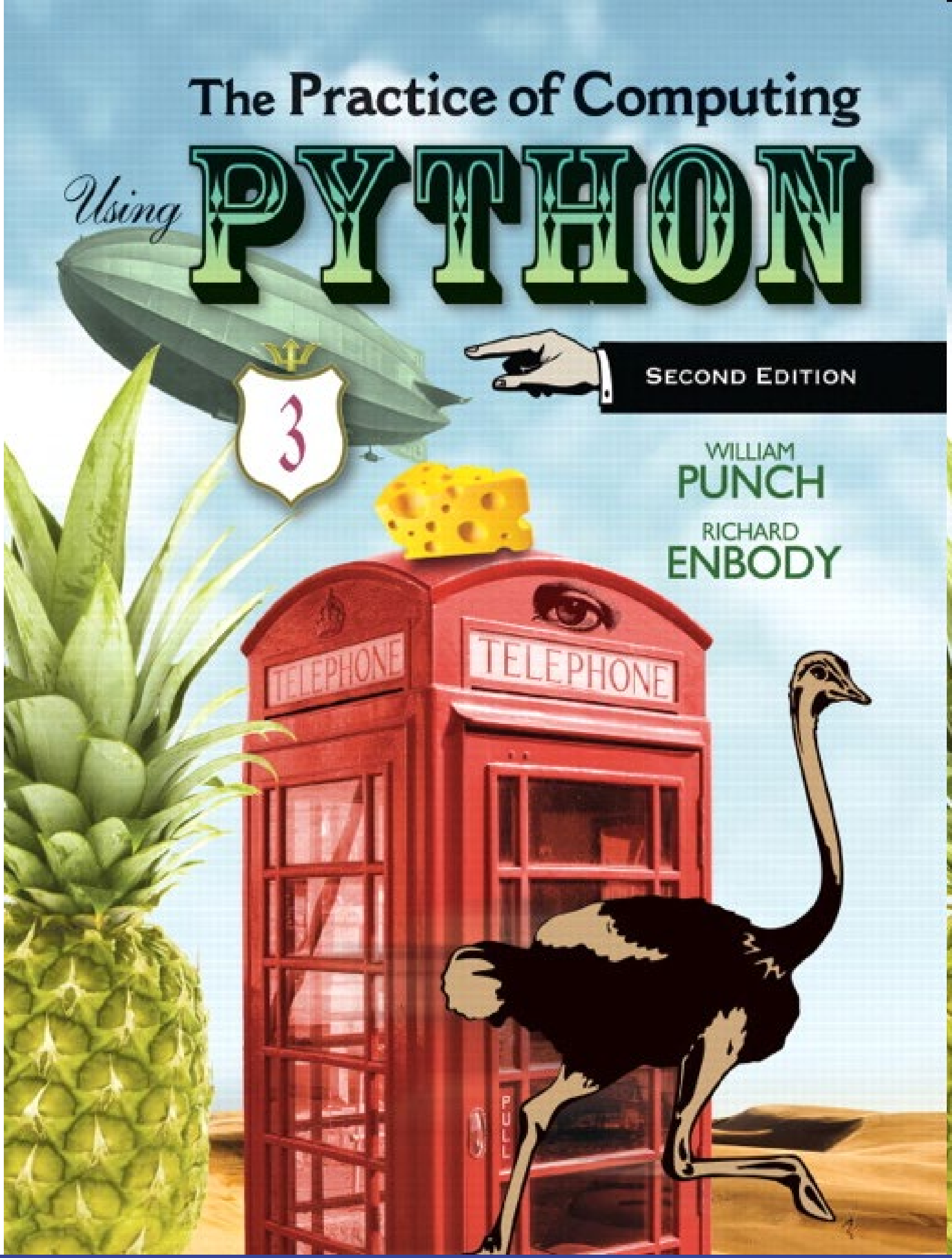
chapter 8

More On Functions



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

Find the namespace

- For Python, there are potentially multiple namespaces that could be used to determine the object associated with a variable.
- Remember, namespace is an association of name and objects
- We will begin by looking at functions.

