Functions in Python

User Defined Functions

Function

- A function is a block of organized, reusable code that is used to perform a single, related action.
- Functions provide better modularity for your application and a high degree of code reusing.

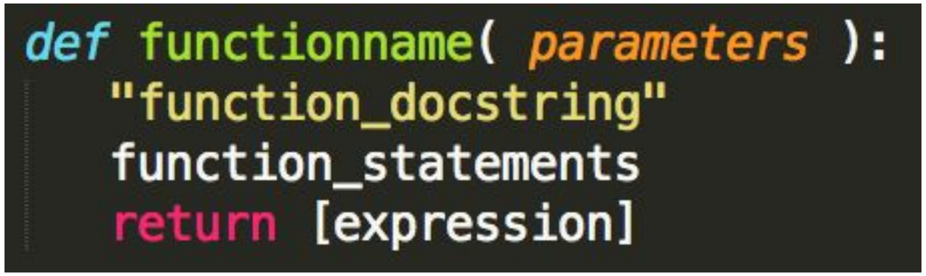
Function

Defining a Function

•The code block within every function starts with a colon (:) and is indented.

•The statement return [expression] exits a function, optionally passing back an expression to the caller.

- A return statement with no arguments is the same as return None.



```
def functionname( parameters ):
   "function_docstring"
   function_statements
   return [expression]
```

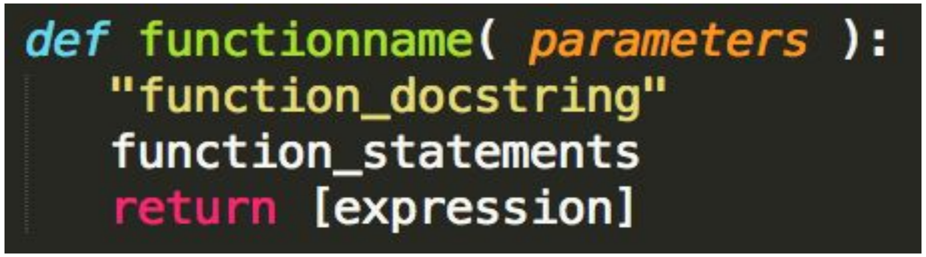
Function

Defining a Function

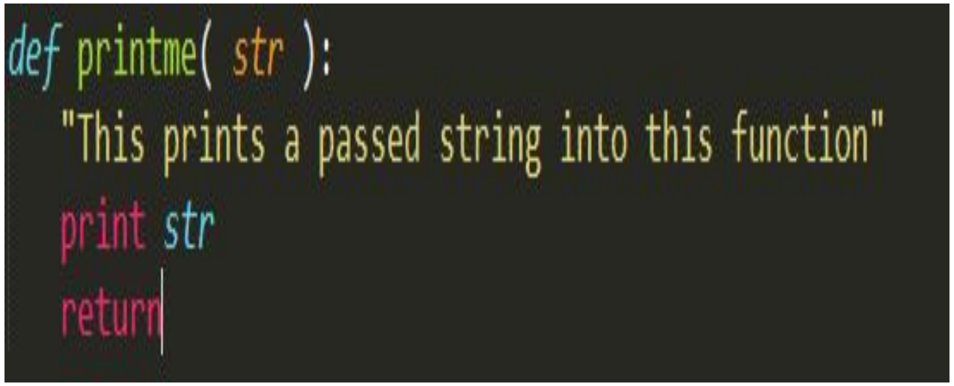
•Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).

•Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

•The first statement of a function can be an optional statement -the documentation string of the function or *docstring*.



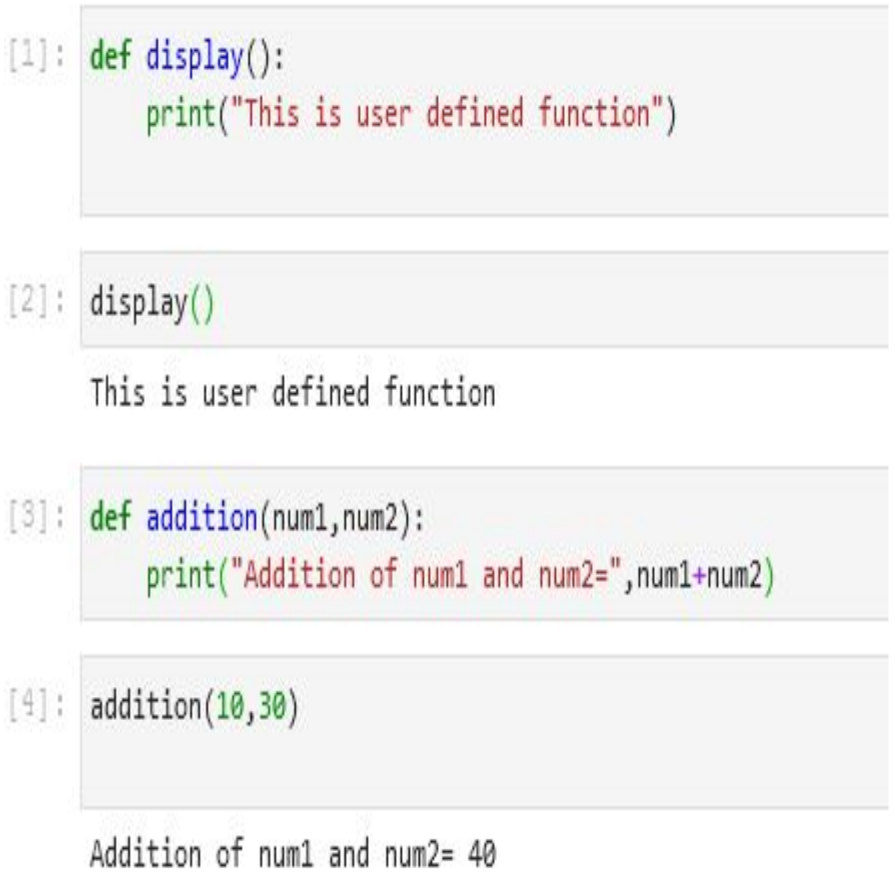
```
def functionname( parameters ):
   "function_docstring"
   function_statements
   return [expression]
```



```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

User Defined Functions

- User defined function without parameters
- User defined function with Parameters



```
[1]: def display():
         print("This is user defined function")
```

```
[2]: display()
```

```
This is user defined function
```

```
[3]: def addition(num1,num2):
         print("Addition of num1 and num2=",num1+num2)
```

```
[4]: addition(10,30)
```

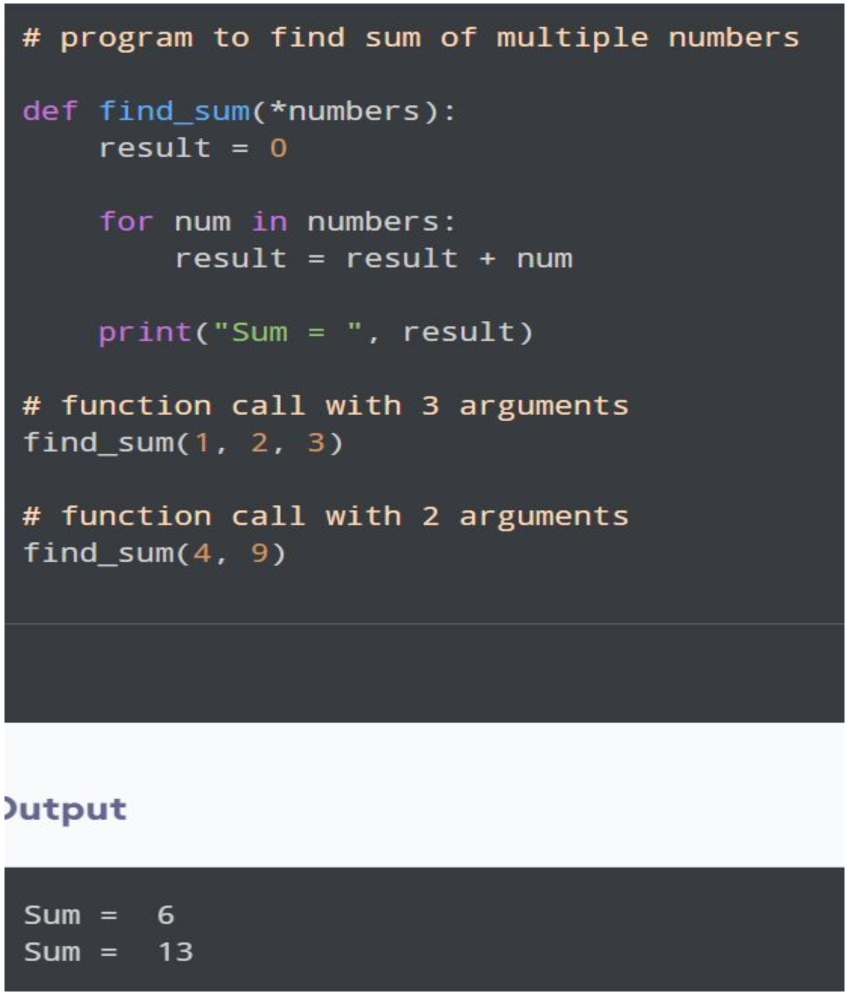
```
Addition of num1 and num2= 40
```

Python Function With Arbitrary Arguments

- Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments in Python.

- Arbitrary arguments allow us to pass a varying number of values during a function call.

- We use an asterisk (*) before the parameter name to denote this kind of argument.

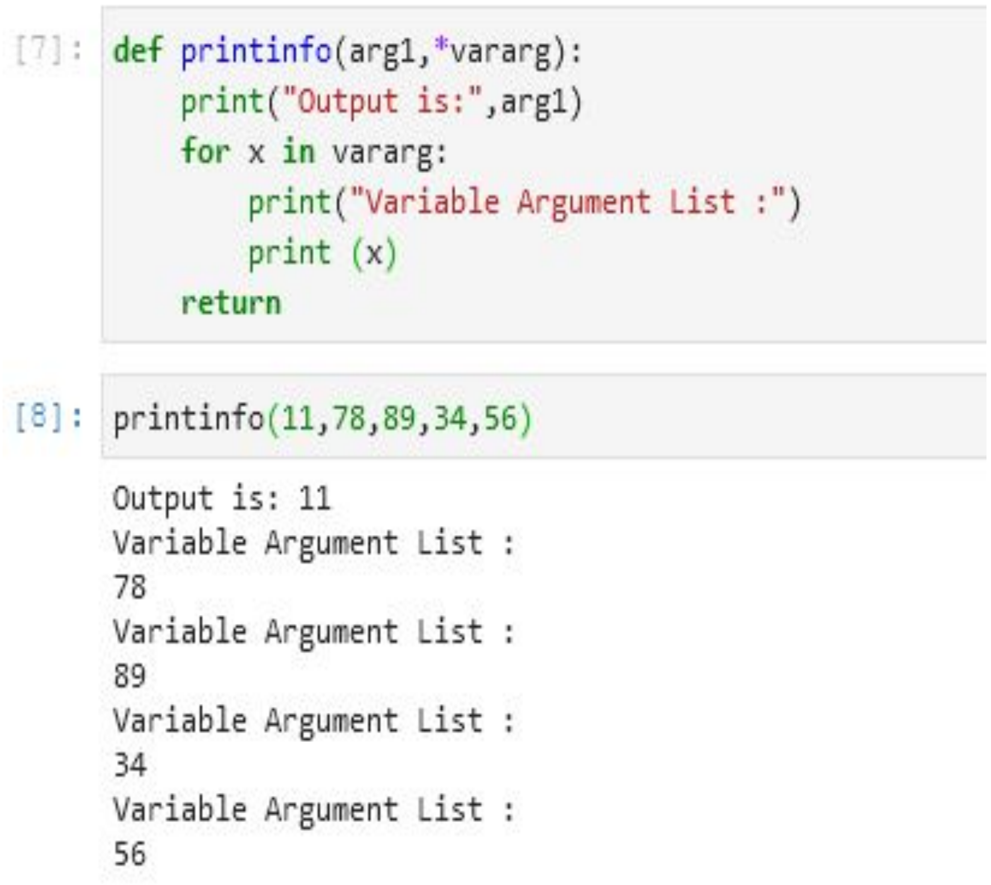


