



**TATA ELXSI**

## Python Scripting

Learning & Development Team

Monday, November 2, 2020

## Module – 3 : Functions

## Disclaimer

- This material is developed for in house training at **TATA ELXSI**.
- *TATA ELXSI is not responsible for any errors or omissions, or for the results obtained from the use of this information. All information in this material is provided "as is", with no guarantee of completeness, timeliness or of the results obtained from the use of this information...*
- *This is a training material and the information contained herein is not intended to be a source of advice or research outcome with respect to the material presented, and the information and/or documents/examples contained in this material do not constitute authors or TATA ELXSI's advice.*
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage.
- **TATA ELXSI**, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books.
- Excerpts of these material has been taken where there is no copy right infringements.

## Agenda

- Built-in functions (bin,hex,oct,min,max,len,id,slice,sorted)
- Creating user defined functions
- python namespace
- variable scope
- recursion
- default arguments
- keyword arguments
  - Use of \*args
  - Use of \*\*kwargs

## Built-in List Functions & Methods:

The Python interpreter has a number of functions and types built into it that are always available.

Function	Description
<code>abs(x)</code>	Return the absolute value of a number. The argument may be an integer or a floating point number.
<code>bin(x)</code>	Convert an integer number to a binary string.
<code>len(s)</code>	Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).
<code>max(iterable, *[, key, default])</code> :	Return the largest item in an iterable or the largest of two or more arguments.
<code>min(iterable, *[, key, default])</code>	Return the smallest item in an iterable or the smallest of two or more arguments.

## Built-in List Functions & Methods:

Function	Description
hex(x)	Convert an integer number to a lowercase hexadecimal string prefixed with "0x".
id(object)	Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime.
class int(x, base=10)	Return an integer object constructed from a number or string x, or return 0 if no arguments are given.
oct(x)	Convert an integer number to an octal string.

## Built-in List Functions & Methods:

Function	Description
<code>class set([iterable])</code>	Return a new set object, optionally with elements taken from iterable.
<code>class slice(start, stop[, step])</code>	Return a slice object representing the set of indices specified by <code>range(start, stop, step)</code> .
<code>sorted(iterable[, key][, reverse])</code>	Return a new sorted list from the items in iterable.
<code>class type(object)</code>	With one argument, return the type of an object.

# Functions, Procedures

```
def name(arg1, arg2, ...):  
    """documentation""" # optional doc string  
    statements
```

```
return # from procedure
```

```
return expression # from function
```



## Example Function

```
def gcd(a, b):  
    "greatest common divisor"  
    while a != 0:  
        a, b = b%a, a    # parallel assignment  
    return b
```

gcd.\_\_doc\_\_

gcd(12, 20)

# Python Namespace

- Namespace is a collection of names
- Name (also called identifier) is simply a name given to objects. (for inst., `a=2`)
- A namespace containing all the built-in names is created when we start the Python interpreter and exists as long we don't exit. This is the reason that built-in functions like `id()`, `print()` etc.