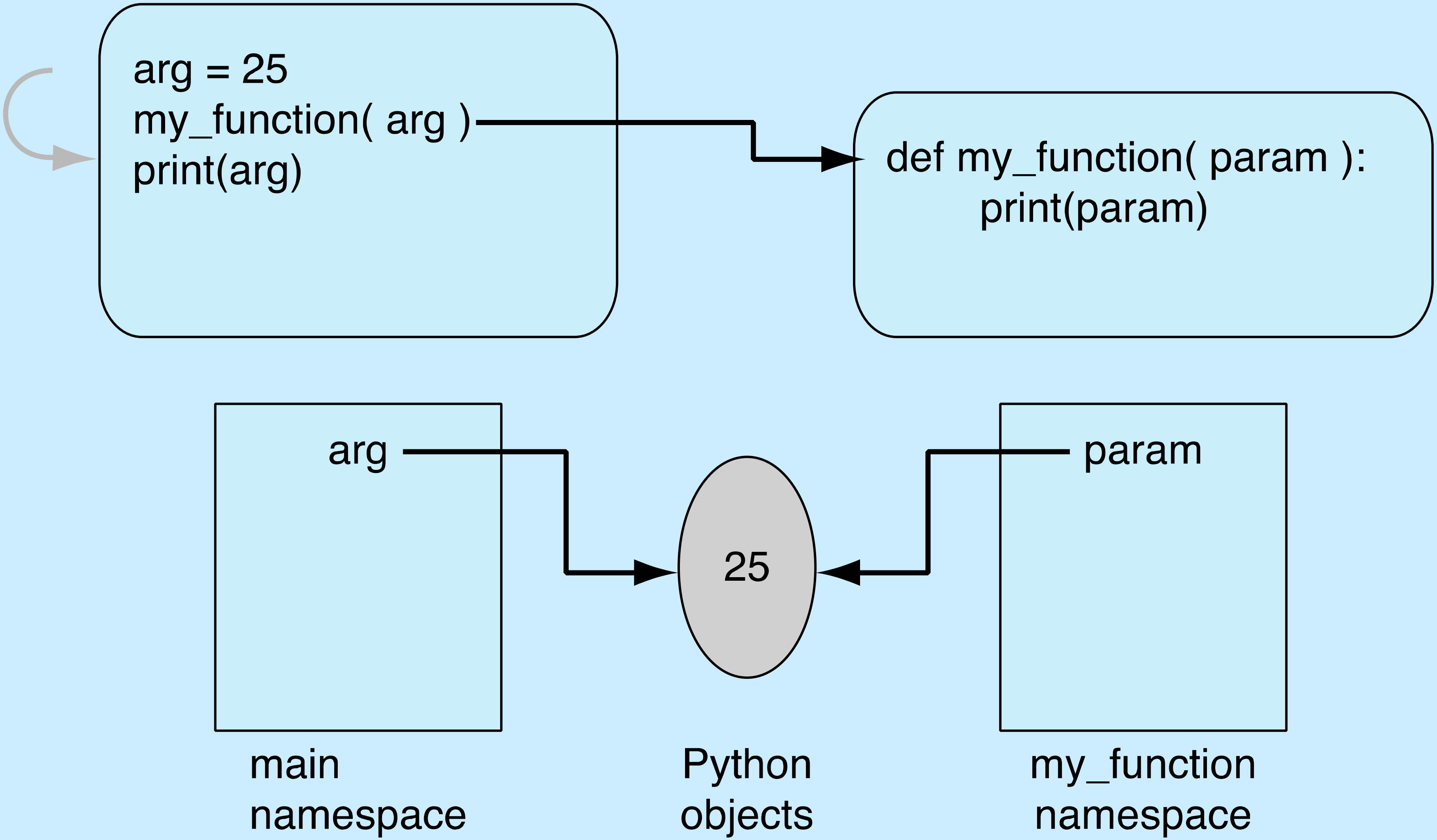
A function’s namespace

- Each function 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

Passing immutable objects



```
arg = 25
my_function( arg )
print(arg)
```

```
def my_function( param ):
    print(param)
```

arg

25

param

main namespace

Python objects

my_function namespace

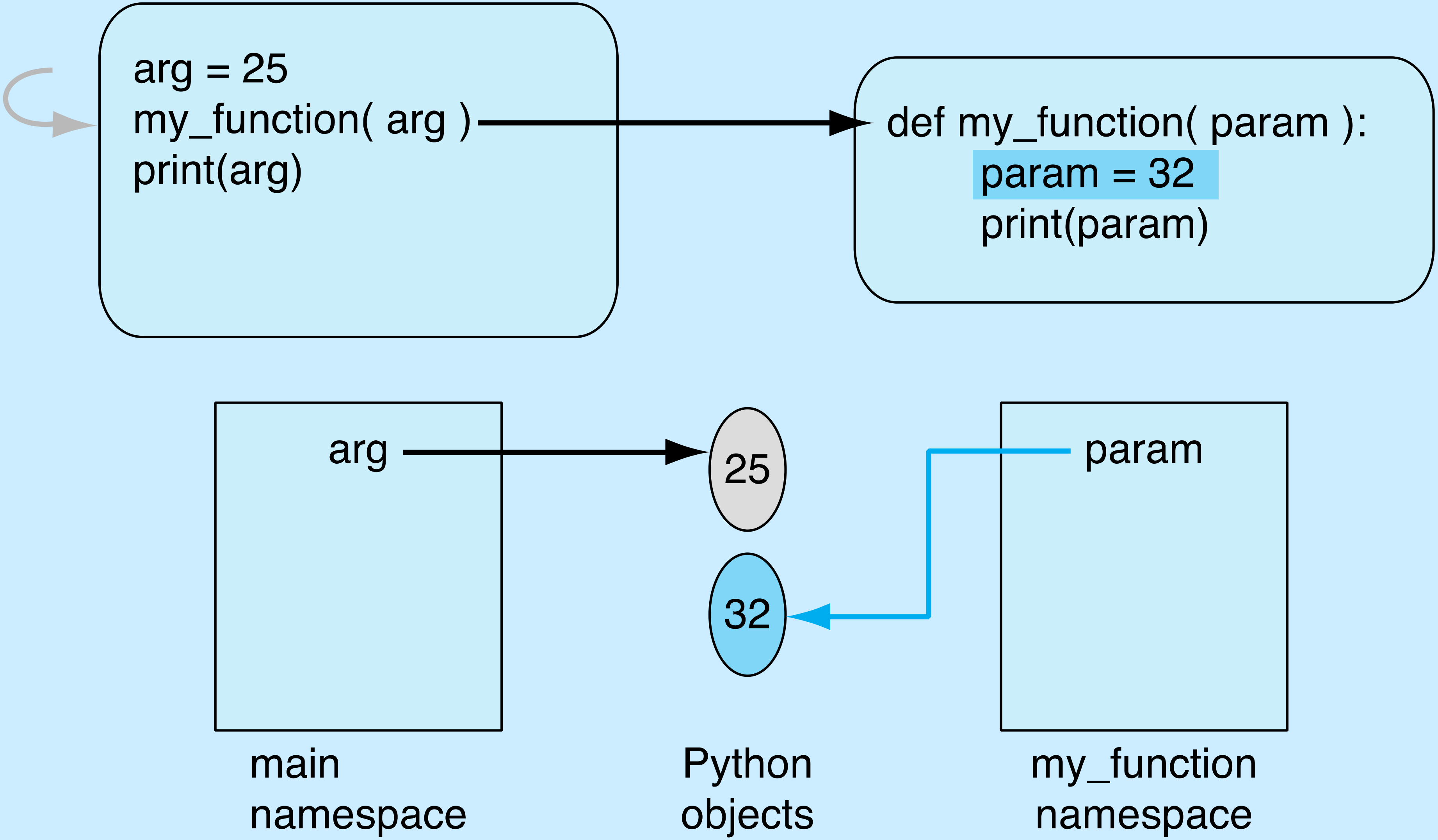
FIGURE 8.1 Function namespace: at function start.

What does “pass” mean?

- The diagram should make it clear that the parameter name is local to the function namespace
- Passing means that the argument and the parameter, named in two different namespaces, share an association with the same object
- So “passing” means “sharing” in Python

