INT3075 Programming and Problem Solving for Mathematics

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

- When a function is executed it creates its own 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
- these variables can only be accessed within that function - local scope
- when a function ends, namespace is hidden

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

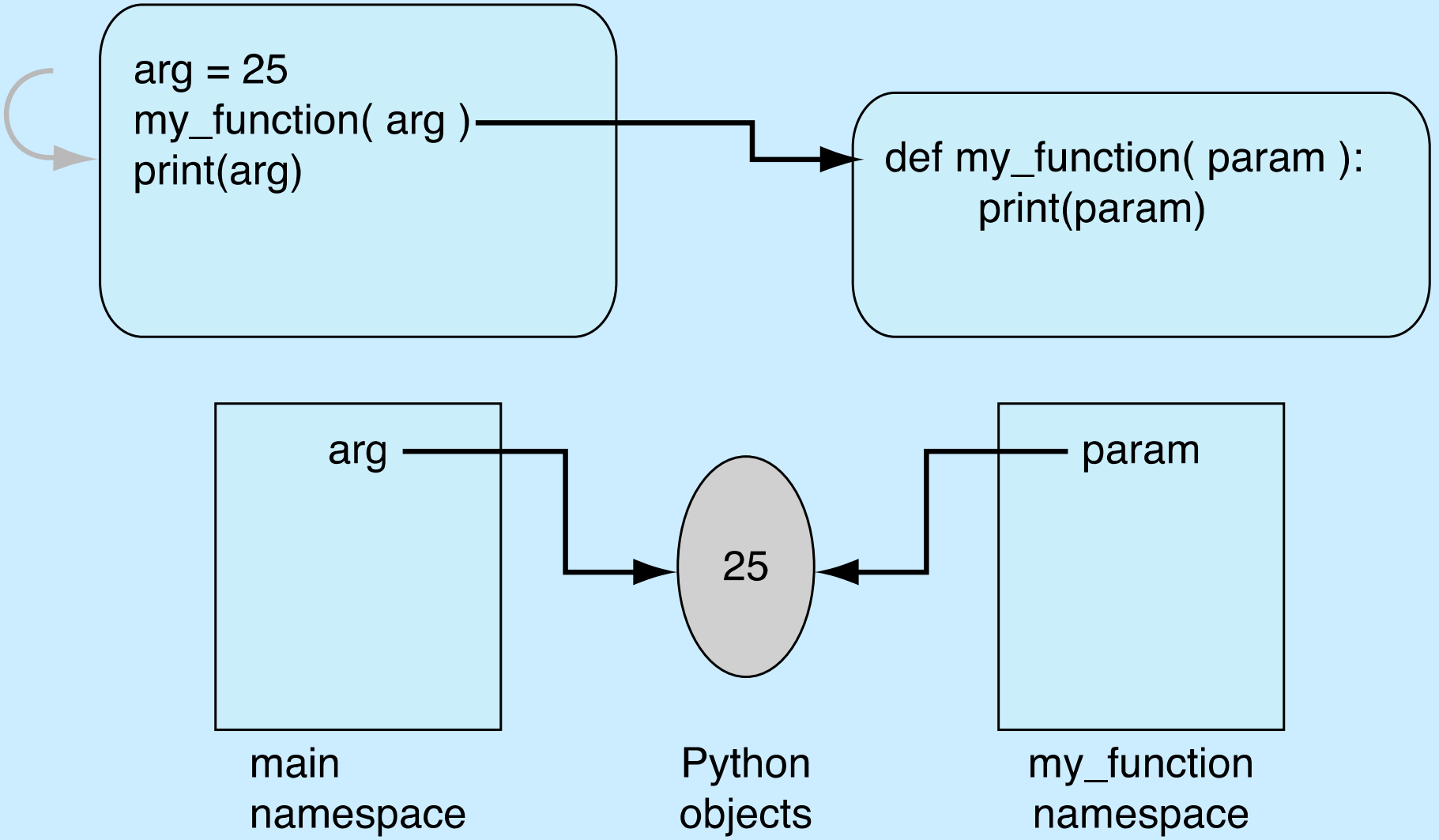


FIGURE 8.1 Function namespace: at function start.

