Python for Data Scientist
L4: Functions, decomposition & abstraction

How to write a code?

So far…

- Know python syntax
- know how to write a code for each problem
- each code is a sequence of instructions

How to write a code?

Problems with this approach

- easy to understand and write for small-scale problems
- messy for larger problems
- hard to keep track of details

Good code

- more code not necessarily a good thing
- measure good programmers by the amount of functionality
- introduce **functions**
- mechanism to achieve **decomposition** and **abstraction**

Decomposition

Divide code into **modules**

- are **self-contained**
- used to **break up** code
- intended to be **reusable**
- keep code **organized**
- keep code coherent

How to achieve decomposition?

→ **functions**

Abstraction

Piece of code as a **black box**

- cannot see details
- do not need/want to see details
- hide tedious coding details

How to achieve abstraction?

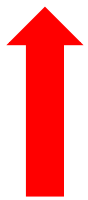
→ **function specifications** or **docstrings**

Functions

- A function is a reusable block of code designed to perform a certain task which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Functions

```
def multiplication( x,y ):
    """
    :param x: first parameter
    :param y: second parameter
    :return: the result for multiplying x and y
    """
    rslt = x * y
    return rslt

print(multiplication(5,9))
```



Calling the function using its name and values for parameters

function characteristics:

- Keyword : def
- has a name
- has parameters(0 or more)
- has a docstring(optional but recommended)
- has a body
- Returns something

Functions: Variables

Formal parameter gets bound to the value of actual parameter when function is called

```
def multiplication( x,y ):
    """
    :param x: first parameter
    :param y: second parameter
    :return: the result for multiplying x
       and y
    """
    return x*y

x= 5
y= 9
z= multiplication( x,y )
```

Formal parameters

Actual parameters

Main program code:

- Initializes the variables
- Call the function
- Assigns return of the function to variable z

Functions: Variables

```
def f(x):
    """
    :return: the result for multiplying
       x by itself
    """
    x *= x
    return x

x= 4
y=f(x)
print(y)
```

f scope:
x -> 4

Global scope:
x -> 4

f -> code for multiplying a number by itself

y ->

Functions: Variables

```
def f(x):
    """
    :return: the result for multiplying
       x by itself
    """
    x *= x
    return x

x= 4
y=f(x)
print(y)
```

f scope:
x -> 16

Global scope:
x -> 4

f -> code for multiplying a number by itself

y -> 16

Functions: Return

- only **one** return executed inside a function
- code inside function but after return statement not executed
- has a value associated with it, **given to function caller**
- Python returns the value **None, if no return is given**

Functions as arguments (Poll)

```
def f_a():
    print ('inside f_a’)

def f_b(y):
    print ('inside f_b')
    return y

def f_c(z):
    print ('inside f_c')
    return z()

print (f_a())
print (5 + f_b(2))
print (f_c(f_a))
```

arguments can take on any type, even functions

: calling f_a with no parameters

: calling f_b with one parameter

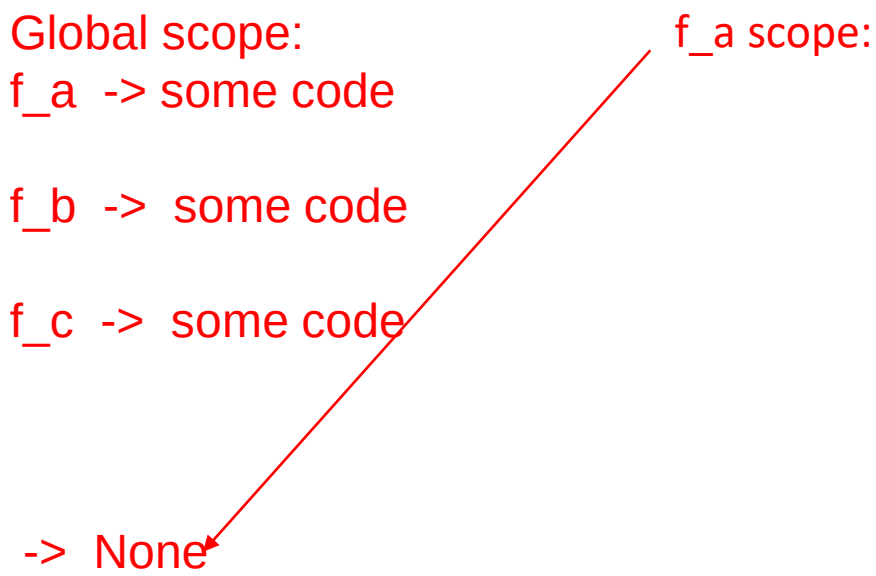
: calling f_c with one parameter which is a function

What will be the result for each print?

A	B	C
None 7 7	7 None 7	None 7 None

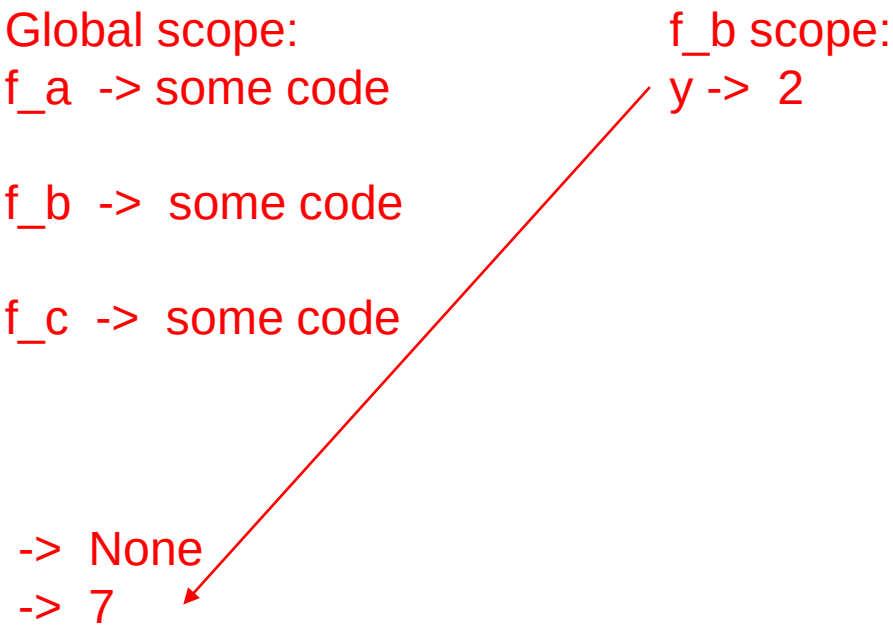
Functions as arguments

```
def f_a():
    print ('inside f_a’)

def f_b(y):
    print ('inside f_b')
    return y

def f_c(z):
    print ('inside f_c')
    return z()

print (f_a())
print (5 + f_b(2))
print (f_c(f_a))
```