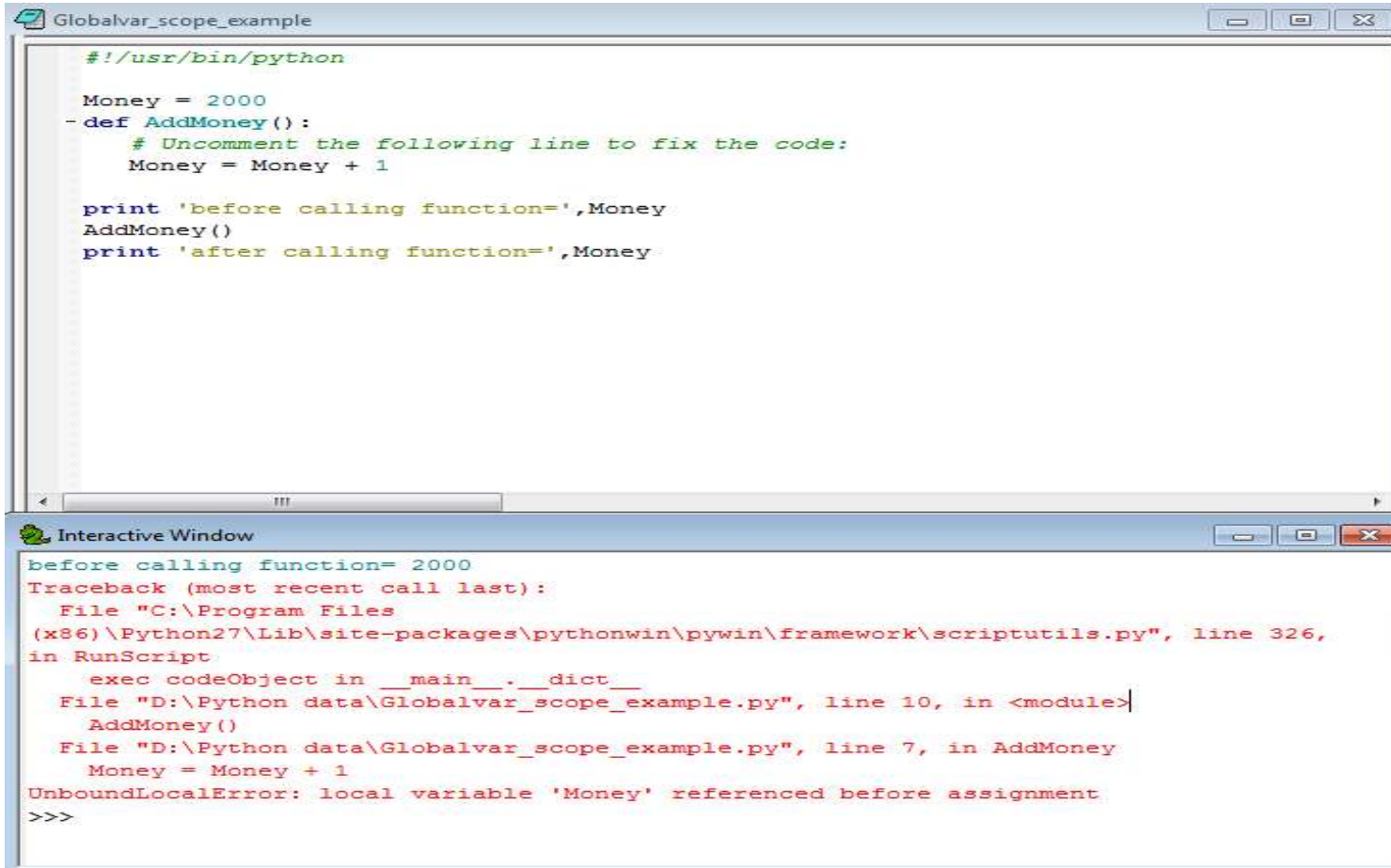
# Namespace

- Each module creates its own global namespace
- A local namespace is created when a function is called, which has all the names defined in it. Similar, is the case with class.
- For example, when we do the assignment `a = 2`, here 2 is an object stored in memory and a is the name we associate it with. We can get the address (in RAM) of some object through the built-in function, `id()`.

# Scope

- Scope is the portion of the program from where a namespace can be accessed directly without any prefix. At any given moment, there are at least three nested scopes.
- Scope of the current function which has local names
- Scope of the module which has global names
- Outermost scope which has built-in names

# Scope-Local



The image shows a Python IDE window titled "Globalvar\_scope\_example" and an "Interactive Window" below it. The script in the IDE contains a function `AddMoney()` that attempts to modify a global variable `Money`. The interactive window shows the output of the script and a detailed traceback of the `UnboundLocalError`.