```
# program to find sum of multiple numbers

def find_sum(*numbers):
    result = 0

    for num in numbers:
        result = result + num

    print("Sum = ", result)

# function call with 3 arguments
find_sum(1, 2, 3)

# function call with 2 arguments
find_sum(4, 9)
```

```
Sum =  6
Sum =  13
```

User Defined Functions

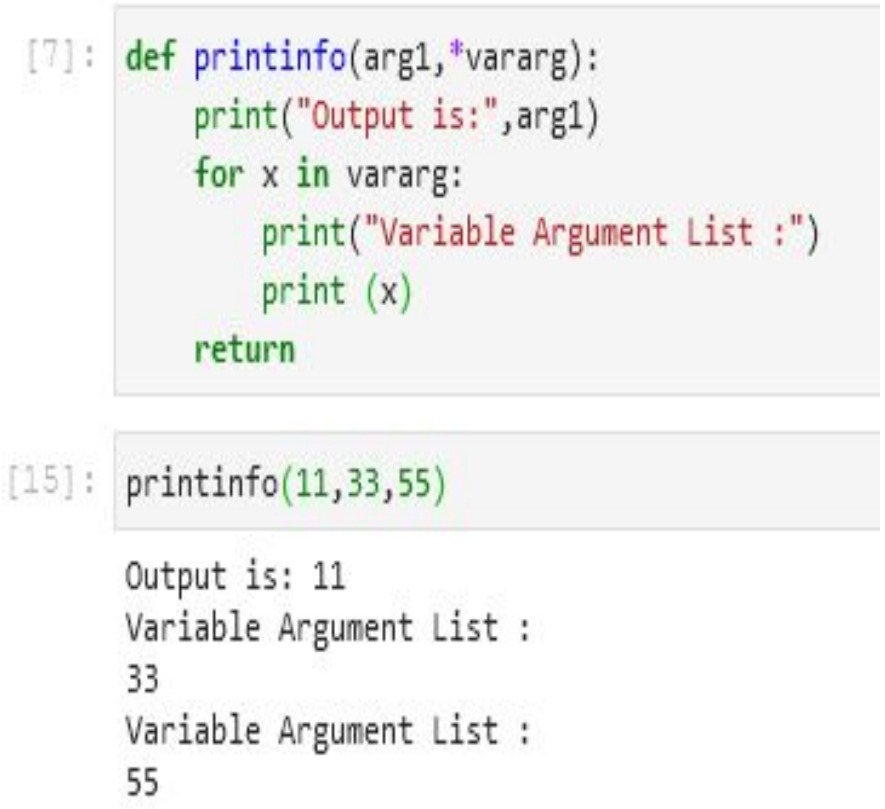
- For variable length arguments
- For unspecified no of arguments, star is used with the variable name



```
[7]: def printinfo(arg1,*vararg):
         print("Output is:",arg1)
         for x in vararg:
             print("Variable Argument List :")
             print (x)
         return
```

```
[8]: printinfo(11,78,89,34,56)
```

```
Output is: 11
Variable Argument List :
78
Variable Argument List :
89
Variable Argument List :
34
Variable Argument List :
56
```



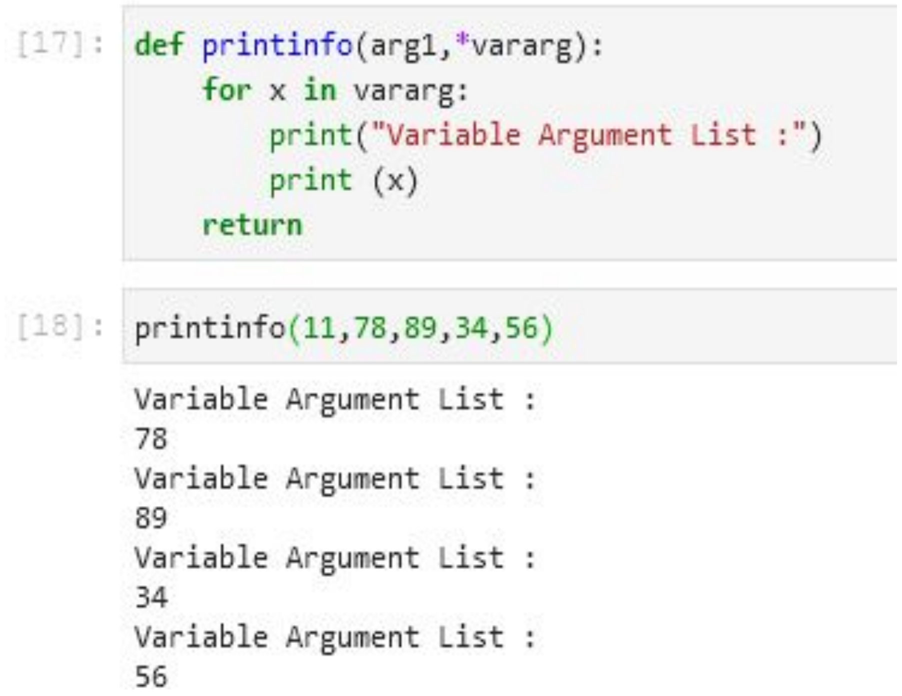
```
[7]: def printinfo(arg1,*vararg):
         print("Output is:",arg1)
         for x in vararg:
             print("Variable Argument List :")
             print (x)
         return
```

```
[15]: printinfo(11,33,55)
```

```
Output is: 11
Variable Argument List :
33
Variable Argument List :
55
```

User Defined Functions

- Without printing arg1

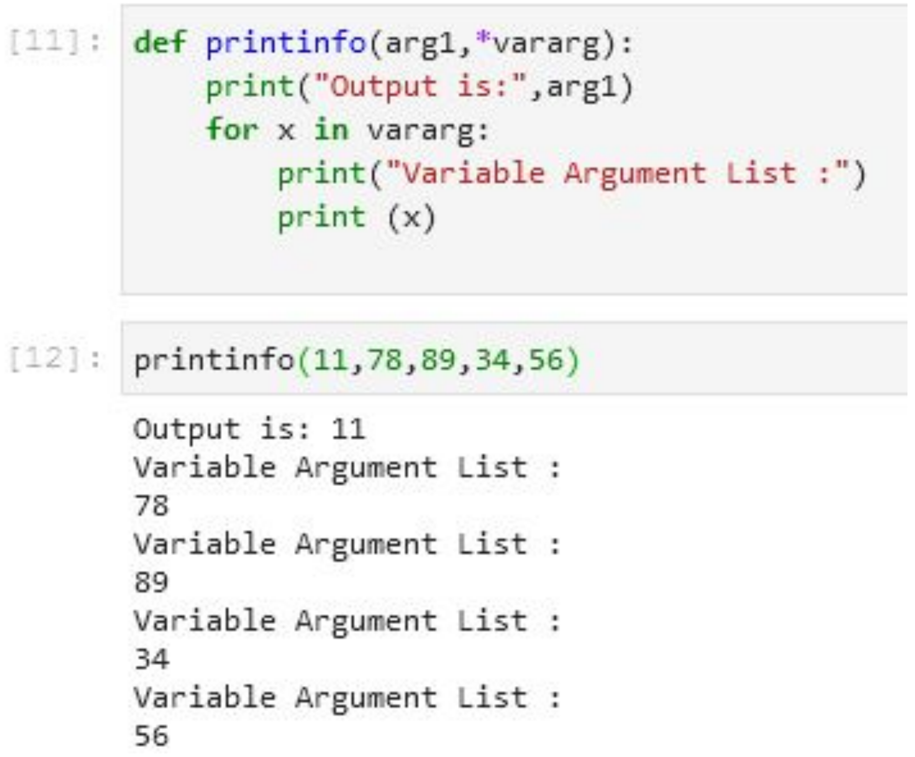