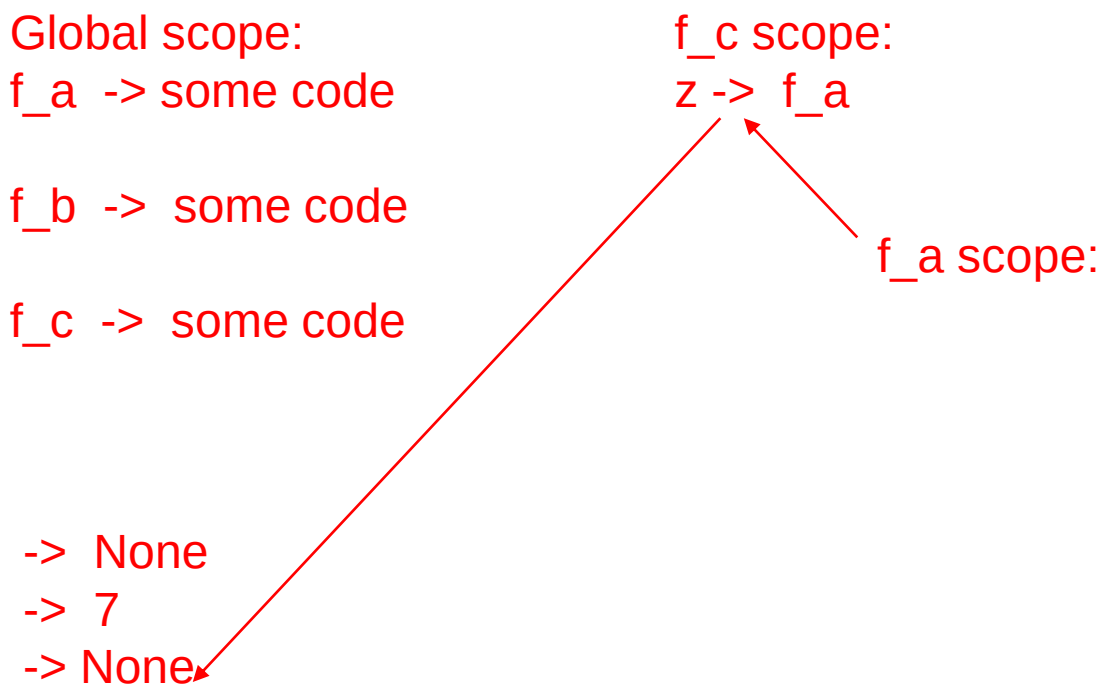
Functions as arguments

```
def f_a():
    print ('inside f_a’)

def f_b(y):
    print ('inside f_b')
    return y

def f_c(z):
    print ('inside f_c')
    return z()

print (f_a())
print (5 + f_b(2))
print (f_c(f_a))
```



Functions as arguments

```
def f_a():
    print ('inside f_a’)

def f_b(y):
    print ('inside f_b')
    return y

def f_c(z):
    print ('inside f_c')
    return z()

print (f_a())
print (5 + f_b(2))
print (f_c(f_a))
```



Call by reference or call by value

In C, Java and some other language, passing a value to a function can be :

- by value : the function receives a copy of the argument objects passed to it by the caller, stored in a new location in memory.
- by reference : the function receives reference to the argument objects passed to it by the caller, both pointing to the same memory location.

Neither of these two concepts are applicable in Python
→ the **values are sent to functions by means of object reference**

Pass-by-object-reference in Python

In Python, values are passed to function by object reference:

- if object is immutable: than the modified value is not available outside the function.
- if object is mutable: than modified value is available outside the function.

Pass-by-object-reference in Python

```
def value(x):
    x = 10
    print(x, id(x))

x = 0
value(x)


print(x, id(x))
```

x = 10
id = 140709803749936

x = 0
id = 140709803749616

A new object is created in the memory because integer objects are immutable

Pass-by-object-reference in Python

```
def value(l):
    l.append(4)
    print(l, id(l))

l = [1,2,3]
value(l)


print(l, id(l))
```

l = [1,2,3,4]
id = 2040455189064

l = [1,2,3,4]
id = 2040455189064

A new object is not created in the memory because list objects are mutable

Local and global variables

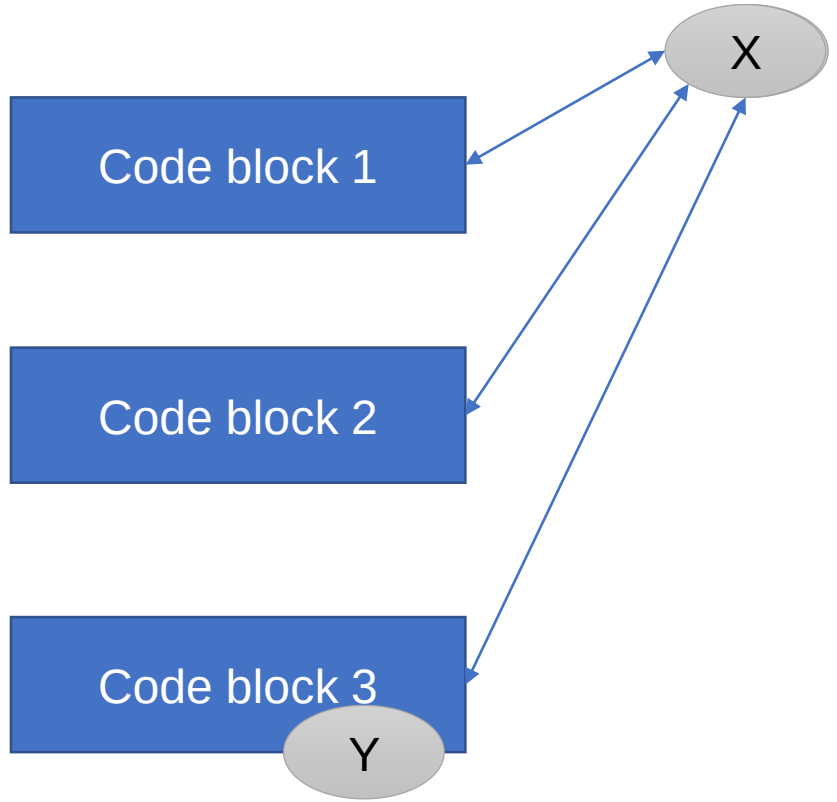
```
def f_c():
    x *= x


x = 10
f_c()
print(x)
```

UnboundLocalError: local variable 'x' referenced before assignment

Inside a function, **cannot modify** a variable defined outside

Local and global variables

- A global variable can be reached and modified anywhere in the code
- Local variables can only be reached in their scope.