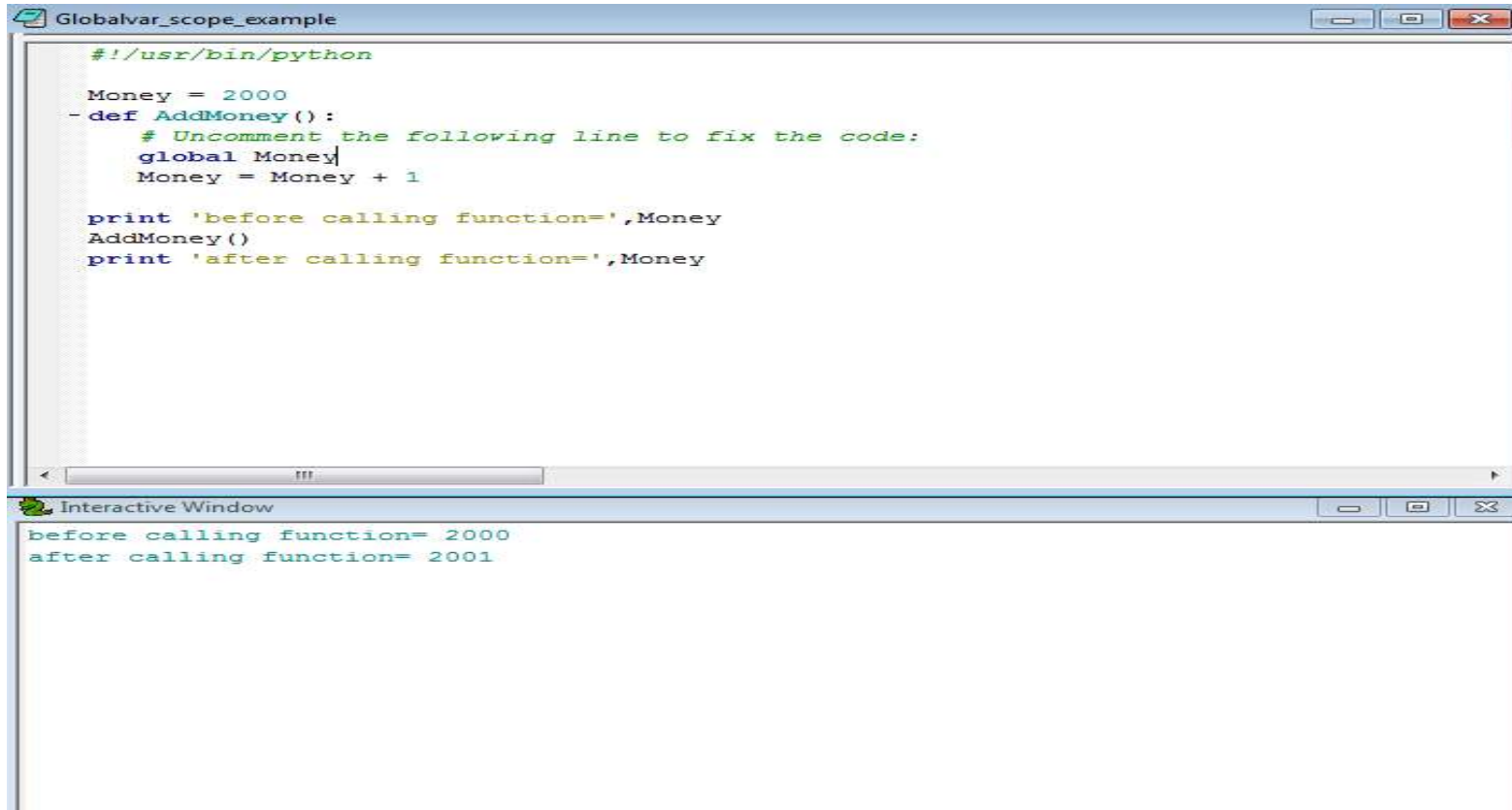
```
#!/usr/bin/python

Money = 2000
def AddMoney():
    # Uncomment the following line to fix the code:
    Money = Money + 1

print 'before calling function=',Money
AddMoney()
print 'after calling function=',Money
```

before calling function= 2000  
Traceback (most recent call last):  
File "C:\Program Files (x86)\Python27\Lib\site-packages\pythonwin\pywin\framework\scriptutils.py", line 326, in RunScript  
exec codeObject in \_\_main\_\_.\_\_dict\_\_  
File "D:\Python data\Globalvar\_scope\_example.py", line 10, in <module>  
AddMoney()  
File "D:\Python data\Globalvar\_scope\_example.py", line 7, in AddMoney  
Money = Money + 1  
UnboundLocalError: local variable 'Money' referenced before assignment  
>>>

# Scope-global



The image shows a screenshot of a Python IDE with two windows. The top window, titled 'Globalvar\_scope\_example', contains the following Python code:

```
#!/usr/bin/python

Money = 2000
def AddMoney():
    # Uncomment the following line to fix the code:
    global Money
    Money = Money + 1

print 'before calling function=',Money
AddMoney()
print 'after calling function=',Money
```

The bottom window, titled 'Interactive Window', shows the output of the code:

```
before calling function= 2000
after calling function= 2001
```

## globals() and locals() Functions

- If `locals()` is called from within a function, it will return all the names that can be accessed locally from that function.
- If `globals()` is called from within a function, it will return all the names that can be accessed globally from that function.
- The return type of both these functions is dictionary. Therefore, names can be extracted using the `keys()` function.

