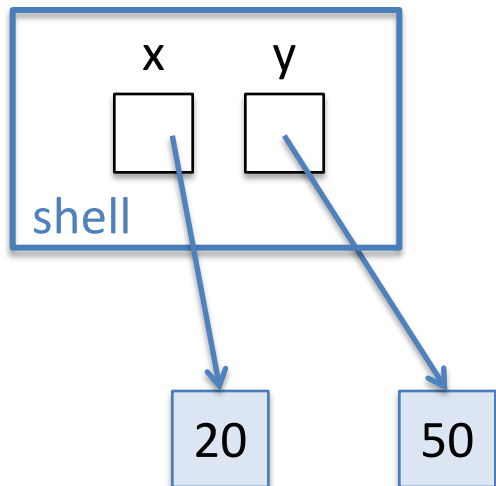
```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

double()-ի կատարման  
ժամանակ local variables x and y  
անտեսանելի են function-ից  
դուրս

# Function call namespace

```
>>> x, y = 20, 50  
>>>
```

```
def double(y):  
    x=2  
    print('x = {}, y = {}'.  
          .format(x,y))  
    return x*y
```

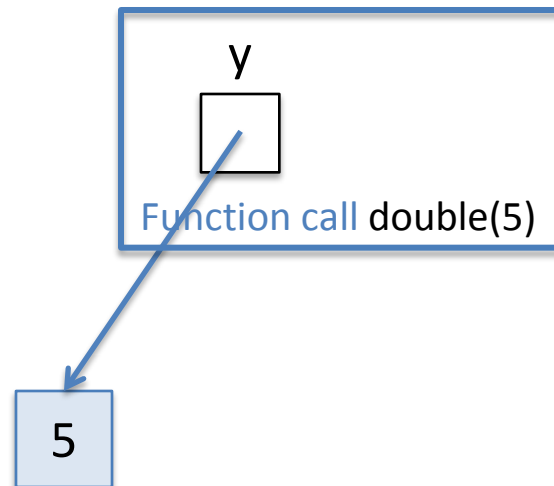
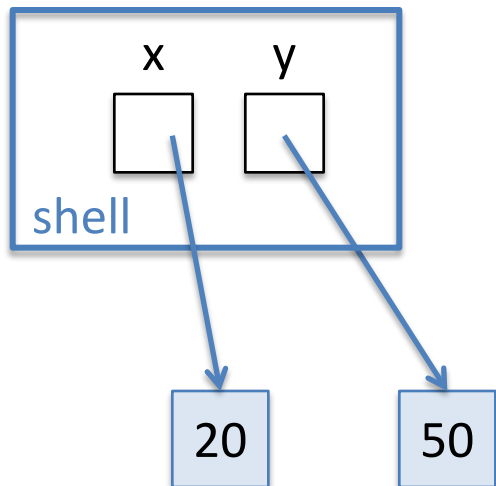




# Function call namespace

```
>>> x, y = 20, 50  
>>> res = double(5)
```

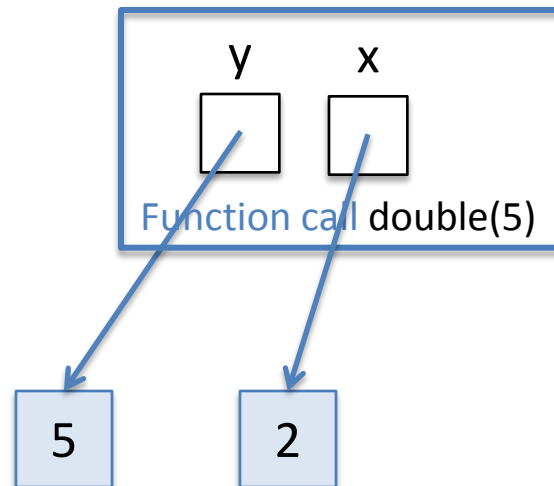
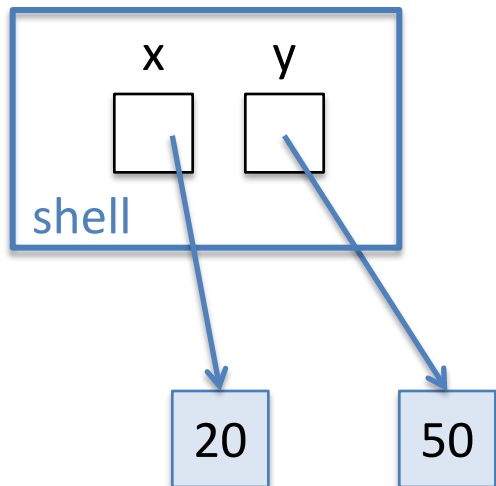
```
def double(y):  
    x=2  
    print('x = {}, y = {}'.  
          .format(x,y))  
    return x*y
```



# Function call namespace

```
>>> x, y = 20, 50  
>>> res = double(5)
```

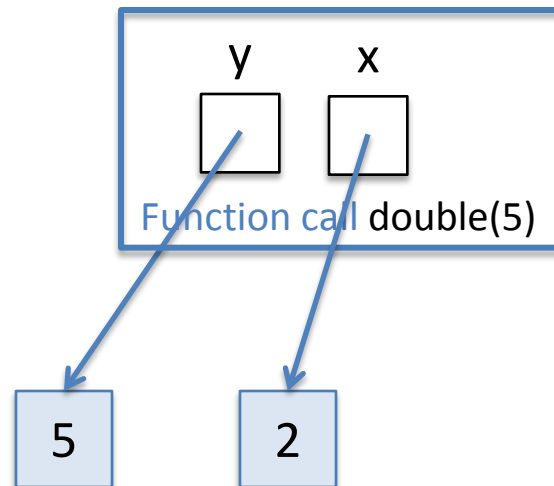
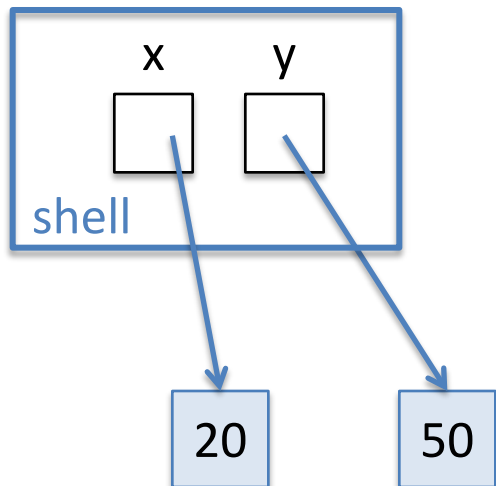
```
def double(y):  
    x=2  
    print('x = {}, y = {}'.  
          .format(x,y))  
    return x*y
```



# Function call namespace

```
>>> x, y = 20, 50
>>> res = double(5)
x = 2, y = 5
>>> x, y
(20, 50)
>>>
```

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

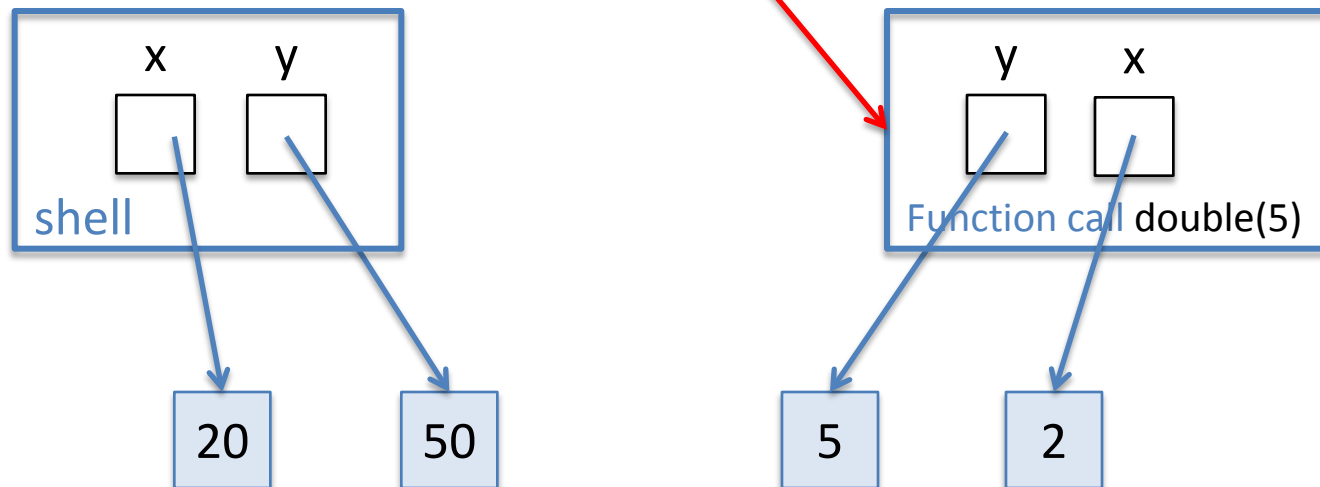


# Function call namespace

```
>>> x, y = 20, 50
>>> res = double(5)
x = 2, y = 5
>>> x, y
(20, 50)
```

```
def double(y):
    x=2
    print('x = {}, y = {}'.format(x,y))
    return x*y
```

Ամեն ֆունկցիայի կանչ ստեղծում է իր անվանատարածքը/**namespace-ը** որում պահվում են **local** փոփոխականները



# Function call namespace

```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)  
  
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)  
  
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

# Function call namespace

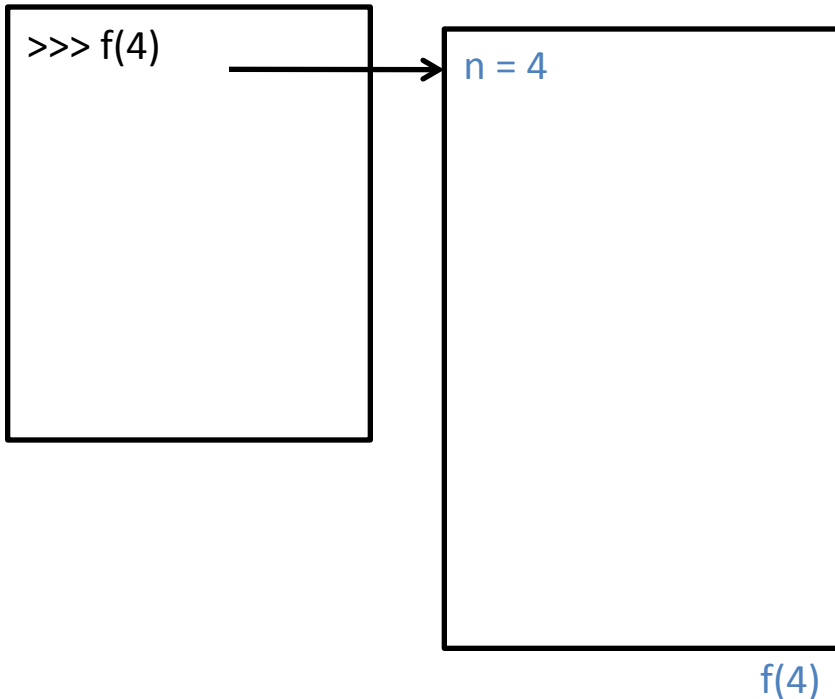
```
>>> f(4)
```

```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

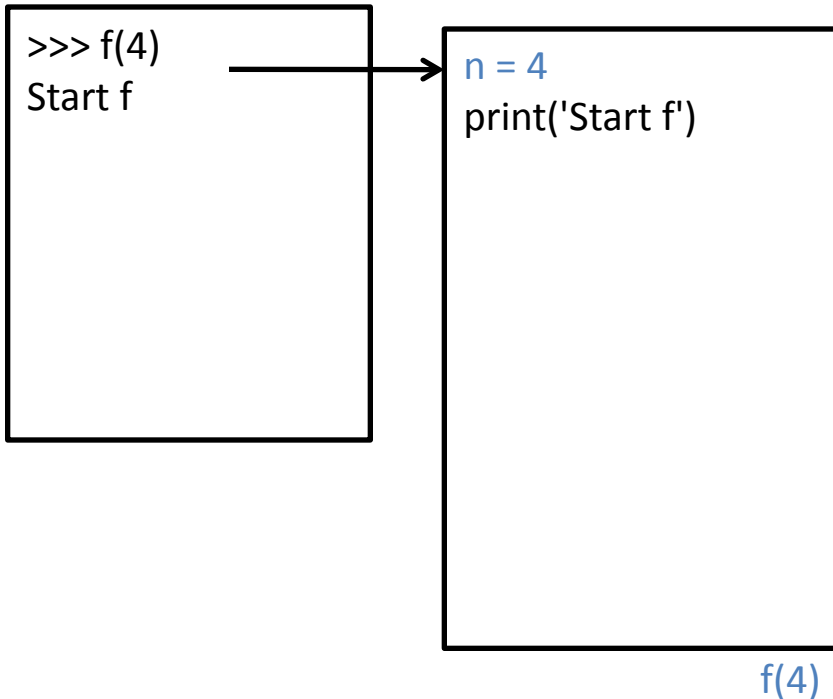
```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

# Function call namespace



```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)  
  
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)  
  
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

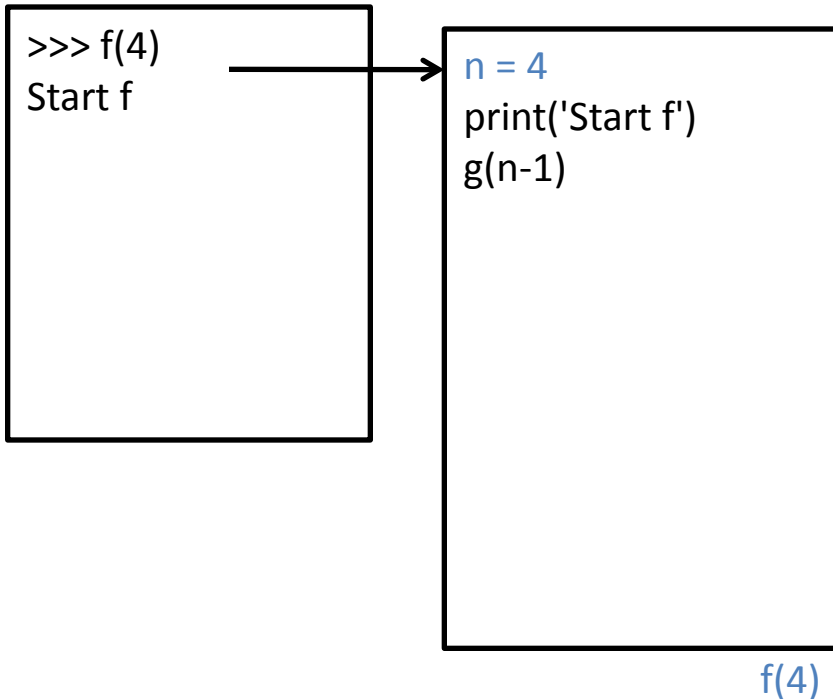
# Function call namespace



```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)  
  
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)  
  
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```



# Function call namespace

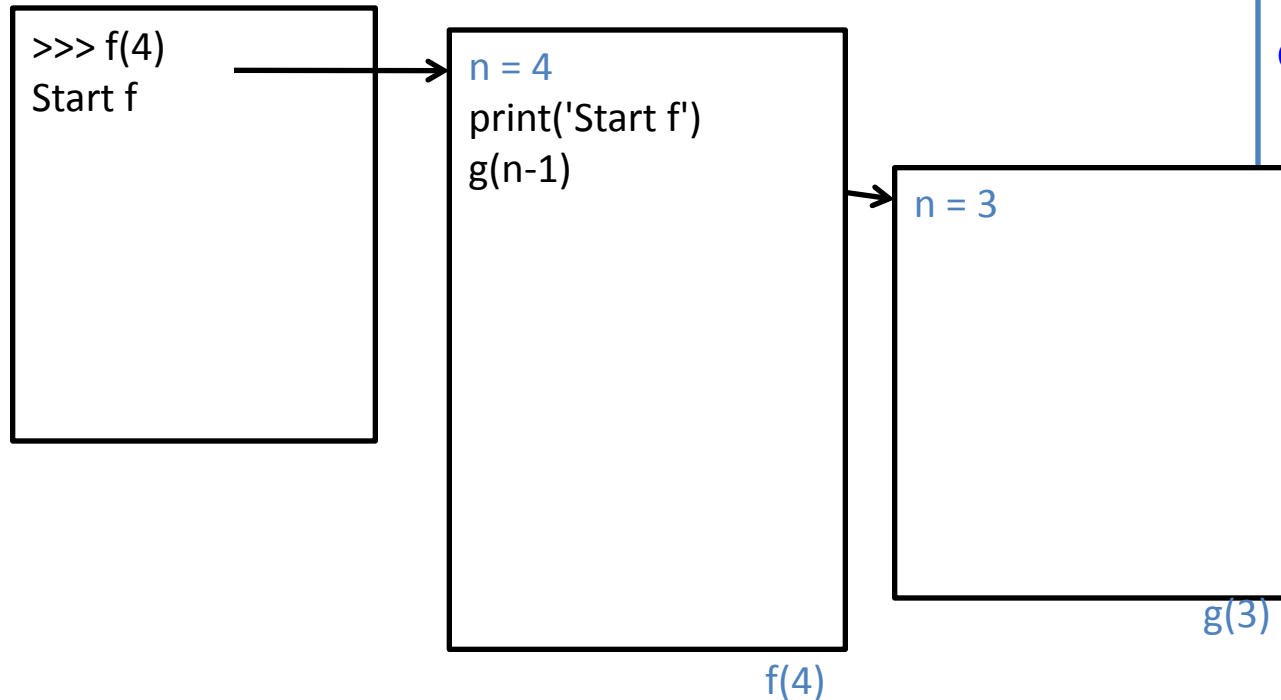


```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

# Function call namespace

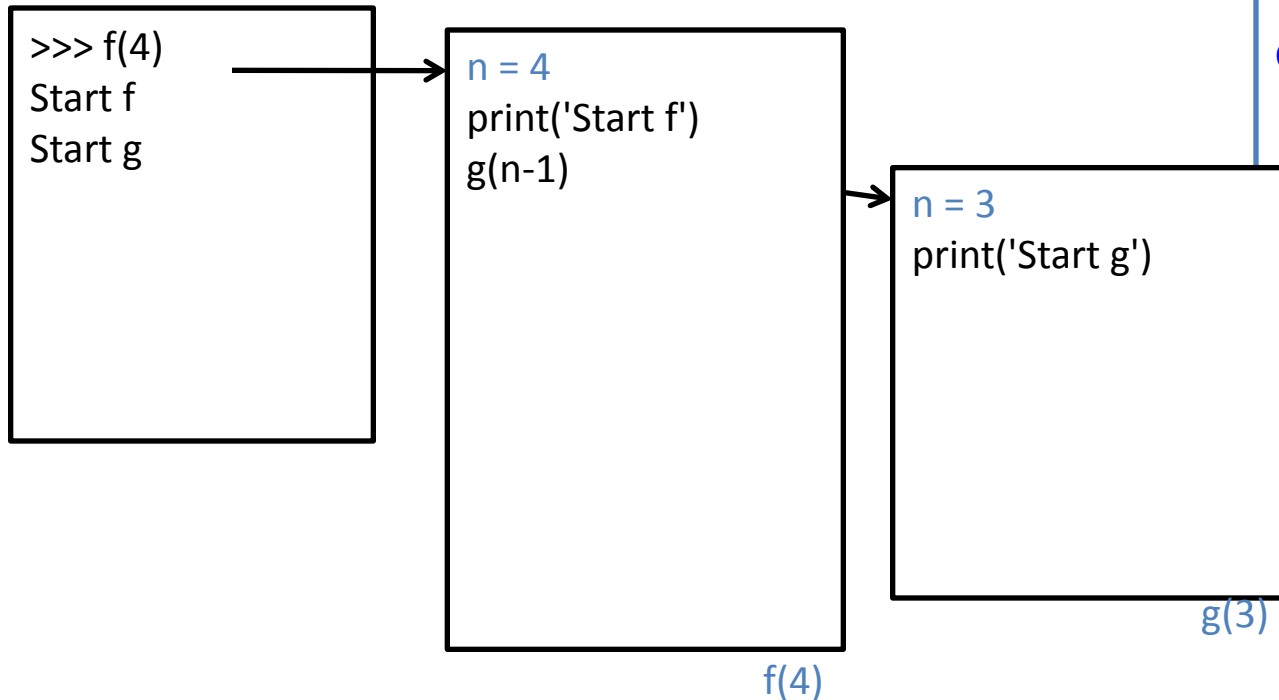