```
[17]: def printinfo(arg1,*vararg):
          for x in vararg:
              print("Variable Argument List :")
              print (x)
          return
```

```
[18]: printinfo(11,78,89,34,56)
```

```
Variable Argument List :
78
Variable Argument List :
89
Variable Argument List :
34
Variable Argument List :
56
```

- Without return statement
- Works fine



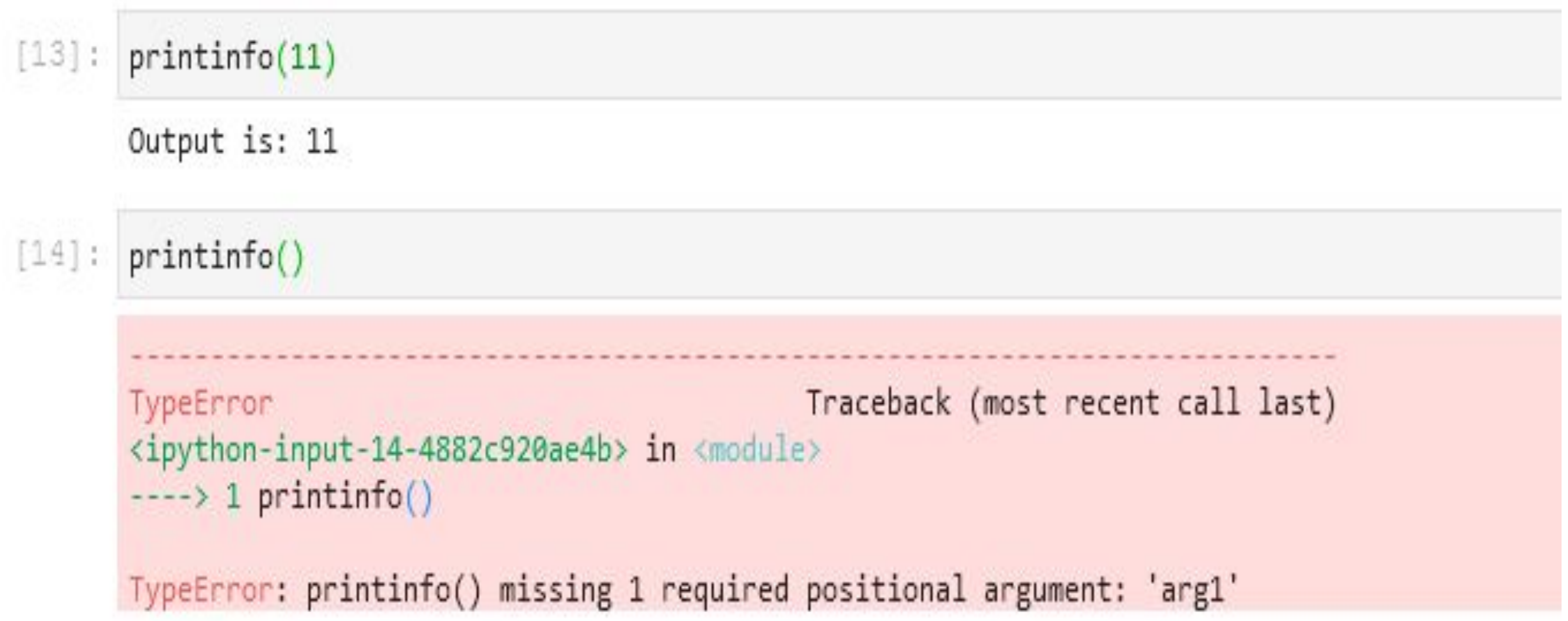
```
[11]: def printinfo(arg1,*vararg):
          print("Output is:",arg1)
          for x in vararg:
              print("Variable Argument List :")
              print (x)
```

```
[12]: printinfo(11,78,89,34,56)
```

```
Output is: 11
Variable Argument List :
78
Variable Argument List :
89
Variable Argument List :
34
Variable Argument List :
56
```

User Defined Functions

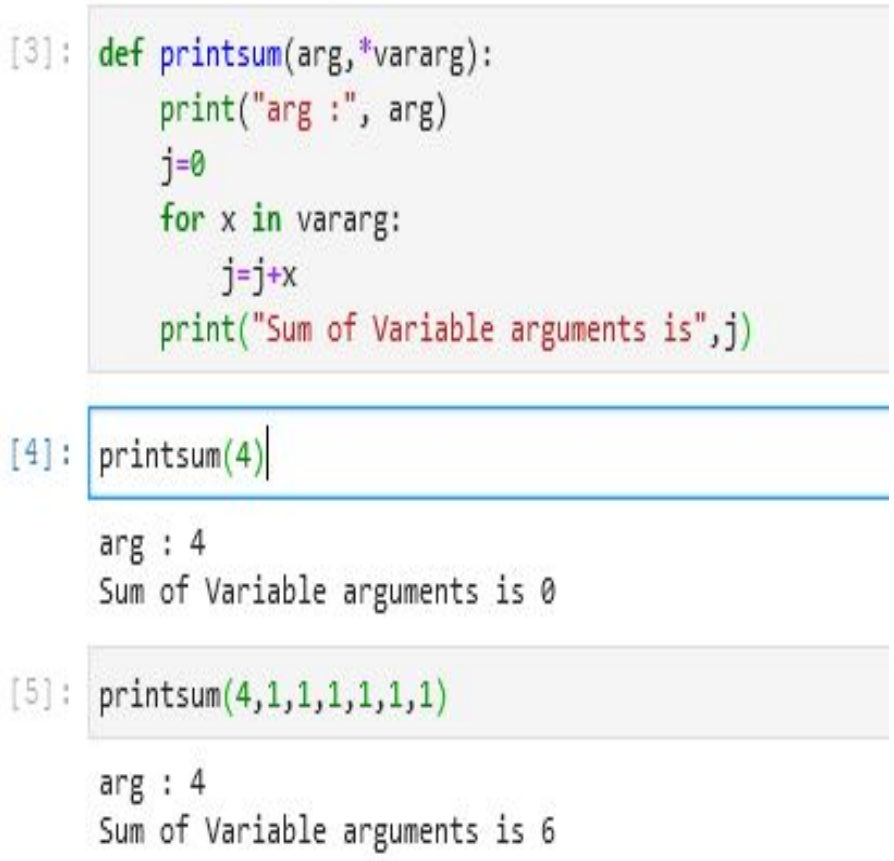
- Works perfectly, if Variable Parameters are NIL
- But needs compulsory Parameter, else raises error



```
[13]: printinfo(11)

Output is: 11

[14]: printinfo()

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-4882c920ae4b> in <module>
----> 1 printinfo()

TypeError: printinfo() missing 1 required positional argument: 'arg1'
```

User Defined Functions

- Function that calculates sum of Variable length parameters



```
[3]: def printsum(arg,*vararg):
         print("arg :", arg)
         j=0
         for x in vararg:
             j=j+x
         print("Sum of Variable arguments is",j)
```

```
[4]: printsum(4)
```

```
arg : 4
Sum of Variable arguments is 0
```

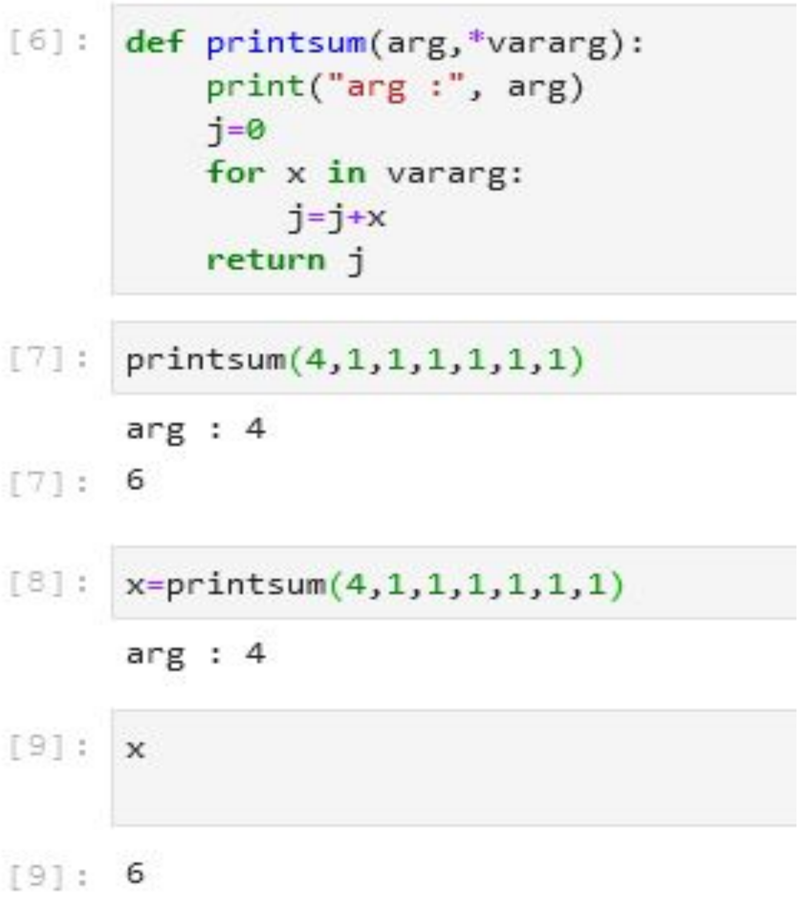
```
[5]: printsum(4,1,1,1,1,1,1)
```