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

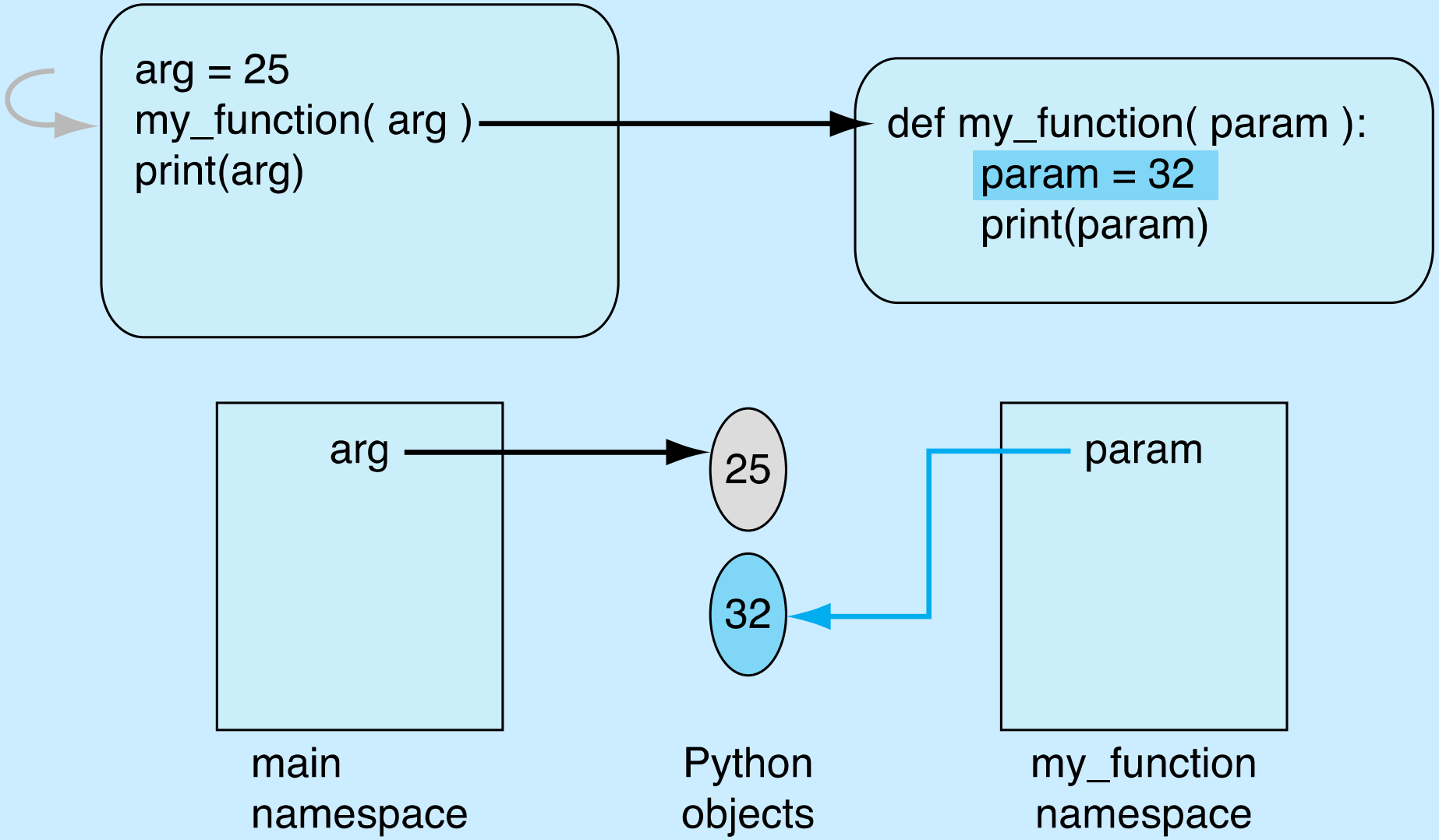
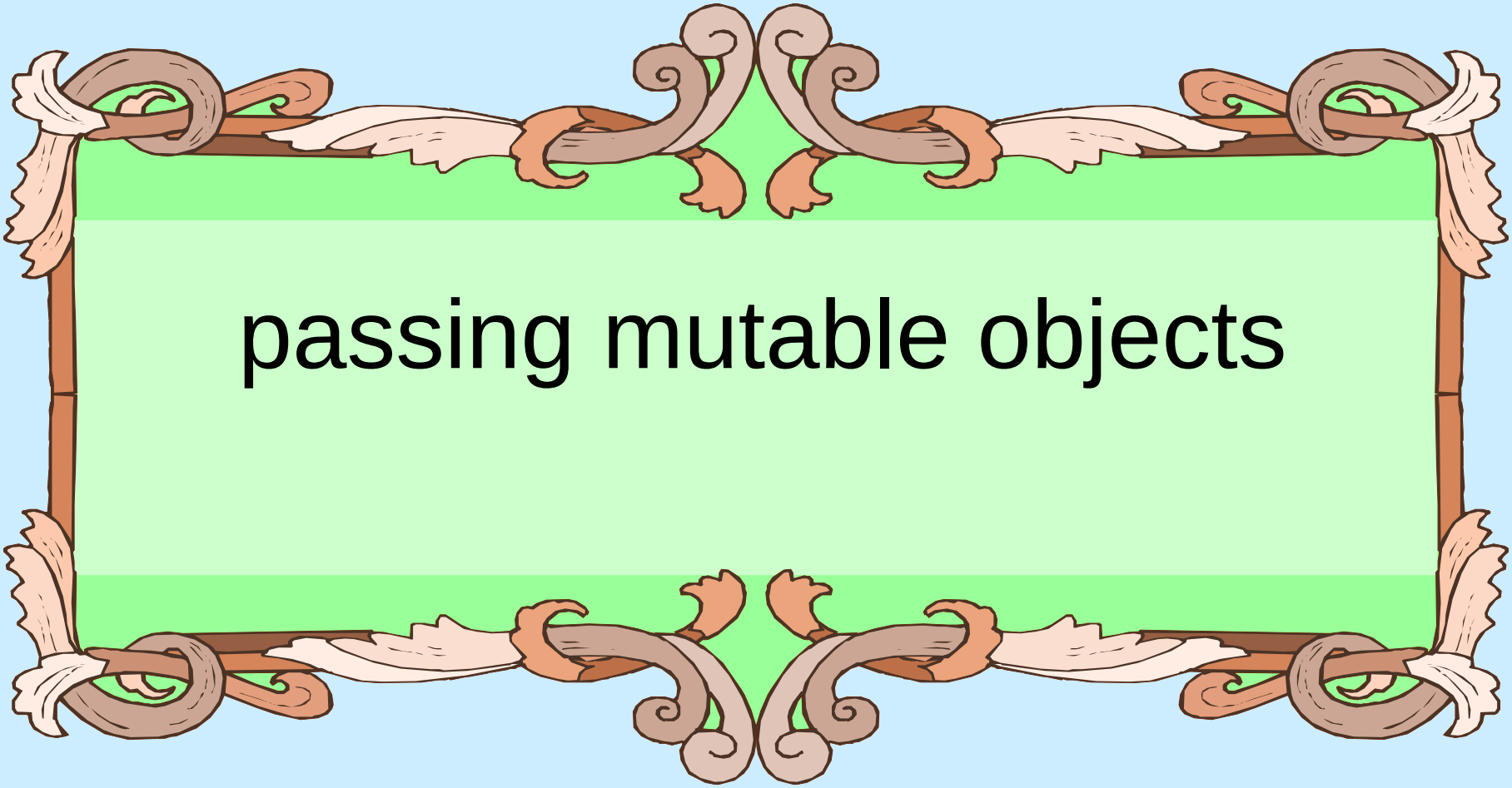


