The Practice of Computing Using

PYTHON

William Punch Richard Enbody

Chapter 7

More on Functions

Addison-Wesley
is an imprint of
PEARSON

Functions Calling Functions

Functions Can Call Functions

- This was first mentioned in chapter 5.
- Functions are made to solve a problem and can be called from other functions.
- Functions calling functions does not do anything we haven’t already seen, but it can make following the flow of a program more difficult.

Example: String isdigit() method

- The isdigit() method returns True if a string contains only digits:
  - Works for integers.
  - Doesn't work for floating-point numbers or negative integers.
- Can we now write a function which cleans text and then calls isdigit() to determine if the text is a floating-point number?

```
def isFloat(aStr):
    """True if aStr is a positive float: digits and at
most one decimal point"""
    print "*** In the isFloat function.”
    # remove the decimal point
    stripped = aStr.replace('.','',1)
    # only digits should remain
    return stripped.isdigit()
```

Example: String isdigit() method

- Now can we write a function to repeatedly prompt the user for a valid floating-point number?

```
def readFloat(prompt):
    """Keep reading until a valid float is entered"""
    print " *** In readFloat function.”
    num_str = raw_input(prompt)
    # keep asking until valid float
    while not isFloat(num_str):
        print ’Invalid float, try again'
        num_str = raw_input(prompt)
    return float(num_str)
```

Chaining Functions

- isFloat checks to see if a string can be converted to a float number.
- readFloat uses isFloat as part of the process of prompting until a float is returned by the user.
- There is no limit to the "depth" of multiple function calls.

First Cut, Scope

Defining Scope

"The set of program statements over which a variable exists, i.e. can be referred to."

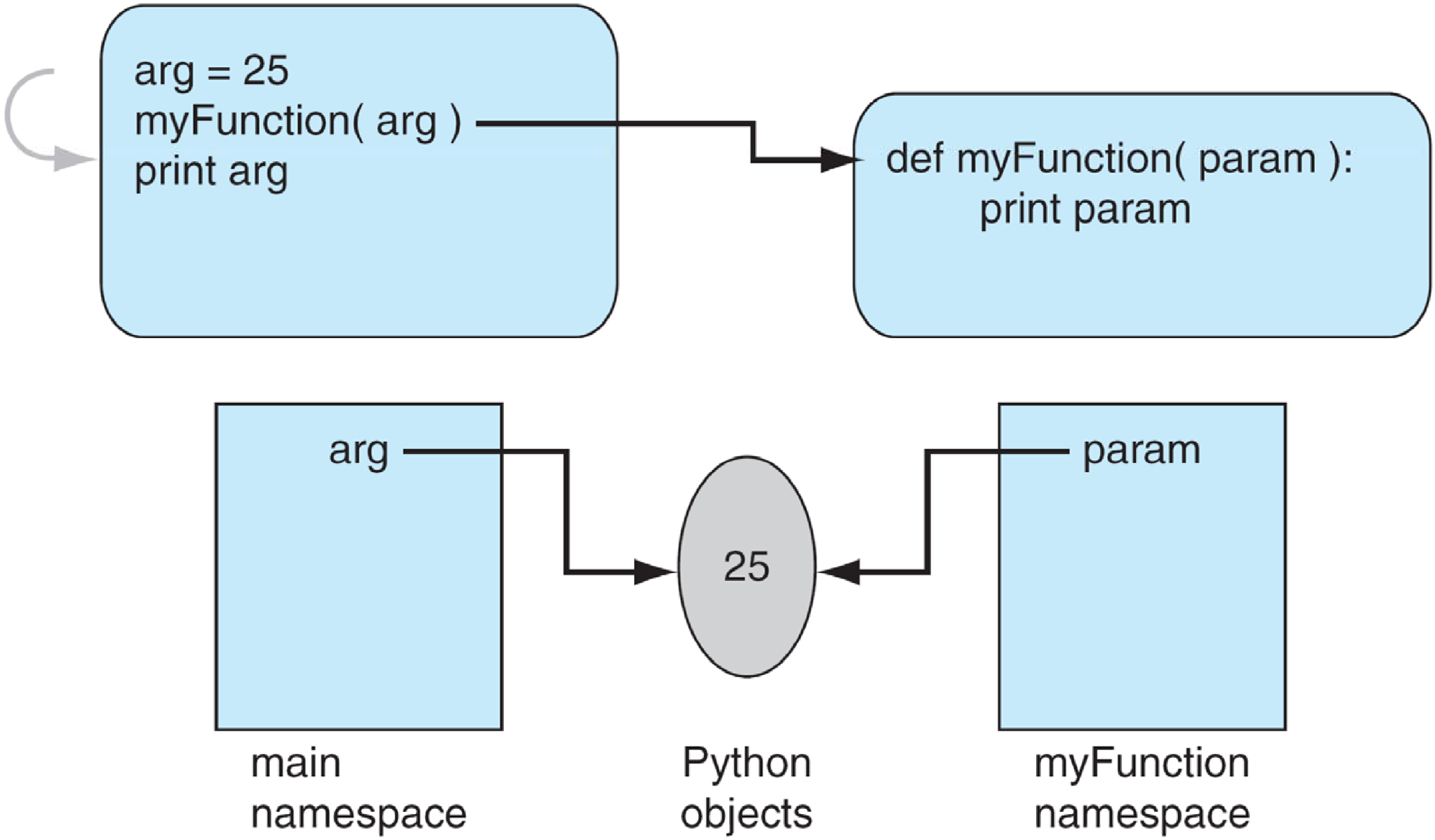
- It is about understanding, for any variable, what its associated value is.
- The problem is that multiple namespaces might be involved.

A Function’s Namespace

- Each function call maintains a namespace for names defined **locally within the function**.
- Locally means one of two things:
  - a name assigned within the function
  - an argument received by invocation of the function

Passing Argument to Parameter

- For each argument in the function invocation, the argument's associated object is passed to the corresponding parameter in the function.