Local and global variables

```
def f_c():
    global x
    x *= x

x = 10
f_c()
print(x)
```

- The global variable x can be used all throughout the program, inside functions or outside.
- A global variable can be modified inside a function and change for the entire program

Functions: Example

What will be the output of this program?

```
def f_a(x):
    def f_b():
        x = 'newValue'
    x *= x
    print('f_a: x =', x)
    f_b()
    return x

x = 3
z = f_a(x)
print(z)
```

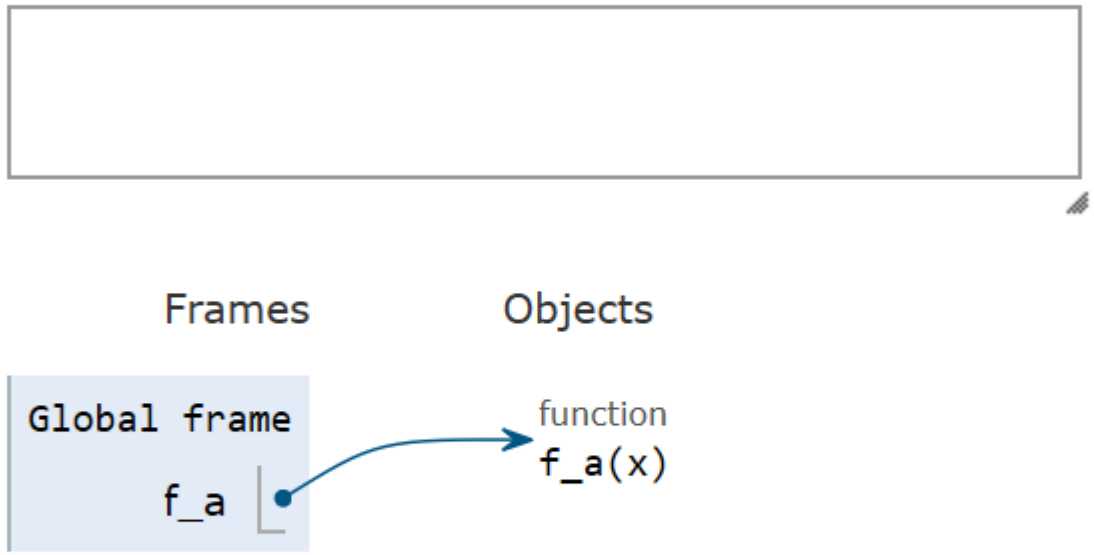
Functions: example

We will use http://www.pythontutor.com/ to solve this problem

Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Print output (drag lower right corner to resize)

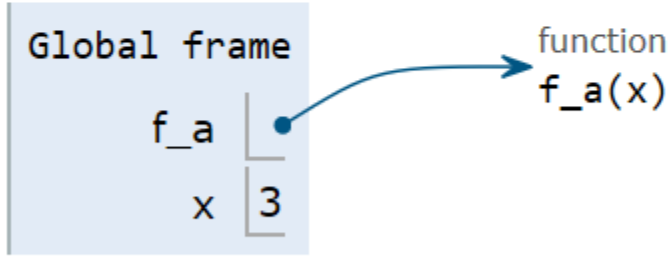


Functions: example

Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Print output (drag lower right corner to resize)



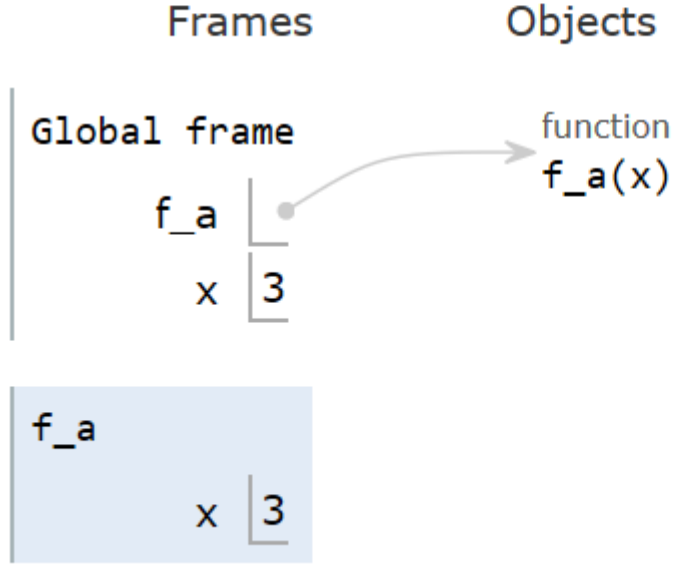
Functions: example

Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)



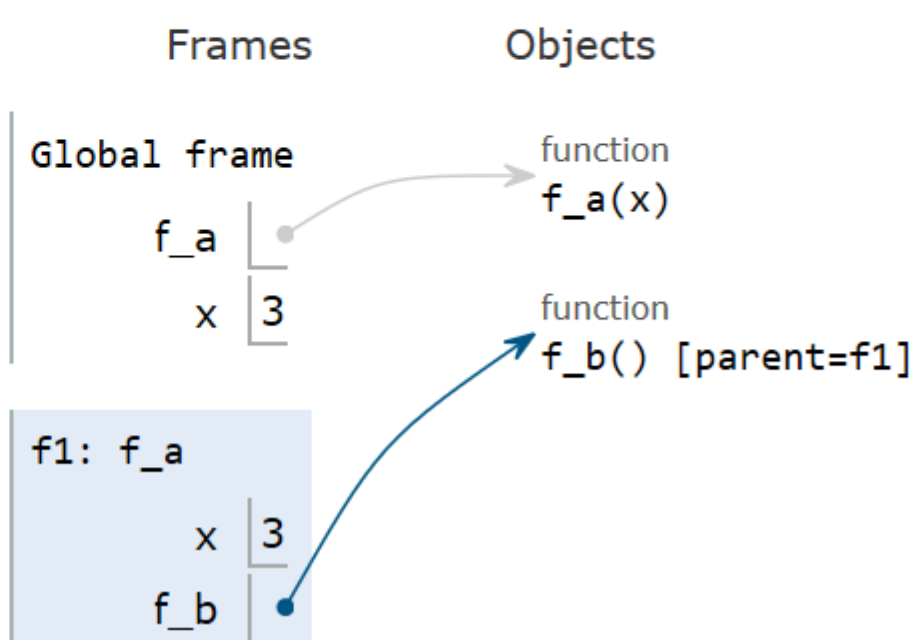
Functions: example

Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)