```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

# Function call namespace

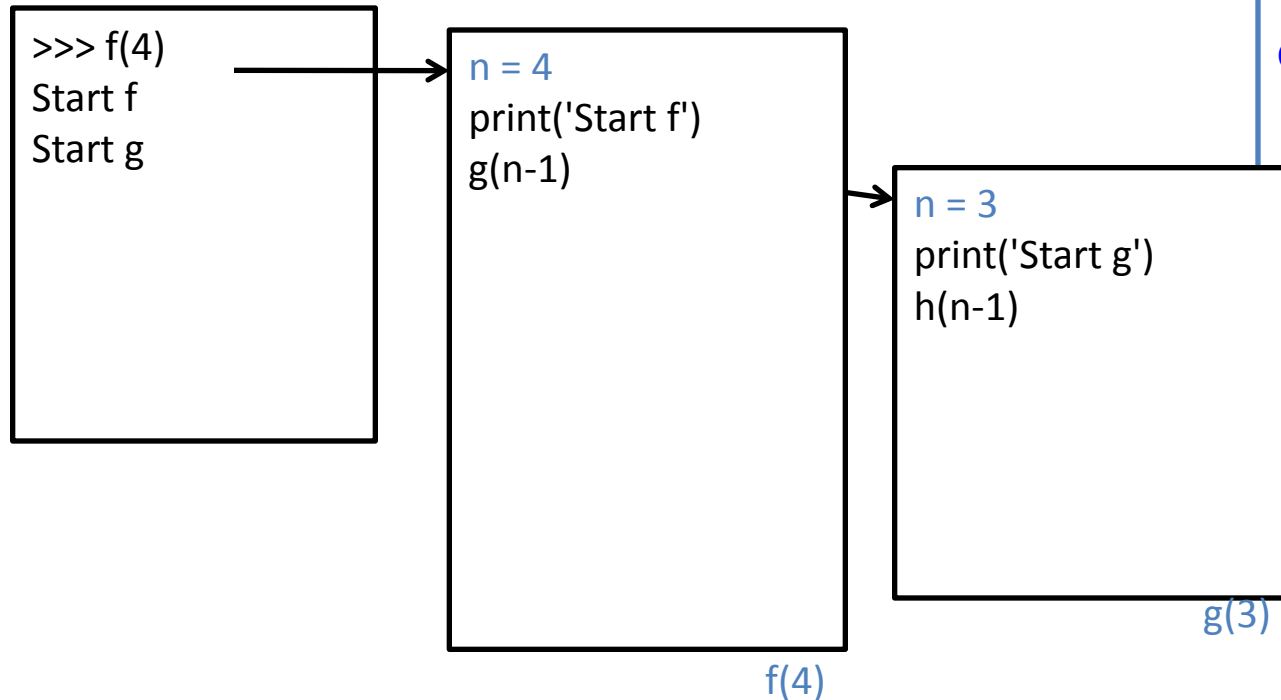


```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

# Function call namespace

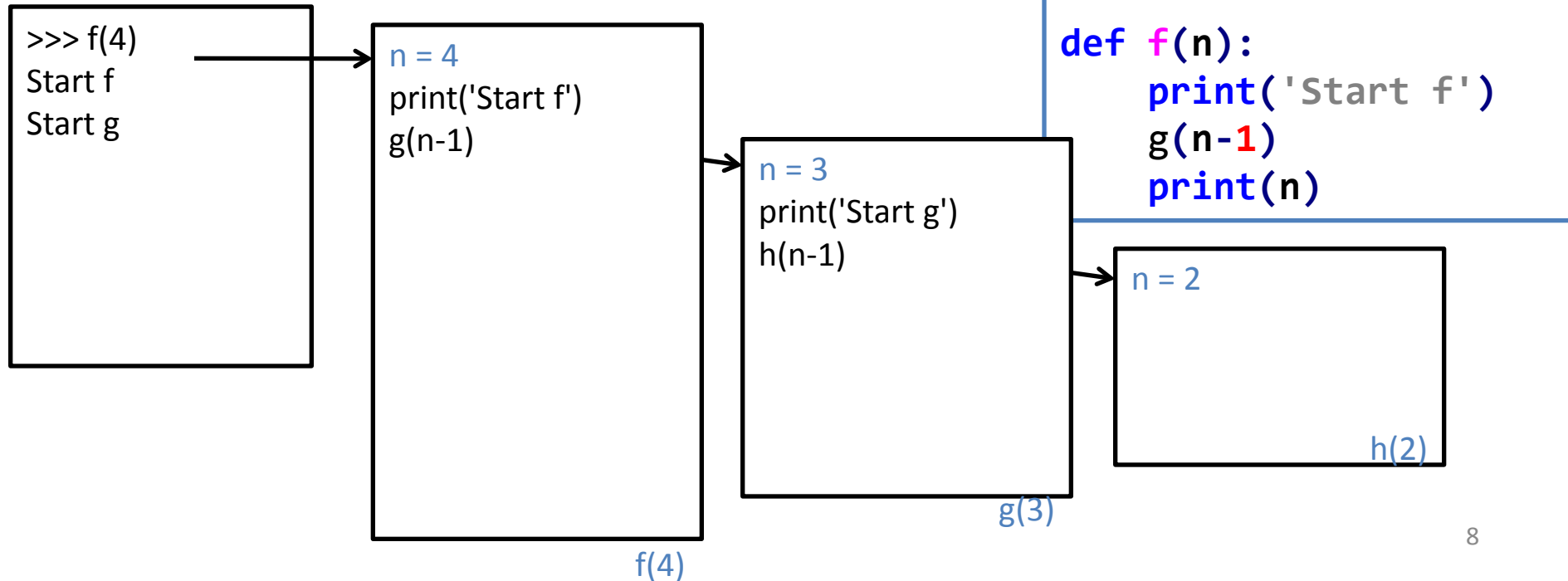


```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

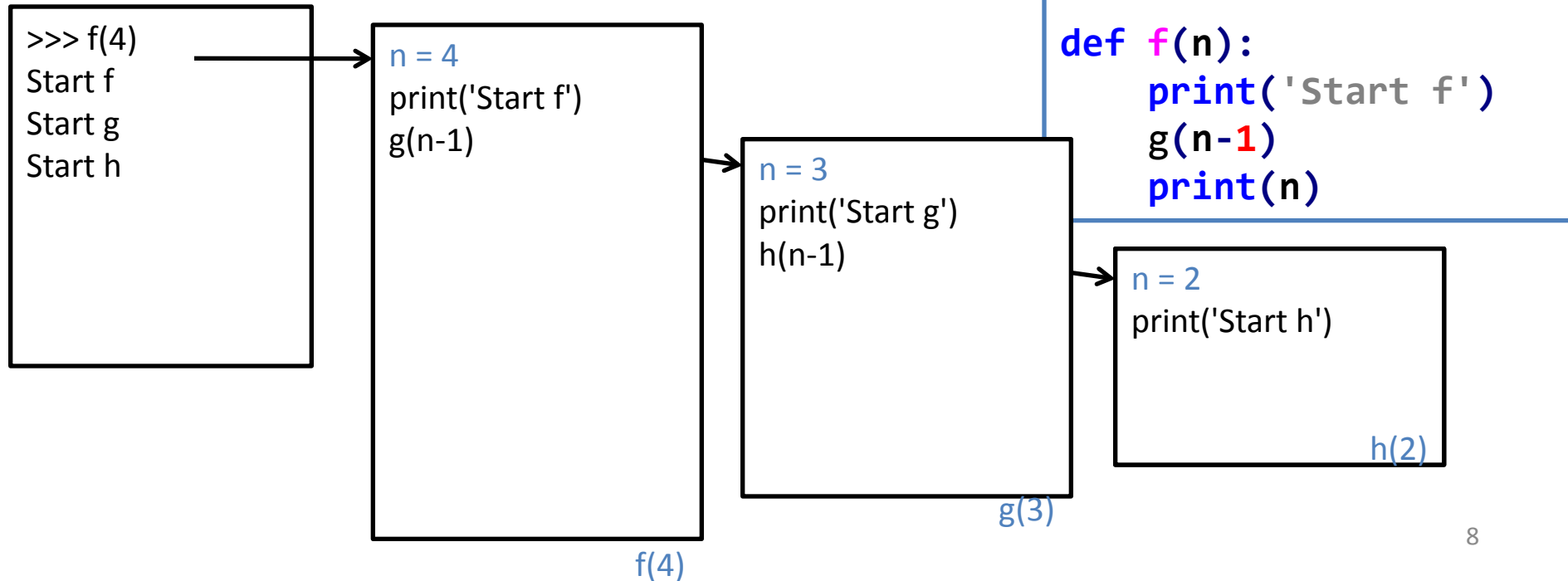
```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

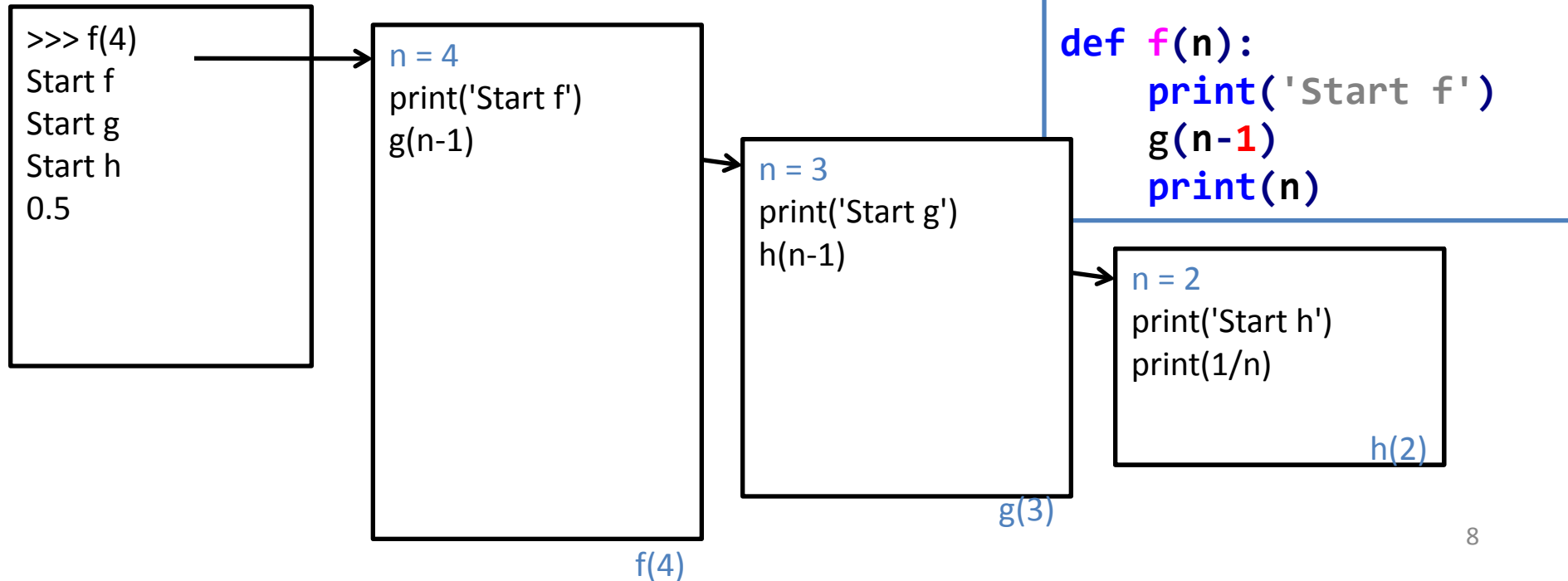
# Function call namespace



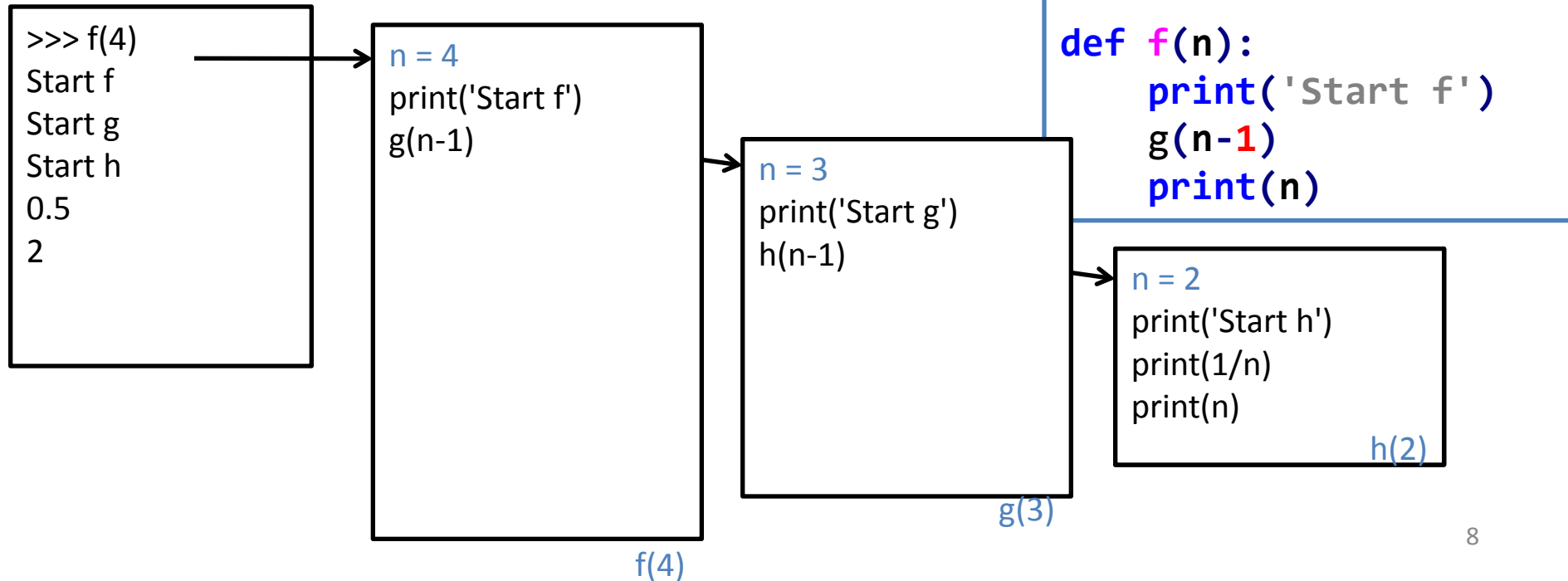
# Function call namespace



# Function call namespace

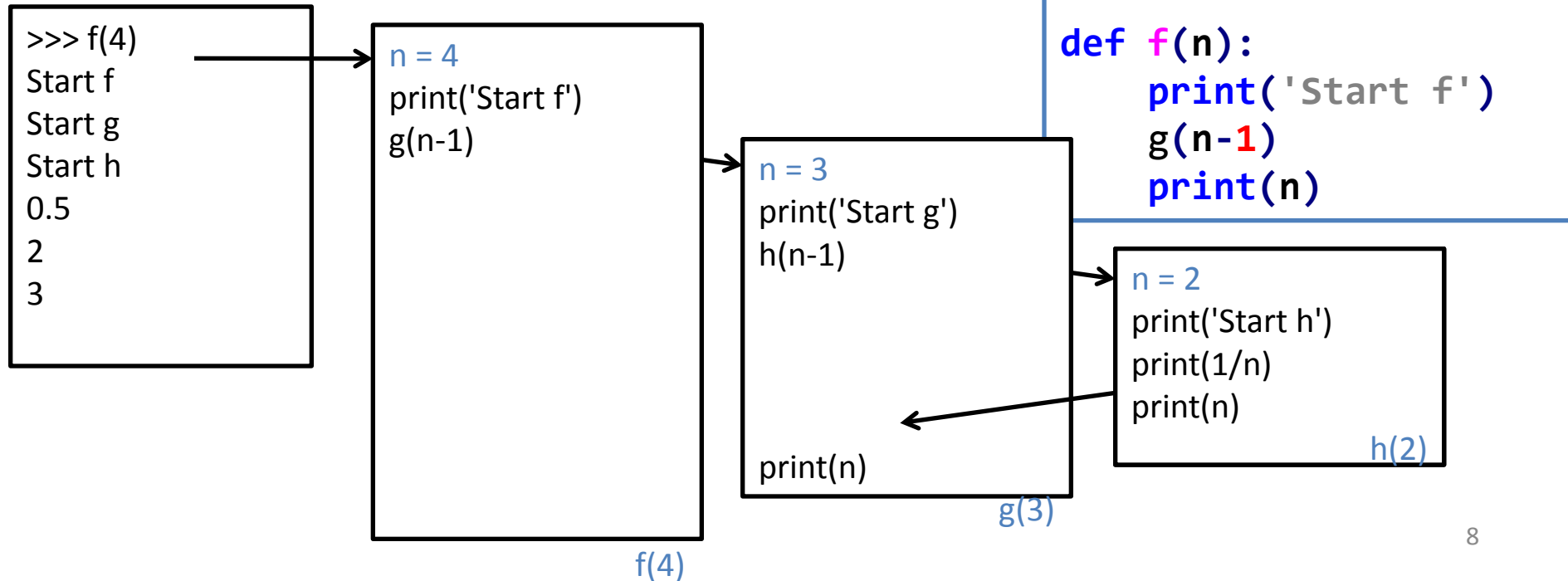


# Function call namespace





# Function call namespace

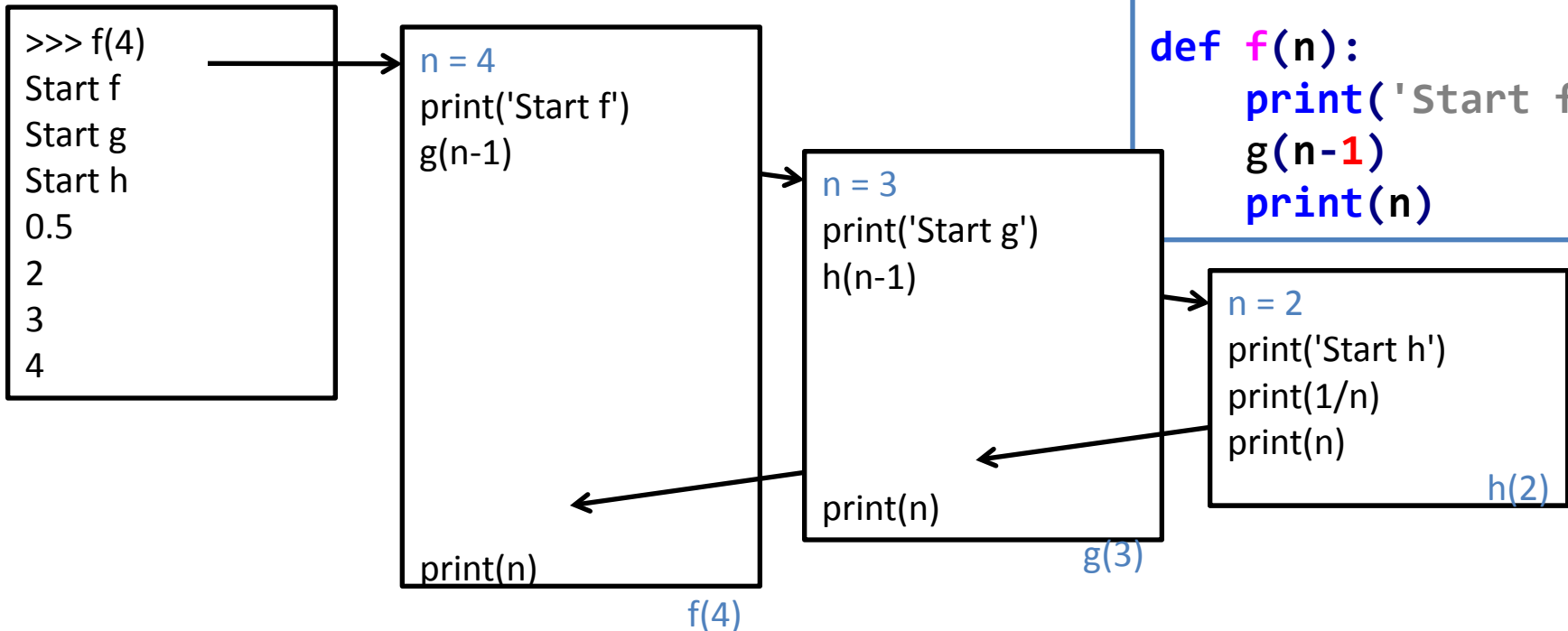


# Function call namespace