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
my_function( arg )
print(arg)
```

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

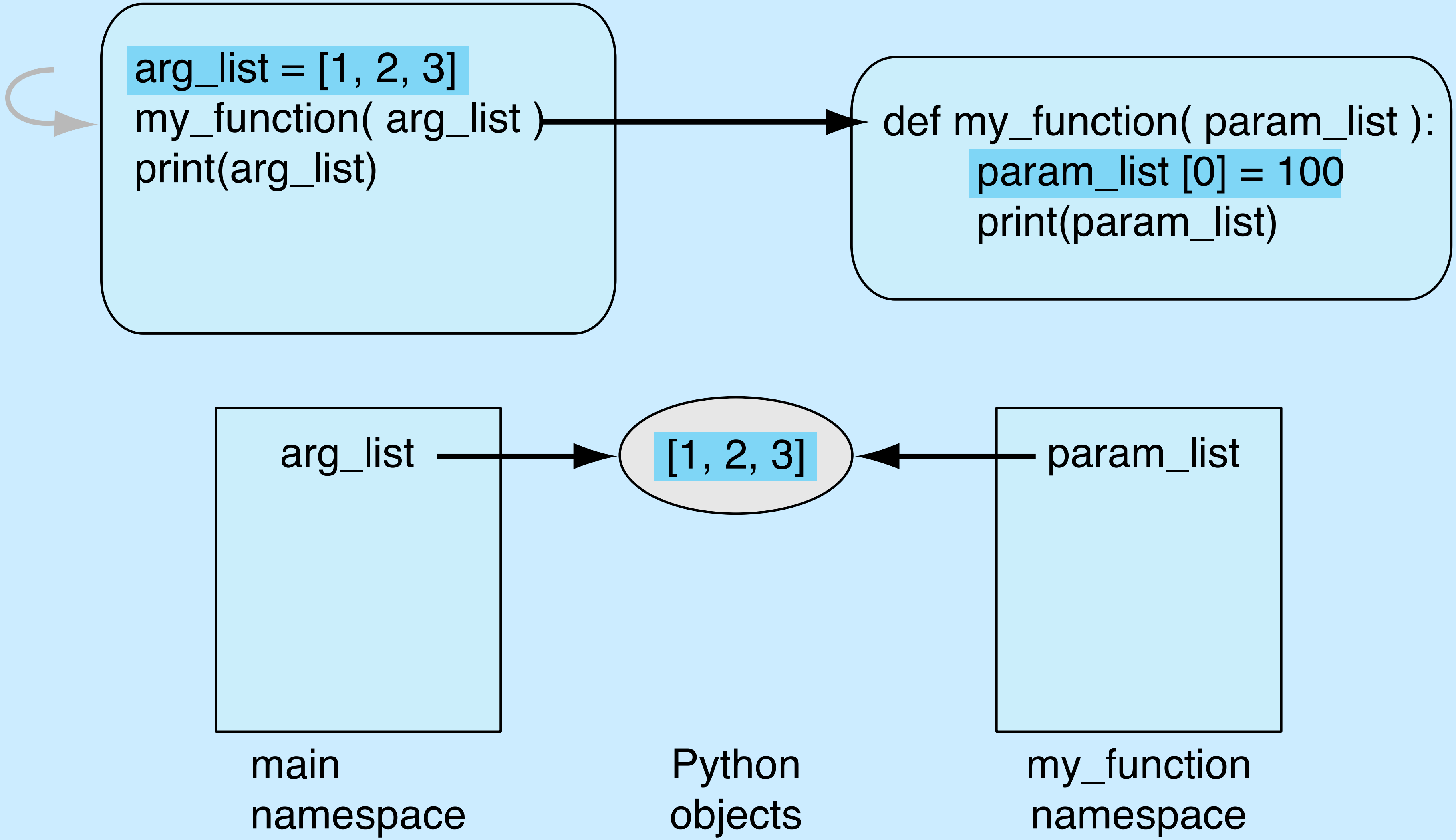
FIGURE 8.2 Function namespace modified.

addA.py

passing mutable objects

Sharing mutables

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



```
arg_list = [1, 2, 3]
my_function( arg_list )
print(arg_list)
```

```
def my_function( param_list ):
    param_list [0] = 100
    print(param_list)
```

FIGURE 8.3 Function namespace with mutable objects: at function start.

```
arg_list = [1, 2, 3]
my_function( arg_list )
print(arg_list)
```

```
def my_function( param_list ):
    param_list [0] = 100
    print(param_list)
```

FIGURE 8.4 Function namespace with mutable objects after `param_list[0]=100`.

"The Practice of Computing Using Python",
Punch & Enbody, Copyright © 2013 Pearson Education, Inc.

More on Functions

Functions return one thing

Functions return one thing, but it can be a ‘chunky’ thing. For example, it can return a tuple

```
>>> def mirror(pair):
      '''reverses first two elements;
         assumes "pair" is as a collection with at least two elements'''
      return pair[1], pair[0]
>>> mirror((2,3))
(3, 2)      # the return was comma separated: implicitly handled as a tuple
>>> first,second = mirror((2,3)) # comma separated works on the left—hand—side als
>>> first
3
>>> second
2
>>> first,second            # reconstruct the tuple
(3, 2)
>>> a_tuple = mirror((2,3)) # if we return and assign to one name, we get a tuple!
>>> a_tuple
(3, 2)
```

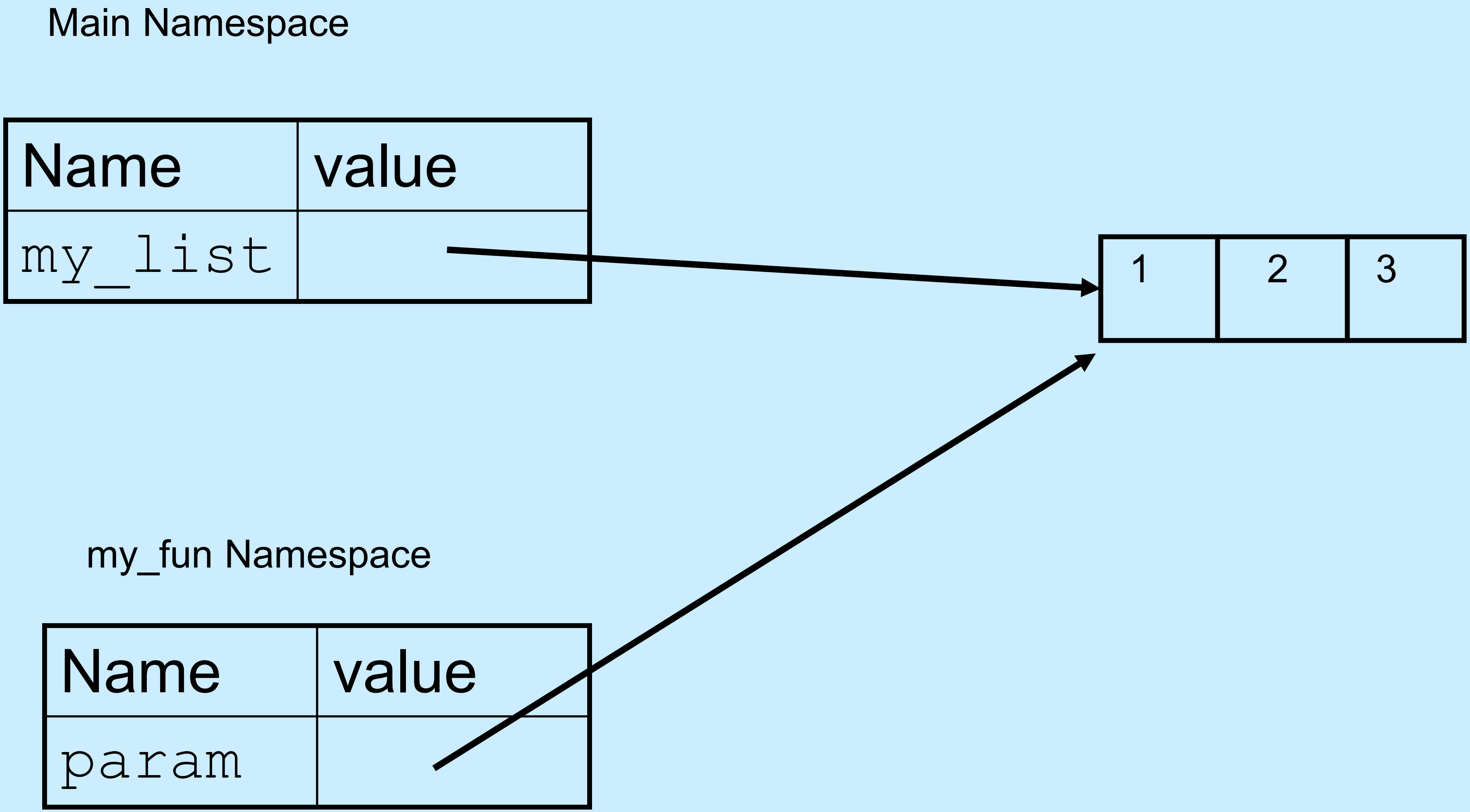
assignment in a function

- 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 my_fun (param):
    param.append(4)
    return param

my_list = [1,2,3]
new_list = my_fun(my_list)
print(my_list,new_list)
```