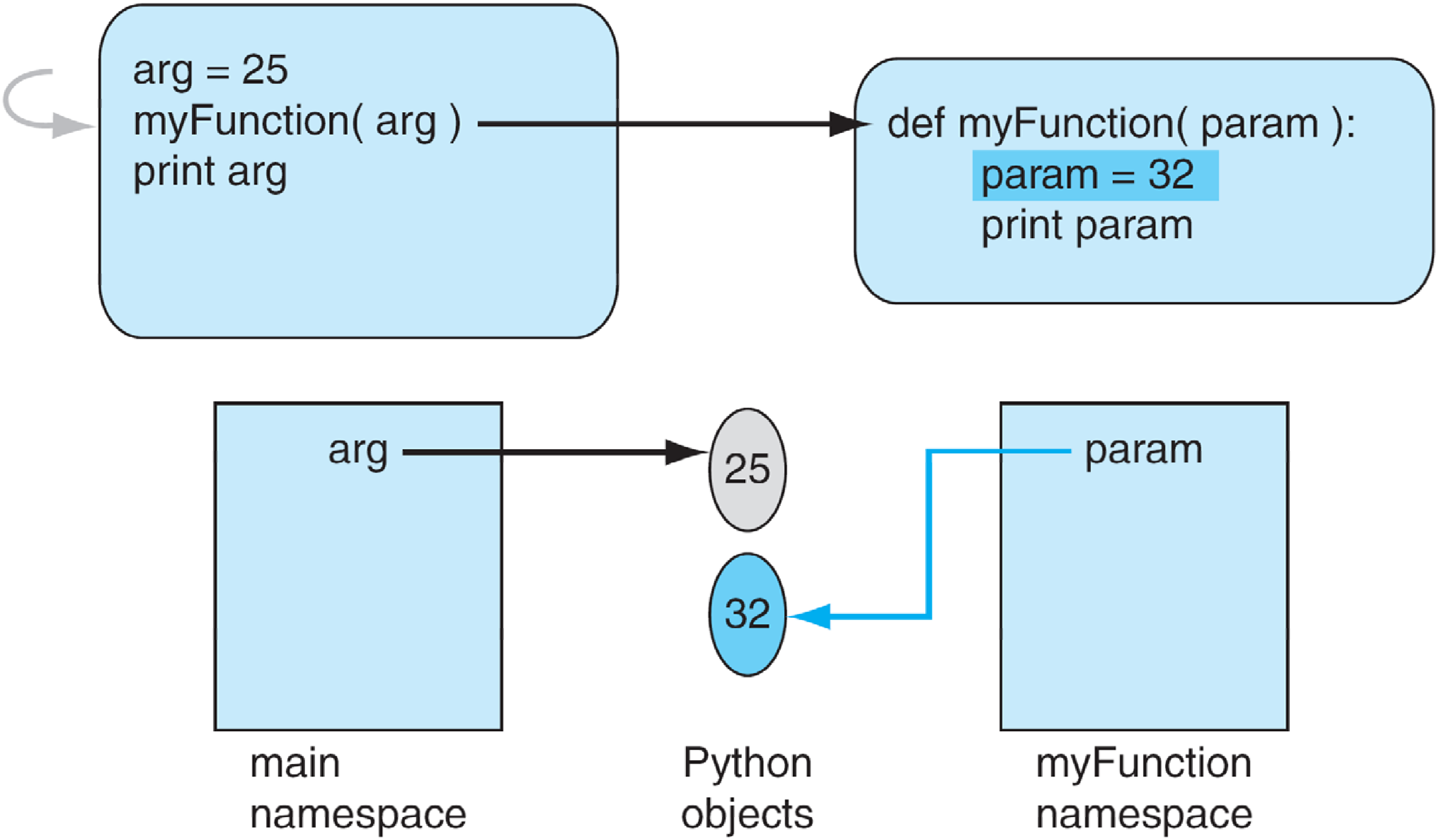
```
arg = 25
myFunction( arg )
print arg
```

```
def myFunction( param ):
        print param
```

FIGURE 7.1 Function namespace: at function start.

Assignment Changes Association

- If a parameter is assigned to a new value, then just like any other assignment, a new association is created.
- This assignment does not affect the object associated with the argument, as a new association was made with the parameter.



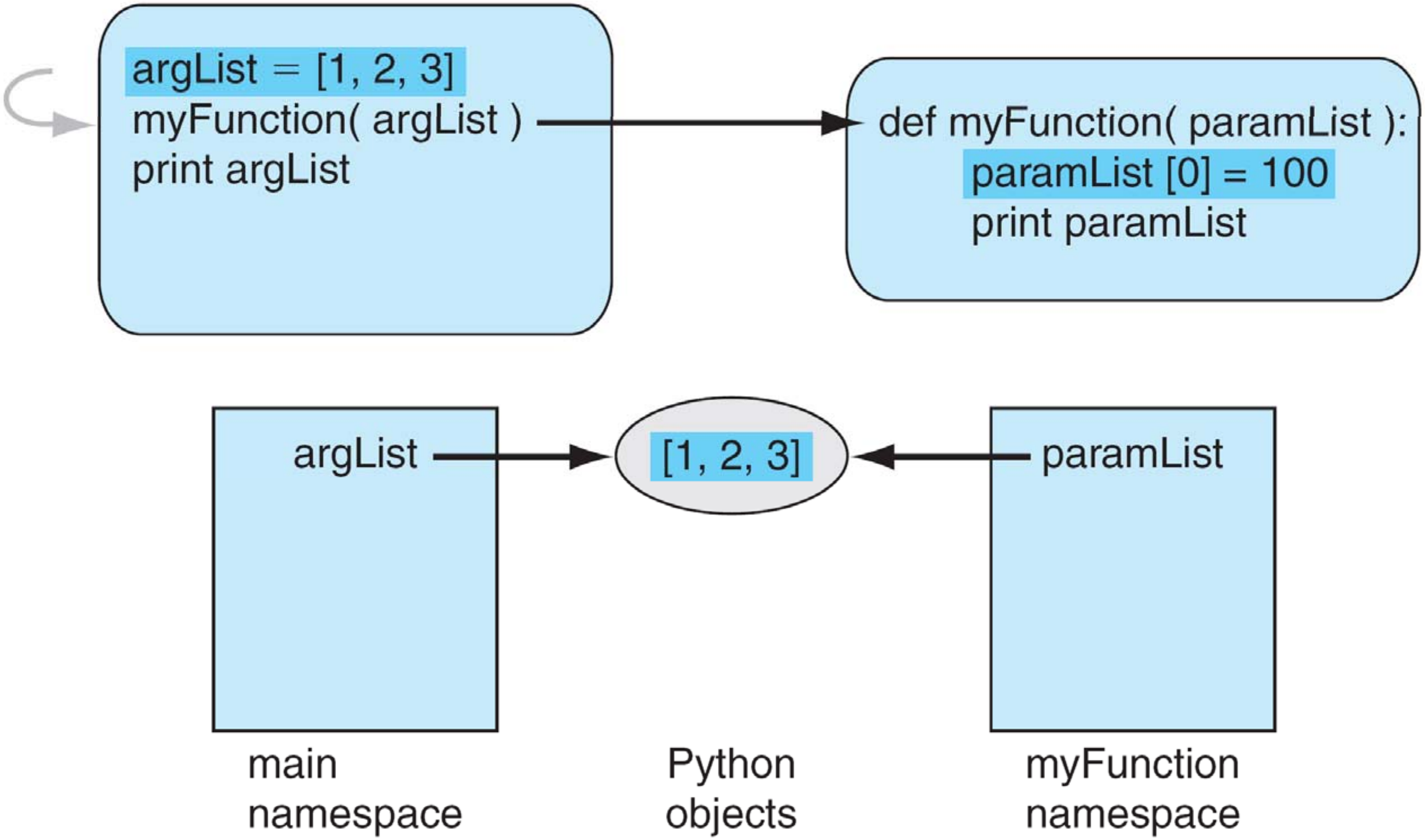
```
arg = 25
myFunction( arg )
print arg
```

```
def myFunction( param ):
    param = 32
    print param
```