```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```



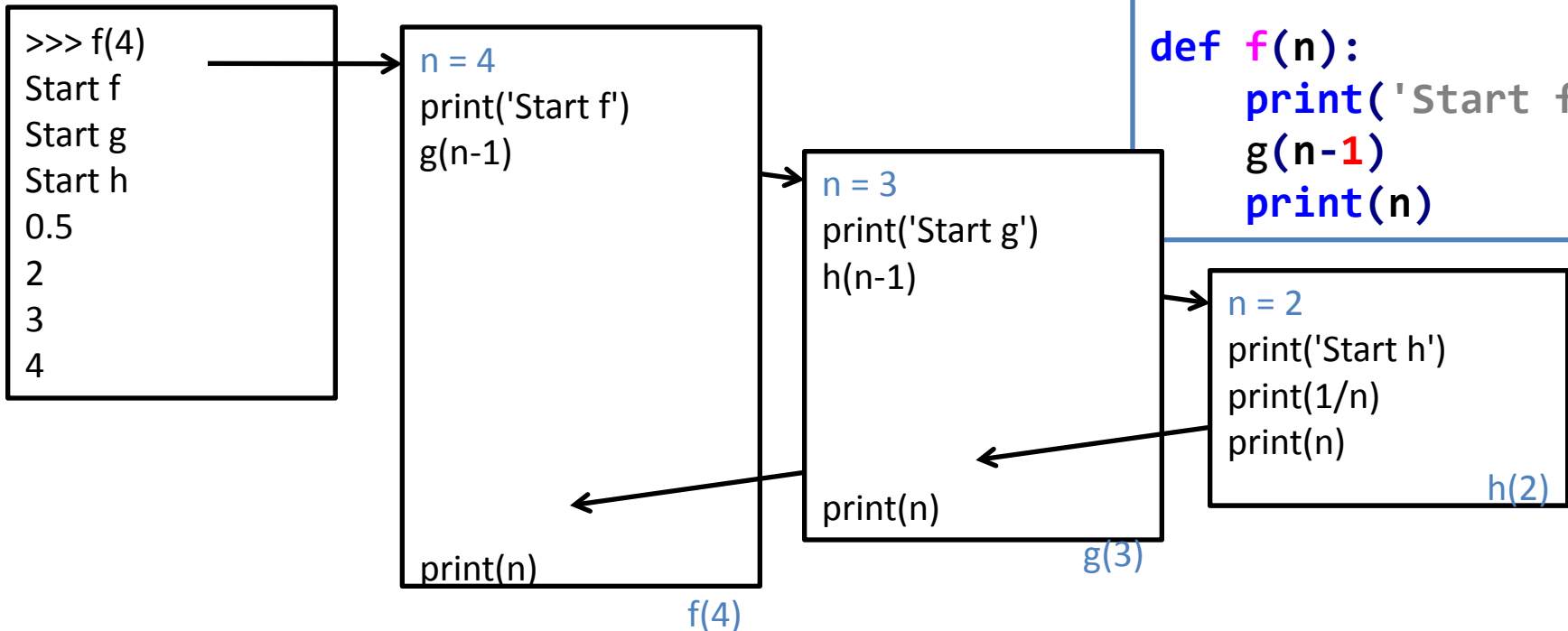
# Function call namespace

կան **n**-ի մի քանի ակտիվ արժեքներ –  
ամեն **namespace**-ում մի հատ.

```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```



# Function call namespace

կան **n**-ի մի քանի ակտիվ արժեքներ –  
ամեն **namespace**-ում մի հատ.

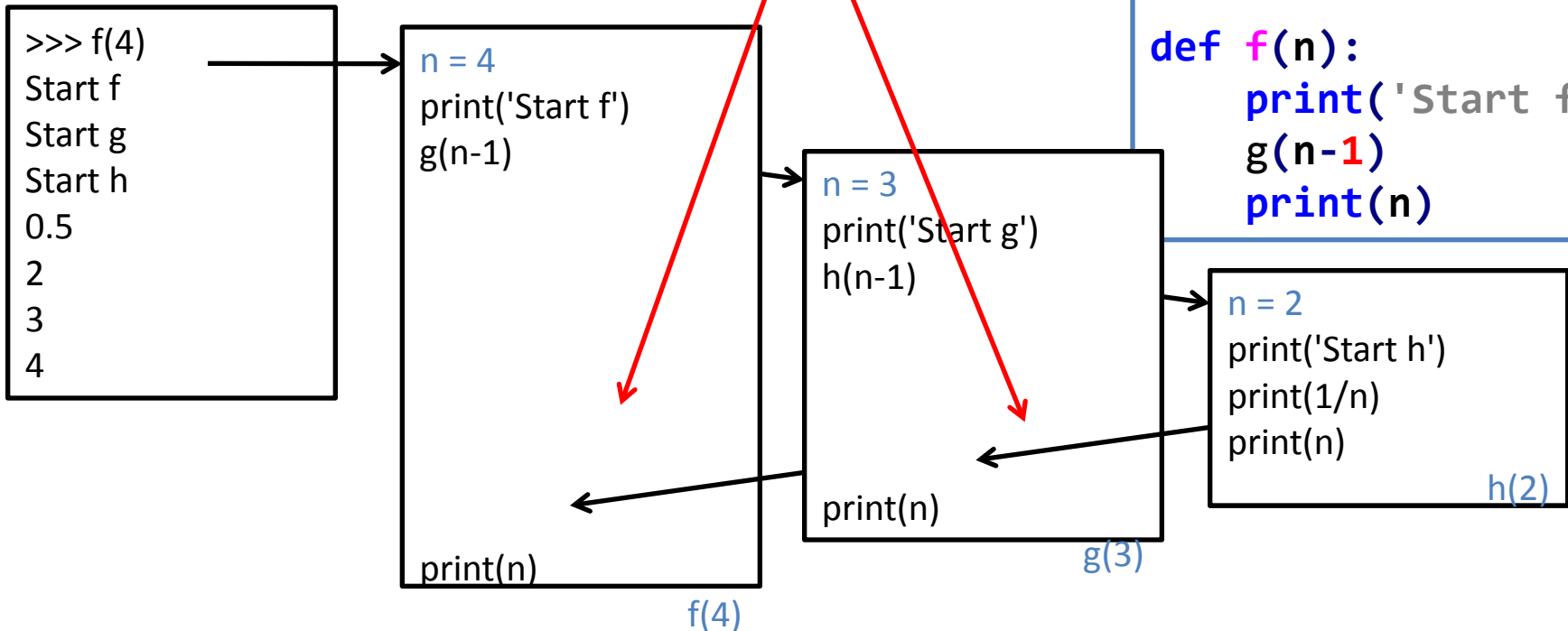
Q: Ինչպես են բոլոր **namespace**-երը  
դեկլարվում Python-ի կոդմիջ?

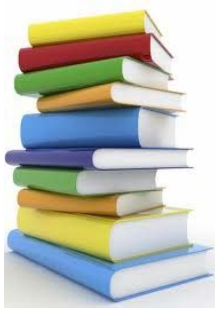
Q: Ինչպես է Python-ը իմանում  
որտեղից շարունակել?

```
def h(n):  
    print('Start h')  
    print(1/n)  
    print(n)
```

```
def g(n):  
    print('Start g')  
    h(n-1)  
    print(n)
```

```
def f(n):  
    print('Start f')  
    g(n-1)  
    print(n)
```

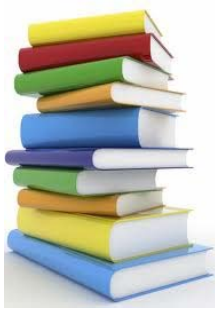




# program stack

System-ը հատկացնում է  
հիշողություն **program stack**-ին, որը  
հիշում է function call-ի ժամանակ  
սահմանված արժեքները

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

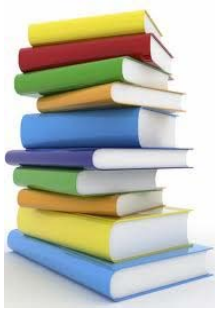


# program stack

System-ը հատկացնում է  
հիշողություն **program stack**-ին, որը  
հիշում է function call-ի ժամանակ  
սահմանված արժեքները

```
>>> f(4)
```

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```



# program stack

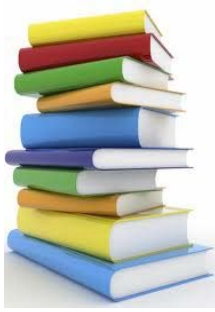
System-ը հատկացնում է  
հիշողություն **program stack**-ին, որը  
հիշում է function call-ի ժամանակ  
սահմանված արժեքները

>>> f(4)

n = 4

f(4)

```
1 def h(n):
2     print('Start h')
3     print(1/n)
4     print(n)
5
6 def g(n):
7     print('Start g')
8     h(n-1)
9     print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```



# program stack

System-ը հատկացնում է  
հիշողություն **program stack**-ին, որը  
հիշում է function call-ի ժամանակ  
սահմանված արժեքները

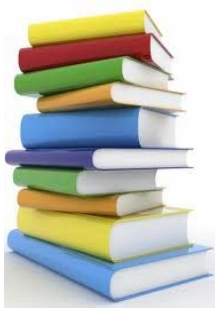
```
>>> f(4)  
Start f
```

```
n = 4  
print('Start f')
```

f(4)

```
1 def h(n):  
2     print('Start h')  
3     print(1/n)  
4     print(n)  
5  
6 def g(n):  
7     print('Start g')  
8     h(n-1)  
9     print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```





# program stack

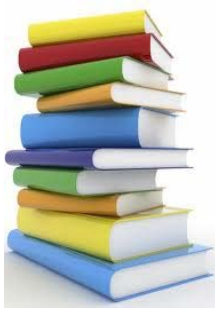
System-ը հատկացնում է  
հիշողություն **program stack**-ին, որը  
հիշում է function call-ի ժամանակ  
սահմանված արժեքները

```
>>> f(4)  
Start f
```

```
n = 4  
print('Start f')  
g(n-1)
```

f(4)

```
1 def h(n):  
2     print('Start h')  
3     print(1/n)  
4     print(n)  
5  
6 def g(n):  
7     print('Start g')  
8     h(n-1)  
9     print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```



# program stack

System-ը հատկացնում է  
հիշողություն **program stack**-ին, որը  
հիշում է function call-ի ժամանակ  
սահմանված արժեքները

```
>>> f(4)  
Start f
```

```
n = 4  
print('Start f')  
g(n-1)
```

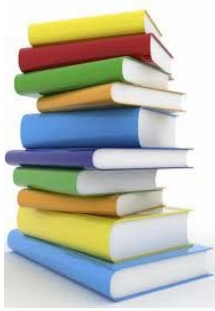
f(4)

line = 14

n = 4

Program stack

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```



# program stack

... կատարի 14 տողը  
էրբ  $g(n-1)$  return անի

```
1 def h(n):  
2     print('Start h')  
3     print(1/n)  
4     print(n)  
5  
6 def g(n):  
7     print('Start g')  
8     h(n-1)  
9     print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

line = 14

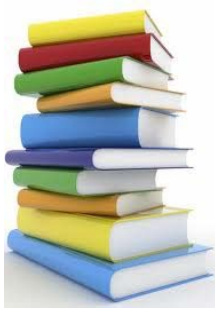
n = 4

Program stack

>>> f(4)  
Start f

n = 4  
print('Start f')  
g(n-1)

f(4)



# program stack

```
>>> f(4)  
Start f
```

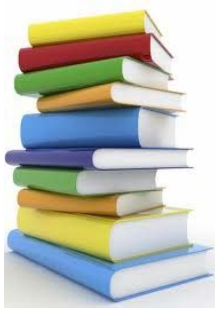
```
n = 4  
print('Start f')  
g(n-1)
```

f(4)

line = 14
n = 4

Program stack

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```



# program stack

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

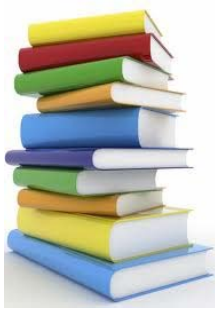
line = 14
n = 4

Program stack

>>> f(4) Start f
---------------------

n = 3
-------

g(3)



# program stack

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

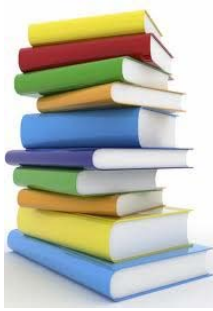
line = 14
n = 4

Program stack

>>> f(4) Start f Start g
--------------------------------

n = 3 print('Start g')
---------------------------

g(3)



# program stack

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

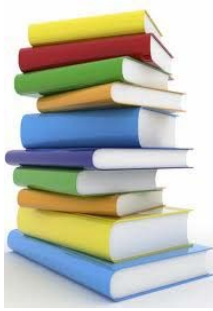
line = 14
n = 4

Program stack

>>> f(4) Start f Start g
--------------------------------

n = 3 print('Start g') h(n-1)
-------------------------------------

g(3)



# program stack

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

line = 9
n = 2
line = 14
n = 4

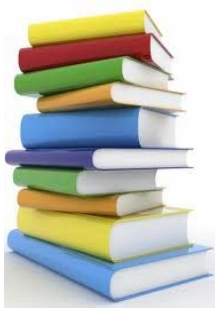
Program stack