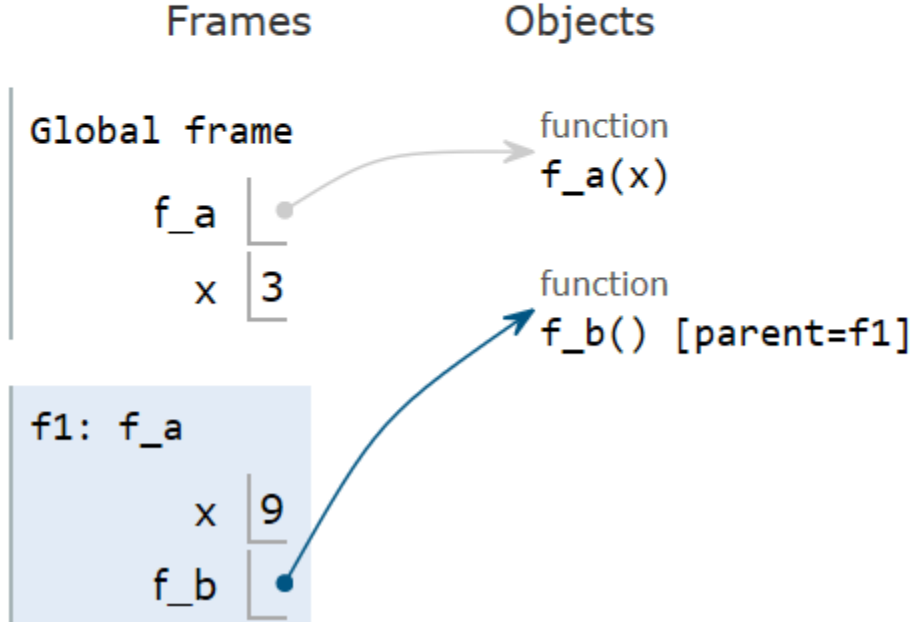
Functions: example

Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)



Functions: example

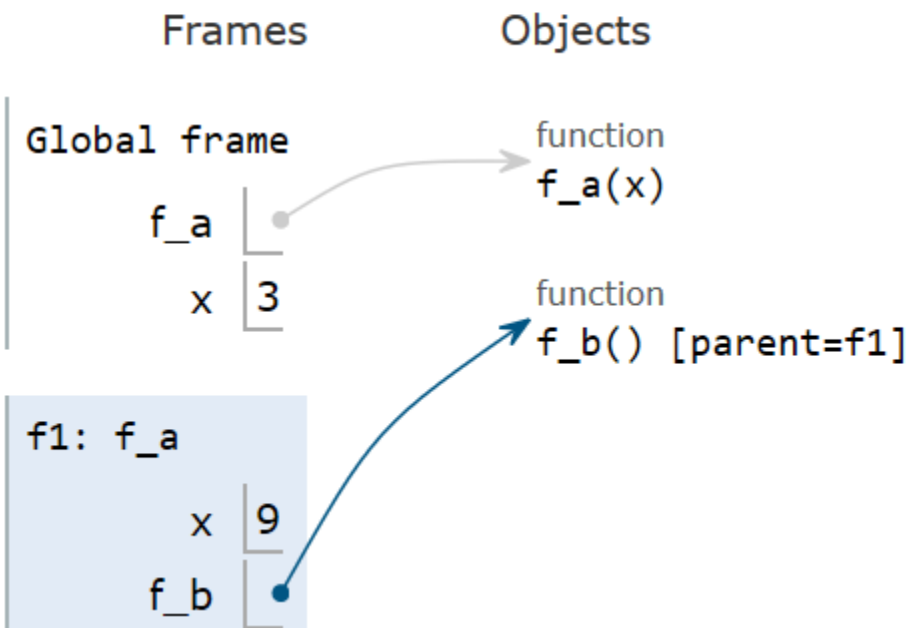
Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)

```
f_a: x = 9
```



Functions: example

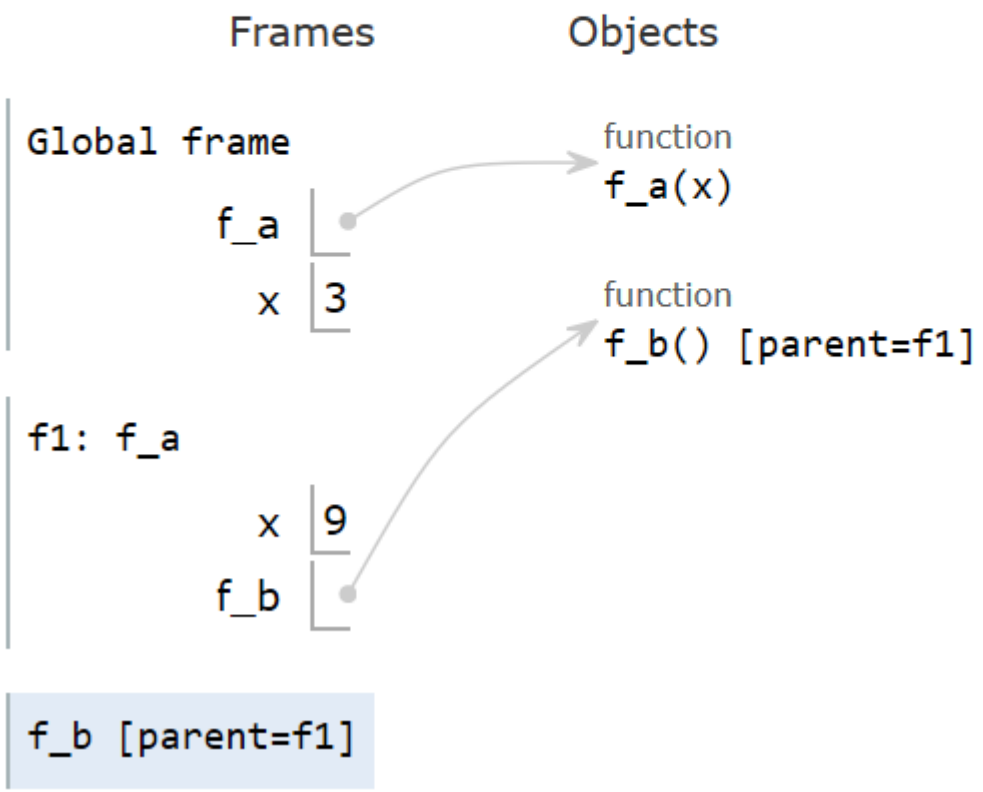
Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)

```
f_a: x = 9
```



f_b [parent=f1]