FIGURE 7.2 Function namespace modified.

Sharing Mutables

- When passing a mutable data structure, it is possible that if the shared object is directly modified, both the parameter and the argument will reflect that change.
- Note that the operation must be a mutable change, a change of the object. An assignment is not such a change.



```
argList = [1, 2, 3]
myFunction( argList )
print argList
```

```
def myFunction( paramList ):
    paramList [0] = 100
    print paramList
```

FIGURE 7.3 Function namespace with mutable objects: at function start.

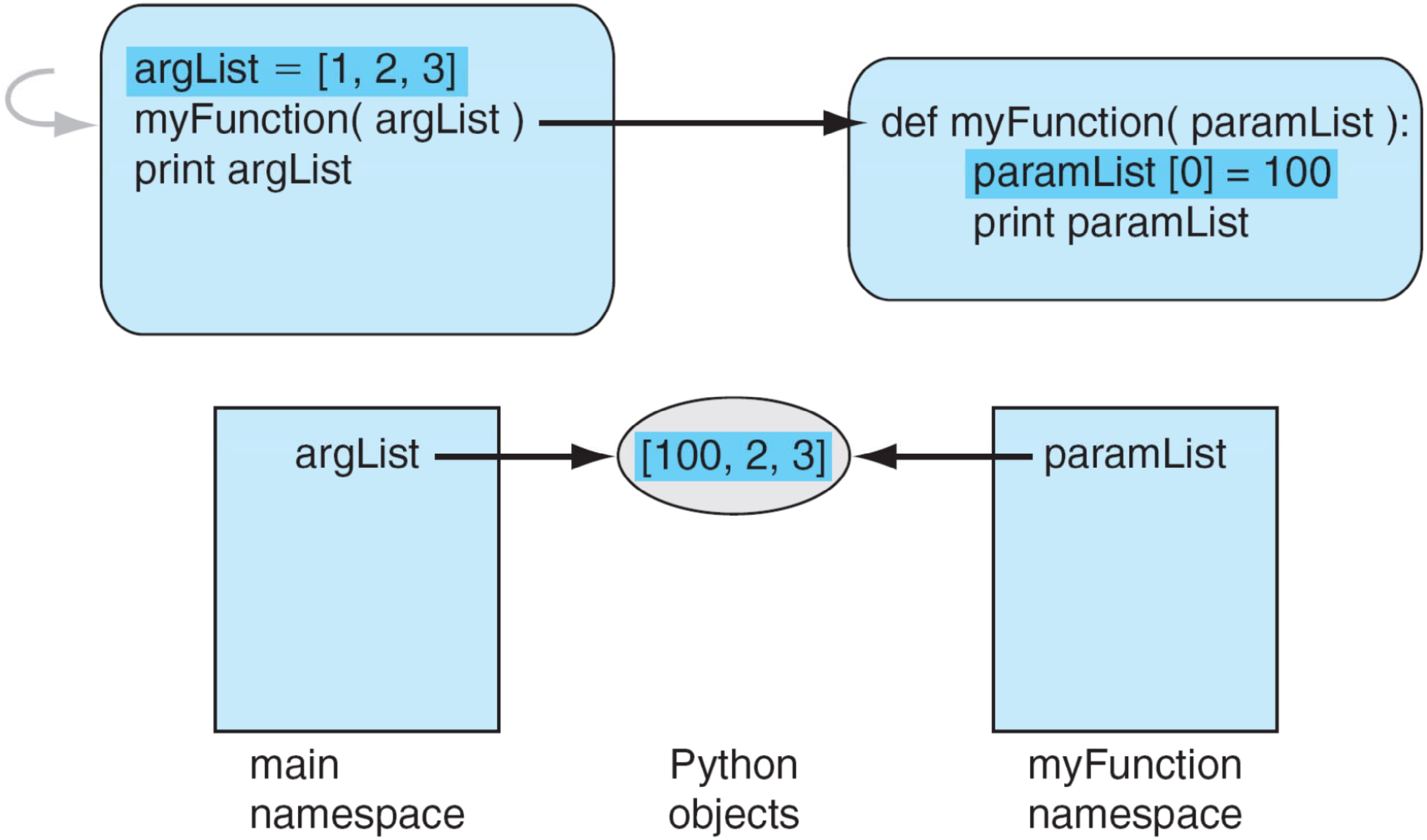


FIGURE 7.4 Function namespace with mutable objects after `paramList[0]=100`.