```
>>> f(4)  
Start f  
Start g
```

```
n = 3  
print('Start g')  
h(n-1)
```

g(3)





# program stack

line = 9
n = 2
line = 14
n = 4

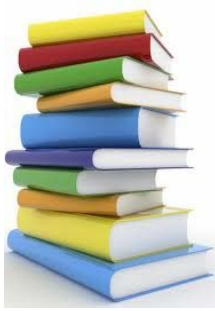
Program stack

```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

```
>>> f(4)
Start f
Start g
```

n = 2

h(2)



# program stack

line = 9
n = 2
line = 14
n = 4

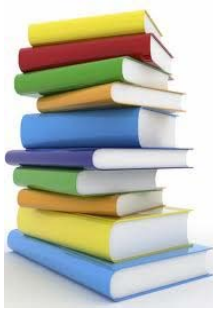
Program stack

```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

```
>>> f(4)
Start f
Start g
Start h
```

```
n = 2
print('Start h')
```

h(2)



# program stack

line = 9
n = 2
line = 14
n = 4

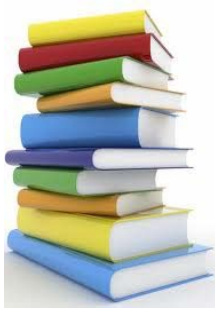
Program stack

```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

```
>>> f(4)
Start f
Start g
Start h
0.5
```

```
n = 2
print('Start h')
print(1/n)
```

h(2)



# program stack

line = 9
n = 2
line = 14
n = 4

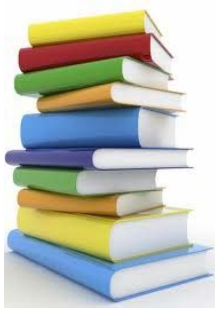
Program stack

```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

```
>>> f(4)
Start f
Start g
Start h
0.5
2
```

```
n = 2
print('Start h')
print(1/n)
print(n)
```

h(2)



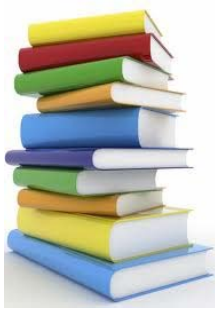
# program stack

```
>>> f(4)
Start f
Start g
Start h
0.5
2
```

line = 9
n = 2
line = 14
n = 4

Program stack

```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```



# program stack

```
>>> f(4)
Start f
Start g
Start h
0.5
2
3
```

line = 14

n = 4

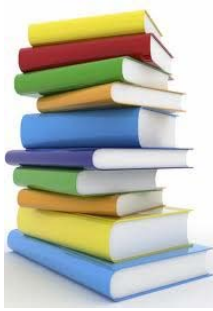
Program stack

```
n = 3
print('Start g')
h(n-1)
```

print(n)

g(3)

```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```



# program stack

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

```
>>> f(4)  
Start f  
Start g  
Start h  
0.5  
2  
3  
4
```

```
n = 4  
print('Start f')  
g(n-1)
```

print(n)

f(4)

Program stack

```
n = 3  
print('Start g')  
h(n-1)
```

print(n)

g(3)

# Scope & global/local namespace

Ամեն function-ի կանչ ունի իր namespace-ը



# Scope & global/local namespace

Ամեն function-ի կանչ ունի իր namespace-ը

- Այս namespace-ում ապրում են function-ի կատարման ժամանակ սահմանված **local variables**

# Scope & global/local namespace

Ամեն function-ի կանչ ունի իր namespace-ը

- Այս namespace-ում ապրում են function-ի կատարման ժամանակ սահմանված **local variables**
- Այս անունների **scope-ը/տեսադաշտը** հենց function-ի **namespace-ն** է

# Scope & global/local namespace

Ամեն function-ի կանչ ունի իր namespace-ը

- Այս namespace-ում ապրում են function-ի կատարման ժամանակ սահմանված **local variables**
- Այս անունների **scope-ը/տեսադաշտը** հենց function-ի **namespace-ն** է

Ամեն name Python-ի ծրագրում ունի scope

# Scope & global/local namespace

Ամեն function-ի կանչ ունի իր namespace-ը

- Այս namespace-ում ապրում են function-ի կատարման ժամանակ սահմանված **local variables**
- Այս անունների **scope-ը/տեսադաշտը** հենց function-ի **namespace-ն** է

Ամեն name Python-ի ծրագրում ունի scope

- Իր scope-ից դուրս name-ը չկա => ամեն reference կգենեռացնի error.

# Scope & global/local namespace

Ամեն function-ի կանչ ունի իր namespace-ը

- Այս namespace-ում ապրում են function-ի կատարման ժամանակ սահմանված **local variables**
- Այս անունների **scope-ը/տեսադաշտը** հենց function-ի **namespace-ն** է

Ամեն name Python-ի ծրագրում ունի scope

- Իր scope-ից դուրս name-ը չկա => ամեն reference կգենեռացնի error.
- Name-երը որոնք սահմանված են interpreter shell –ում կամ module-ում և չեն գտնվում function-ի մեջ ունեն **global scope.**

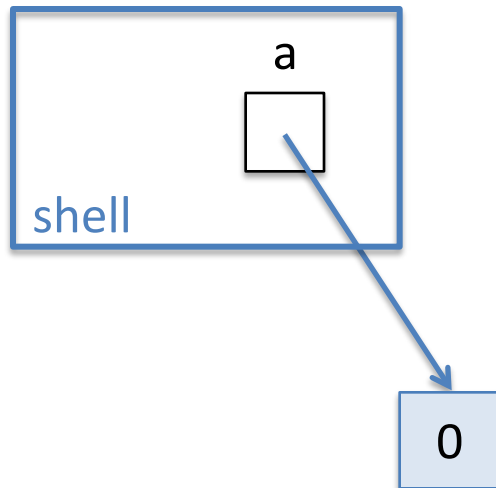
# Example: local scope

```
def f(b): # f : global scope, b : local scope  
    a = 6 # a : local scope  
    return a*b
```

# Example: local scope

```
def f(b):# f : global scope, b : local scope  
    a = 6 # a : local scope  
    return a*b
```

```
>>> a = 0 # global scope
```



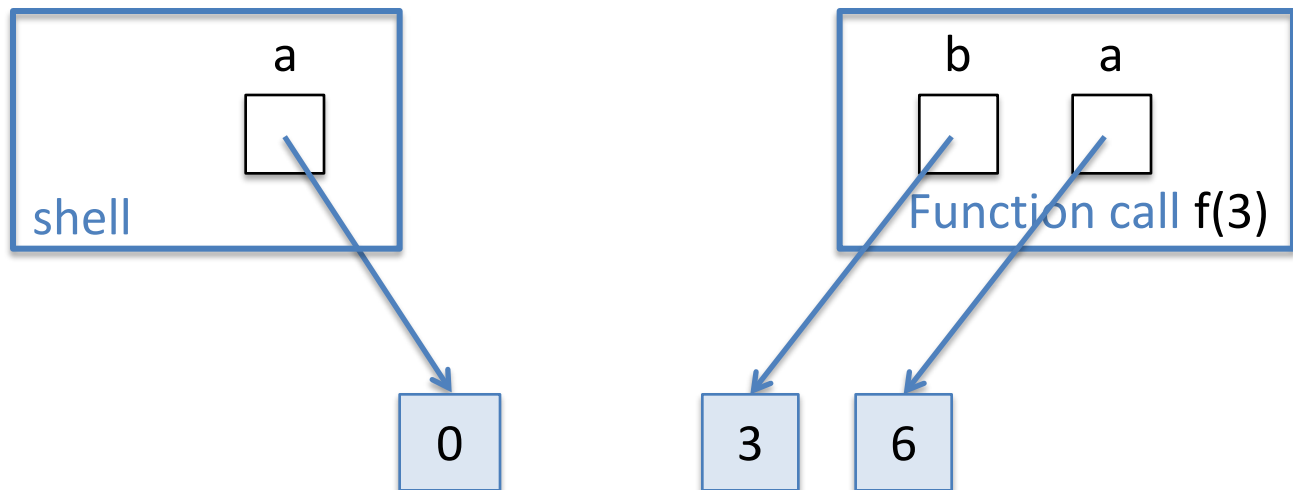
# Example: local scope

```
def f(b):# f : global scope, b : local scope  
    a = 6 # a : local scope  
    return a*b
```

```
>>> a = 0 # global scope
```

```
>>> f(3)
```

```
18
```





# Example: local scope

```
def f(b):# f : global scope, b : local scope
    a = 6 # a : local scope
    return a*b
```

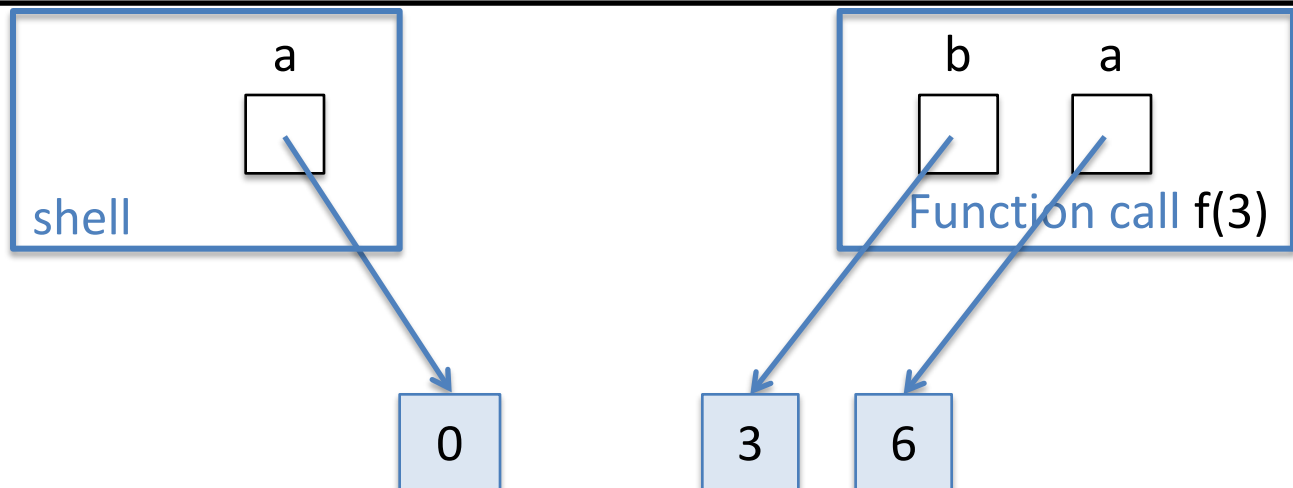
```
>>> a = 0 # global scope
```

```
>>> f(3)
```

```
18
```

```
>>> a
```

```
0
```



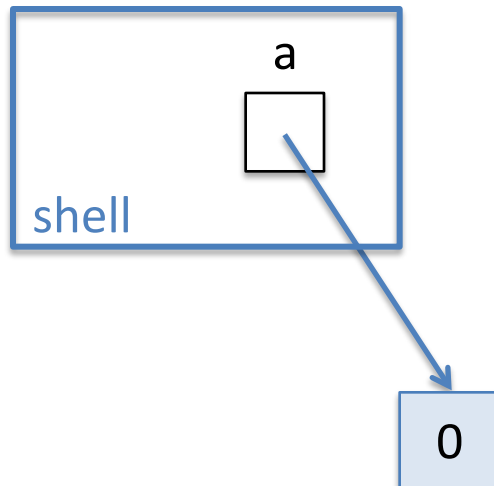
# Example: global scope

```
def f(b): # f : global scope, b : local scope  
    return a*b
```

# Example: global scope

```
def f(b): # f : global scope, b : local scope  
    return a*b
```

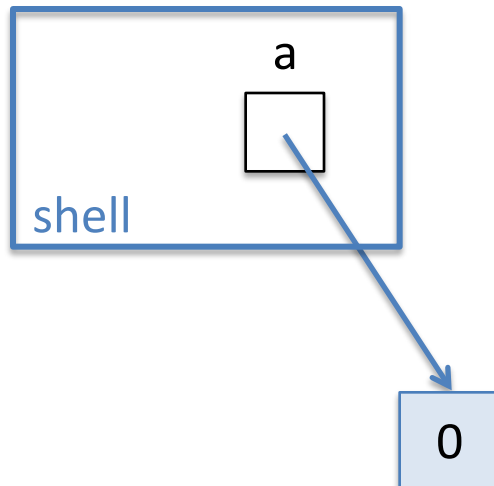
```
>>> a = 0 # global scope
```



# Example: global scope

```
def f(b): # f : global scope, b : local scope  
    return a*b
```

```
>>> a = 0 # global scope  
>>> f(3)
```



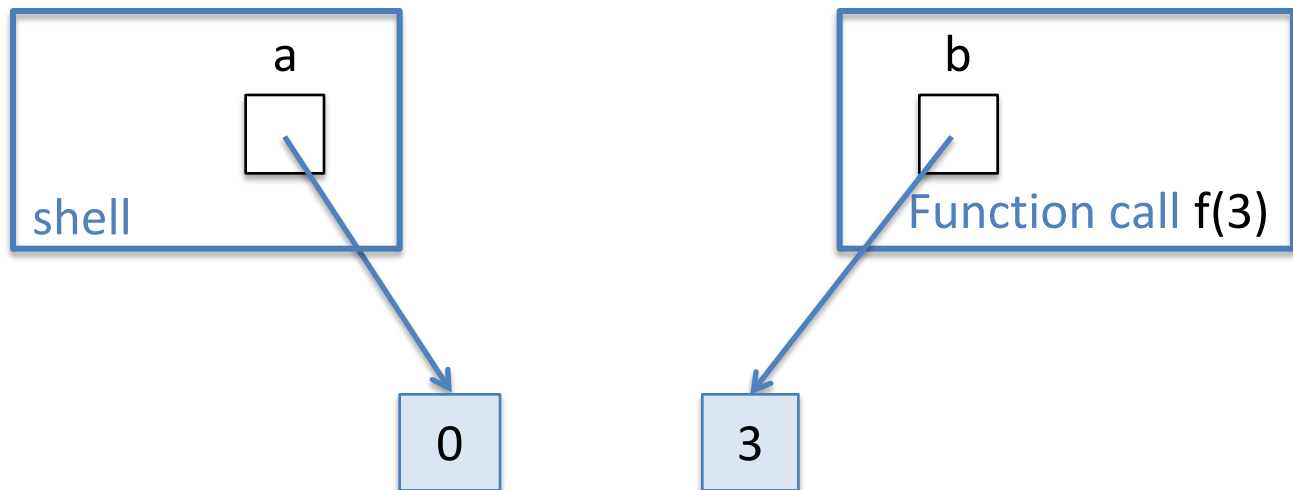
# Example: global scope

```
def f(b): # f : global scope, b : local scope
    return a*b
```

```
>>> a = 0 # global scope
```

```
>>> f(3)
```

```
0
```



# Example: global scope

```
def f(b):#f : global scope, b : local scope  
    return a*b
```

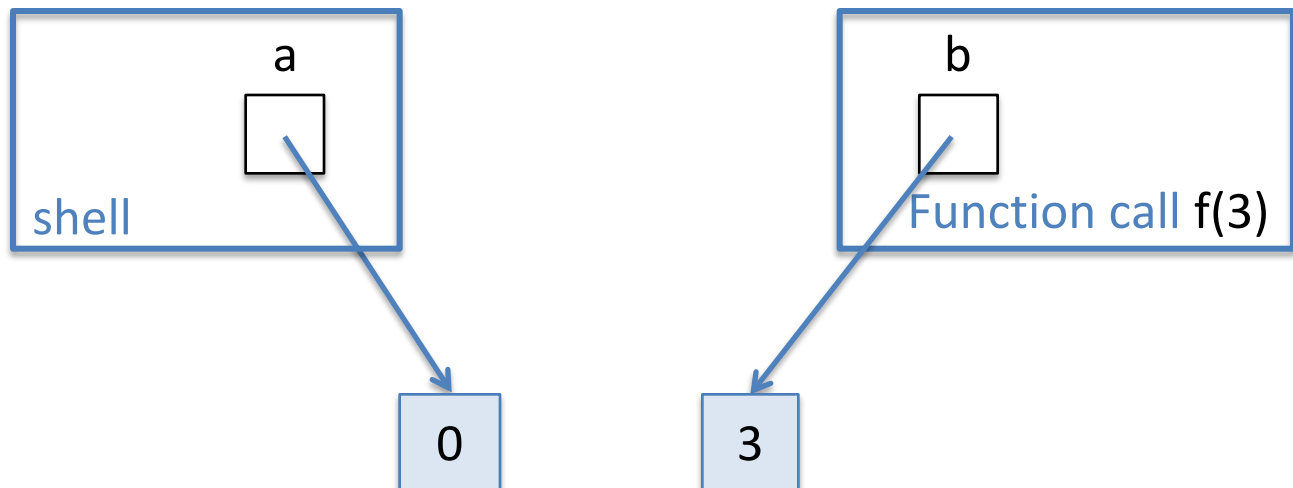
```
>>> a = 0 # global scope
```

```
>>> f(3)
```

```
0
```

```
>>> a
```

```
0
```



# How Python evaluates names

Ինչպես է Python interpreter-ը որոշում որ name-ը  
օգտագործել local թե global?