FIGURE 8.2 Function namespace modified.


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.***

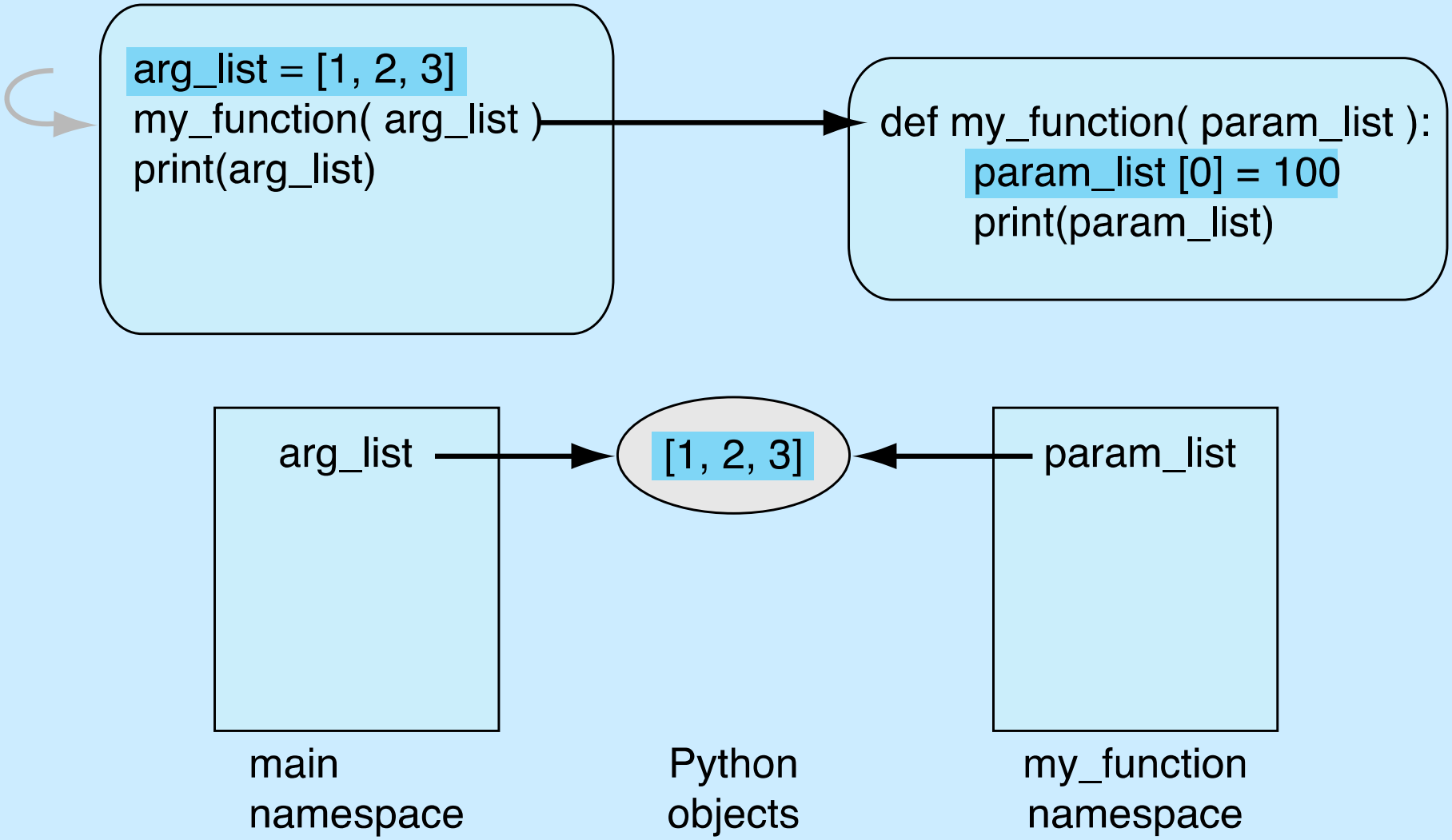


FIGURE 8.3 Function namespace with mutable objects: at function start.