A Function Which Only Takes Mutables

```
>>> def foo(a):
            a[1]='x'
            return a

>>> foo(2)                  # Error
>>> foo('abc')                   # Error
>>> foo([1,2,3])
[1, 'x', 3]
>>> foo((1,2,3))            # Error
```

More on Functions

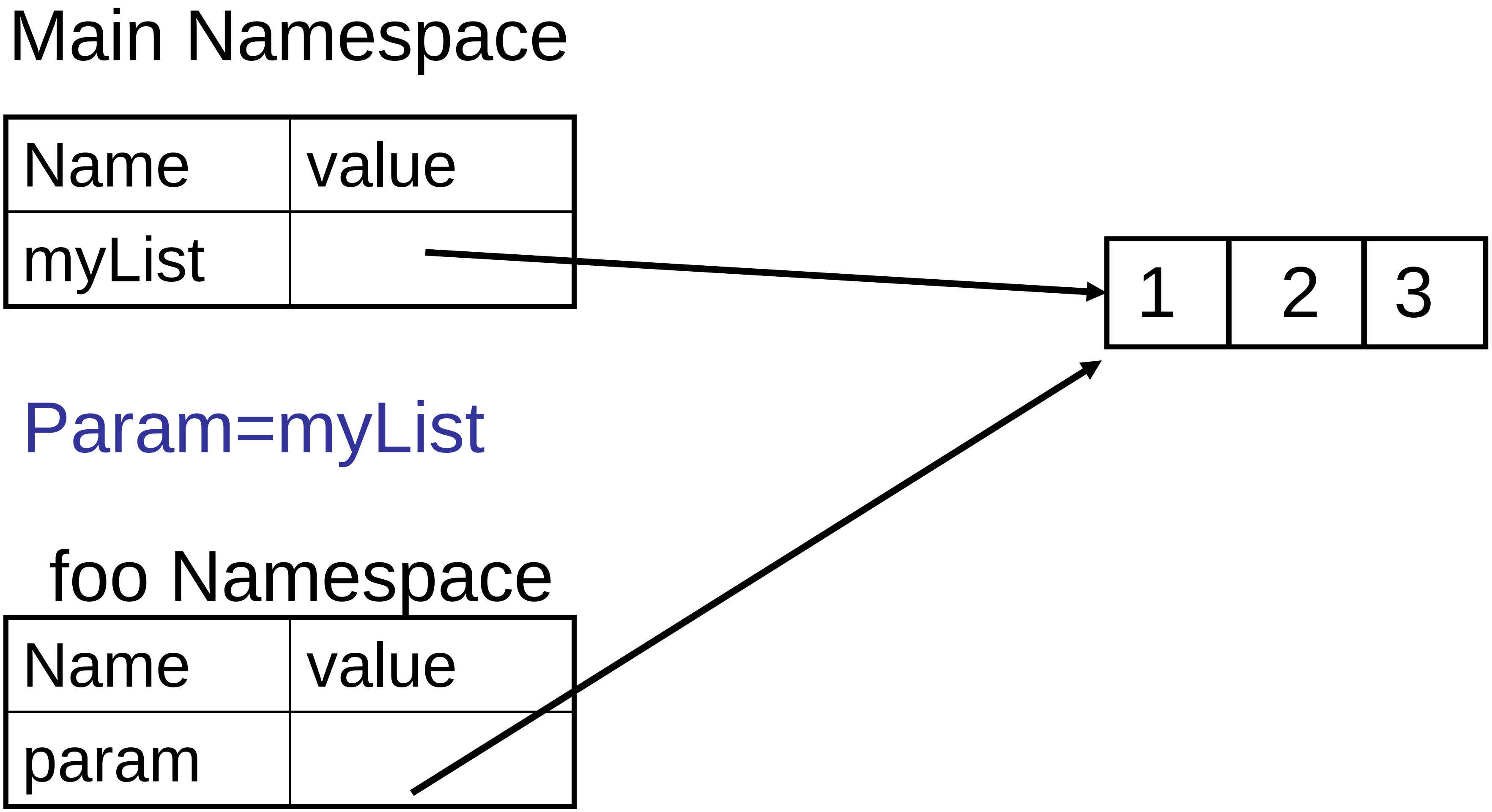
Assignment in a Function

- If you assign a value in a function, that name becomes part of the local namespace of the function.
- It can have some odd effects.

Example

```
def myFun (param):
   param.append(4)
   return param

myList = [1,2,3]
newList = myFun(myList)
print myList,newList
```