```
def f(b):  
    return a*b  
  
>>> a = 0 #global scope  
>>> f(3)  
0  
>>> a  
0
```

# How Python evaluates names

Ինչպես է Python interpreter-ը որոշում որ name-ը  
օգտագործել local թե global?

Python interpreter-ը search է անում  
name-ը հետևյալ կարգով

```
def f(b):  
    return a*b  
  
>>> a = 0 #global scope  
>>> f(3)  
0  
>>> a  
0
```



# How Python evaluates names

Ինչպես է Python interpreter-ը որոշում որ name-ը  
օգտագործել local թե global?

Python interpreter-ը search է անում  
name-ը հետևյալ կարգով

1. Առաջինը փնտրում է  
պարփակող function-ի  
namespace-ը

```
def f(b):  
    return a*b  
  
>>> a = 0 #global scope  
>>> f(3)  
0  
  
>>> a  
0
```

# How Python evaluates names

Ինչպես է Python interpreter-ը որոշում որ name-ը  
օգտագործել local թե global?

Python interpreter-ը search է անում  
name-ը հետևյալ կարգով

1. Առաջինը փնտրում է  
պարփակող function-ի  
namespace-ը
2. Եթե չգտավ փնտրում է global  
(module) namespace-ը

```
def f(b):  
    return a*b  
  
>>> a = 0 #global scope  
>>> f(3)  
0  
>>> a  
0
```

# How Python evaluates names

Ինչպես է Python interpreter-ը որոշում որ name-ը  
օգտագործել local թե global?

Python interpreter-ը search է անում  
name-ը հետևյալ կարգով

1. Առաջինը փնտրում է  
պարփակող function-ի  
namespace-ը
2. Եթե չգտավ փնտրում է global  
(module) namespace-ը
3. Եթե չգտավ փնտրում է  
builtins module-ի namespace-ը  
և վերջ

```
def f(b):  
    return a*b  
  
>>> a = 0 #global scope  
>>> f(3)  
0  
>>> a  
0
```

# Global variable inside a function

```
def f(b):  
    global a  
    a = 6  
    return a*b
```

# Global variable inside a function

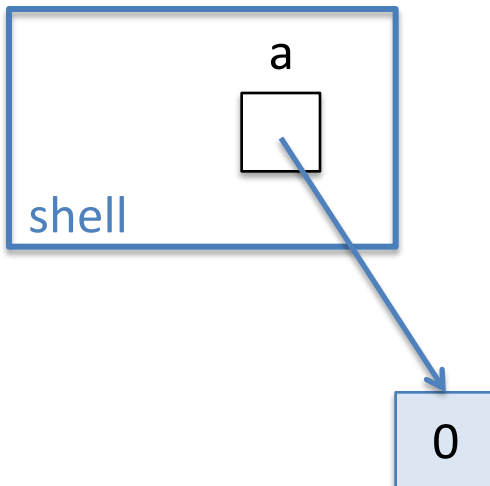
a-ի reference-ը f()-ում գնվց է  
ստացվում global a

```
def f(b):  
    global a  
    a = 6  
    return a*b
```

# Global variable inside a function

a-h reference-u f()-nu gnug t  
unuhu global a

```
def f(b):  
    global a  
    a = 6  
    return a*b  
  
>>> a = 0
```

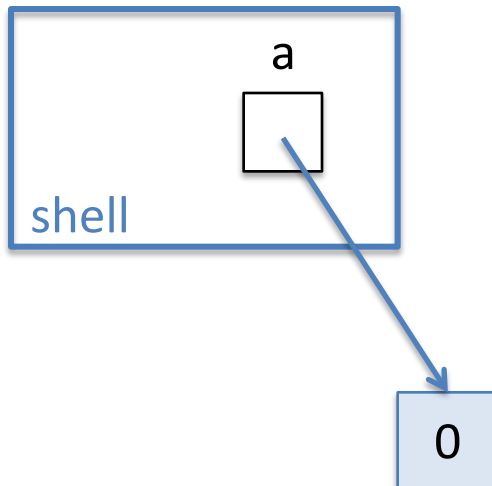


# Global variable inside a function

a-ի reference-ը f()-ում գնվց է  
տալիս global a

```
def f(b):  
    global a  
    a = 6  
    return a*b
```

```
>>> a = 0  
>>> f(3)
```

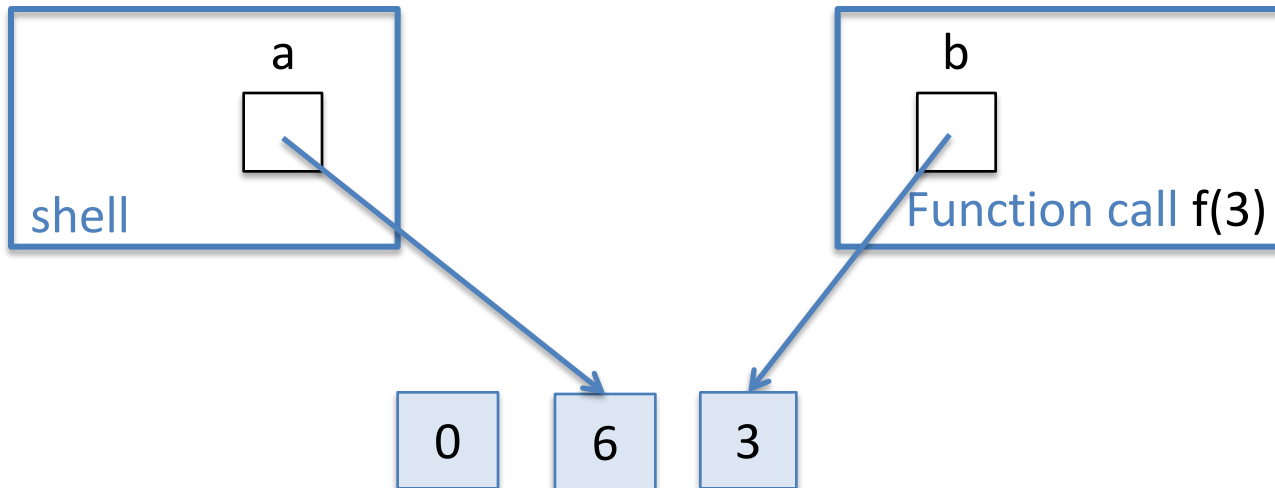


# Global variable inside a function

a-h reference-u f()-nu gnug t  
unuhu global a

```
def f(b):  
    global a  
    a = 6  
    return a*b
```

```
>>> a = 0  
>>> f(3)  
18
```



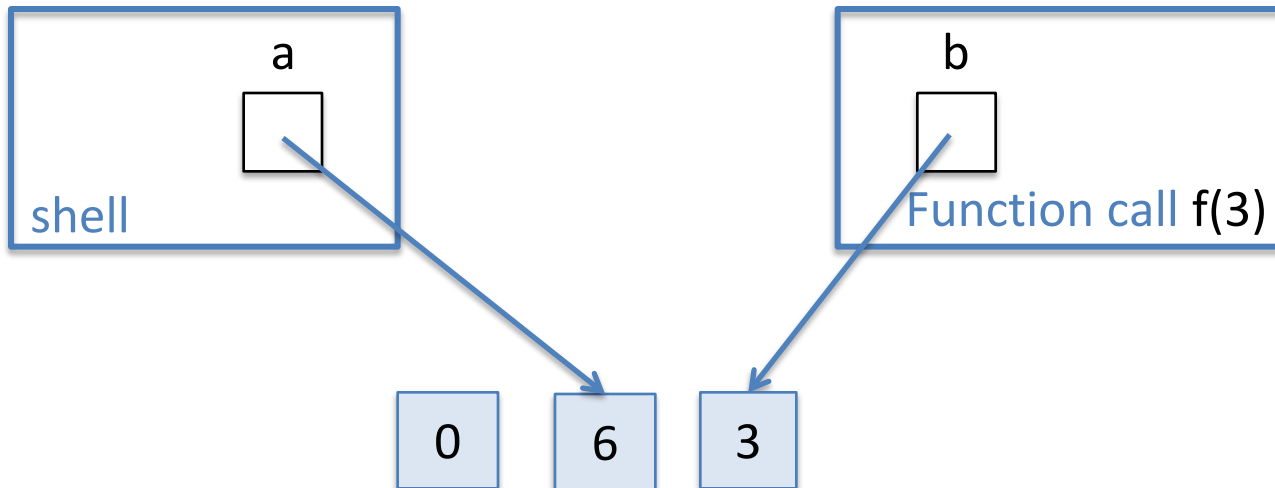


# Global variable inside a function

a-h reference-u f()-nu gnug t  
unuhu global a

```
def f(b):  
    global a  
    a = 6  
    return a*b
```

```
>>> a = 0  
>>> f(3)  
18  
>>> a  
6
```



# Exceptions revisited

Երբ ծրագրի կատարումը ընդհատվում է error-ի պատճառով, exception օբյեկտ է ստեղծվում

- այս object-ը ունի type որը հենց *type of error*
- object-ը պարունակում է *information* error-ի մասին
- *default behavior: print* this information and *stop*

# Exceptional control flow

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

# Exceptional control flow

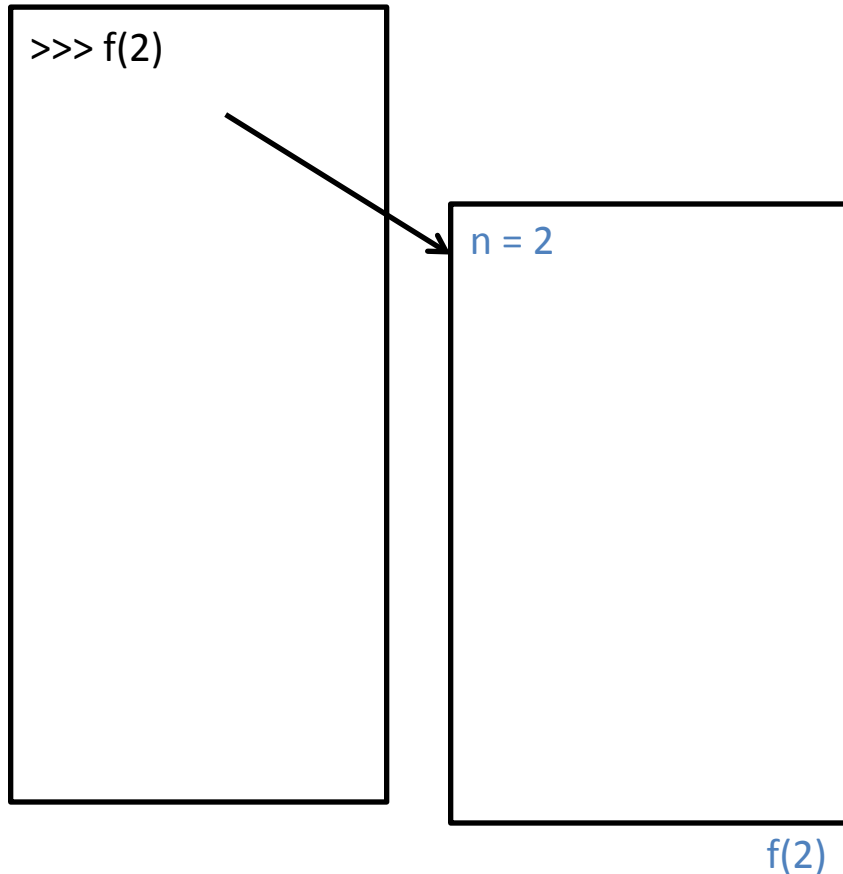
## Normal control flow

```
>>> f(2)
```

```
1  def h(n):  
2      print('Start h')  
3      print(1/n)  
4      print(n)  
5  
6  def g(n):  
7      print('Start g')  
8      h(n-1)  
9      print(n)  
10  
11 def f(n):  
12     print('Start f')  
13     g(n-1)  
14     print(n)
```

# Exceptional control flow

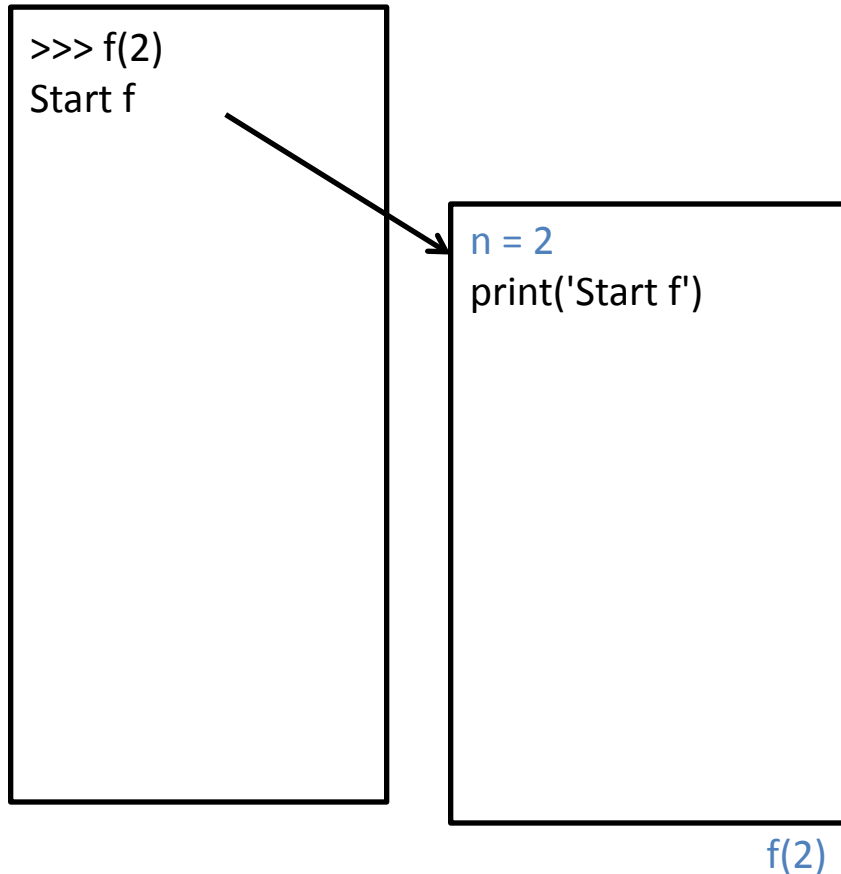
## Normal control flow



```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

# Exceptional control flow

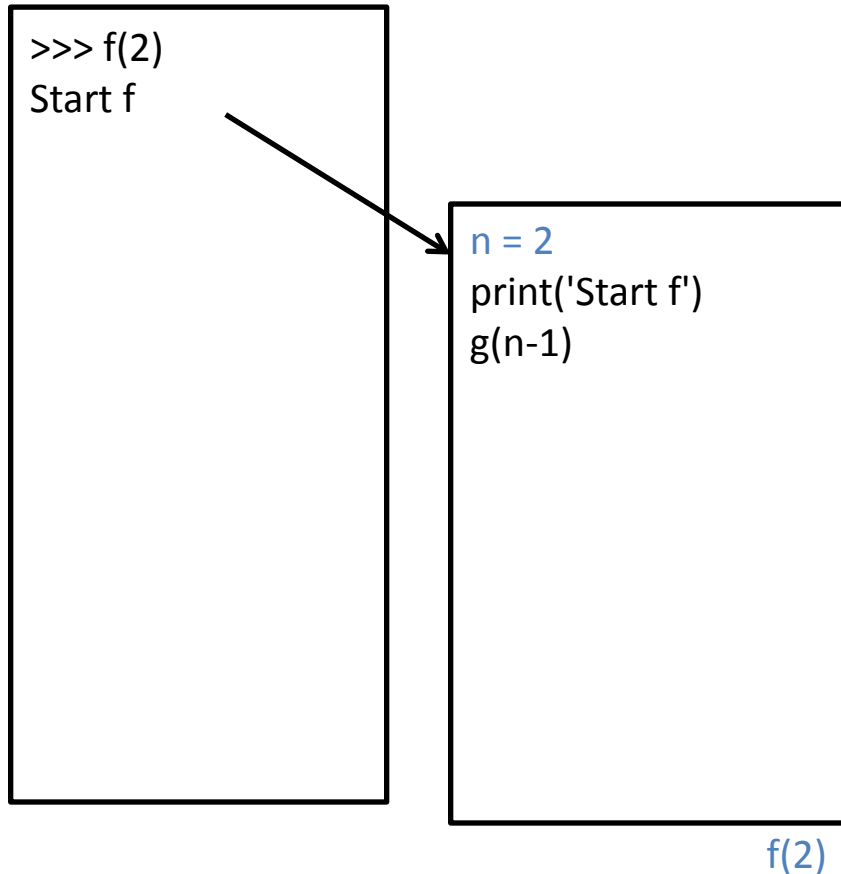
## Normal control flow



```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

# Exceptional control flow

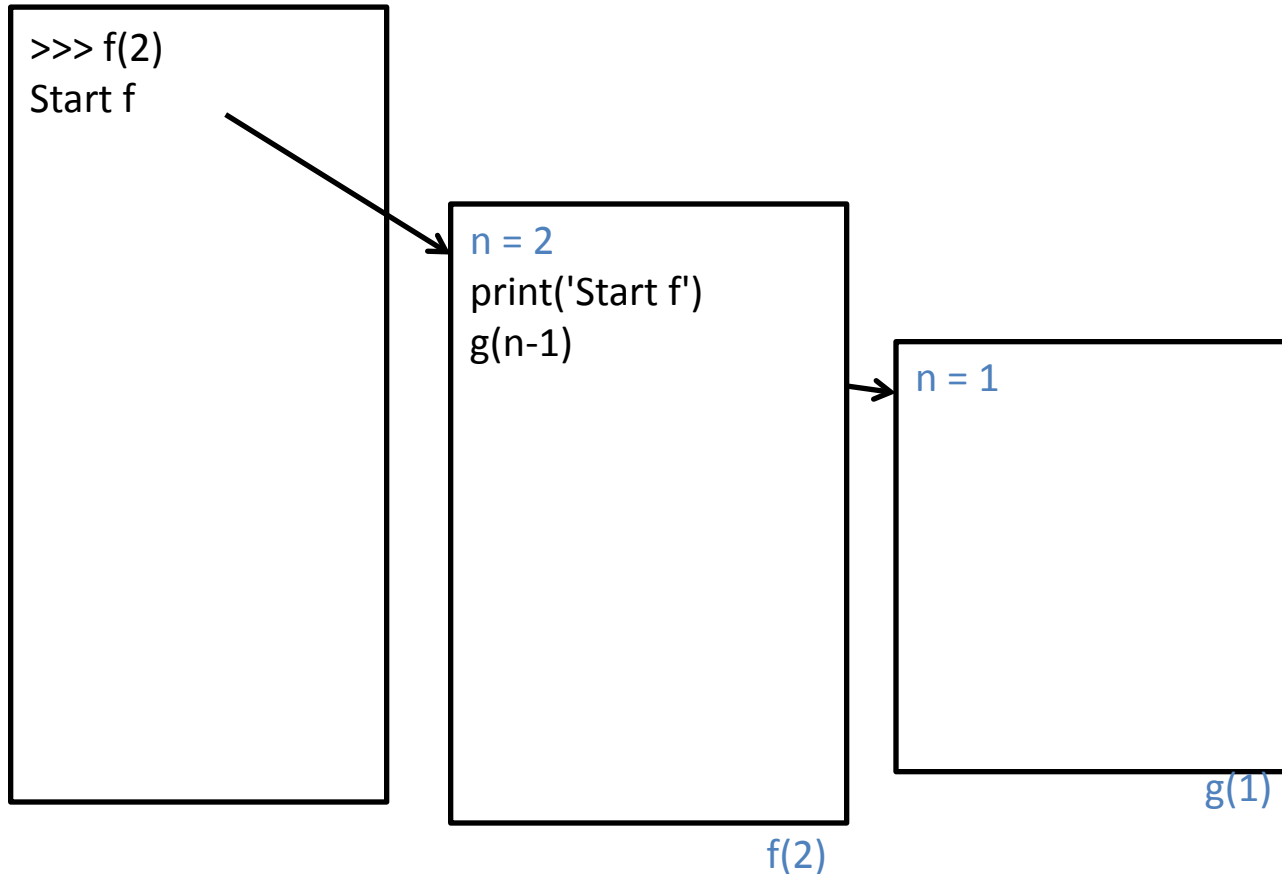
## Normal control flow