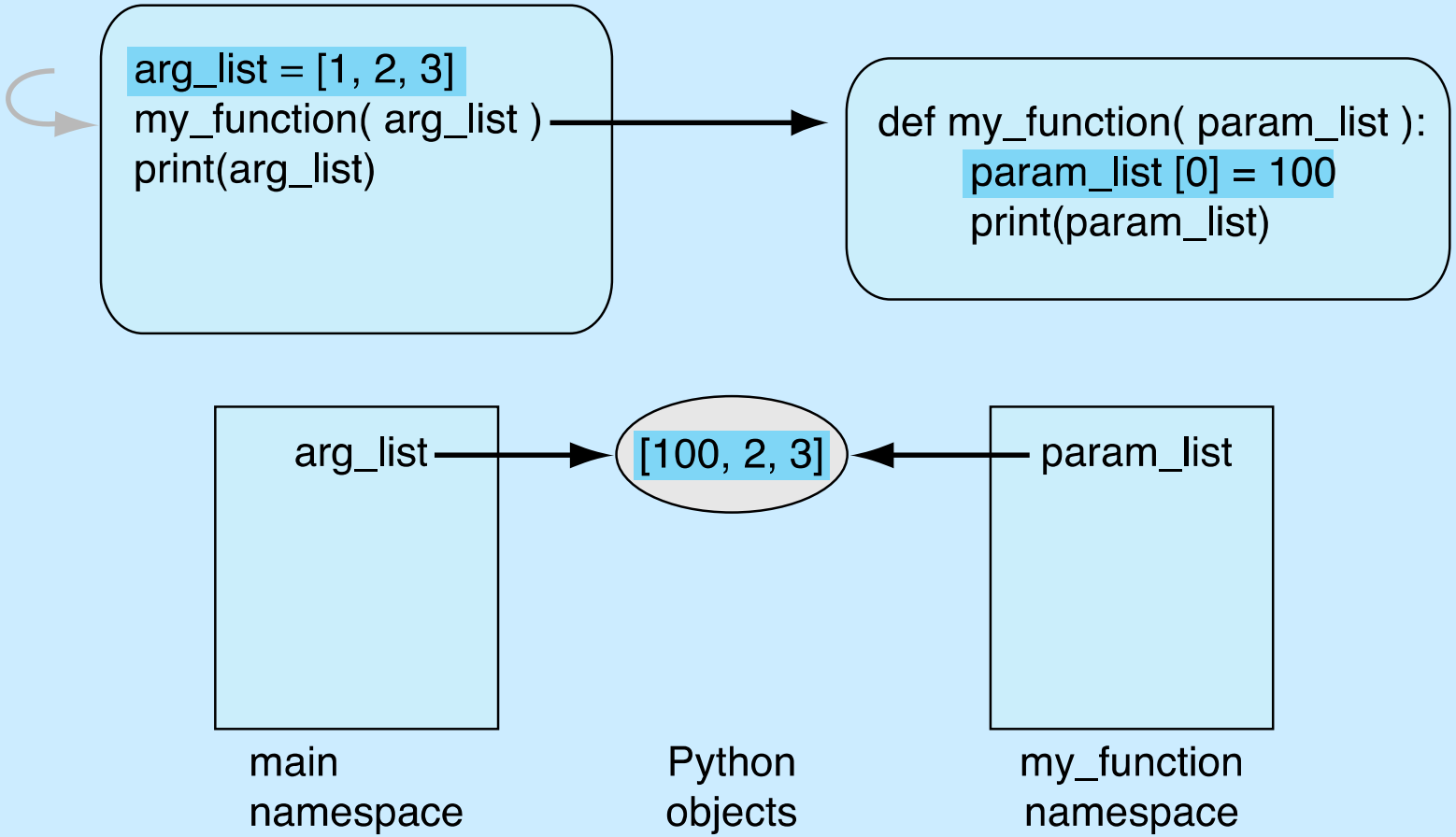