Main Namespace

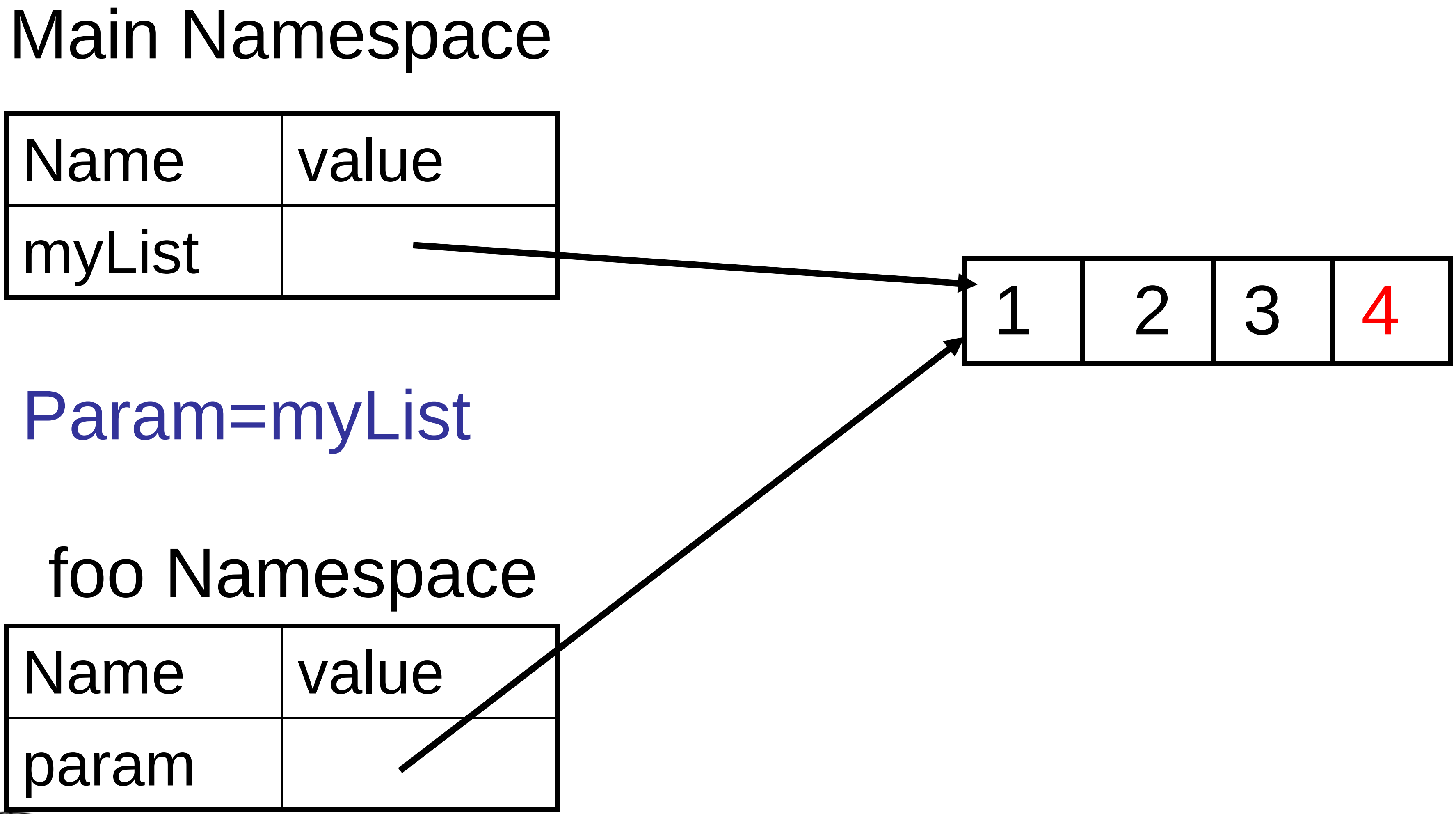
Name	value
myList	

1	2	3

Param=myList

foo Namespace

Name	value
param	



Main Namespace

Name	value
myList	

Param=myList

foo Namespace

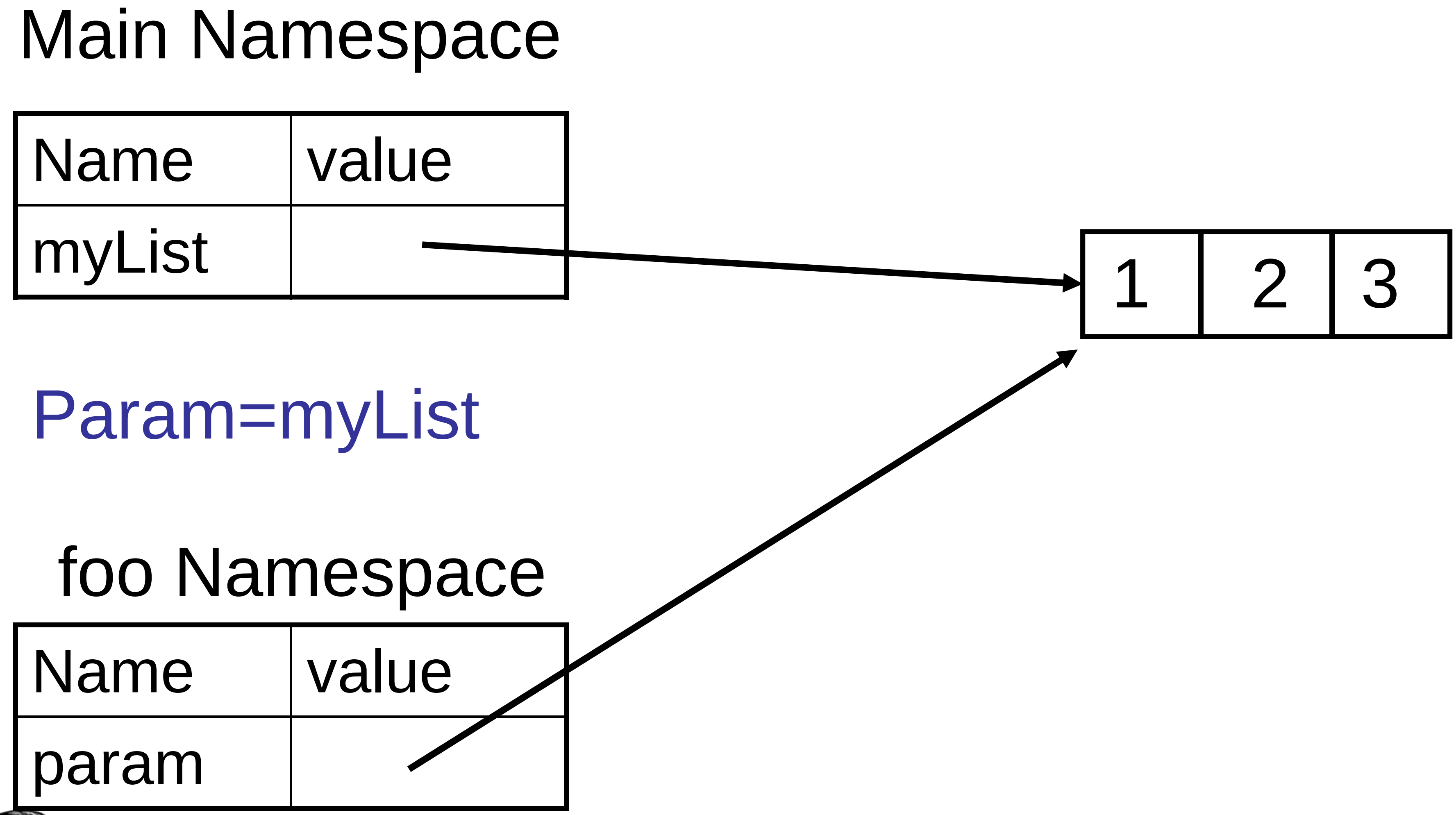
Name	value
param	

1	2	3	4

Example

```
def myFun (param):
    param=[1,2,3]
    param.append(4)
    return param

myList = [1,2,3]
newList = myFun(myList)
print myList,newList
```