Functions: example

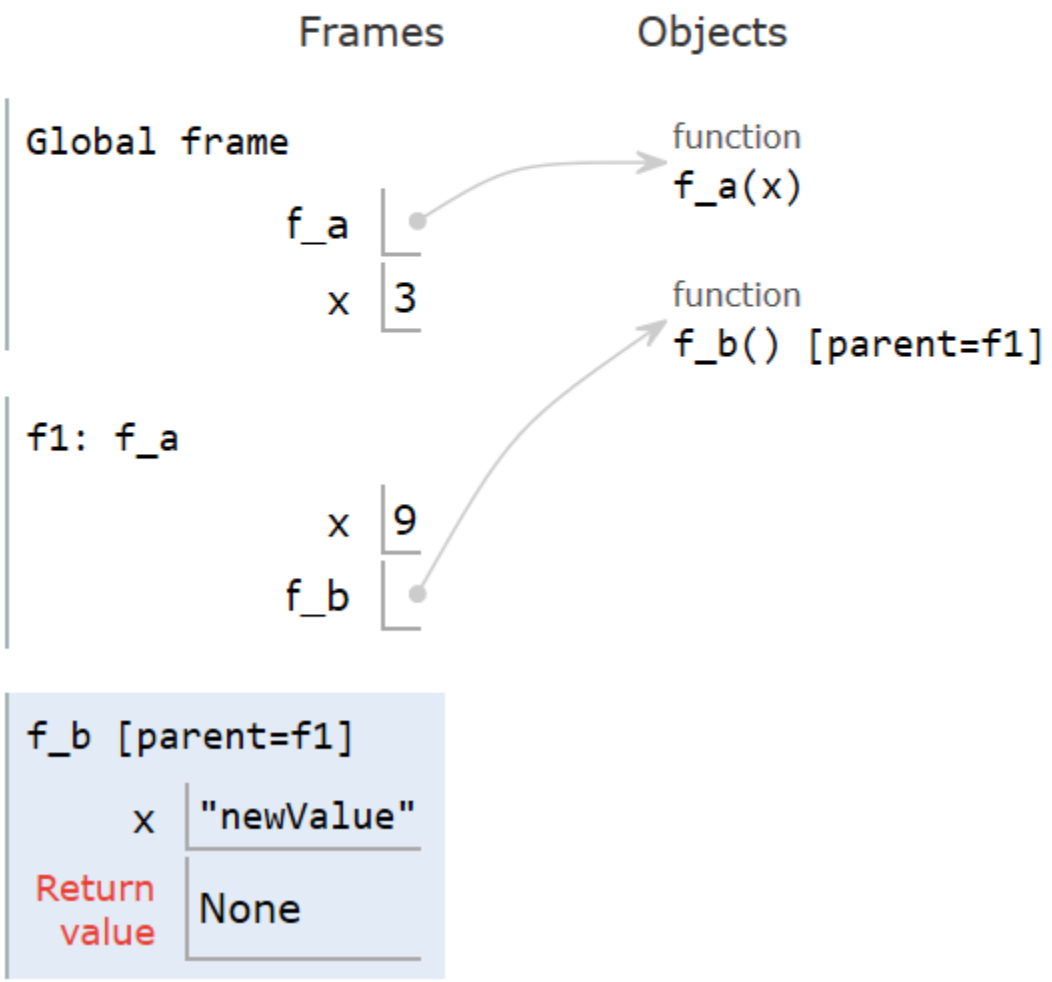
Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)

```
f_a: x = 9
```



Functions: example

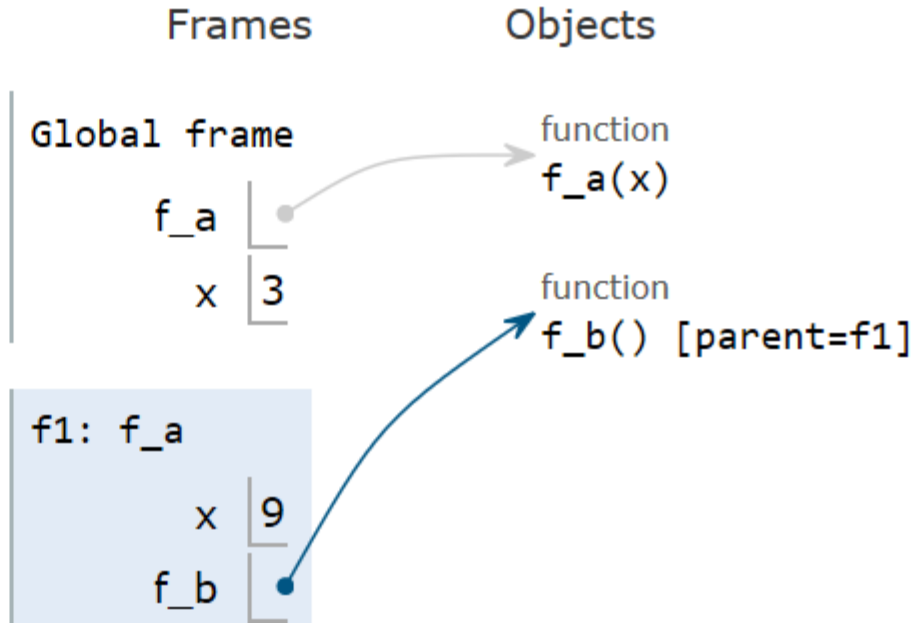
Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)

```
f_a: x = 9
```