```
1  def h(n):
2      print('Start h')
3      print(1/n)
4      print(n)
5
6  def g(n):
7      print('Start g')
8      h(n-1)
9      print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

# Exceptional control flow

## Normal control flow

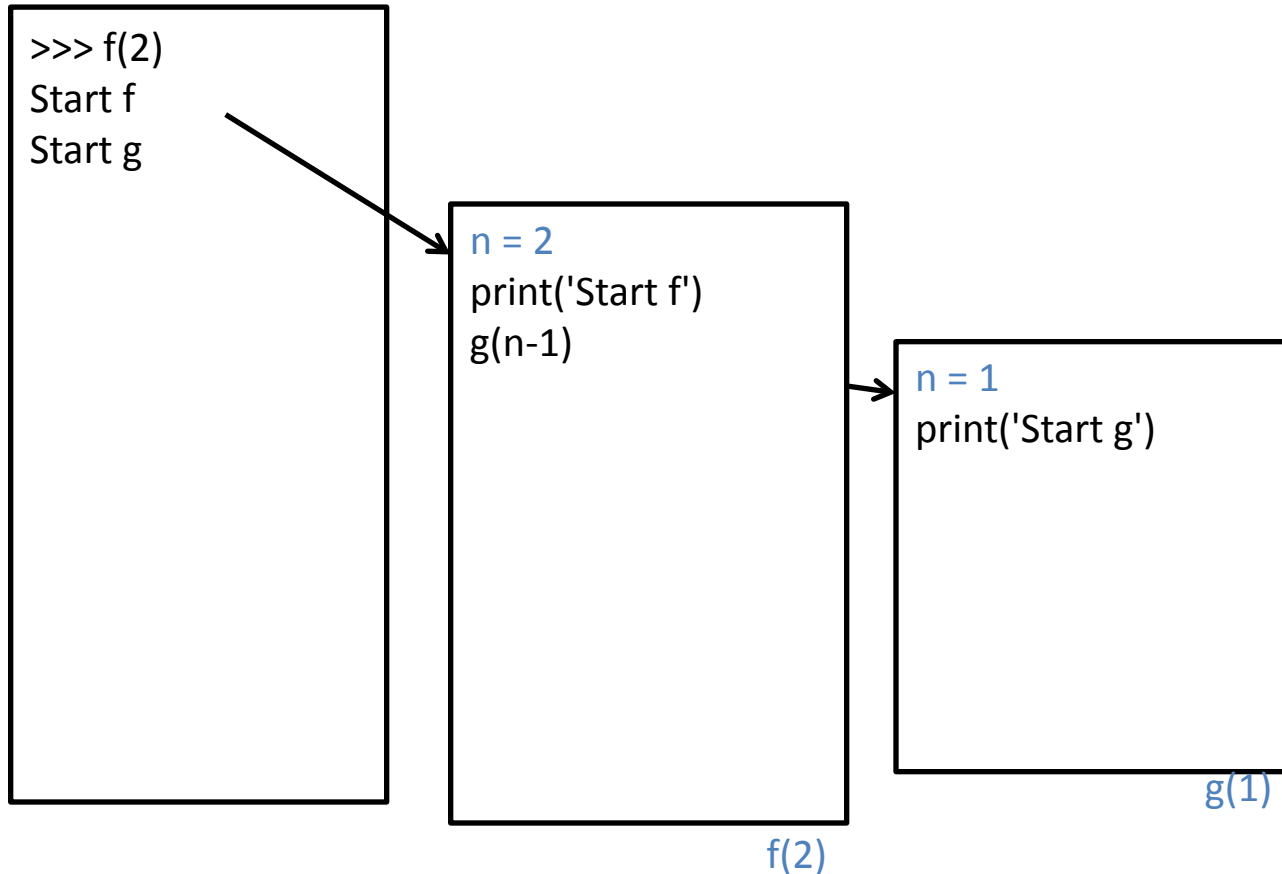


```
1 def h(n):
2     print('Start h')
3     print(1/n)
4     print(n)
5
6 def g(n):
7     print('Start g')
8     h(n-1)
9     print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```



# Exceptional control flow

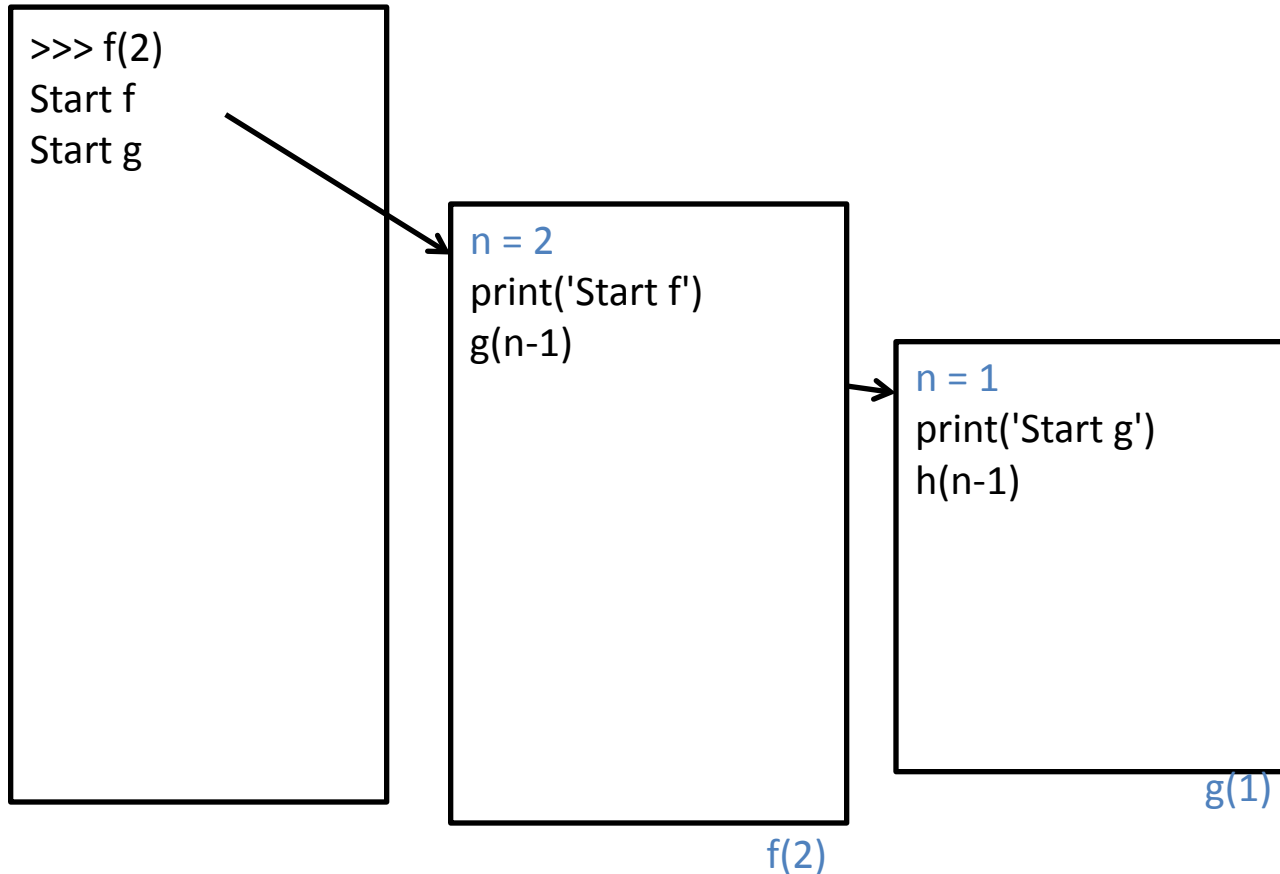
## Normal control flow



```
1 def h(n):
2     print('Start h')
3     print(1/n)
4     print(n)
5
6 def g(n):
7     print('Start g')
8     h(n-1)
9     print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

# Exceptional control flow

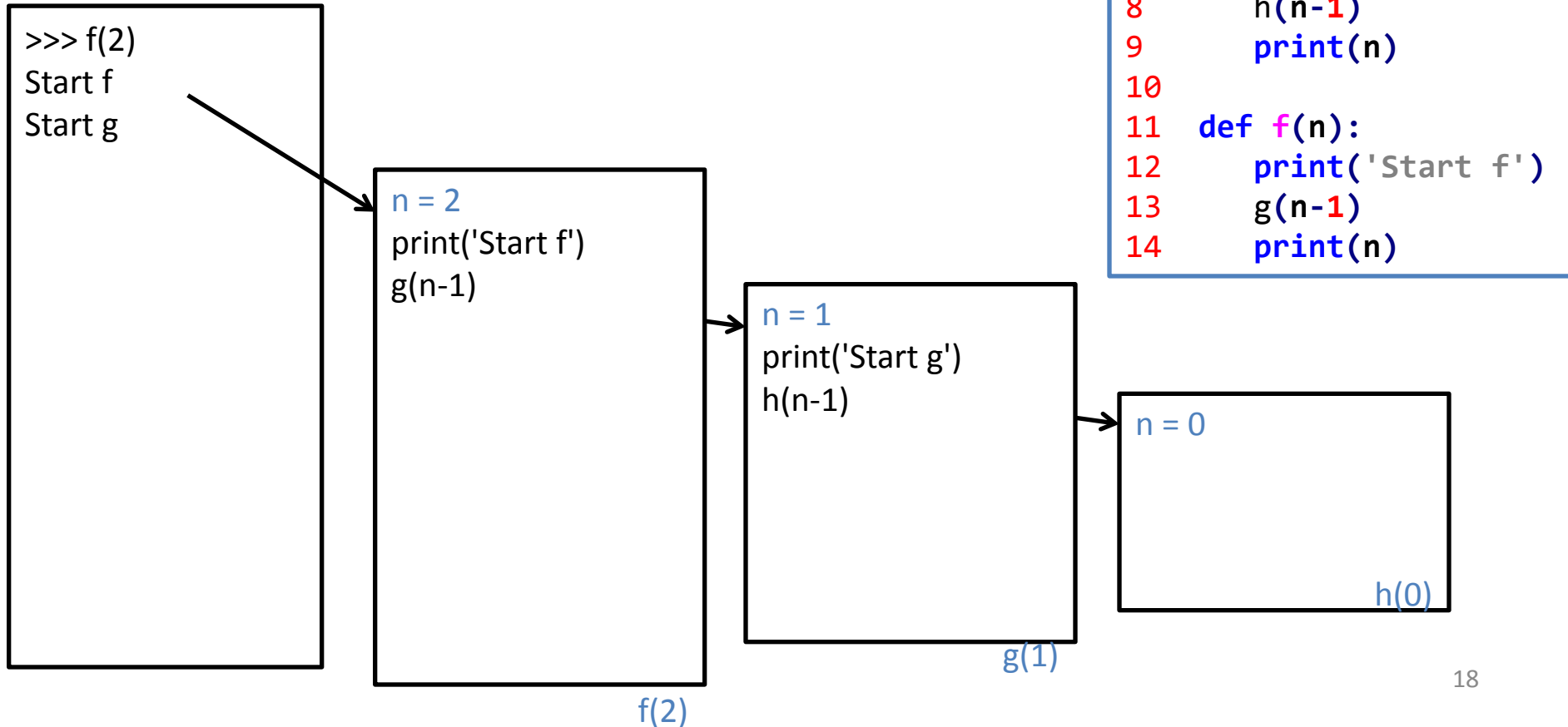
## Normal control flow



```
1 def h(n):
2     print('Start h')
3     print(1/n)
4     print(n)
5
6 def g(n):
7     print('Start g')
8     h(n-1)
9     print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

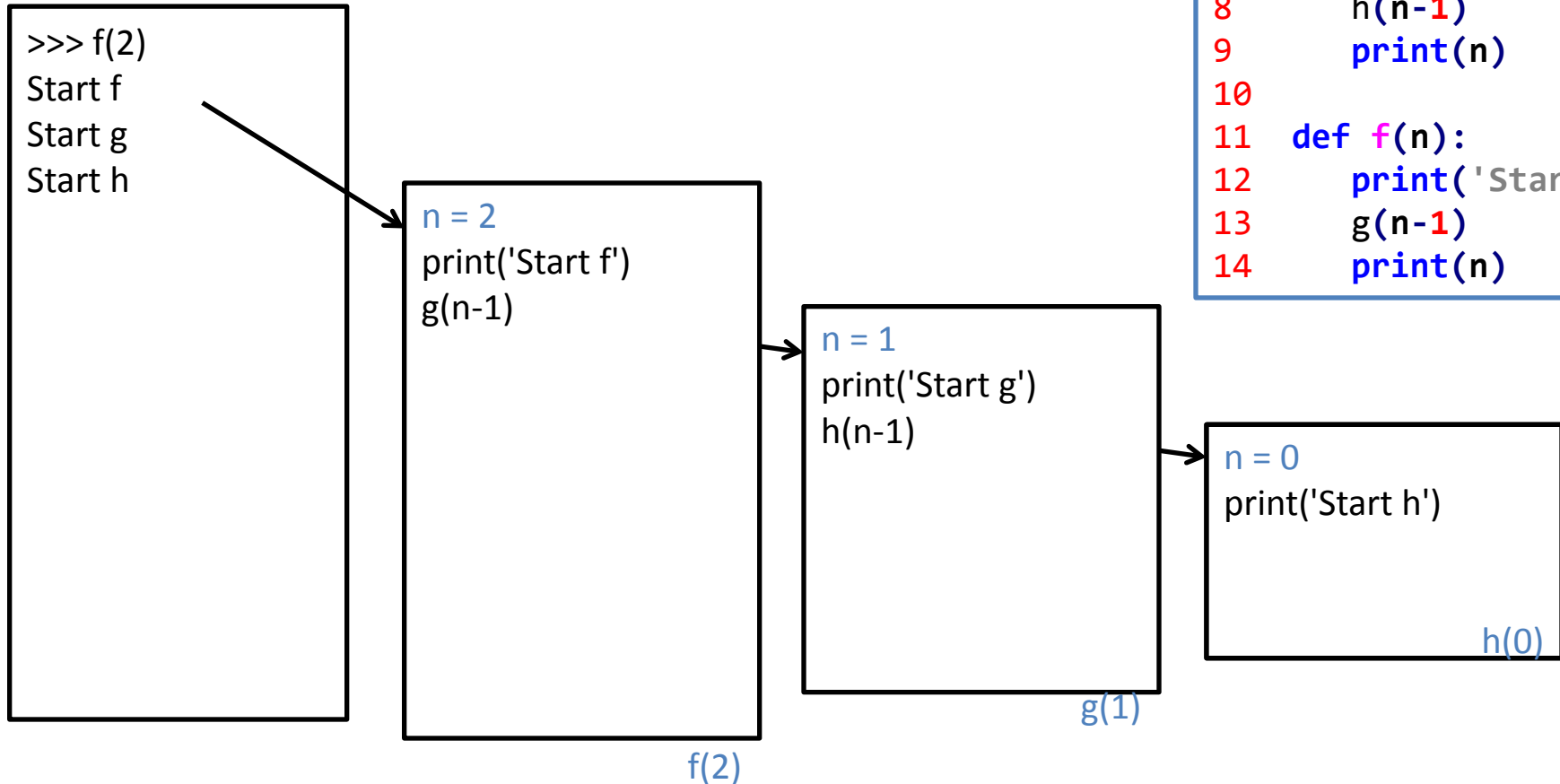
# Exceptional control flow

## Normal control flow



# Exceptional control flow

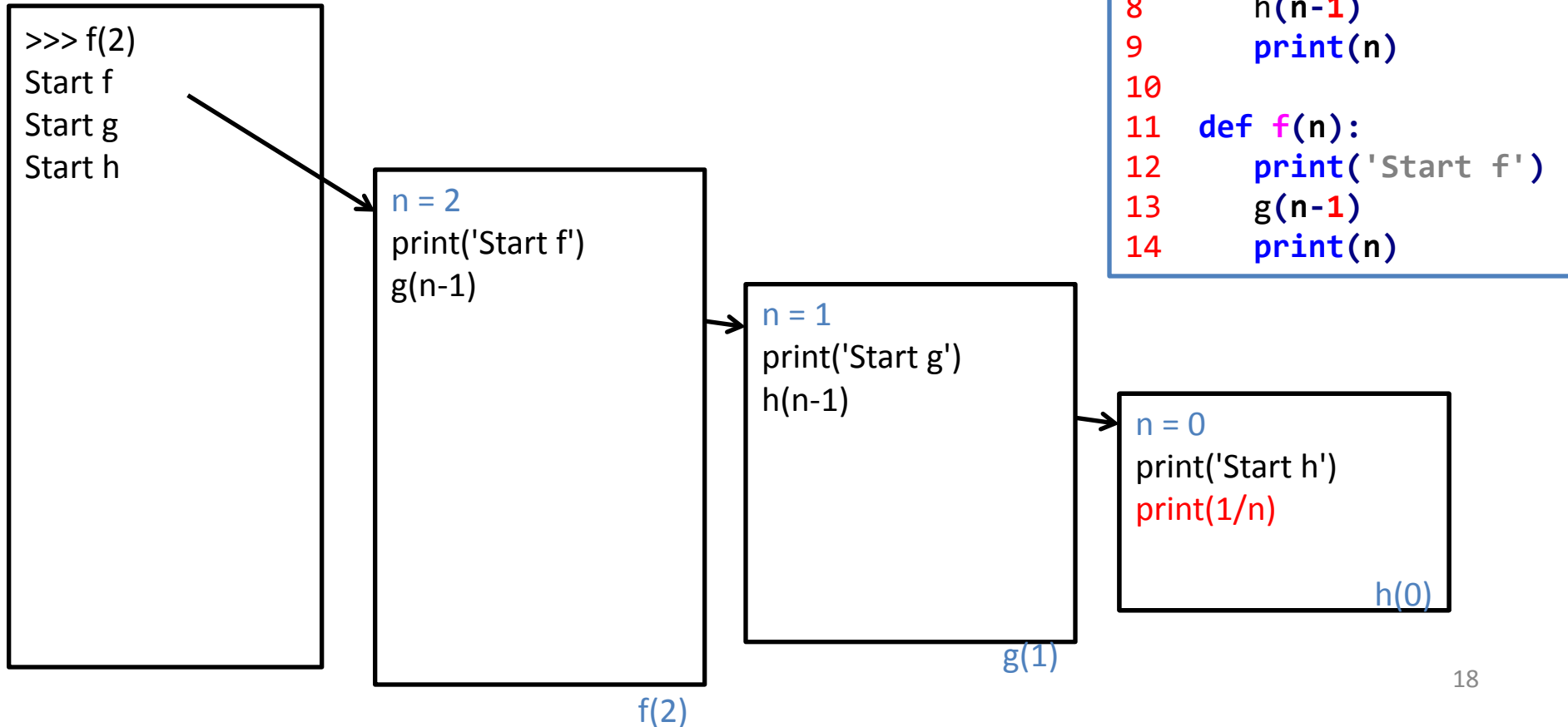
## Normal control flow