Main Namespace

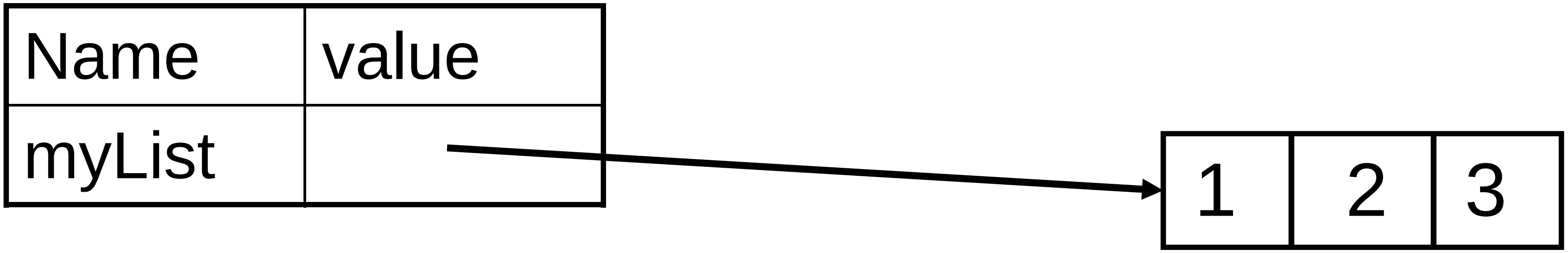
Name	value
myList	

Param=myList

foo Namespace

Name	value
param	

Main Namespace

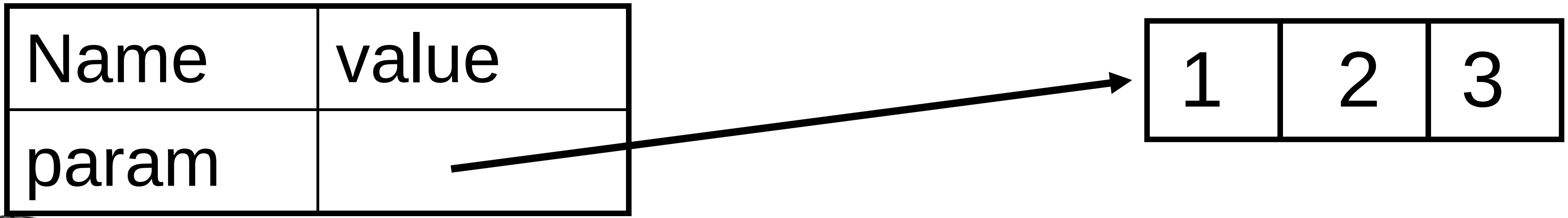


Name	value
myList	

1	2	3

Param=myList

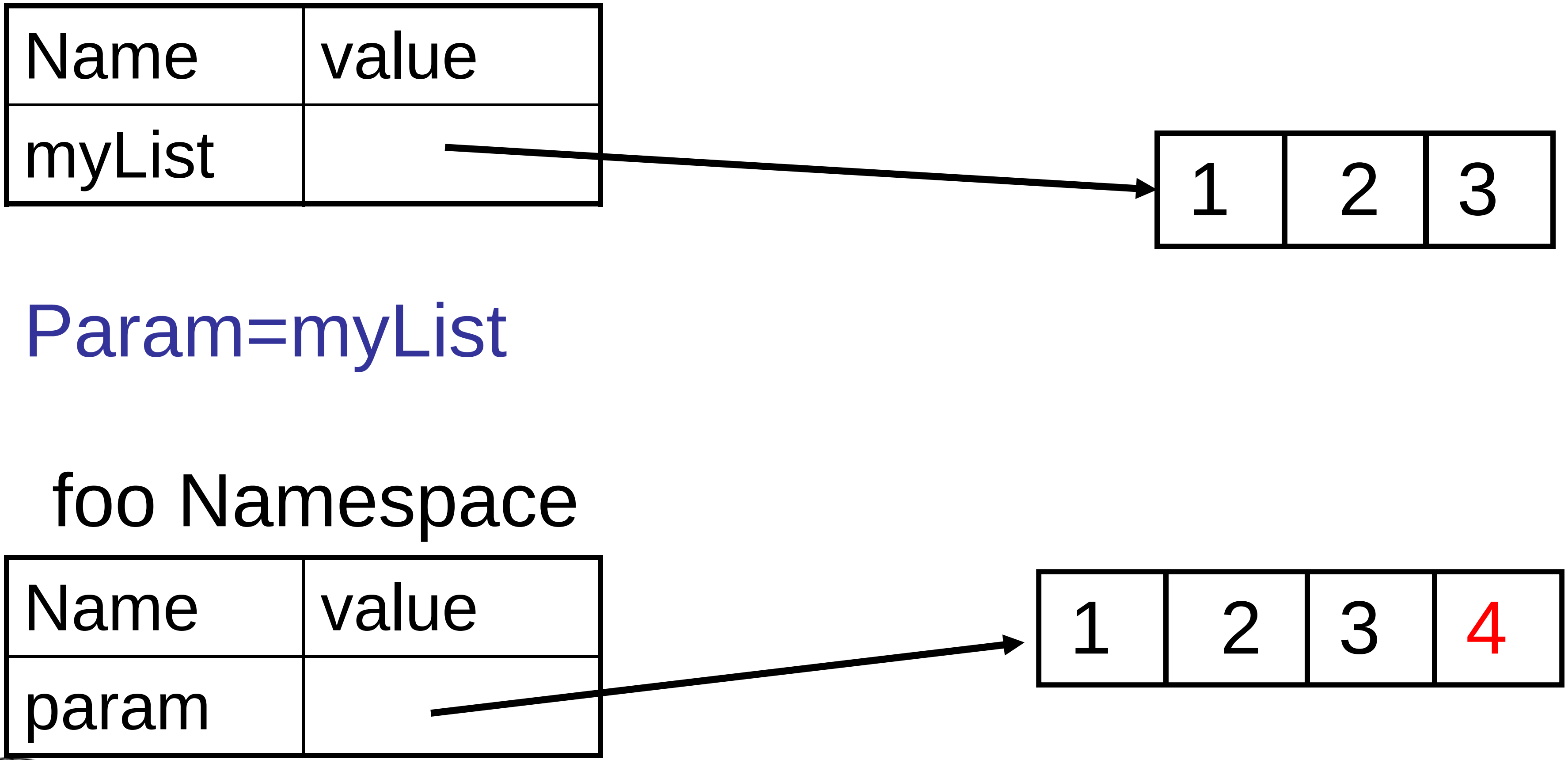
foo Namespace



Name	value