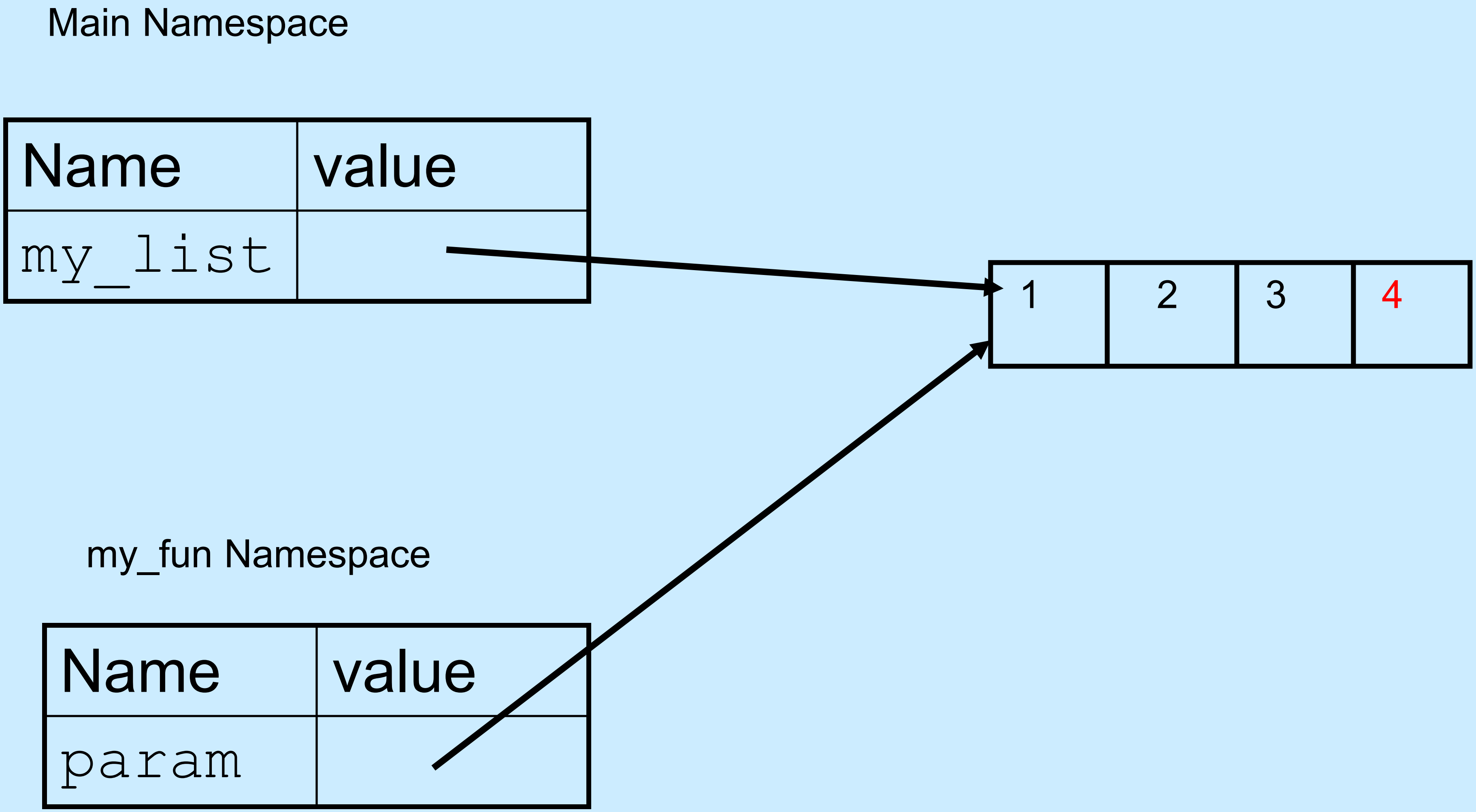
Main Namespace

Name	value
`my_list`	

my_fun Namespace

Name	value
`param`	

1	2	3



Main Namespace

Name	value
`my_list`	

my_fun Namespace

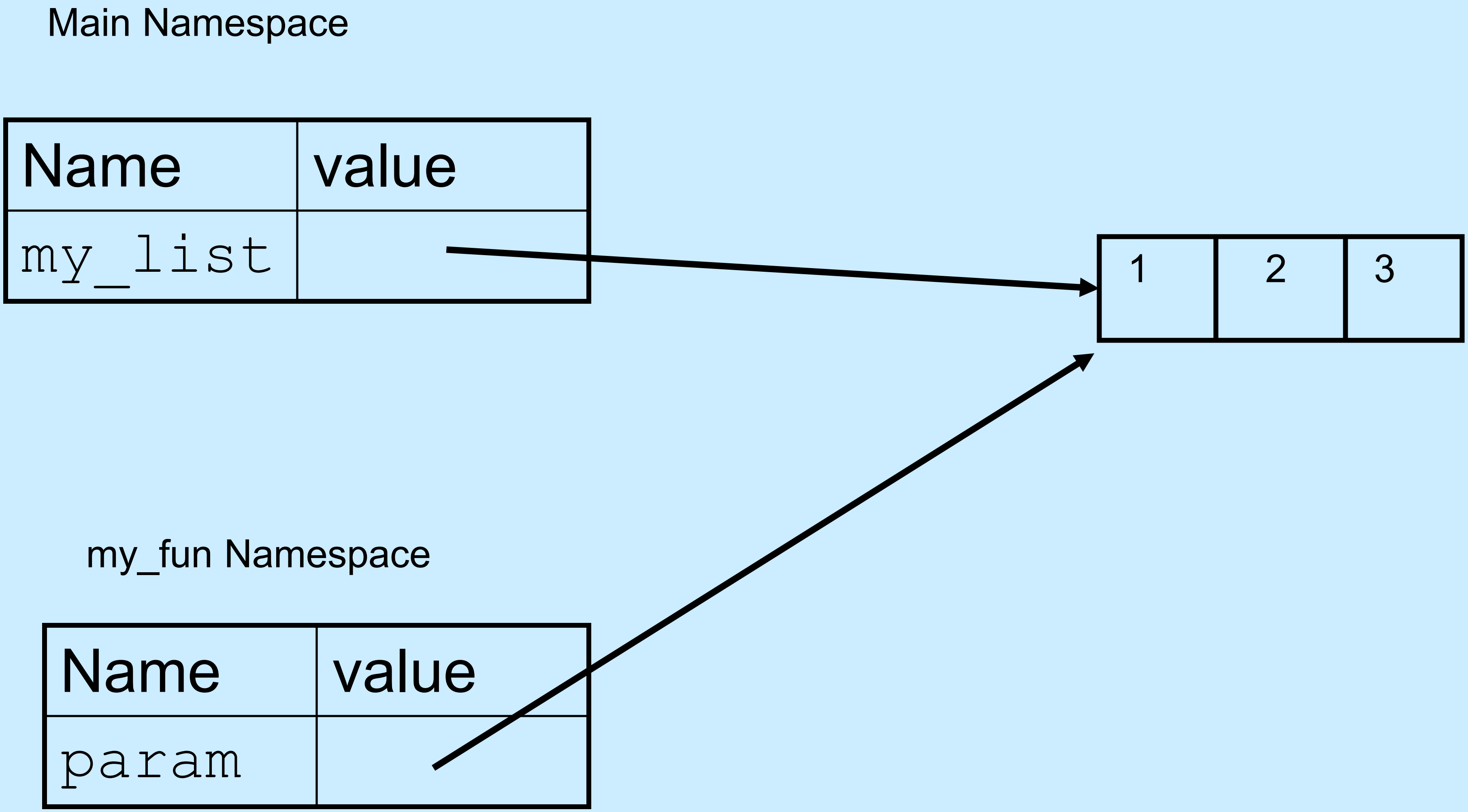
Name	value
`param`	

1	2	3	4

Example

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