## Example Function

```
a_var = 10
b_var = 15
e_var = 25
def a_func(a_var):
    print "in a_func a_var = ",a_var
    b_var = 100 + a_var
    d_var = 2*a_var
    print "in a_func b_var = ",b_var
    print "in a_func d_var = ",d_var
    print "in a_func e_var = ",e_var
    return b_var + 10
```

```
c_var = a_func(b_var)
print "a_var = ",a_var
print "b_var = ",b_var
print "c_var = ",c_var
print "d_var = ",d_var
```



## Recursion Function

#defines a function that  
calculates the factorial

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

```
print "2! = ",factorial(2)  
print "3! = ",factorial(3)  
print "4! = ",factorial(4)  
print "5! = ",factorial(5)
```

### *Values of factorials*

```
0! = 1  
1! = 1  
2! = 2  
3! = 6  
4! = 24  
5! = 120  
6! = 720  
7! = 5 040  
8! = 40 320  
9! = 362 880  
10! = 3 628 800  
11! = 39 916 800  
12! = 479 001 600  
13! = 6 227 020 800  
14! = 87 178 291 200  
15! = 1 307 674 368 000  
16! = 20 922 789 888 000  
17! = 355 687 428 096 000  
18! = 6 402 373 705 728 000  
19! = 121 645 100 408 832 000  
20! = 2 432 902 008 176 640 000
```

## The range() Function

- If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy.
- It generates arithmetic progressions:

```
>>> for i in range(5):
```

```
...     print(i)
```

```
...
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

## The range() Function

- range(5, 10)  
5 through 9
- range(0, 10, 3)  
0, 3, 6, 9
- range(-10, -100, -30)  
-10, -40, -70

Example: >>> a = ['Mary', 'had', 'a', 'little', 'lamb']  
>>> for i in range(len(a)):  
... print(i, a[i])  
...

### **Output:**

0 Mary  
1 had  
2 a  
3 little  
4 lamb

## Range()

```
>>> print(range(10))  
range(0, 10)
```

- In many ways the object returned by range() behaves as if it is a list, but in fact it isn't.
- It is an object which returns the successive items of the desired sequence , thus saving space.
- The function list() is another; it creates lists from iterables:

```
>>> list(range(5))  
[0, 1, 2, 3, 4]
```

OR

- function enumerate(iterable, start=0) is useful

## enumerate()

- `enumerate(iterable, start=0)`
- Returns an enumerate object.
- `enumerate` essentially turns each element of the input list into a list of tuples with the first element as the index and the element as the second.
- `enumerate(['zero','one', 'two', 'three'])` therefore turns into
- `[(0,'zero'),(1, 'one'), (2, 'two'), (3, 'three')]`

## enumerate()

- Example :

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
```

```
print(list(enumerate(seasons)))
```

```
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
```

- `list(enumerate(seasons, start=1))`
- `[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]`

## Default Argument Values

- This creates a function that can be called with fewer arguments than it is defined to allow.
- The default values are evaluated at the point of function definition in the defining scope

```
i = 5
def f(arg=i):
    print(arg)
i = 6
f()
```

- will print 5.

## Default Argument Values

- For example:

```
def ask_ok(prompt,      retries=4   , complaint='Yes or no, please!') :  
    while True:  
        ok = raw_input(prompt)  
        if ok in ('y', 'ye', 'yes'):  
            return True  
        if ok in ('n', 'no', 'nop', 'nope'):  
            return False  
        retries = retries - 1  
        if retries < 0:  
            raise OSError('uncooperative user')  
    print(complaint)
```



## Default Argument Values

- This function can be called in several ways:
- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

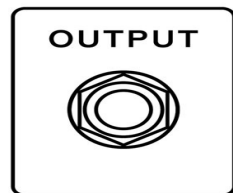
## Default Argument Values



- The default value is evaluated only once.
- This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):  
    L.append(a)  
    return L  
print(f(1))  
print(f(2))  
print(f(3))
```

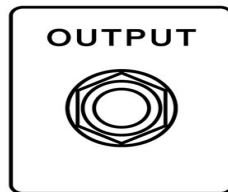


This will print  
[1]  
[1, 2]  
[1, 2, 3]

## Default Argument Values

- This is how you can prevent default values to be shared between subsequent calls,

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

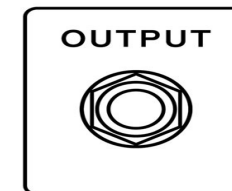


This will print  
[1]  
[2]  
[3]

## Usage of \*\*kwargs

- \*\*kwargs allows you to pass keyworded variable length of arguments to a function.