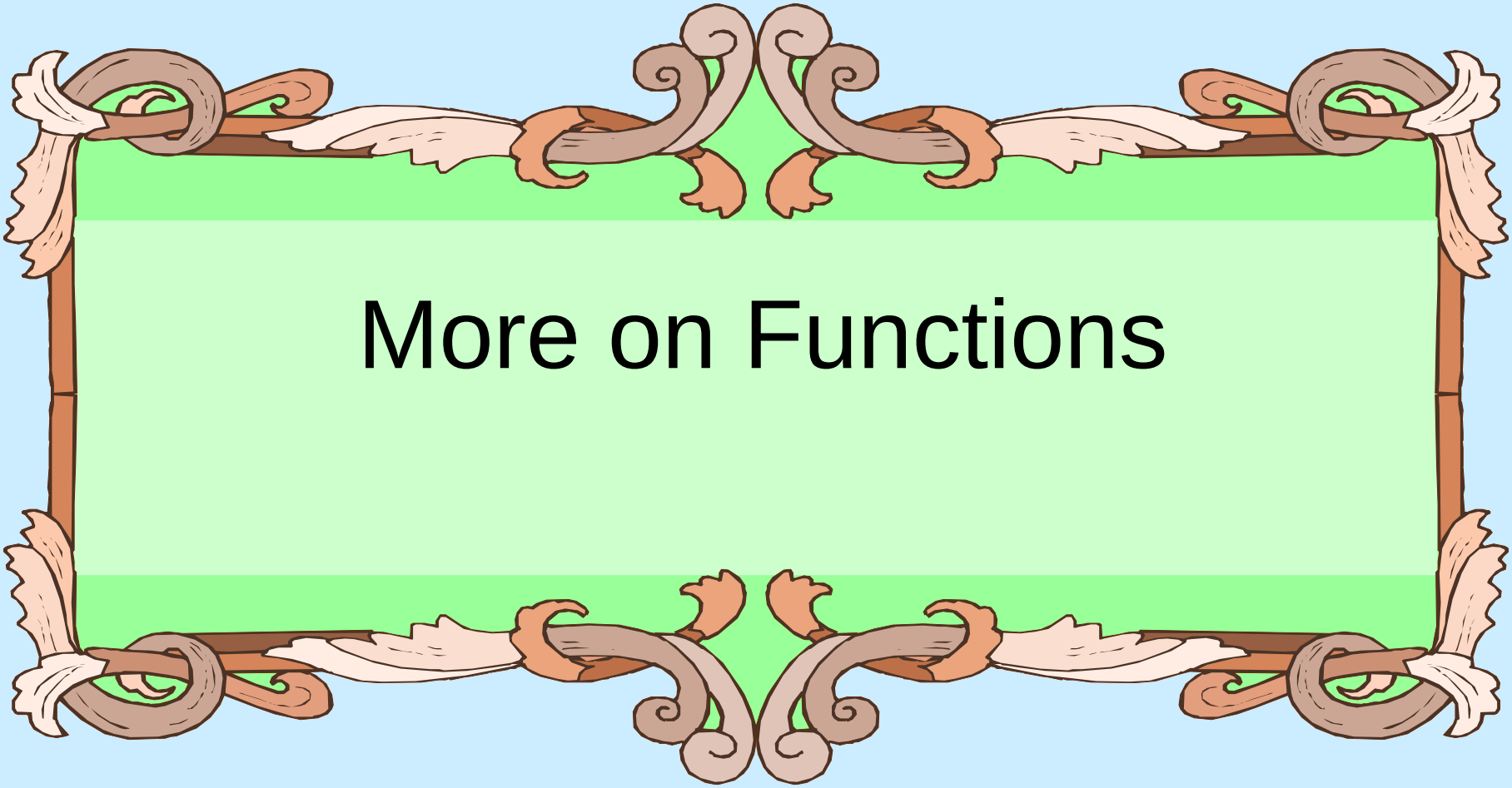