my_list = [1,2,3]
new_list = my_fun(my_list)
print(my_list,new_list)
```



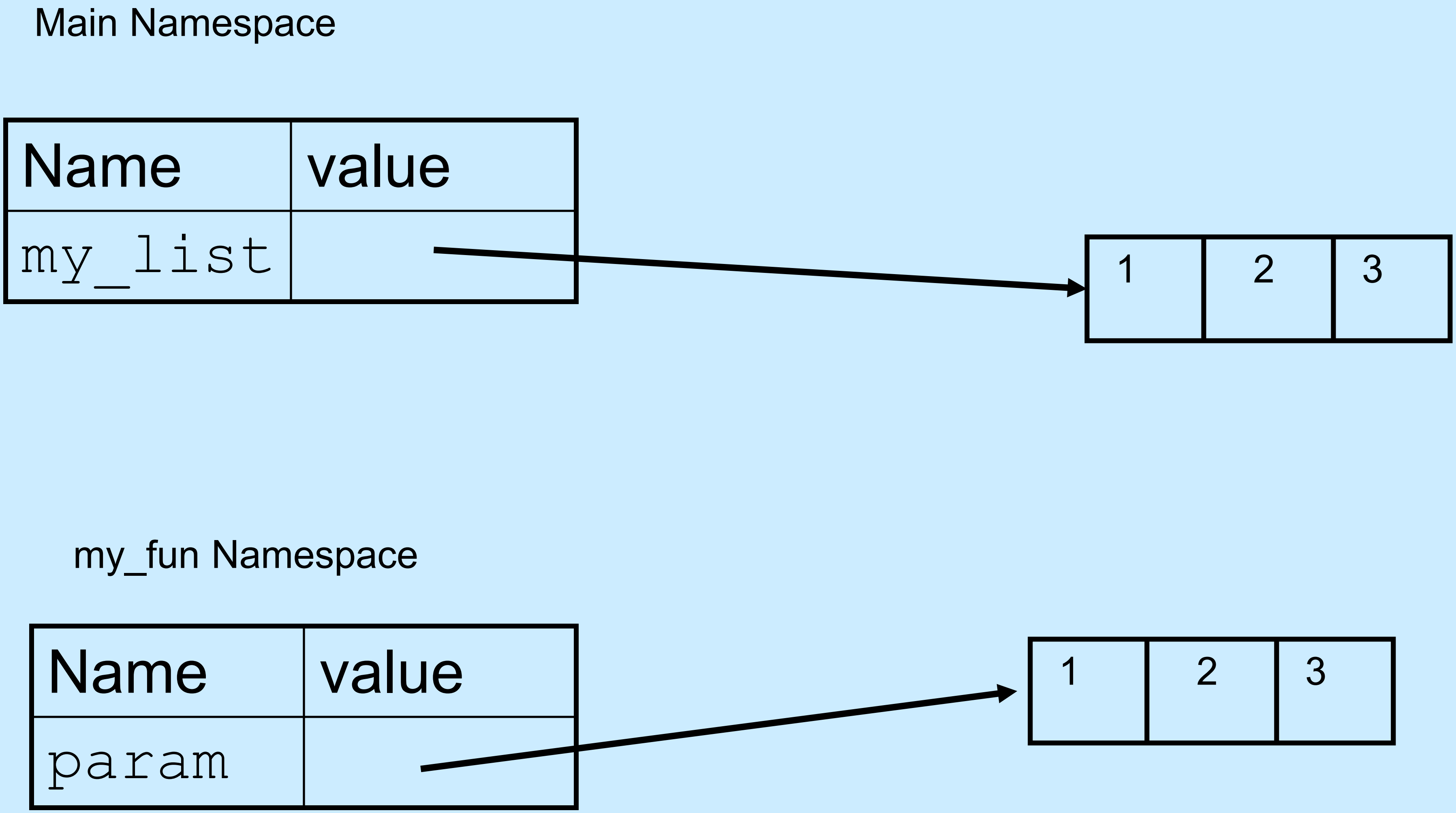
Main Namespace

Name	value
`my_list`	

1	2	3

my_fun Namespace

Name	value
`param`	



Main Namespace

Name	value
`my_list`	

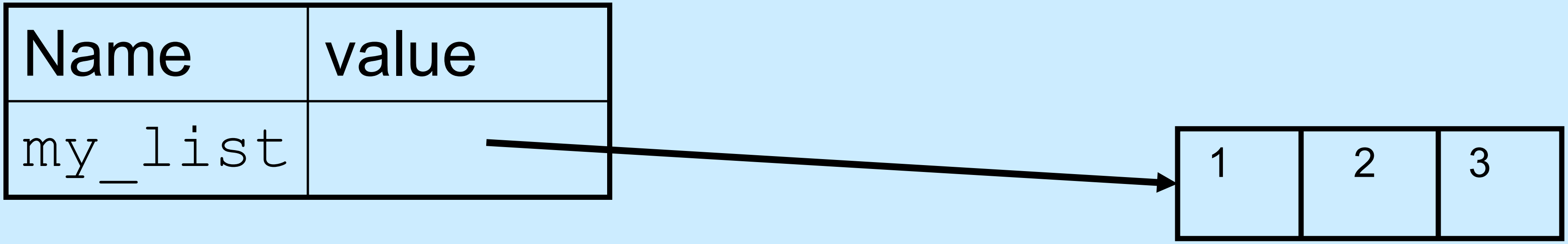
1	2	3

my_fun Namespace

Name	value
`param`	