```
arg : 4
Sum of Variable arguments is 6
```

User Defined Functions

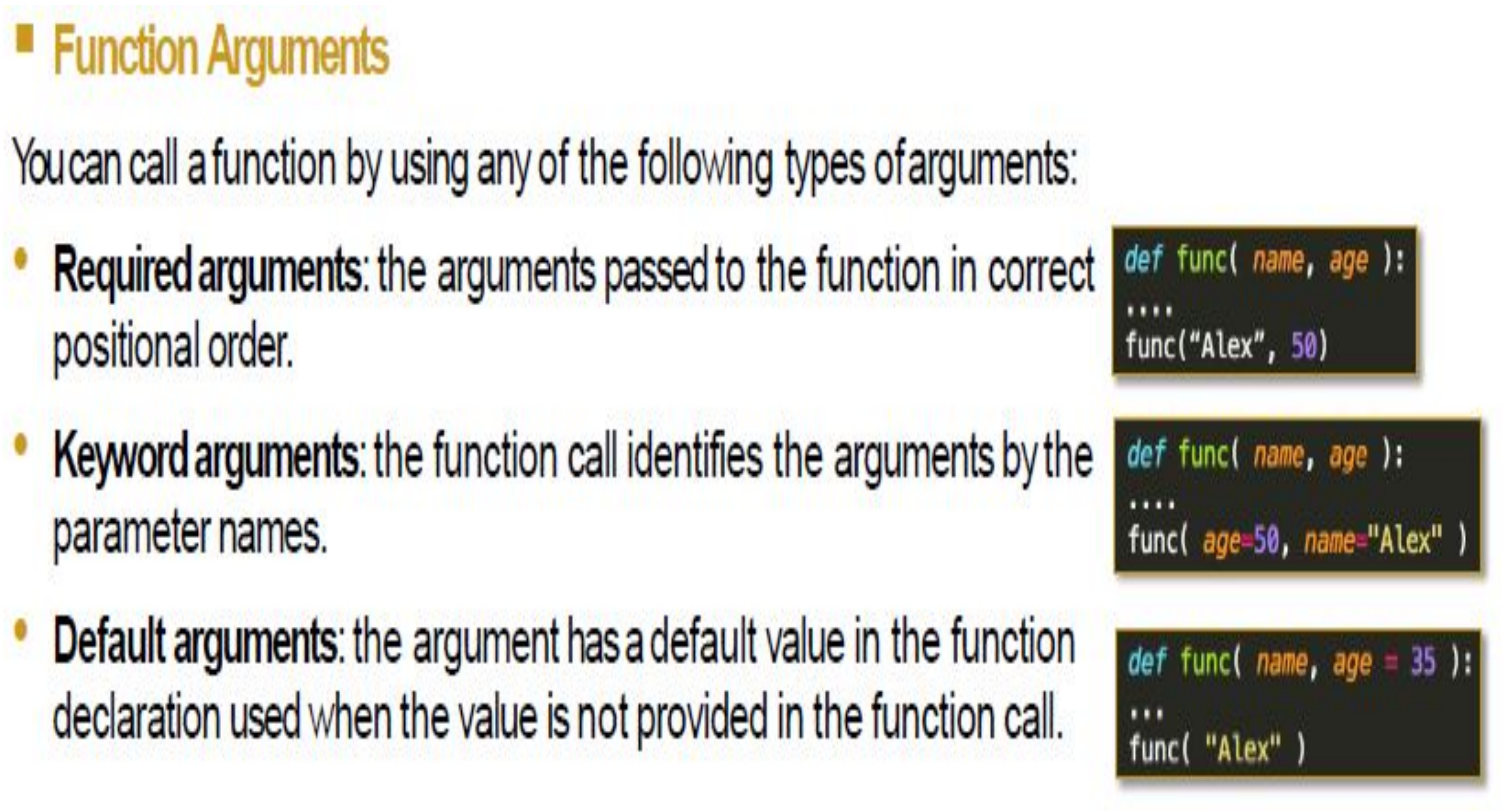
- Using Return statement in the function
- Value 6 gets printed on executing the function

- Storing the Value returned in some variable x
- Printing the value of that Variable



```
[6]: def printsum(arg,*vararg):
         print("arg :", arg)
         j=0
         for x in vararg:
             j=j+x
         return j

[7]: printsum(4,1,1,1,1,1,1)

     arg : 4
[7]: 6

[8]: x=printsum(4,1,1,1,1,1,1)

     arg : 4

[9]: x

[9]: 6
```

User Defined Functions



Function Arguments

You can call a function by using any of the following types of arguments:

- **Required arguments**: the arguments passed to the function in correct positional order.

```
def func( name, age ):
....
func("Alex", 50)
```

- **Keyword arguments**: the function call identifies the arguments by the parameter names.

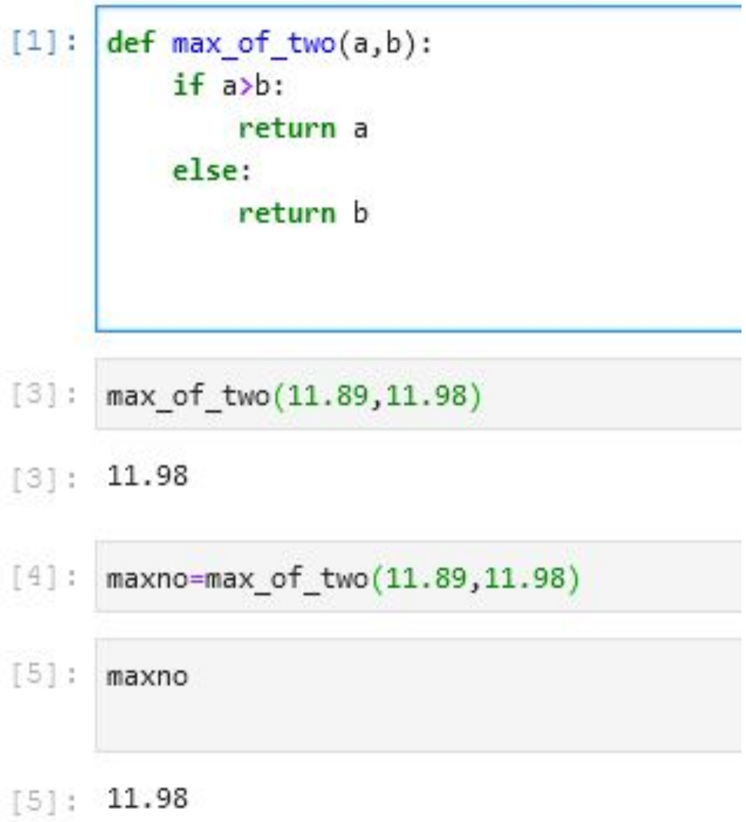
```
def func( name, age ):
....
func( age=50, name="Alex" )
```

- **Default arguments**: the argument has a default value in the function declaration used when the value is not provided in the function call.

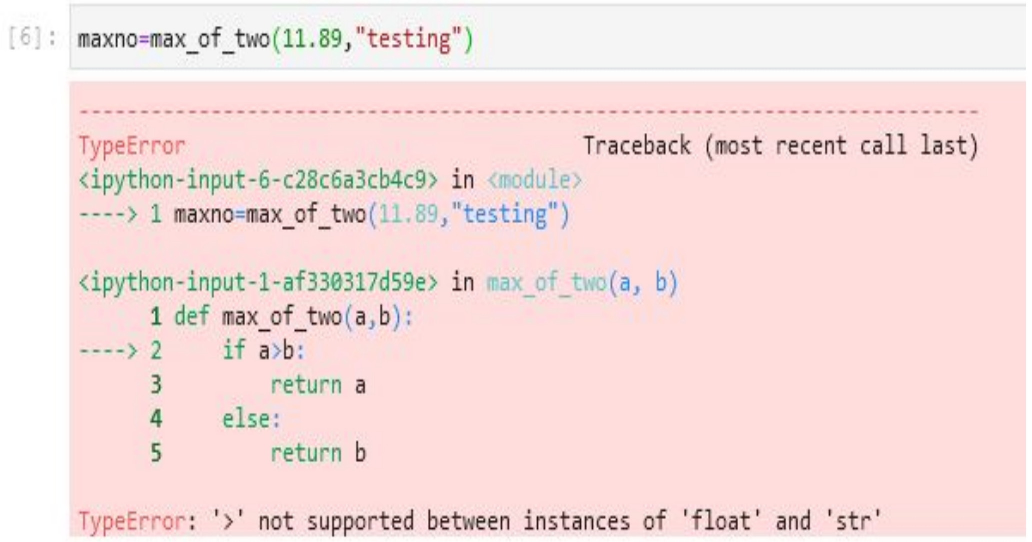
```
def func( name, age = 35 ):
...
func( "Alex" )
```

User Defined Functions

- Function to find max of 2 nos with return statement
- Return value stored in variable
- Error on Passing Strings as a parameter



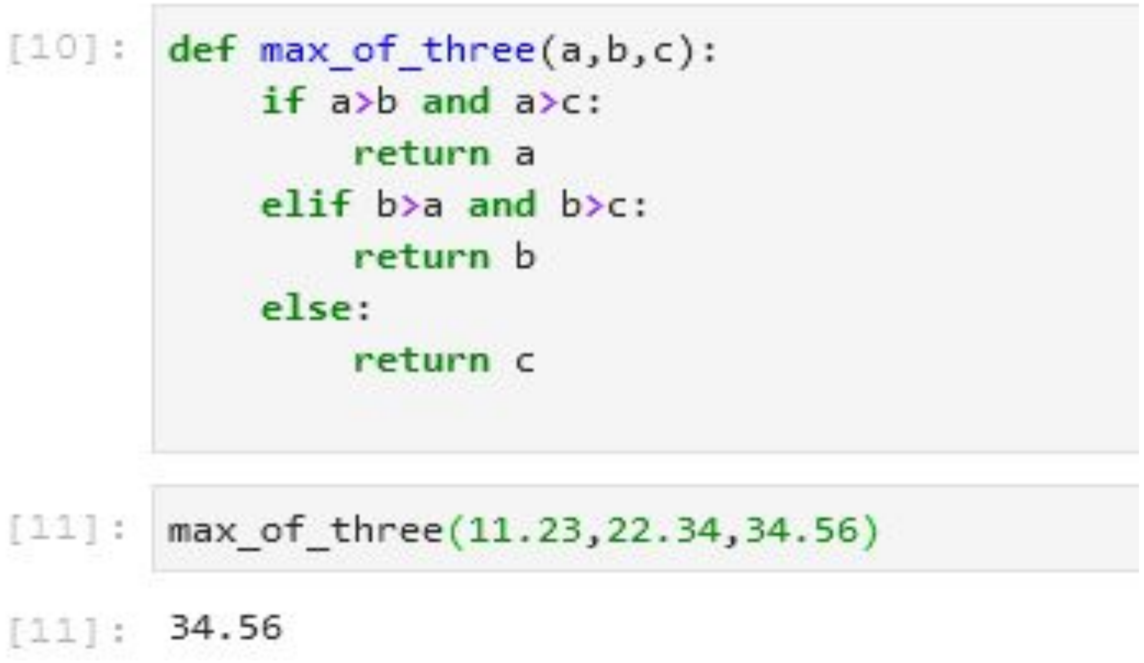
```
[1]: def max_of_two(a,b):
         if a>b:
             return a
         else:
             return b