```
1 def h(n):
2     print('Start h')
3     print(1/n)
4     print(n)
5
6 def g(n):
7     print('Start g')
8     h(n-1)
9     print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

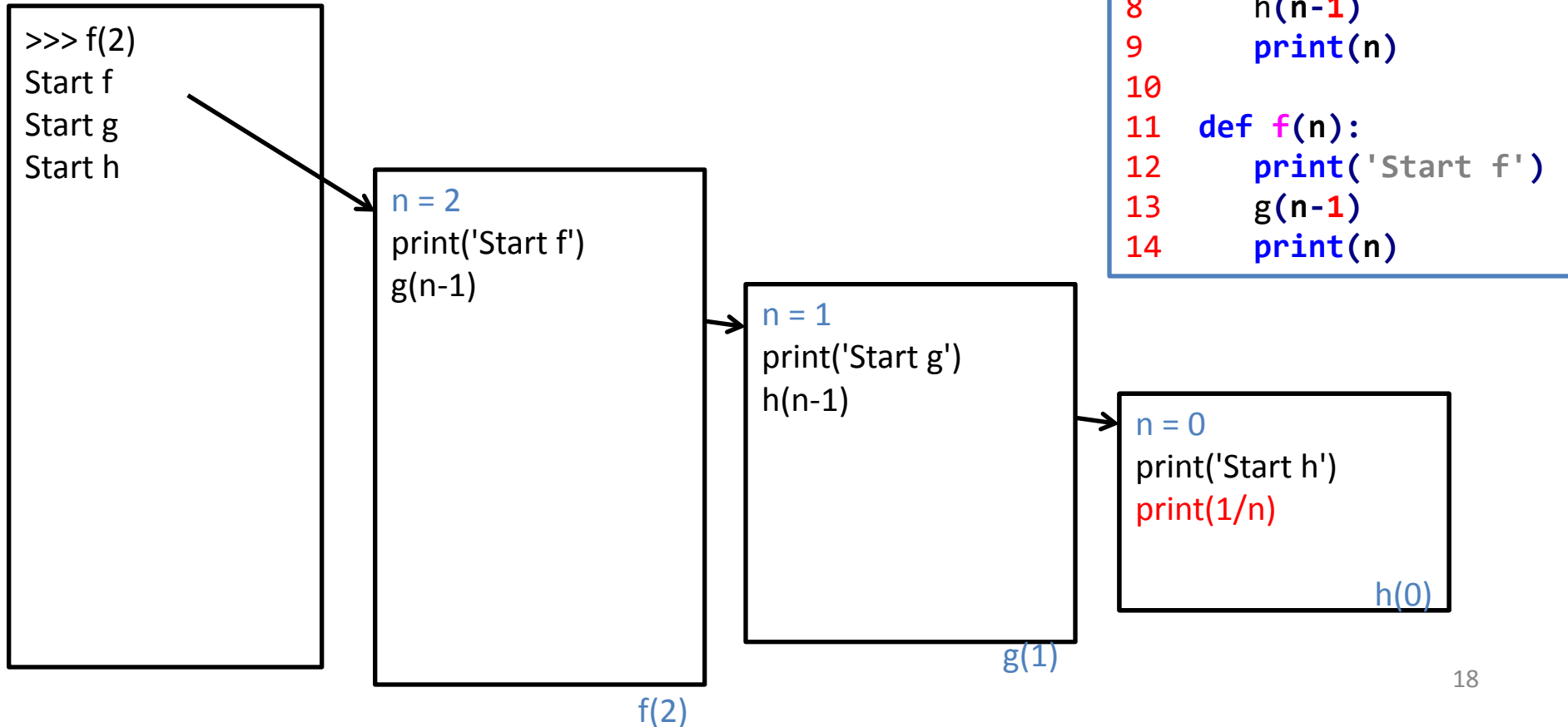
# Exceptional control flow

## Normal control flow



# Exceptional control flow

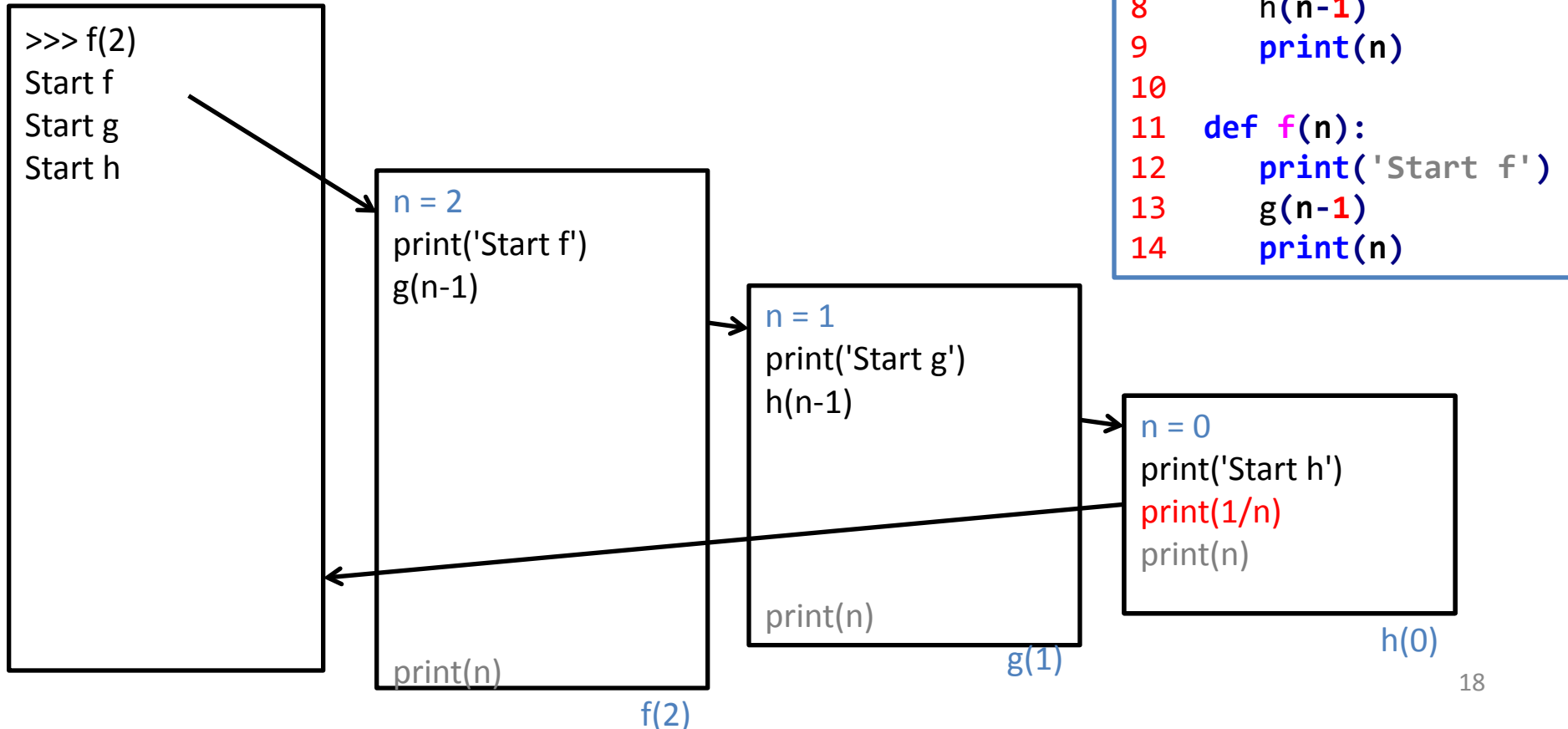
## Exceptional control flow



# Exceptional control flow

## Exceptional control flow

Լռակյաց վարքը: ընդհատել կատարումը  
յուրաքանչյուր «ակտիվ» statement-ի և տպել  
exception object-ի սխալի վերաբերյալ  
ինֆորմացիան.



# Exceptional control flow

## Exceptional control flow

Լռակյաց վարքը: ընդհատել կատարումը  
յուրաքանչյուր «ակտիվ» statement-ի և տպել  
exception object-ի սխալի վերաբերյալ  
ինֆորմացիան.

```
>>> f(2)
Start f
Start g
Start h
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    f(2)
  File "/Users/me/ch7/stack.py", line 13, in f
    g(n-1)
  File "/Users/me/ch7/stack.py", line 8, in g
    h(n-1)
  File "/Users/me/ch7/stack.py", line 3, in h
    print(1/n)
ZeroDivisionError: division by zero
>>>
```

```
1 def h(n):
2     print('Start h')
3     print(1/n)
4     print(n)
5
6 def g(n):
7     print('Start g')
8     h(n-1)
9     print(n)
10
11 def f(n):
12     print('Start f')
13     g(n-1)
14     print(n)
```

n = 0  
print('Start h')  
print(1/n)  
print(n)

h(0)

g(1)

f(2)



# Catching & handling exceptions

Հնարավոր է փոխել **default վարքը** (տպիք սխալը և crash) երբ exception է բարձրացվում, օգտագործելով **try / except** statement-ները

```
strAge = input('Enter your age: ')\nintAge = int(strAge)\nprint('You are {} years old.'.format(intAge))
```

Default behavior/վարք

```
>>>\nEnter your age: fifteen\nTraceback (most recent call last):\n  File "/Users/me/age1.py", line 2, in <module>\n    intAge = int(strAge)\nValueError: invalid literal for int() with base 10: 'fifteen'\n>>>
```

# Catching & handling exceptions

Հնարավոր է փոխել **default վարքը** (տպիք սխալը և crash)  
երբ exception է բարձրացվում, օգտագործելով **try / except**  
statement-ները

```
strAge = input('Enter your age: ')\nintAge = int(strAge)\nprint('You are {} years old.'.format(intAge))
```

```
>>>\nEnter your age: fifteen\nTraceback (most recent call last):\n  File "/Users/me/age1.py", line 2, in <module>\n    intAge = int(strAge)\nValueError: invalid literal for int() with base 10: 'fifteen'\n>>>
```

# Catching & handling exceptions

Հնարավոր է փոխել **default վարքը** (տպիր սխալը և crash)  
երբ exception է բարձրացվում, օգտագործելով **try / except**  
statement-ները

```
try:
    strAge = input('Enter your age: ')
    intAge = int(strAge)
    print('You are {} years old.'
          .format(intAge))
except:
    print('Enter your age using digits 0-9!')
```

# Catching & handling exceptions

Հնարավոր է փոխել **default վարքը** (տպիք սխալը և crash)  
երբ exception է բարձրացվում, օգտագործելով **try / except**  
statement-ները

Եթե exception է  
բարձրացվում **try**-ի  
block-ում, ապա  
հաջորդում է **except**-ի  
բլոկի կատարումը

```
try:
    strAge = input('Enter your age: ')
    intAge = int(strAge)
    print('You are {} years old.'
          .format(intAge))
except:
    print('Enter your age using digits 0-9!')
```

# Catching & handling exceptions

Հնարավոր է փոխել **default վարքը** (տպիր սխալը և crash) երբ exception է բարձրացվում, օգտագործելով **try / except** statement-ները

Եթե exception է բարձրացվում **try**-ի block-ում, ապա հաջորդում է **except**-ի բլոկի կատարումը

```
try:
    strAge = input('Enter your age: ')
    intAge = int(strAge)
    print('You are {} years old.'
          .format(intAge))
except:
    print('Enter your age using digits 0-9!')
```

**Custom behavior:**

**except** code block-ը exception-ին տեր է կանգնում

```
>>>
```

```
Enter your age: fifteen
```

```
Enter your age using digits 0-9!
```

```
>>>
```

# Syntax of try/except

# Syntax of try/except

```
try:  
    <indented code block>  
except:  
    <exception handler block>  
<non-indented statement>
```

# Syntax of try/except

```
try:  
    <indented code block>  
except:  
    <exception handler block>  
<non-indented statement>
```

except statement-ը քննում է **ցանկացած** exception, որը բարձրացվել է try block-ում



# Syntax of try/except

```
try:  
    <indented code block>  
except:  
    <exception handler block>  
<non-indented statement>
```

except statement-ը քննում է **ցանկացած** exception, որը բարձրացվել է try block-ում

Հնարավոր է քննել որոշակի type-ի exception-ներ նշելով except statement-ը և exception-ի type-ը, e.g. `NameError`, `IndexError`, `ValueError`, ...

# Syntax of try/except

```
try:
    <indented code block>
except:
    <exception handler block>
<non-indented statement>
```

except statement-ը քննում է **ցանկացած** exception, որը բարձրացվել է try block-ում

Հնարավոր է քննել որոշակի type-ի exception-ներ նշելով except statement-ը և exception-ի type-ը, e.g. `NameError`, `IndexError`, `ValueError`, ...

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
<non-indented statement>
```

# try/except

```
def readAge(filename):  
    '''converts first line of file filename to an integer  
    and prints it'''  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is', age)  
    except ValueError:  
        print('Value cannot be converted to integer.')
```

# try/except

```
def readAge(filename):  
    '''converts first line of file filename to an integer  
    and prints it'''  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is', age)  
    except ValueError:  
        print('Value cannot be converted to integer.')
```

1 fifteen

age.txt

```
>>> readAge('age.txt')  
Value cannot be converted to integer.  
>>>
```

# try/except

```
def readAge(filename):  
    '''converts first line of file filename to an integer  
    and prints it'''  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is', age)  
    except ValueError:  
        print('Value cannot be converted to integer.')
```

1 fifteen

age.txt

```
>>> readAge('age.txt')  
Value cannot be converted to integer.  
>>> readAge('age.text')  
Traceback (most recent call last):  
  File "<pyshell#11>", line 1, in <module>  
    readAge('age.text')  
  File "/Users/me/ch7.py", line 12, in readAge  
    infile = open(filename)  
IOError: [Errno 2] No such file or directory: 'age.text'  
>>>
```

# try/except

```
def readAge(filename):  
    '''converts first line of file filename to an integer  
    and prints it'''  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is', age)  
    except ValueError:  
        print('Value cannot be converted to integer.')
```

1 fifteen

age.txt

default exception  
handler

```
>>> readAge('age.txt')  
Value cannot be converted to integer.  
>>> readAge('age.text')  
Traceback (most recent call last):  
  File "<pyshell#11>", line 1, in <module>  
    readAge('age.text')  
  File "/Users/me/ch7.py", line 12, in readAge  
    infile = open(filename)  
IOError: [Errno 2] No such file or directory: 'age.text'  
>>>
```

# try/except

```
%%writefile age.txt  
fifteen
```

Writing age.txt

```
def readAge(filename):  
    '''converts first line of file filename to an integer  
    and prints it'''  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is', age)  
    except ValueError:  
        print('Value cannot be converted to integer.')
```

```
readAge('age.txt')
```

Value cannot be converted to integer.

```
readAge('age.text')
```

-----  
FileNotFoundError

Traceback (most recent call last)

# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```




# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```


# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```



# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```

A diagram consisting of three red curved arrows pointing from the left towards the code. The top arrow points to the `try:` block. The middle arrow points to the `except IOError:` block. The bottom arrow points to the `except ValueError:` block.

# Multiple exception handlers

```
def readAge(filename):
    'converts first line of file filename to an integer and
    prints it'
    try:
        infile = open(filename)
        strAge = infile.readline()
        age = int(strAge)
        print('age is',age)
    except IOError:
        # executed only if an IOError exception is raised
        print('Input/Output error.')
    except ValueError:
        # executed only if a ValueError exception is raised
        print('Value cannot be converted to integer.')
    except:
        # executed for any other exception
        print('Other error.')
```

# Multiple exception handlers


```
def readAge(filename):  
    'converts first line of file filename to an integer and  
    prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```

# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and  
    prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```

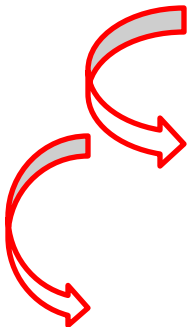
# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and  
    prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```



# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and  
    prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```

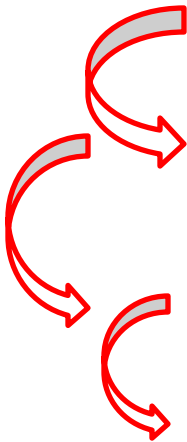


The diagram consists of three red curved arrows. The first arrow starts at the line `age = int(strAge)` and points to the `except IOError:` block. The second arrow starts at the `except IOError:` block and points to the `except ValueError:` block. The third arrow starts at the `except ValueError:` block and points to the final `except:` block. This illustrates the flow of exception handling from the point of error to the appropriate handler.