1	2	3

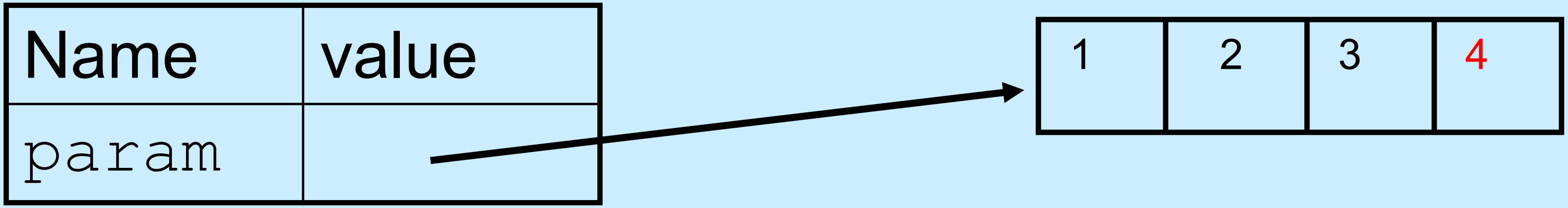
Main Namespace



Name	value
my_list	

1	2	3

my_fun Namespace



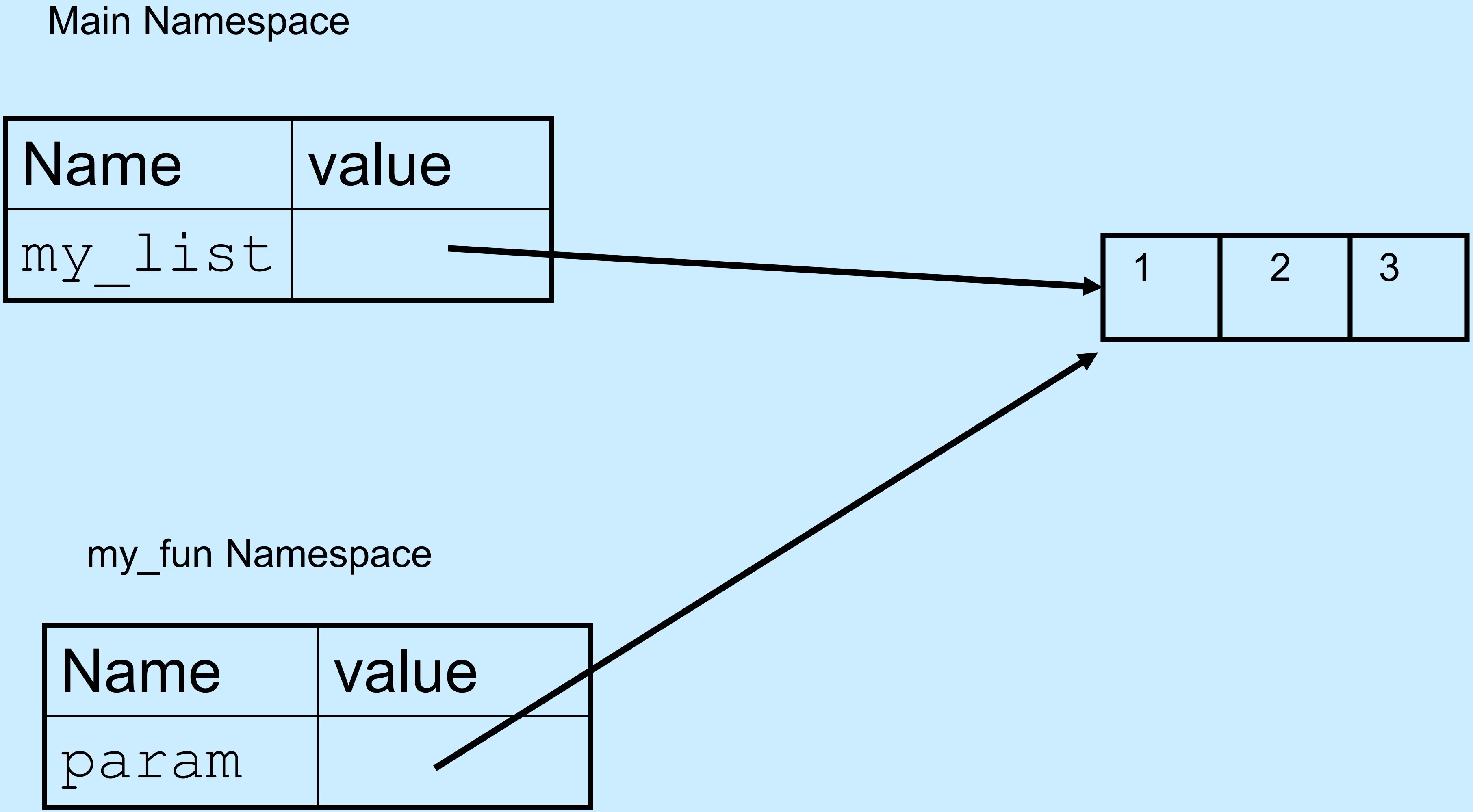
Name	value
param	

1	2	3	4

Example

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

my_list = [1,2,3]
new_list = my_fun(my_list)
print(my_list,new_list)
```