param	

1	2	3

Main Namespace



Name	value
myList	

1	2	3

Param=myList

foo Namespace

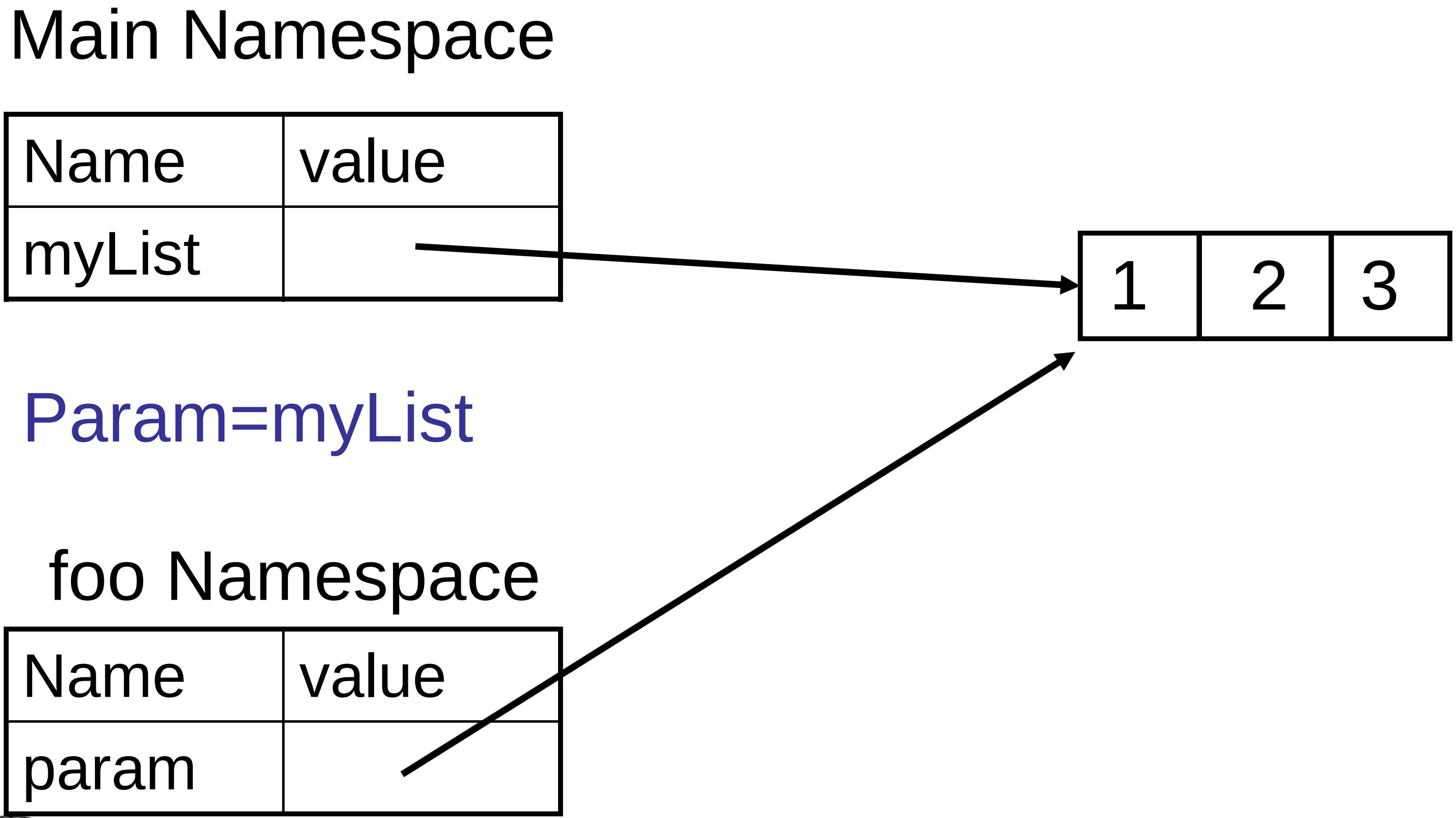
Name	value
param	

1	2	3	4

Example

```
def myFun (param):
   param=param.append(4)
   return param

myList = [1,2,3]
newList = myFun(myList)
print myList,newList
```