[3]: max_of_two(11.89,11.98)

[3]: 11.98

[4]: maxno=max_of_two(11.89,11.98)

[5]: maxno

[5]: 11.98
```



```
[6]: maxno=max_of_two(11.89,"testing")

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-c28c6a3cb4c9> in <module>
----> 1 maxno=max_of_two(11.89,"testing")

<ipython-input-1-af330317d59e> in max_of_two(a, b)
      1 def max_of_two(a,b):
----> 2     if a>b:
      3         return a
      4     else:
      5         return b

TypeError: '>' not supported between instances of 'float' and 'str'
```

User Defined Functions

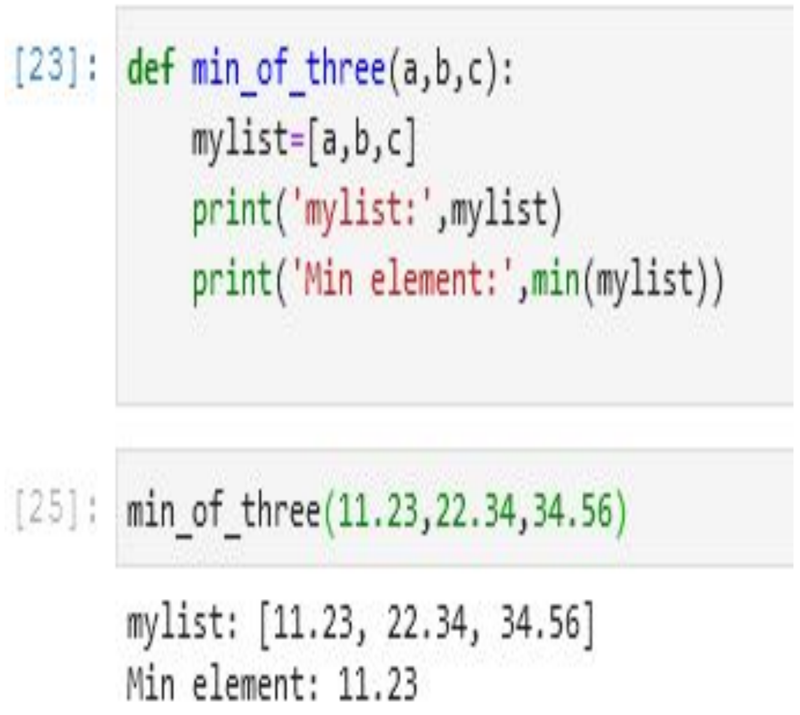
- Function to fins max of 3 nos with return statement



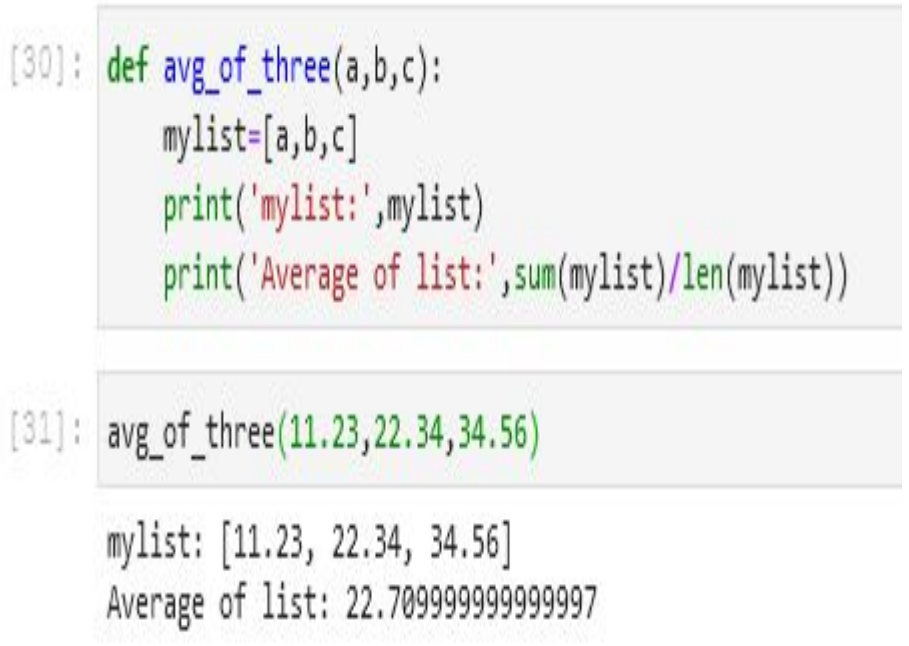
```
[10]: def max_of_three(a,b,c):
          if a>b and a>c:
              return a
          elif b>a and b>c:
              return b
          else:
              return c

[11]: max_of_three(11.23,22.34,34.56)

[11]: 34.56
```

User Defined Functions

- Minimum of 3 nos using List
- Average of 3 nos using List



```
[23]: def min_of_three(a,b,c):
          mylist=[a,b,c]
          print('mylist:',mylist)
          print('Min element:',min(mylist))

[25]: min_of_three(11.23,22.34,34.56)

mylist: [11.23, 22.34, 34.56]
Min element: 11.23
```



```
[30]: def avg_of_three(a,b,c):
          mylist=[a,b,c]
          print('mylist:',mylist)
          print('Average of list:',sum(mylist)/len(mylist))

[31]: avg_of_three(11.23,22.34,34.56)

mylist: [11.23, 22.34, 34.56]
Average of list: 22.709999999999997
```

Lambda Function

Lambda Function

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Syntax

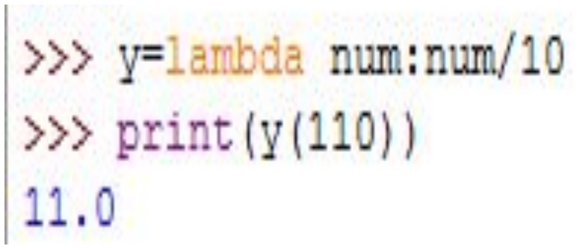
lambda *arguments* : *expression*

- The expression is executed and the result is returned:

Example

- Add 10 to argument a, and return the result:

```
x = lambda a : a + 10
print(x(5))
```



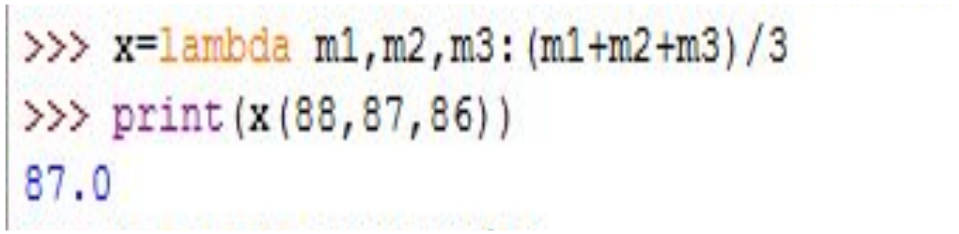
```
>>> y=lambda num:num/10
>>> print(y(110))
11.0
```

Lambda Function

Lambda functions can take any number of arguments:

Example

- Multiply argument a with argument b and return the result:
- x = lambda a, b : a * b
  print(x(5, 6))