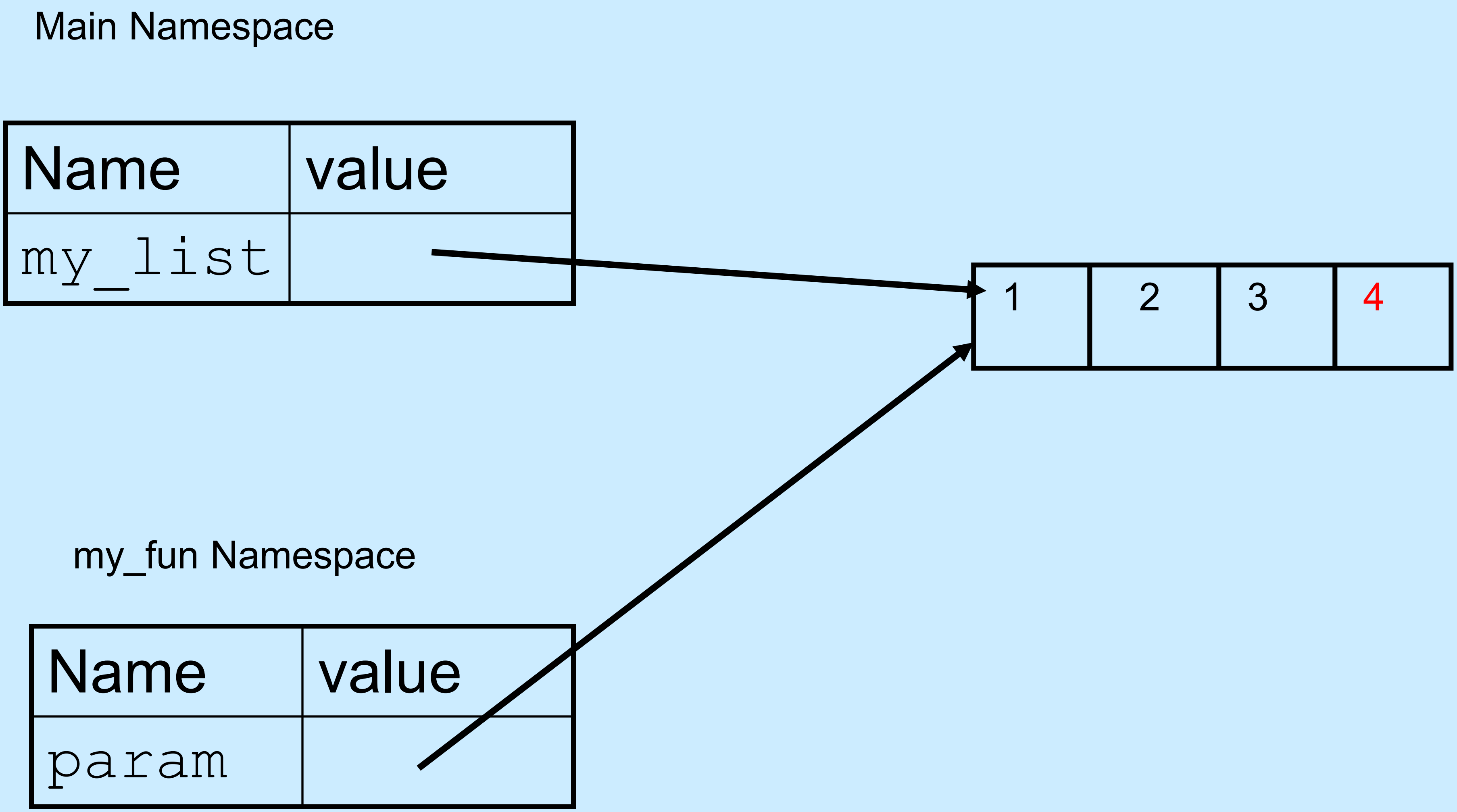
Main Namespace

Name	value
`my_list`	

1	2	3

my_fun Namespace

Name	value
`param`	



Main Namespace

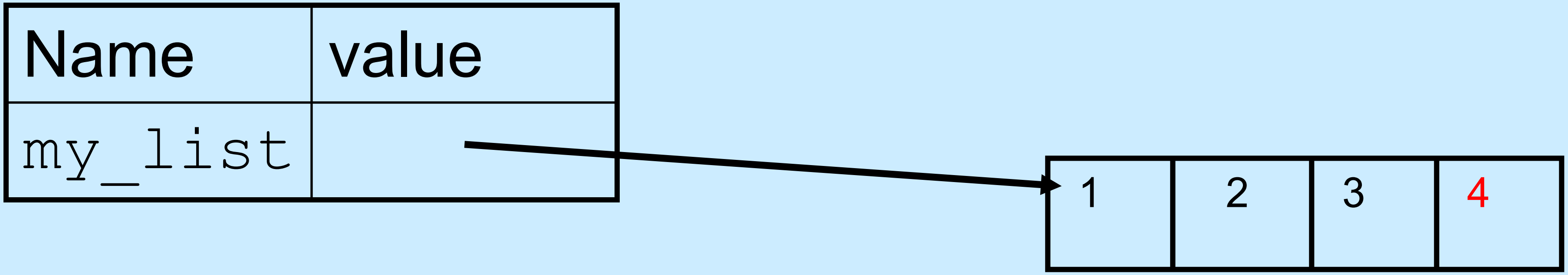
Name	value
`my_list`	

1	2	3	4

my_fun Namespace

Name	value
`param`	

Main Namespace



Name	value
my_list	

1	2	3	4

my_fun Namespace

Name	value
param	None

assignment to a local

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

Default and Named parameters

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

The parameter assignment means two things:

- if the caller does not provide a value, the default is the parameter assigned value
- you can get around the order of parameters by using the name