# Multiple exception handlers

```
def readAge(filename):  
    'converts first line of file filename to an integer and  
    prints it'  
    try:  
        infile = open(filename)  
        strAge = infile.readline()  
        age = int(strAge)  
        print('age is',age)  
    except IOError:  
        # executed only if an IOError exception is raised  
        print('Input/Output error.')  
    except ValueError:  
        # executed only if a ValueError exception is raised  
        print('Value cannot be converted to integer.')  
    except:  
        # executed for any other exception  
        print('Other error.')
```



The diagram consists of three red curved arrows pointing from the line `age = int(strAge)` to the `except` blocks. The first arrow points to `except IOError:`, the second to `except ValueError:`, and the third to `except:`. This illustrates how different types of exceptions (IOError, ValueError, and any other) are handled by different blocks of code.

# Exercise 1A

- Գրեք ծրագիր, որը user-ից վերցնում է 2 թիվ և կատարում է բաժանում.
- օգտագործեք try/except կառավարելու համար սխալ input-ները
- օգտագործեք infinite loop pattern

```
>>>
Enter 2 digits (format: x y):5 3
1.6666666666666667
Enter 2 digits (format: x y):5 0
ZeroDivisionError
try again without 0
Enter 2 digits (format: x y):a b
ValueError
try again without letters
Enter 2 digits (format: x y):4 b
ValueError
try again without letters
Enter 2 digits (format: x y):
```

# Solution

```
while True:
    nums = input('Enter 2 digits (format: x y):')
    (x,y) = nums.split()
    try:
        # code
    except
        # code
    except
        # code
    except
        # code
```

# Solution

```
while True:
    nums = input('Enter 2 digits (format: x y):')
    (x,y) = nums.split()
    try:
        x = int(x)
        y = int(y)
        res = x/y
        print(res)
    except ZeroDivisionError:
        print('ZeroDivisionError')
        print('try again without 0')
    except ValueError:
        print('ValueError')
        print('try again without letters')
    except:
        print("Something went wrong!")
        raise
```

# Exercise 1B

Ձևափոխեք նախորդ խնդիրը ամփոփելով  
այն divisor() function-ի մեջ

```
def divider():  
    while True:  
        nums = input('Enter 2 digits (format: x y):')  
        (x,y) = nums.split()  
        try:  
            # code  
        except:  
            # code  
        except:  
            # code  
        except:  
            # code
```

# Solution

```
def divider():  
    while True:  
        nums = input('Enter 2 digits (format: x y):')  
        (x,y) = nums.split()  
        try:  
            x = int(x)  
            y = int(y)  
            res = x/y  
            print(res)  
        except ZeroDivisionError:  
            print('ZeroDivisionError')  
            print('try again without 0')  
        except ValueError:  
            print('ValueError')  
            print('try again without letters')  
        except:  
            print("Something went wrong!")  
            raise
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```



# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```



# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```



# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Syntax of try/except/finally

```
try:
    <indented code block>
except <ExceptionType>:
    <exception handler block>
finally:
    <clean up code block>
<non-indented statement>
```

**finally**-ն միշտ կատարվում է անկախ ամեն ինչից

## Normal Flow

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Handled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# continue
```

## Unhandled Exception

```
try:
    # process
except exception:
    # handle
finally:
    # cleanup
# throw call stack
```

# Example try/except/finally

```
def read_data(filename):  
    lines = []  
    infile = None  
    try:  
        infile = open(filename, encoding="utf8")  
        for line in infile:  
            lines.append(line)  
    except (IOError, OSError) as err:  
        print(err)  
        return []  
    finally:  
        if infile is not None:  
            infile.close()  
    return lines
```

old style => use **with** statement

# Example try/except + with

```
def read_data_right(filename):  
    lines = []  
    try:  
        with open(filename, encoding="utf8") as infile:  
            for line in infile:  
                lines.append(line)  
    except (IOError, OSError) as err:  
        print(err)  
        return []  
    return lines
```

# Example try/except + with

```
def read_data_right(filename):  
    lines = []  
    try:  
        with open(filename, encoding="utf8") as infile:  
            for line in infile:  
                lines.append(line)  
    except (IOError, OSError) as err:  
        print(err)  
        return []  
    return lines
```

# Example try/except + with

```
def read_data_right(filename):  
    lines = []  
    try:  
        with open(filename, encoding="utf8") as infile:  
            for line in infile:  
                lines.append(line)  
    except (IOError, OSError) as err:  
        print(err)  
        return []  
    return lines
```



# Example try/except + with

```
def read_data_right(filename):  
    lines = []  
    try:  
        with open(filename, encoding="utf8") as infile:  
            for line in infile:  
                lines.append(line)  
    except (IOError, OSError) as err:  
        print(err)  
        return []  
    return lines
```

# Exercise 2

Debug արեք `read_data(filename)` function-ը

- օգտագործեք `age.txt` file-ը
- օգտագործեք PyCharm-ի debugger-ը

# Default arguments

- Function-ի argument-ները կարող են ունենալ default value-ներ
- Այս argument-ները պարտադիր չեն function call-ի ժամանակ

```
def f(a, b=7, c="Hi"):  
    return a + b
```

```
>>> f(5,6)
```

```
11
```

```
>>> f(5,5,"dummy")
```


```
10
```

```
>>> f(5)
```

```
12
```

# Default arguments


Default values



The diagram consists of a horizontal red line with two vertical red arrows pointing downwards from its ends. The left arrow points to the default value 'Hallo' for the 'greeting' parameter, and the right arrow points to the default value '!' for the 'punct' parameter.

```
def salute(name, greeting='Hallo', punct='!'):
    return greeting + ' ' + name + punct
```

# Default arguments

 Default values

```
def salute(name, greeting='Hallo', punct='!'):
    return greeting + ' ' + name + punct
```

**greeting** and **punct** : keyword arguments

# Unpack Syntactic Rule

```
def f(a, b):  
    print(a, b)  
  
>>> f('apple', 'appricot')  
apple appricot  
>>> l = ['apple', 'appricot']  
>>> f(*l)  
apple appricot  
>>> f(1)  
TypeError: f() missing 1 required  
positional argument: 'b'  
>>>
```

\* unpack "operator"/syntactic rule:

վերածույթ է collection/sequence-ը positional argument-ներին

# Unpack Syntactic Rule

```
def f(a='apple', b='apricot'):  
    print(a, b)  
  
>>> f(b='lemon', a='orange')  
orange lemon  
>>> d = {'a': 'orange', 'b': 'lemon'}  
>>> f(**d)  
orange lemon
```

**\*\*** unpack "operator"/syntactic rule for dictionaries:  
վերածույմ է dict-ը keyword argument-ների

# args & kwargs

Function-ները կարող են ընդունել փոփոխական թվով **argument**-ներ (variable number of arguments)

```
def f(*args):  
    print("args is ", args)  
    print("type of args is ", type(args))  
  
>>> f(1, 2, 3, 4)  
args is (1, 2, 3, 4)  
type of args is <class 'tuple'>
```



# args & kwargs

Function-ները կարող են ընդունել փոփոխական թվով **argument**-ներ (variable number of arguments)

```
def f(*args):  
    print("args is ", args)  
    print("type of args is ", type(args))  
  
>>> f(1, 2, 3, 4)  
args is (1, 2, 3, 4)  
type of args is <class 'tuple'>  
def f(*args):  
    for arg in args:  
        print(arg, end=' ')  
  
>>> f(1, 2, 3, 4)  
1 2 3 4
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)  
  
>>> f(1,2,3,4)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1,2,3,4)  
a = 1  
args = (2, 3, 4)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1,2,3,4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)  
TypeError: f() missing 1 required  
keyword-only argument: 'b'
```



# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)  
TypeError: f() missing 1 required  
keyword-only argument: 'b'  
def f(a, b, *args):
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)  
TypeError: f() missing 1 required  
keyword-only argument: 'b'
```

```
def f(a, b, *args):  
    print("a = ", a)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)  
TypeError: f() missing 1 required  
keyword-only argument: 'b'
```

```
def f(a, b, *args):  
    print("a = ", a)  
    print("args = ", args)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)  
TypeError: f() missing 1 required  
keyword-only argument: 'b'
```

```
def f(a, b, *args):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)
```

# args & kwargs

```
def f(a, *args):  
    print("a = ", a)  
    print("args = ", args)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (2, 3, 4)
```

```
def f(a, *args, b):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)  
TypeError: f() missing 1 required  
keyword-only argument: 'b'
```

```
def f(a, b, *args):  
    print("a = ", a)  
    print("args = ", args)  
    print("b = ", b)
```

```
>>> f(1, 2, 3, 4)  
a = 1  
args = (3, 4)  
b = 2  
>>>
```

# Exercise 3

Գրեք function `sum_of_powers`,

- որը ընդունում է տարբեր քանակությամբ թվեր և
- և հաշվում է գումարը հետեյալ կերպ

```
>>> sum_of_powers(1,2)
```

```
3
```

```
>>> sum_of_powers(1,2,power=2)
```

```
5
```

```
>>> sum_of_powers(1,2,power=3)
```

```
9
```

# Solution 3

```
def sum_of_powers(*args, power=1):  
    #accumulator  
    #for each  
        #do sum of powers  
    #return result
```

# Solution 3

```
def sum_of_powers(*args, power=1):  
    result = 0  
    for arg in args:  
        result += arg ** power  
    return result
```



# args & kwargs

Function-ները կարող են ընդունել փոփոխական թվով  
**keyword argument**-ներ

(variable number of keyword arguments)

```
def f(**kwargs):  
    # kwargs is a dict of all keyword arguments  
    for (k,v) in kwargs.items():  
        print('{}={}'.format(k,v))  
    print('type of kwargs is ', type(kwargs))  
  
>>> f(a=1,b=2,c=3)
```

# args & kwargs

Function-ները կարող են ընդունել փոփոխական թվով  
**keyword argument**-ներ

(variable number of keyword arguments)

```
def f(**kwargs):  
    # kwargs is a dict of all keyword arguments  
    for (k,v) in kwargs.items():  
        print('{}={}'.format(k,v))  
    print('type of kwargs is ', type(kwargs))
```

```
>>> f(a=1,b=2,c=3)  
b=2  
a=1  
c=3
```

# args & kwargs

Function-ները կարող են ընդունել փոփոխական թվով  
**keyword argument**-ներ

(variable number of keyword arguments)

```
def f(**kwargs):  
    # kwargs is a dict of all keyword arguments  
    for (k,v) in kwargs.items():  
        print('{}={}'.format(k,v))  
    print('type of kwargs is ', type(kwargs))  
  
>>> f(a=1,b=2,c=3)  
b=2  
a=1  
c=3  
type of kwargs is <class 'dict'>  
>>>
```

# args & kwargs

Function-ները կարող են ընդունել

# args & kwargs

Function-ները կարող են ընդունել

- փոփոխական թվով argument-ներ  
    \*args -ով

# args & kwargs

Function-ները կարող են ընդունել

- փոփոխական թվով keyword argument-ներ  
**\*\*kwargs**-ով

# args & kwargs

Function-ները կարող են ընդունել

- փոփոխական թվով argument-ներ  
    \*args -ով
- փոփոխական թվով keyword argument-ներ  
    \*\*kwargs-ով

# args & kwargs

Function-ները կարող են ընդունել

- փոփոխական թվով argument-ներ  
    \*args -ով
- փոփոխական թվով keyword argument-ներ  
    \*\*kwargs-ով

```
def functionName(fargs,*args,**kwargs):  
    <indented code block>
```



# args & kwargs

Function-ները կարող են ընդունել

- փոփոխական թվով argument-ներ  
    \*args -ով
- փոփոխական թվով keyword argument-ներ  
    \*\*kwargs-ով

```
def functionName(fargs,*args,**kwargs):  
    <indented code block>
```

fargs : arguments with fixed position

# args & kwargs

```
def print_args(*args, **kwargs):  
    for i, arg in enumerate(args):  
        print("positional argument {0} = {1}"  
              .format(i, arg))  
    for key in kwargs:  
        print("keyword argument {0} = {1}"  
              .format(key, kwargs[key]))  
  
>>> print_args(0,1,2,d=3,e=4,f=5)
```

# args & kwargs

```
def print_args(*args, **kwargs):  
    for i, arg in enumerate(args):  
        print("positional argument {0} = {1}"  
              .format(i, arg))  
    for key in kwargs:  
        print("keyword argument {0} = {1}"  
              .format(key, kwargs[key]))
```

```
>>> print_args(0,1,2,d=3,e=4,f=5)
```

```
positional argument 0 = 0
```

```
positional argument 1 = 1
```

```
positional argument 2 = 2
```

```
keyword argument f = 5
```

```
keyword argument e = 4
```

```
keyword argument d = 3
```

# Exercise 4

Գրեք function sum,

- որը ընդունում է տարբեր քանակությամբ թվեր և
- հաշվում այդ թվերի գումարը և return անում
- կարող է ընդունել լրացուցիչ neg default parameter

```
>>> sum(1, 2, 3)
```

```
6
```

```
>>> sum(1, 2, 3, neg=True)
```

```
-6
```

```
>>> sum(1, 2, 3, neg=False)
```

```
6
```

# Solution 4

```
def sum(*args, **kwargs):  
    s = 0  
    for i in args:  
        s = s + i  
    if kwargs.get("neg", False):  
        s = -s  
    return s
```

# Solution 4

```
def sum(*args, **kwargs):  
    s = 0  
    for i in args:  
        s = s + i  
    if kwargs.get("neg", False):  
        s = -s  
    return s
```

```
def sum(*args, neg=False):  
    s = 0  
    for i in args:  
        s = s + i  
    if neg:  
        s = -s  
    return s
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
```



# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!".join(words)
'to!_be!_or!_not!_to!_be'
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

```
>>> s = ' hallo !!'
>>> s.strip(' !')
'hallo'
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

```
>>> s = ' hallo !!'
>>> s.strip(' !')
'hallo'
>>> s.lstrip(' ')
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

```
>>> s = ' hallo !!'
>>> s.strip(' !')
'hallo'
>>> s.lstrip(' ')
'hallo !!'
```



# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

```
>>> s = ' hallo !!'
>>> s.strip(' !')
'hallo'
>>> s.lstrip(' ')
'hallo !!'
>>> s = 'abc'
>>> s.isalpha()
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

```
>>> s = ' hallo !!'
>>> s.strip(' !')
'hallo'
>>> s.lstrip(' ')
'hallo !!'
>>> s = 'abc'
>>> s.isalpha()
True
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

```
>>> s = ' hallo !!'
>>> s.strip(' !')
'hallo'
>>> s.lstrip(' ')
'hallo !!'
>>> s = 'abc'
>>> s.isalpha()
True
>>> s.isdigit()
```

# Strings revisited

```
>>> s = 'to be or not to be'
>>> words = s.split(' ')
>>> words
['to', 'be', 'or', 'not', 'to', 'be']
>>> " ".join(words)
'to be or not to be'
>>> "_!_".join(words)
'to!_be!_or!_not!_to!_be'
>>> lst = [1,2,3,4]
>>> "-".join([str(x) for x in lst])
'1-2-3-4'
```

```
>>> s = ' hallo !!'
>>> s.strip(' !')
'hallo'
>>> s.lstrip(' ')
'hallo !!'
>>> s = 'abc'
>>> s.isalpha()
True
>>> s.isdigit()
False
```

# Tuples & zip function

---

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
```

# Tuples & zip function

---

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
```

# Tuples & zip function

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
>>> a
'a'
>>> b
'b'
>>> c
'c'
```

# Tuples & zip function

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
>>> a
'a'
>>> b
'b'
>>> c
'c'
```

```
>>> zip(u,v)
<zip object at 0x0232CF58>
```



# Tuples & zip function

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
>>> a
'a'
>>> b
'b'
>>> c
'c'
```

```
>>> zip(u,v)
<zip object at 0x0232CF58>
>>> list(zip(u,v))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

# Tuples & zip function

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
>>> a
'a'
>>> b
'b'
>>> c
'c'
```

```
>>> zip(u,v)
<zip object at 0x0232CF58>
>>> list(zip(u,v))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(zip(u,v,w))
```

# Tuples & zip function

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
>>> a
'a'
>>> b
'b'
>>> c
'c'
```

```
>>> zip(u,v)
<zip object at 0x0232CF58>
>>> list(zip(u,v))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(zip(u,v,w))
[(1, 'a', 4), (2, 'b', 5), (3, 'c', 6)]
```

# Tuples & zip function

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
>>> a
'a'
>>> b
'b'
>>> c
'c'
```

```
>>> zip(u,v)
<zip object at 0x0232CF58>
>>> list(zip(u,v))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(zip(u,v,w))
[(1, 'a', 4), (2, 'b', 5), (3, 'c', 6)]
>>> e = dict(zip(u,v))
>>> e
```

# Tuples & zip function

```
>>> u = [1,2,3]
>>> v = ('a', 'b', 'c')
>>> w = [4,5,6]
>>> (a,b,c) = v
>>> a
'a'
>>> b
'b'
>>> c
'c'
```

```
>>> zip(u,v)
<zip object at 0x0232CF58>
>>> list(zip(u,v))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(zip(u,v,w))
[(1, 'a', 4), (2, 'b', 5), (3, 'c', 6)]
>>> e = dict(zip(u,v))
>>> e
{1: 'a', 2: 'b', 3: 'c'}
```

# References

1. Franek. “CS 1MD3 Introduction to Programming.” Accessed July 8, 2014.
2. “The Python Tutorial — Python 3.4.1 Documentation.” Accessed August 2, 2014.  
<https://docs.python.org/3.4/tutorial/>.