```
>>> x=lambda m1,m2,m3:(m1+m2+m3)/3
>>> print(x(88,87,86))
87.0
```

Example

- Summarize argument a, b, and c and return the result:
- x = lambda a, b, c : a + b + c
  print(x(5, 6, 2))

Why Use Lambda Functions?

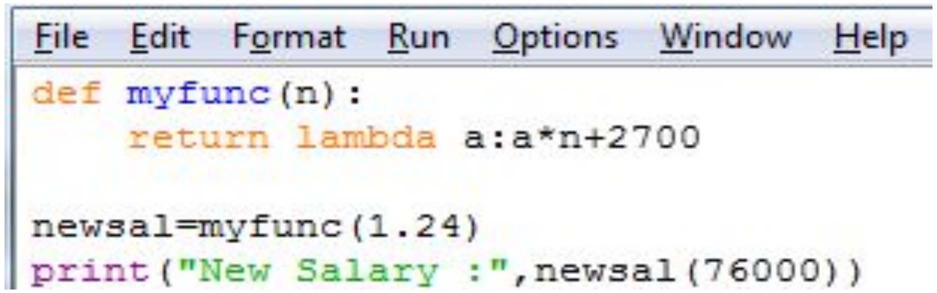
- The power of lambda is better shown when you use them as an anonymous function inside another function.
- Say you have a function definition that takes one argument, and <u>that argument will be multiplied with an unknown number:</u>

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

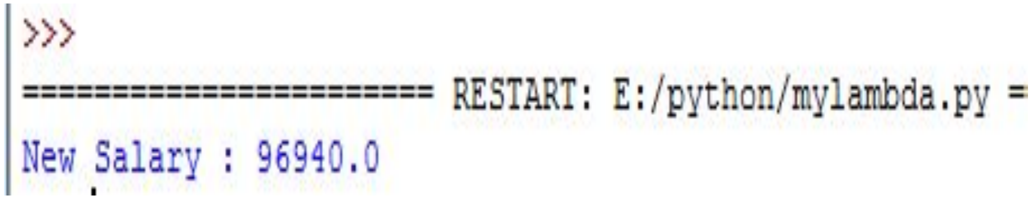
- Use that function definition to make a function that always doubles the number you send in:

Example

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

mydoubler = myfunc(2)
print(mydoubler(11))
```



```
def myfunc(n):
    return lambda a:a*n+2700

newsal=myfunc(1.24)
print("New Salary :",newsal(76000))
```



```
>>>
====================== RESTART: E:/python/mylambda.py =
New Salary : 96940.0
```

- Or, use the same function definition to make a function that always *triples* the number you send in:

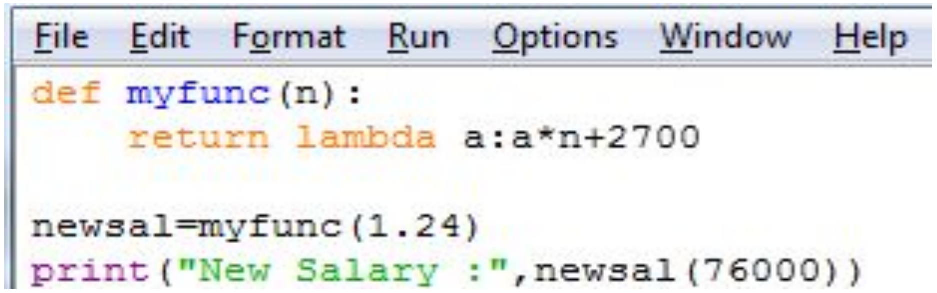
Example

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

mytripler = myfunc(3)
print(mytripler(11))
```

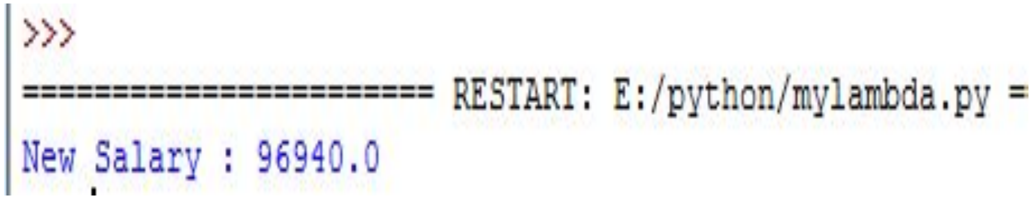
Example

HRA=24% of Basic or a

Ta=2700 Rs



```
def myfunc(n):
    return lambda a:a*n+2700

newsal=myfunc(1.24)
print("New Salary :",newsal(76000))
```



```
>>>
====================== RESTART: E:/python/mylambda.py =
New Salary : 96940.0
```

- Or, use the same function definition to make both functions, in the same program:

Example

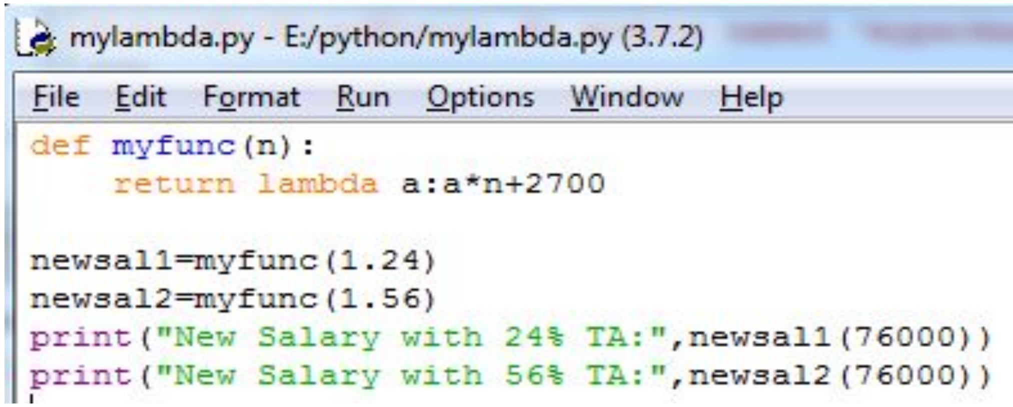
```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

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

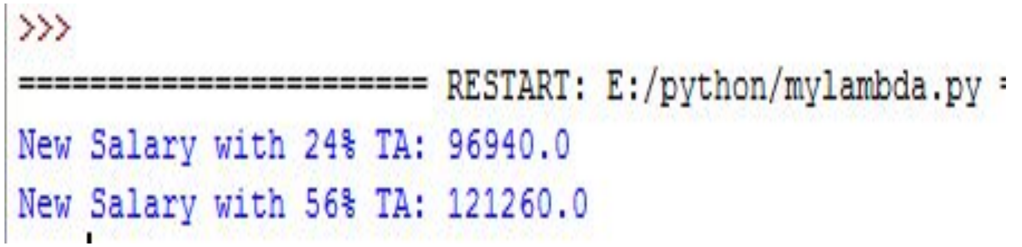
- Or, use the same function definition to make both functions, in the same program:

Example

- HRA=24% of Basic or a Ta=2700 Rs
- HRA=56% of Basic or a Ta=2700 Rs


mylambda.py - E:/python/mylambda.py (3.7.2)
File Edit Format Run Options Window Help

```python
def myfunc(n):
    return lambda a:a*n+2700

newsal1=myfunc(1.24)
newsal2=myfunc(1.56)
print("New Salary with 24% TA:",newsal1(76000))
print("New Salary with 56% TA:",newsal2(76000))
```



```
>>>
======================= RESTART: E:/python/mylambda.py =
New Salary with 24% TA: 96940.0
New Salary with 56% TA: 121260.0
```

Python filter() Function

- Python filter() function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence from those elements of iterable for which function returns **True**.
- The first argument can be None if the function is not available and returns only elements that are True.