```
def bar(first, second, third, **options):  
    if options.get("action") == "sum":  
        print("The sum is: %d" % (first + second + third))  
  
    if options.get("number") == "first":  
        return first  
  
result = bar(1, 2, 3, action = "sum", number = "first")  
print ("Result: %d" % result)
```

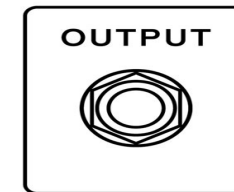


The sum is: 6  
Result: 1

## Usage of \*args

- \*args is used to send a non-keyworded variable length argument list to the function.

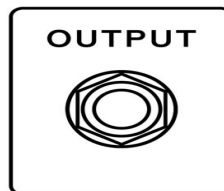
```
def foo(first, second, third, *therest):  
    print("First: %s" % first)  
    print("Second: %s" % second)  
    print("Third: %s" % third)  
    print("And all the rest... %s" % list(therest))  
  
foo('one','two','three','1','2','3','4')
```



```
First: one  
Second: two  
Third: three  
And all the rest... ['1', '2', '3', '4']
```

## Usage of \*\*kwargs

```
def test_var_kwargs(farg, **kwargs):  
    print ("formal arg:", farg)  
    for key in kwargs:  
        print ("another keyword arg: %s: %s" % (key, kwargs[key]))  
  
test_var_kwargs(farg=1, myarg2="two", myarg3=3)
```

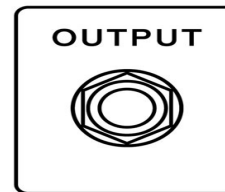


formal arg: 1  
another keyword arg: myarg2: two  
another keyword arg: myarg3: 3

## Using \*args when calling a function

- This special syntax can be used, not only in function definitions, but also when calling a function.

```
def test_var_args_call (arg1, arg2, arg3):  
    print("arg1:", arg1)  
    print("arg2:", arg2)  
    print("arg3:", arg3)  
  
args = ("two", 3)  
test_var_args_call('blahblah', *args)
```

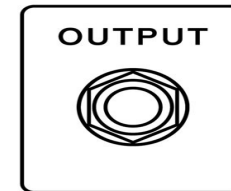


```
arg1: blahblah  
arg2: two  
arg3: 3
```

## Using **\*\*kwargs** when calling a function

```
def test_var_args_call(arg1, arg2, arg3):  
    print ("arg1:", arg1)  
    print ("arg2:", arg2)  
    print ("arg3:", arg3)
```

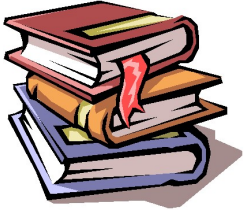
```
kwargs = {"arg3": 3, "arg2": "two"}  
test_var_args_call('blahblah', **kwargs)
```



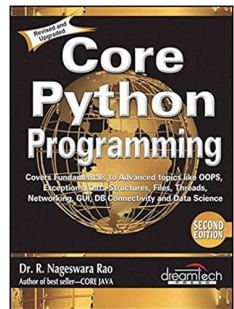
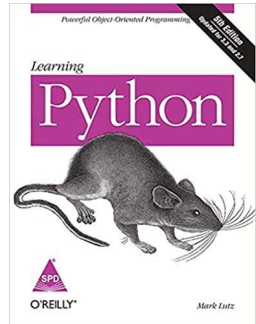
```
arg1: blahblah  
arg2: two  
arg3: 3
```



## References



- Python 3.x.x documentation: <https://docs.python.org/3/>
- Learning Python: Powerful Object-Oriented Programming: 5th Edition
- Core Python Programming [by [R. Nageswara Rao](#) (Author)]



# Thank you

Learning & Development

Tata Elxsi

Bangalore

[www.tataelxsi.com](http://www.tataelxsi.com)

---

**Confidentiality Notice**

This document and all information contained herein is the sole property of Tata Elxsi Limited and shall not be reproduced or disclosed to a third party without the express written consent of Tata Elxsi Limited.

---

**TATA ELXSI**