Functions: example

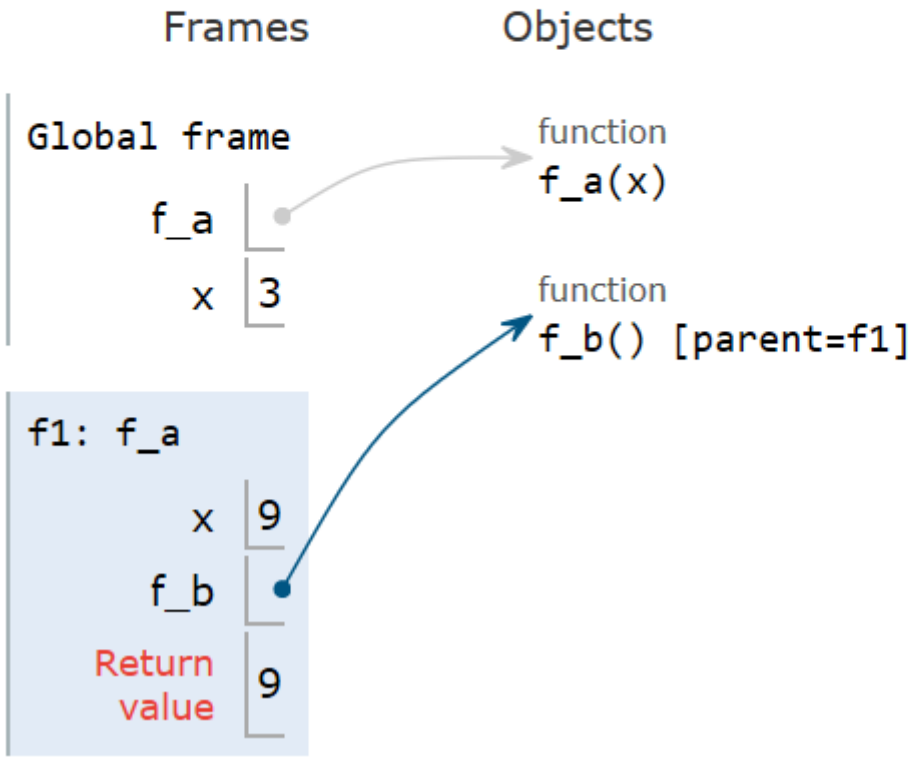
Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)

```
f_a: x = 9
```



Functions: example

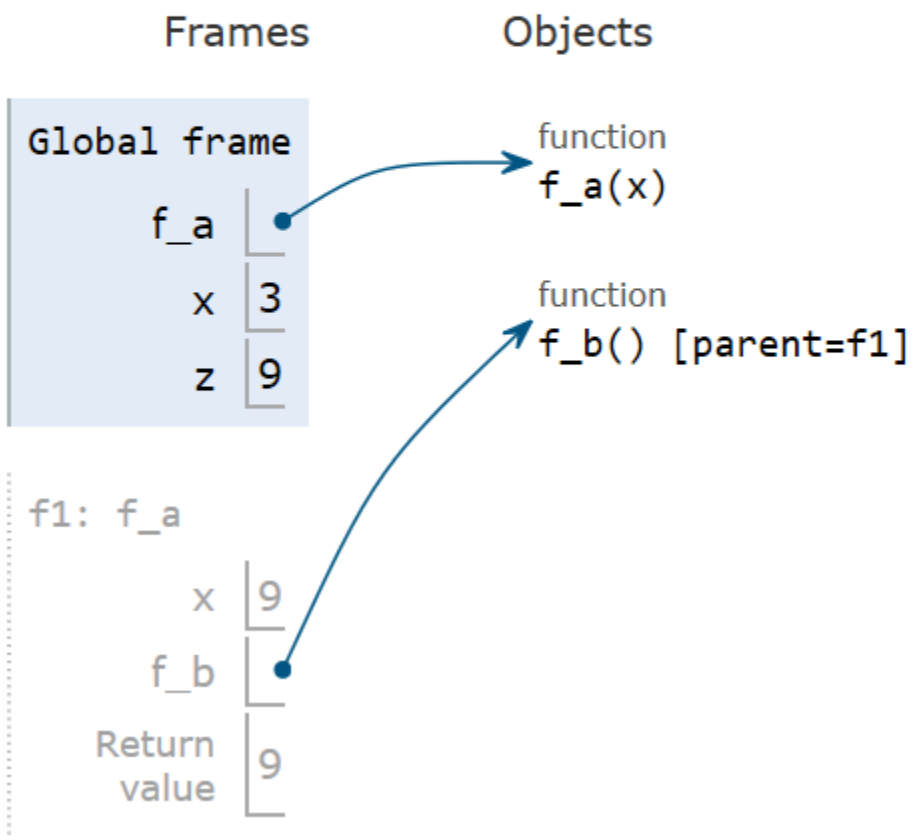
Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)

```
f_a: x = 9
```



Functions: example

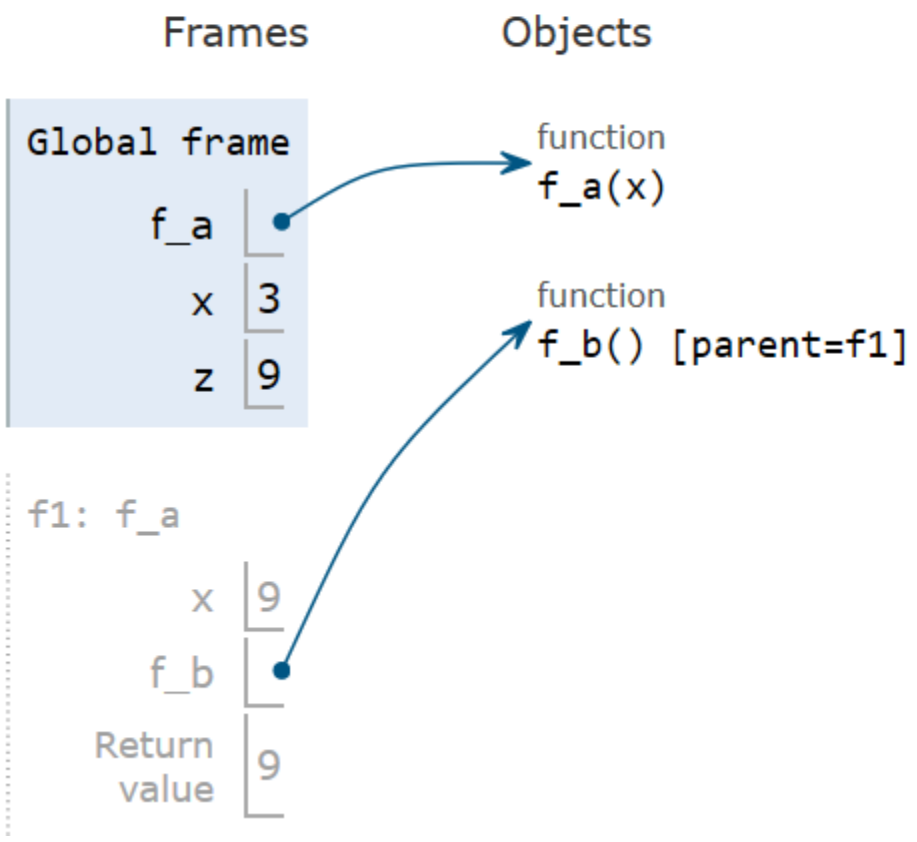
Python 3.6

```
1  def f_a(x):
2      def f_b():
3          x = 'newValue'
4      x *= x
5      print('f_a: x =', x)
6      f_b()
7      return x
8
9  x = 3
10 z = f_a(x)
11 print(z)
```