Main Namespace

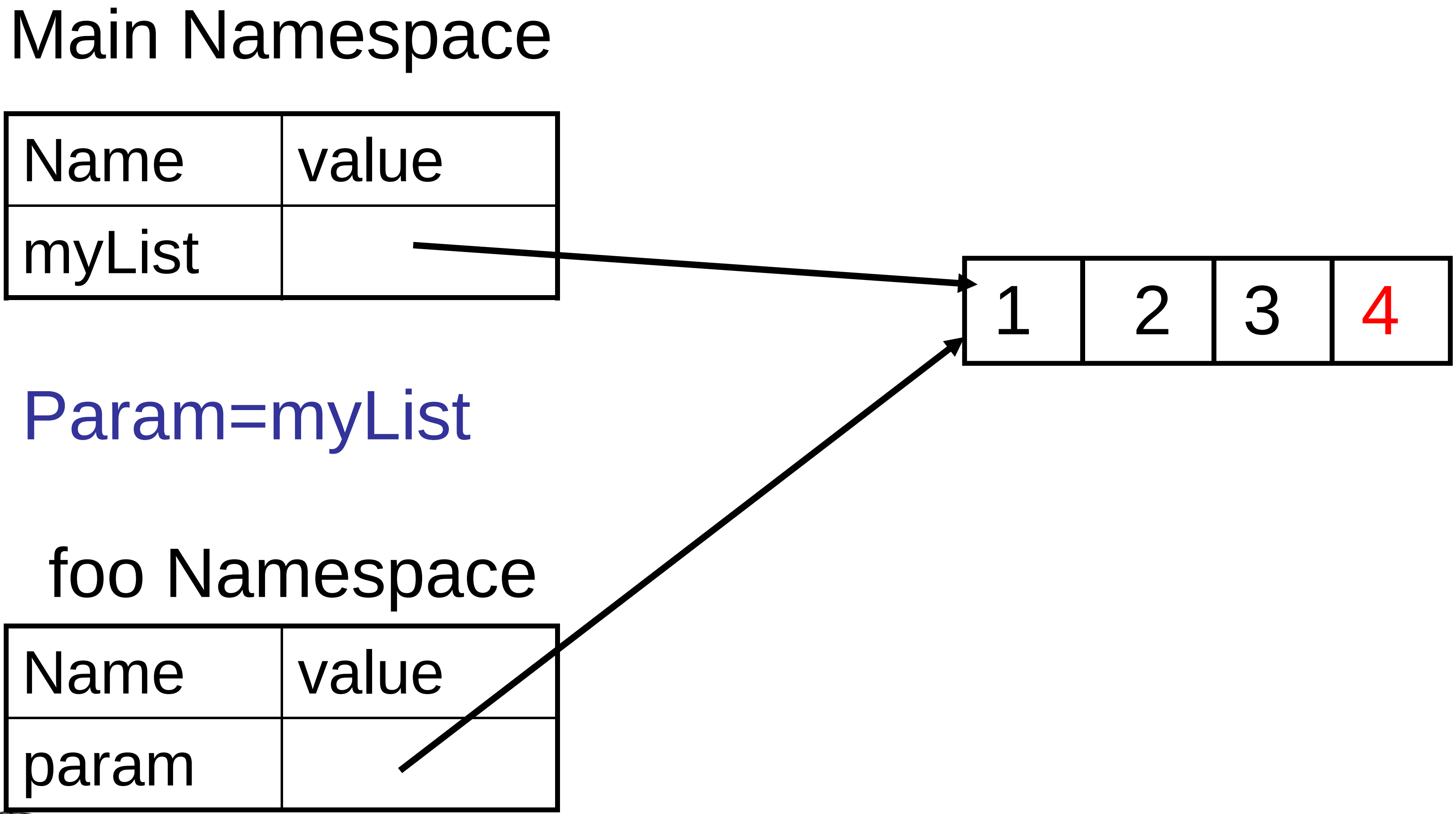
Name	value
myList	

Param=myList

foo Namespace

Name	value
param	

1	2	3



Main Namespace

Name	value
myList	

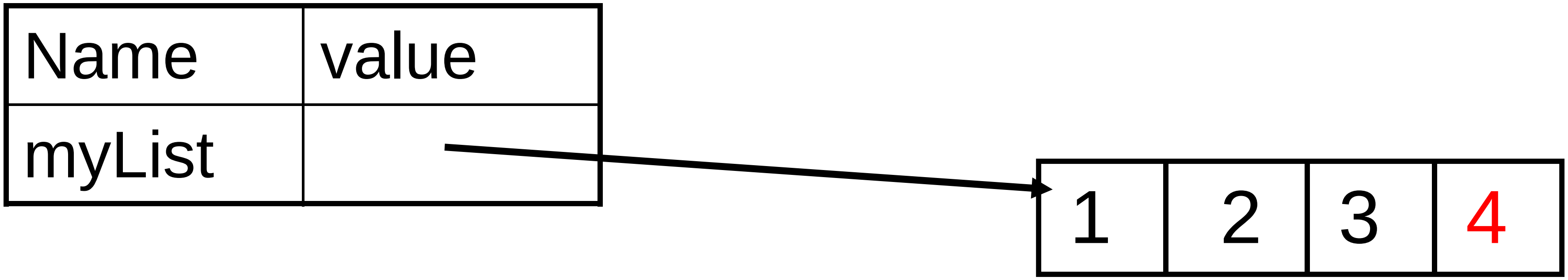
Param=myList

foo Namespace

Name	value
param	

1	2	3	4

Main Namespace



Name	value
myList	

1	2	3	4

Param=myList

foo Namespace

Name	value
param	None

Assignment to a Local

- Assignment creates a local variable.
- Changes to a local variable affect only the local context, even if it is a parameter and mutable.
- If a variable is assigned locally, you cannot reference it before this assignment, even if it exists in main as well.

Example

```
myList = [1,2,3]

def myFun():
   myList.append(4) # error!
   myList = [4, 5, 6]
   return myList

myFun()
```

Default Parameters

```
def box(height=10,width=10,depth=10,
        color= "blue“ ):
         ... do something ...
```

If the caller does not provide a value, the default is the parameter assigned value

Defaults

```
def box (height=10,width=10,length=10):
    print height,width,length

box()        # prints 10 10 10
```

Named Arguments

```
def box (height=10,width=10,length=10):
    print height,width,length


box(length=25,height=25)
   # prints 25 10 25


box(15,15,15)  # prints 15 15 15
```

Name Use Works in General Cases

```
def foo(a,b):
    print a,b

foo(1,2)          # prints 1 2
foo(b=1,a=2)      # prints 2 1
```

Default args and Mutables

- There's an issue with using mutables as default args. This is because:
  - the default value is created once, when the function is defined, and stored in the function name space
  - a mutable can change the value of that default

Weird…

```
def fn1 (arg1=[], arg2=27):
   arg1.append(arg2)
   return arg1

myList = [1,2,3]
print fn1(myList,4)         # [1, 2, 3, 4]
print fn1(myList)  # [1, 2, 3, 4, 27]
print fn1()                 # [27]
print fn1()                 # [27, 27]???
```

Functions Return One Thing

- Functions return one thing, but it can be a ‘chunky’ thing. For example, it can return a tuple.
- Thus, multiple things can be returned by being packed into a tuple or other data structure.

Functions Can Return Tuples

```
>>> def foo():
            a = 2
            b = 3
            return a,b

>>> T = foo()
>>> print T                  # (2, 3)
>>> print foo()              # (2, 3)
>>> x,y = foo()
>>> print x                  # 2
>>> print y                  # 3
```

Functions as Objects and Docstrings

Functions are Objects, Too!

- Functions are objects, just like anything else in Python.
- As such, they have attributes:
  - __name__ : function name
  - __doc__ : docstring

Can ask for Docstring

- Every object (function, whatever) can have a docstring. It is stored as an attribute of the function (the __doc__ attribute)
- listMean.__doc__
  - ‘Takes a list of integers, returns the average of the list.’
- Other programs can use the docstring to report to the user (for example, IDLE).