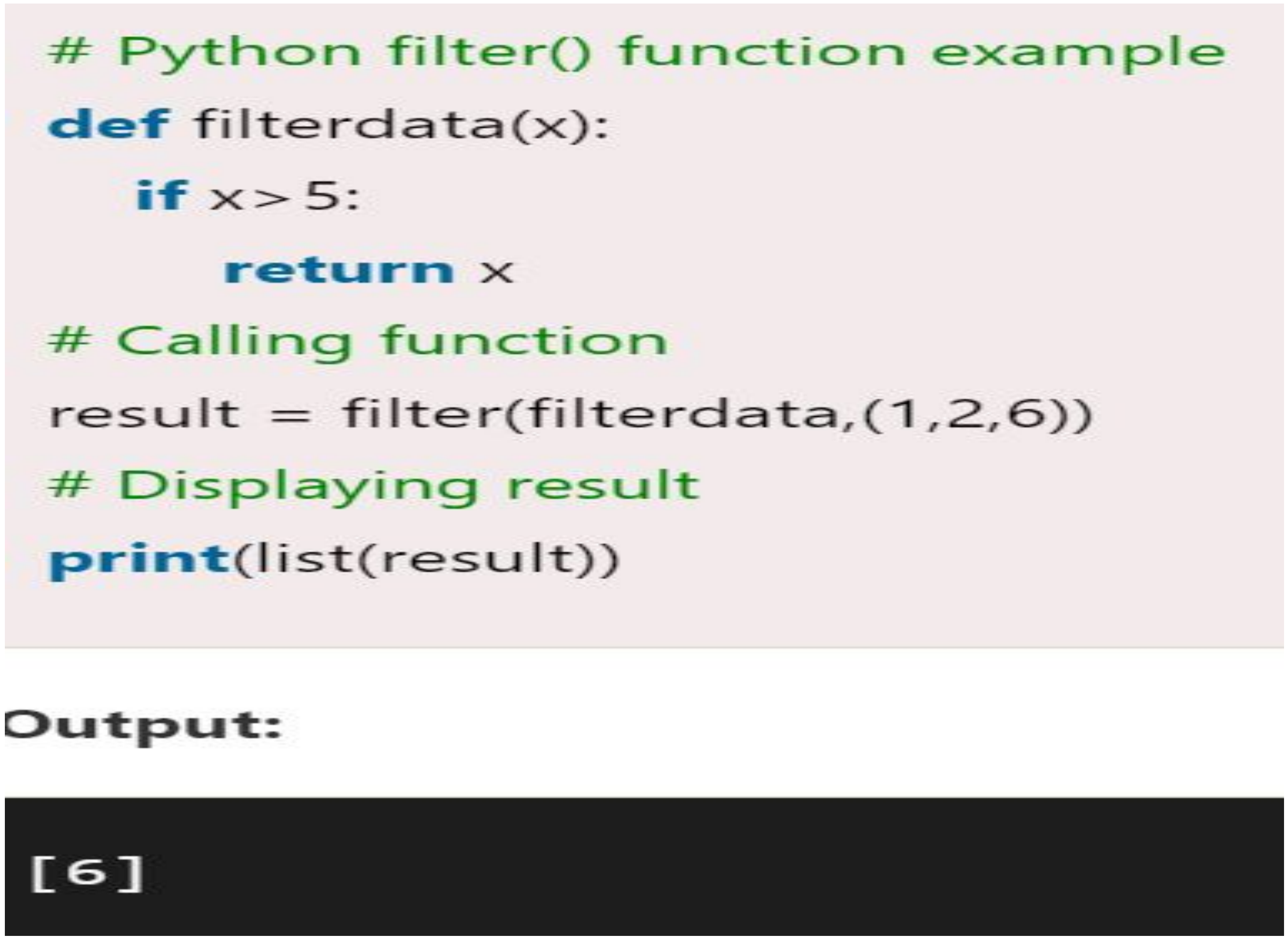
Syntax:

filter (function, iterable)

Parameters:

- function: It is a function. If set to None returns only elements that are True.
- Iterable: Any iterable sequence like list, tuple, and string.

Example filter()

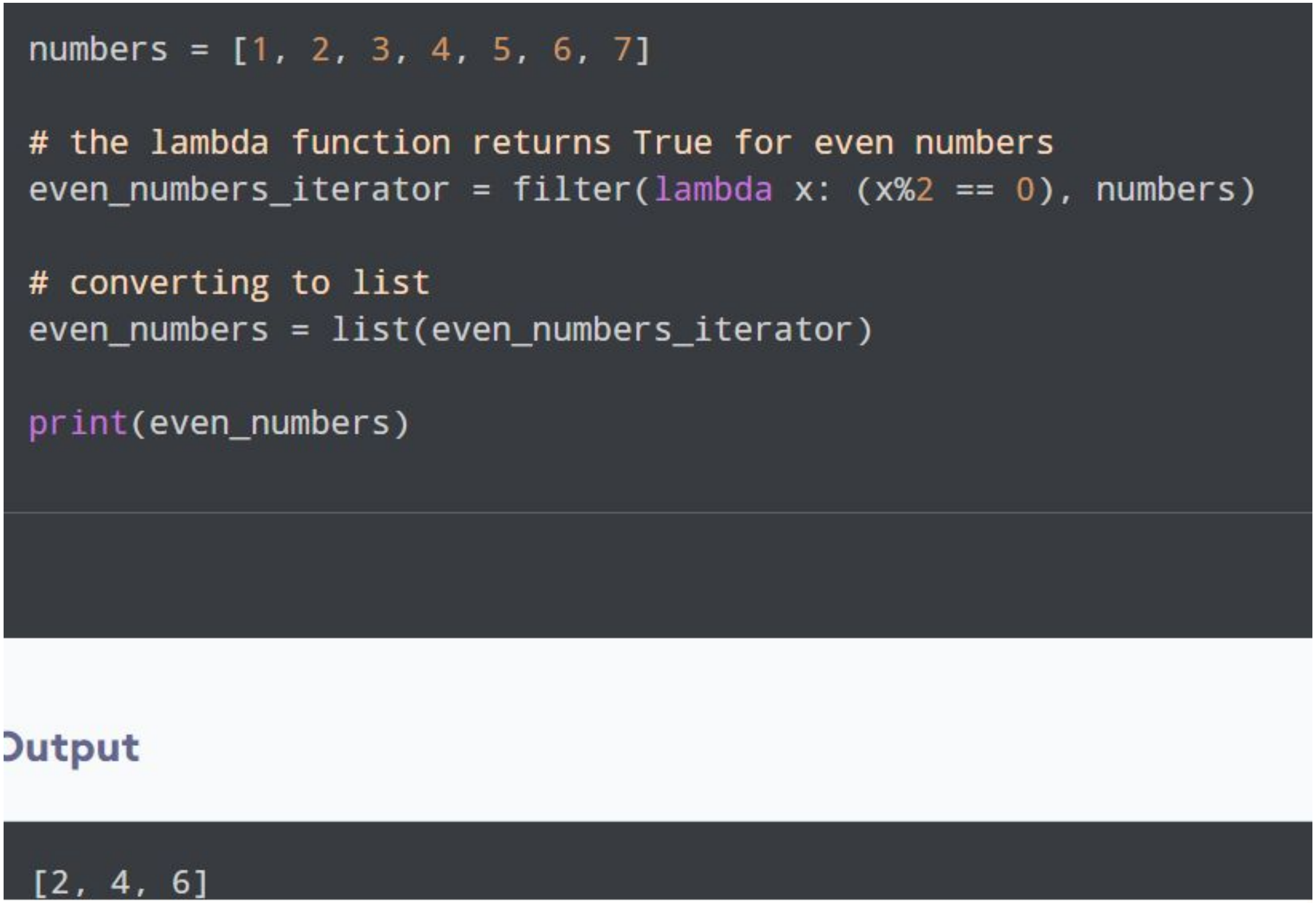


```python
# Python filter() function example
def filterdata(x):
    if x>5:
        return x
# Calling function
result = filter(filterdata,(1,2,6))
# Displaying result
print(list(result))
```

Output:

```
[6]
```

Use filter() with lambda



```python
numbers = [1, 2, 3, 4, 5, 6, 7]

# the lambda function returns True for even numbers
even_numbers_iterator = filter(lambda x: (x%2 == 0), numbers)

# converting to list
even_numbers = list(even_numbers_iterator)

print(even_numbers)
```

Output

```
[2, 4, 6]
```

Map() function

- The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.)

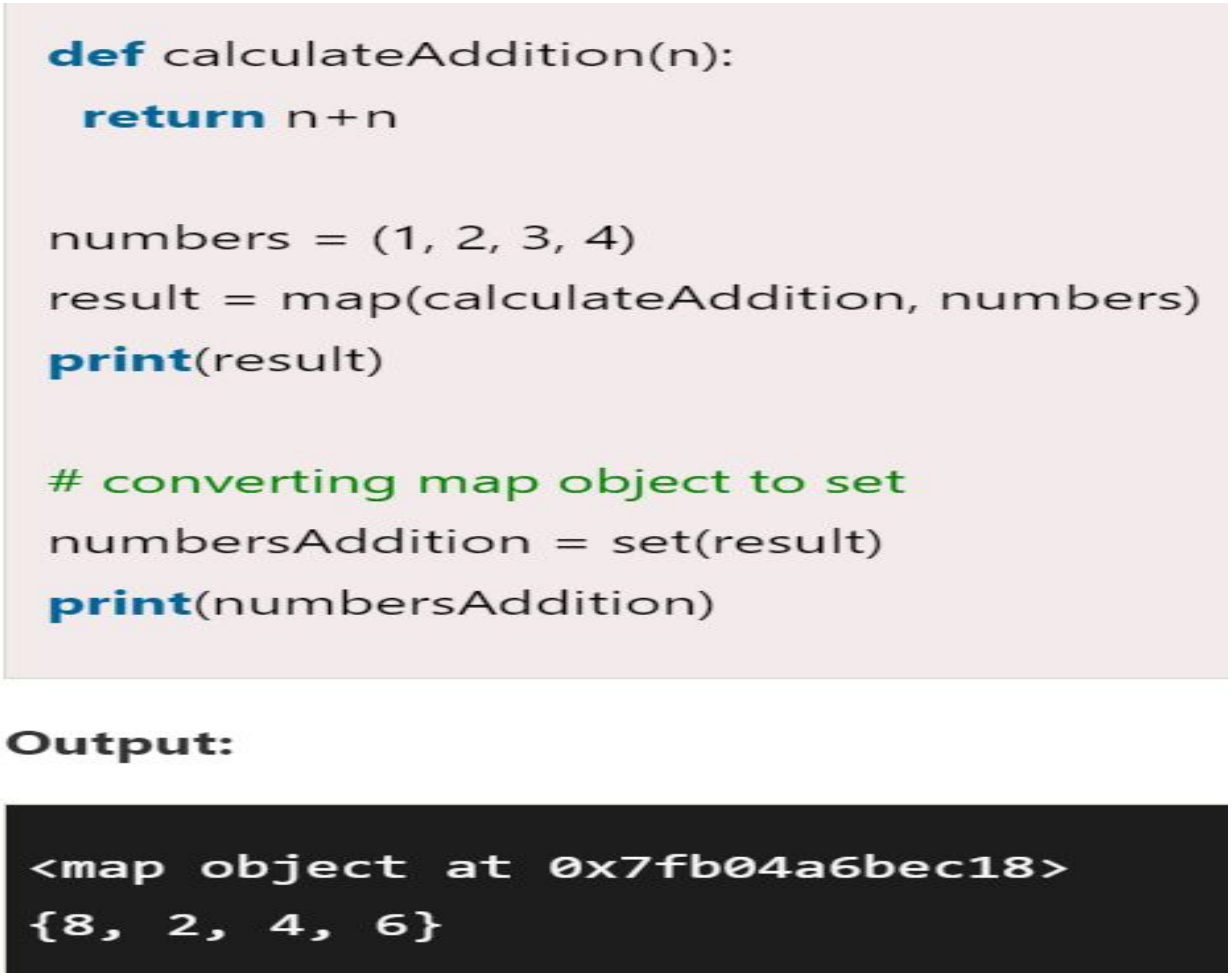
Syntax:

```
map(function, iterables)
```