Edit this code

Print output (drag lower right corner to resize)

```
f_a: x = 9
9
```



Functions with default values

- Default value will be substituted if an appropriate actual argument is passed when the function is called.
- If the actual argument is not provided, the default value will be used inside the function.

Functions with default values

```
def Hello(name='Guest'):
    print ("Hello dear " + name)
    return

Hello()
name = "Alex"
Hello(name)
```

Hello dear Guest
Hello dear Alex

Functions with command line arguments

- They are arguments which are added after the function call in the same line.
- If you call a Python script from a shell, the arguments are placed after the script name. The arguments are separated by spaces.
- Inside the script these arguments are accessible through the list variable sys.argv.

Functions with command line arguments

```
import sys

for eachArg in sys.argv:
        print(eachArg)
```

```
C:\Users\naili\PycharmProjects\PythonCourse>python testArg.py Python for Data Scientist
testArg.py
Python
for
Data
Scientist
```

Functions with variable length of parameters

- A function with an arbitrary number of arguments (called a variadic function): is a function of indefinite arity.

- The asterisk "*" is used to define a variable number of arguments.

Functions with variable length of parameters

```
def varfun(x):
    print(x)
varfun("Python", "for", "data", "scientist")
```

TypeError: varpafu() takes 1 positional argument but 4 were given

```
def varfun(*x):
    print(x)
varfun("Python", "for", "data", "scientist")
```

('Python', 'for', 'data', 'scientist')

Functions with * in the function call

→ An argument will be unpacked : the elements of the list or tuple are singularized

```
def f(x,y,z):
    print(x,y,z)

p = (0,1,2)
f(*p)
```

```
0 1 2
```

Functions with ** in the function call

```
def f(a,b,x,y):
    print(a,b,x,y)

t = (47,11)
d = {'x':'extract','y':'yes'}
f(*t, **d)
```

47 11 extract yes

Generators

- A generator : special type of function which does not return a single value, instead it returns an iterator object with a sequence of values.

- In a generator function, a yield statement is used rather than a return statement.

Generators

```
def myGenerator():
    print('First element')
    yield 10

    print('Second element')
    yield 20

    print('Third element')
    yield 30

gen = myGenerator()
x =next(gen)
print(x)

x =next(gen)
print(x)

x =next(gen)
print(x)
```

```
First element
10
Second element
20
Third element
30
```

Generators

- yield : returns a value and pauses the execution while maintaining the internal states
- return : returns a value and terminates the execution of the function.

Advantage of generators :
- Elements are generated dynamically.
- The next item is generated only after the first is consumed, it is more memory efficient than the iterator.

Generators

```
def myGenerator():
    print('First element')
    yield 10
    return

    print('Second element')
    yield 20

    print('Third element')
    yield 30

gen = myGenerator()
x =next(gen)
print(x)

x =next(gen)
print(x)

x =next(gen)
print(x)
```

x =next(gen)
StopIteration
First element
10

Generators: Exception Handling

- A program suddenly terminates if it encounters an exception (wrong input, …) → may cause damage to system resources.

→ Solution: the exceptions should be properly handled so that an sudden termination of the program is prevented.

Generators: Exception Handling

Python uses **try** and **except** keywords to handle exceptions:

```
try :
        #statements in try block
```

```
except (type of exception):
        #executed when error in try block
```

Generators: Exception Handling

```
def loopGenerator(x):
    for i in range(x):
        yield i

it=loopGenerator(6)

while True:
    try:
        print ("Received on next(): ",next(it))
    except StopIteration:
        break
```

try..except block will handle the StopIteration error. It will break the while loop once it catches the StopIteration error.

Defined functions

- Python includes many built-in functions.

- These functions perform a predefined task and can be called upon in any program.

Lambda functions

- a small anonymous function.

- can take any number of arguments, but can only have one expression.