Defaults

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

box()    # prints 10 10 10
```

Named parameter

```
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 case

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


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

Default args and mutables

- One of the problem with default args occurs with mutables. 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 that value of that default

weird

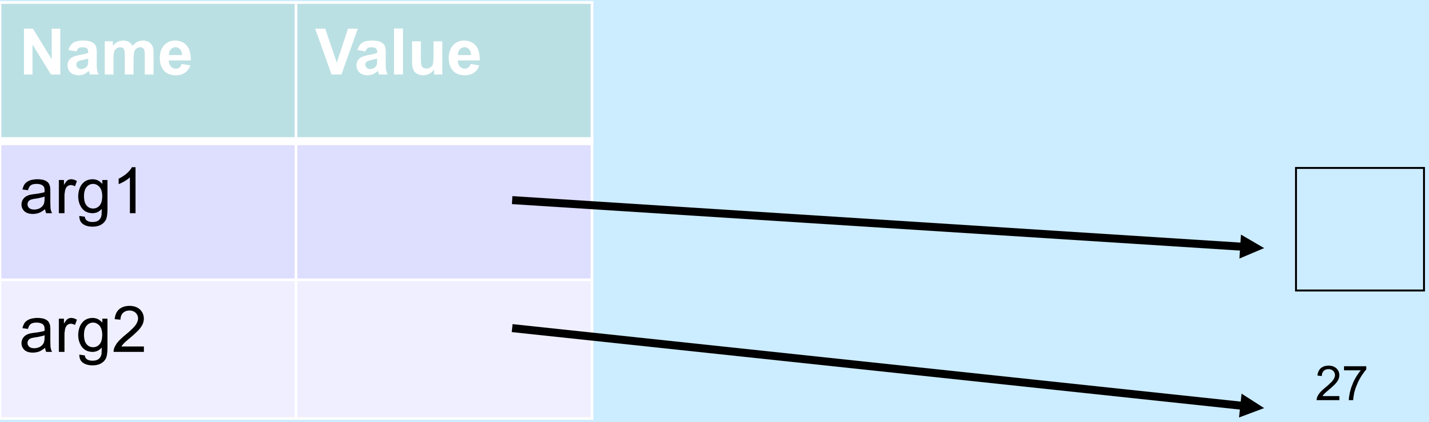
```
def fn1 (arg1=[], arg2=27):
  arg1.append(arg2)
  return arg1

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

mutable.py

arg1 is either assigned to the passed arg or to the function default for the arg

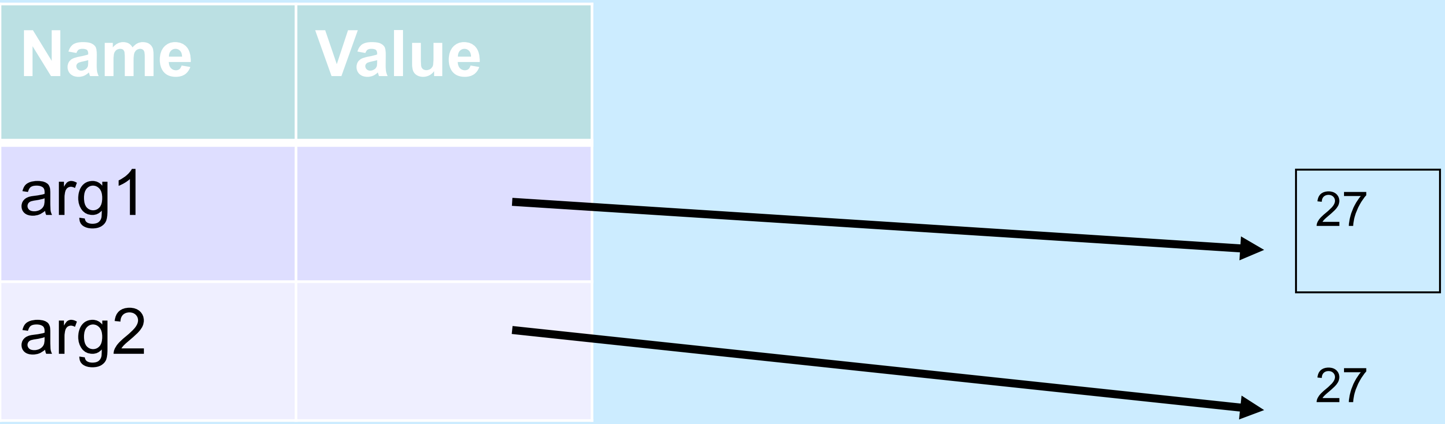
fn1 Namespace



Name	Value
arg1	
arg2	

Now the function default, a mutable, is updated and will remain so for the next call

fn1 Namespace



Name	Value
arg1	
arg2	

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
__str__ : string function
__dict__ : function namespace
__doc__ : docstring
```

function annotations

You can associate strings of information, ignored by Python, with a parameter

- to be used by the reader or user the colon ":" indicates the parameter annotation
- the "->" the annotation is associated with the return value
- stored in dictionary `name_fn.__annotations__`

```
def my_func (param1 : int, param2 : float) -> None :
    print('Result is:', param1 + param2)

>>> my_func(1, 2.0)
Result is: 3.0
>>> my_func(1, 2)
Result is: 3
>>> my_func('a', 'b')
Result is: ab
>>>
```

```
def my_func (param1 : int, param2 : float) -> None :
    print('Result is:', param1 + param2)

>>> my_func.__annotations__
{'return': None, 'param2': <class 'float'>, 'param1': <class 'int'>}
>>>
```

addA.py

Docstring

- If the first item after the def is a string, then that string is specially stored as the docstring of the function
- This string describes the function and is what is shown if you do a help on a function
- Usually triple quoted since it is multilined

Code Listing 8.2
Weighted Grade Function

```python
def weighted_grade(score_list, weights_tuple=(0.3,0.3,0.4)):
    '''Expects 3 elements in score_list. Multiples each grade
    by its weight. Returns the sum.'''
    grade_float = \
        (score_list[0]*weights_tuple[0]) +\
        (score_list[1]*weights_tuple[1]) +\
        (score_list[2]*weights_tuple[2])
    return grade_float
```

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).

Arbitrary arguments

- it is also possible to pass an arbitrary number of arguments to a function
- the function simply collects all the arguments (no matter how few or many) into a tuple to be processed by the function
- tuple parameter preceeded by a * (which is not part of the param name, its part of the language)
- positional arguments only

example

```
def aFunc(fixedParam,*tupleParam):
     print ‘fixed =‘,fixedParam
     print ‘tuple=‘,tupleParam
aFunc(1,2,3,4)
prints    fixed=1
     tuple=(2,3,4)
aFunc(1)
prints    fixed=1
          tuple=()
aFunc(fixedParam=4)
prints    fixed=1
          tuple=()
aFunc(tupleParam=(1,2,3),fixedParam=1)
          Error!
```

"The Practice of Computing Using Python",
Punch & Enbody, Copyright © 2013 Pearson Education, Inc.

Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to imporve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.