Parameters:

- **function**- It is a function in which a map passes each item of the iterable.
- **iterables**- It is a sequence, collection or an iterator object which is to be mapped

Example

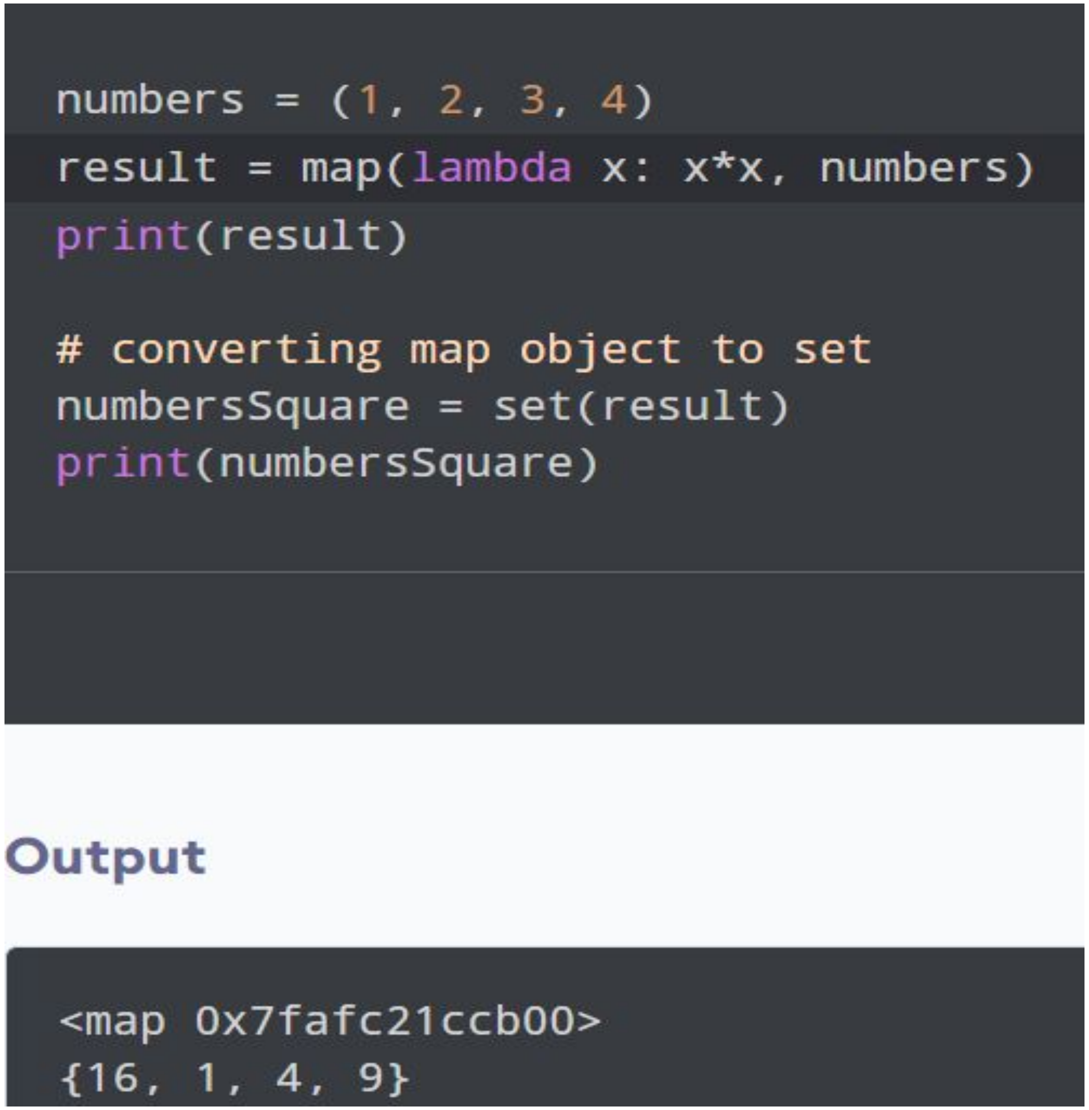


```python
def calculateAddition(n):
  return n+n

numbers = (1, 2, 3, 4)
result = map(calculateAddition, numbers)
print(result)

# converting map object to set
numbersAddition = set(result)
print(numbersAddition)
```

Output:

```
<map object at 0x7fb04a6bec18>
{8, 2, 4, 6}
```

Use lambda function with map() function



```python
numbers = (1, 2, 3, 4)
result = map(lambda x: x*x, numbers)
print(result)

# converting map object to set
numbersSquare = set(result)
print(numbersSquare)
```

Output

```
<map 0x7fafc21ccb00>
{16, 1, 4, 9}
```

Range() function

- Python **range()** function returns an immutable sequence of numbers starting from 0, increments by 1 and ends at a specified number.

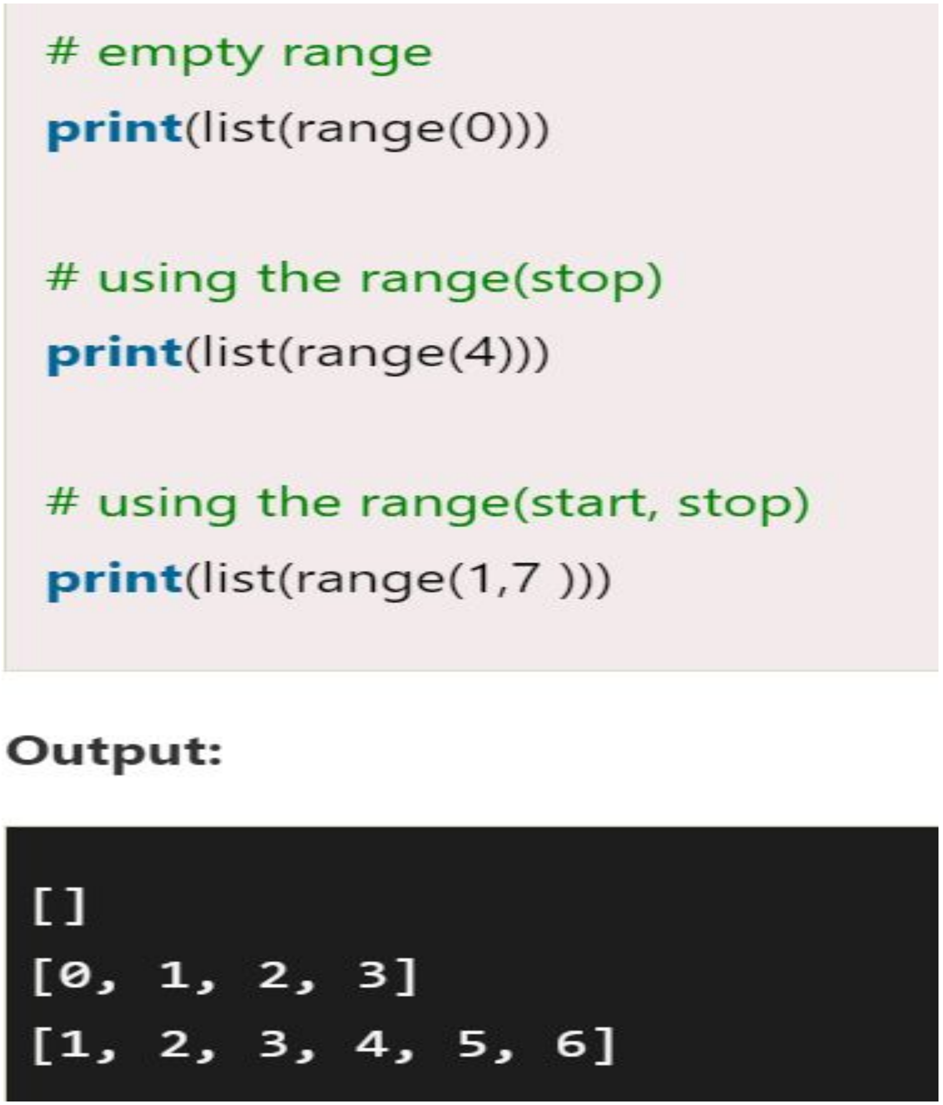
Syntax:

range(start, stop, step)

Parameters:

- **start** (optional) : It is an integer number that specifies the starting position. The Default value is 0.
- **stop** (optional) : It is an integer that specifies the ending position.
- **step** (optional) : It is an integer that specifies the increment of a number. The Default value is 1.

Example



```
# empty range
print(list(range(0)))

# using the range(stop)
print(list(range(4)))

# using the range(start, stop)
print(list(range(1,7 )))
```

Output:

```
[]
[0, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```