lambda <*arguments*> : <*expression*>

Lambda functions

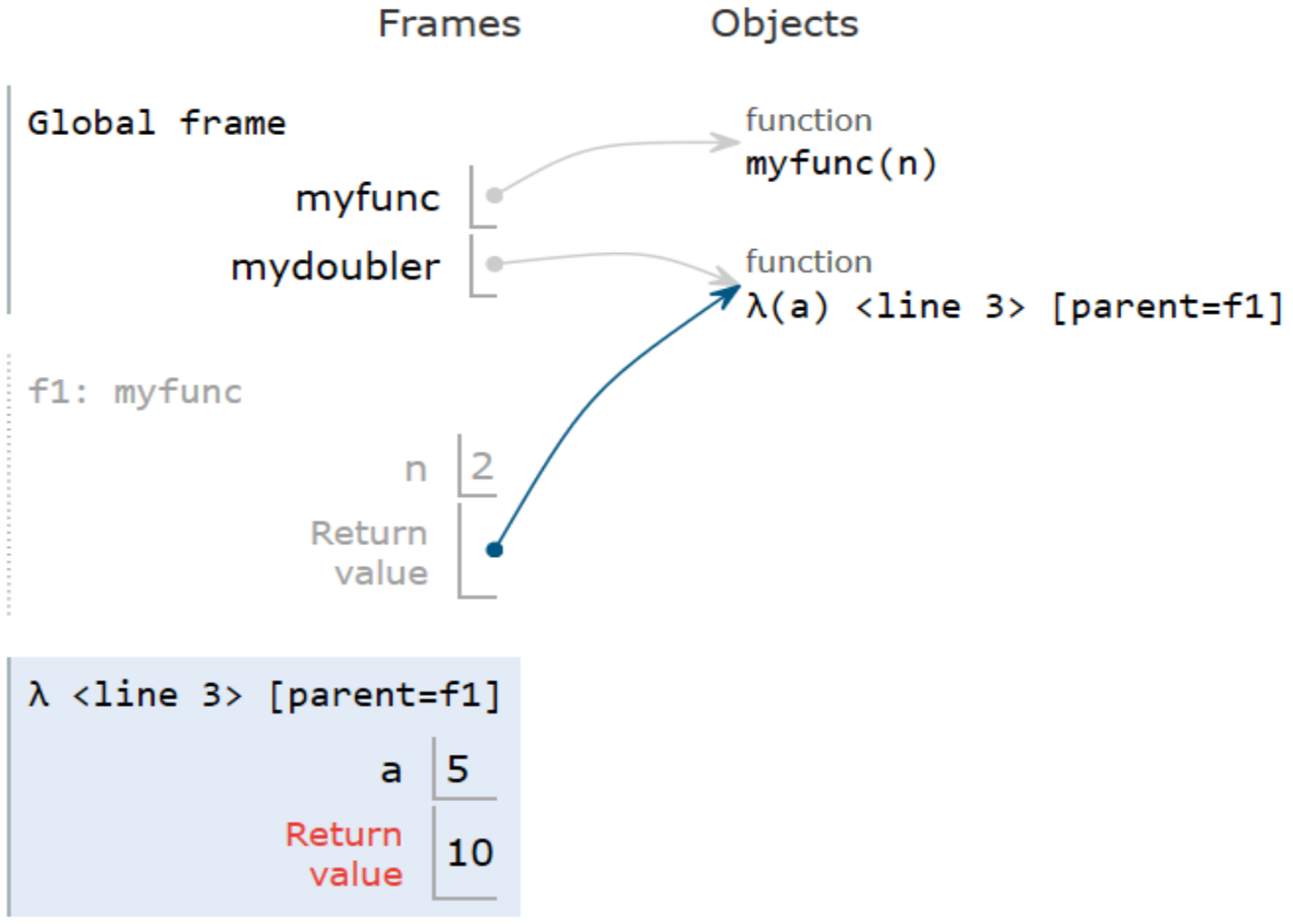
```
x = lambda a, b : a * b
print(x(2, 3))
```

Use lambda functions when an anonymous function is required for a short period of time.

Lambda functions

Lambda functions are mostly used as an anonymous function inside another function.

```
def myfunc(n):
    print(n)
    return lambda a : a * n


mydoubler = myfunc(2)

print(mydoubler(5))
```



Lambda functions (Poll)

What will be the output of this code:

```
func = lambda x: return x
print(func(2))
```

A- 2

B- 2.0

C- none of the above

A lambda function can’t contain the return statement. In a lambda function, statements like return, pass will raise a SyntaxError exception.

Lambda functions (Poll)

What will be the output of this code:

```
print((lambda x: (x + 3) * 5 / 2)(3))
```

A- syntaxError

B- 0

C- 15.0

Map function

- Calls the specified function for each item of an iterable
- returns a list of results.

```
def square(x):
    return x*x

numbers=[1, 2, 3, 4]
sqrList=map(square, numbers)

while True:
    try:
        print ("Received on next(): ",next(sqrList))
    except StopIteration:
        break
```

```
Received on next():  1
Received on next():  4
Received on next():  9
Received on next():  16
```

Map with Lambda Expression

```
def square(x):
    return x*x

numbers=[1, 2, 3, 4]
sqrList=map(square, numbers)
```

```
sqrList = map(lambda x: x*x, [1, 2, 3, 4])
```

```
while True:
    try:
        print ("Received on next(): ",next(sqrList))
    except StopIteration:
        break
```

Map with Built-in Function

```
l1 = [1, 2, 3, 4, 5]
l2 = [6, 7, 8, 9, 10]
powers=list(map(pow, l1, l2))

print(powers)
```

Filter function

Calls the specified function which returns Boolean for each item of the specified iterable

```
def prime(x):
    for d in range(2,x):
        if x % d ==0:
            return False
        else:
            return True

rslt=filter(prime, range(15))
print ('Prime numbers:', list(rslt))
```

Prime numbers: [3, 5, 7, 9, 11, 13]

Reduce function

- Receives two arguments, a function and an iterable object
- Returns a single value.

```
import functools
def multiplication(x,y):
    print("x=",x," y=",y)
    return x*y

fact=functools.reduce(multiplication, range(1, 5))
print ('Factorial of 4: ', fact)
```

```
x= 1  y= 2
x= 2  y= 3
x= 6  y= 4
Factorial of 4:  24
```

Reduce function (Poll)

What will be the output of this code:

```
from functools import reduce
l = [1, 2, 3]
reduce(lambda x, y: x * y, l)
```

A- 6

B- 3

C- syntaxError

List comprehension

- Very concise way to create a new list by performing an operation on each item in the existing list.

- Faster than processing a list using the for loop.

[expression for item in iterable]

List comprehension

Example (1):

Create a list of squares of the numbers between 1 and 5?

```
# for loop
l1 = []

for i in range(6):
    l1.append(i*i)

print(l1)
```

[0, 1, 4, 9, 16, 25]

```
# list comprehension
l1= [x*x for x in range(6)]

print(l1)
```

List comprehension :

Example (2):

Combinations of items from two lists (list of integers, list of strings) in the form of a tuple are added in a new list.

```
numList=[1,2,3]
alphaList=["a", "b", "c"]
CombList=[(x,y) for x in numList for y in alphaList]
print(CombList)
```

```
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]
```

List comprehension: if…else

Example (3):

Return a list of strings where you specify if the numbers between 1 and 10 are odd or even numbers

```
l=[str(i)+" = Even" if i%2==0 else str(i)+" = Odd" for i in range(11)]
print(l)
```

```
['0 = Even', '1 = Odd', '2 = Even', '3 = Odd', '4 = Even', '5 = Odd', '6 = Even', '7 = Odd', '8 = Even', '9 = Odd', '10 = Even']
```

Next Module

Object oriented programming!