FIGURE 8.4 Function namespace with mutable objects after `param_list[0]=100`.


More on Functions

Functions return one thing

Functions return one thing, but 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 also
>>> 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

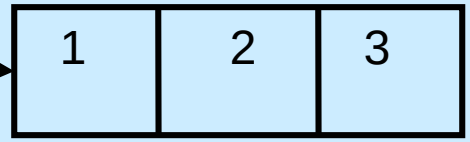
- if you assign a value in a function, that name becomes part of the local namespace of the function
- it can have some odd effects

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`	

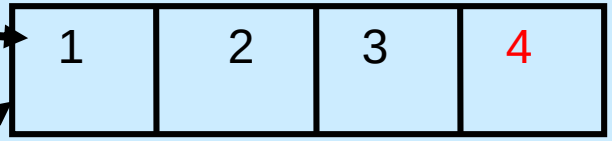


my_fun Namespace

Name	value
`param`	

Main Namespace

Name	value
`my_list`	



my_fun Namespace

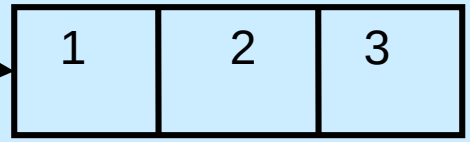
Name	value
`param`	

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`	

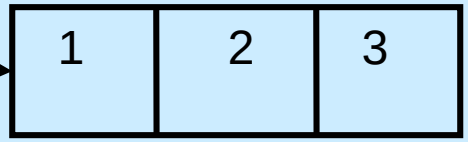


my_fun Namespace

Name	value
`param`	

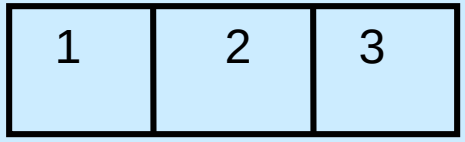
Main Namespace

Name	value
`my_list`	



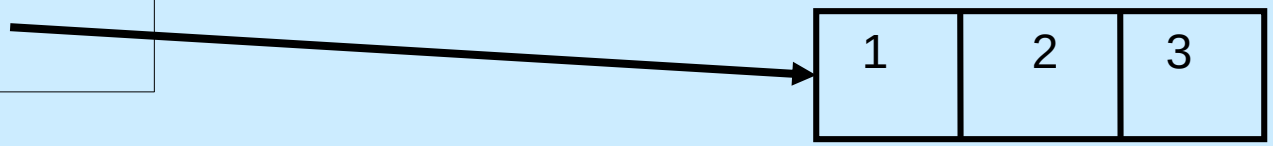
my_fun Namespace

Name	value
`param`	



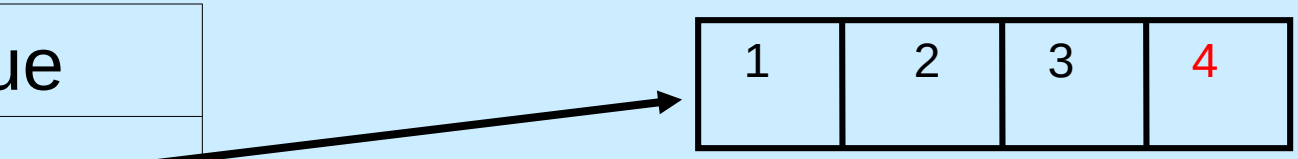
Main Namespace

Name	value
`my_list`	



my_fun Namespace

Name	value
`param`	

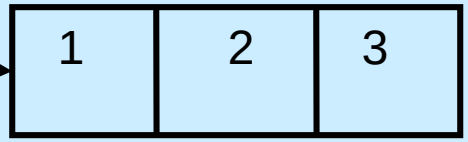


Example

```
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`	

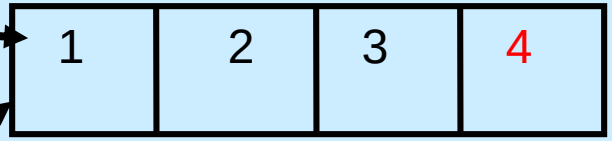


my_fun Namespace

Name	value
`param`	

Main Namespace

Name	value
`my_list`	

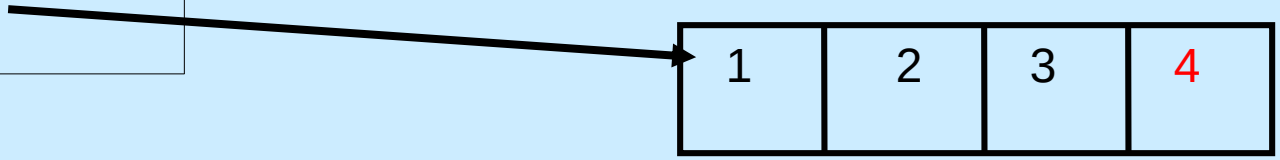


my_fun Namespace

Name	value
`param`	

Main Namespace

Name	value
`my_list`	



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 which then is preserved between invocations

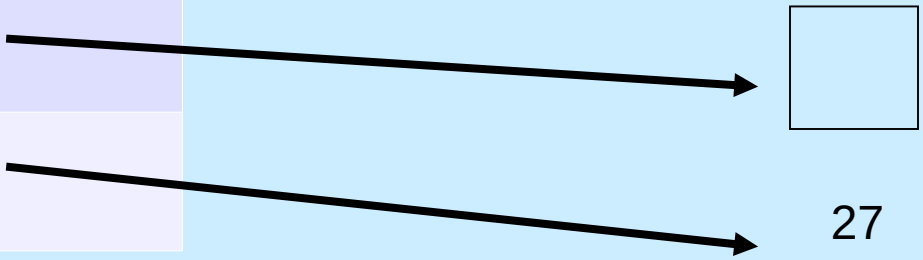
weird

```
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]
```

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

fn1 Namespace

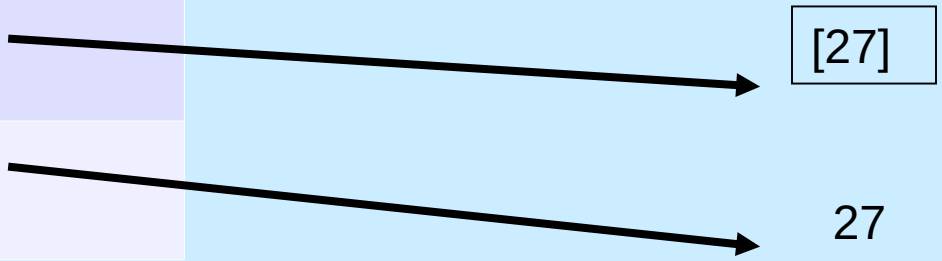
Name	Value
arg1	
arg2	

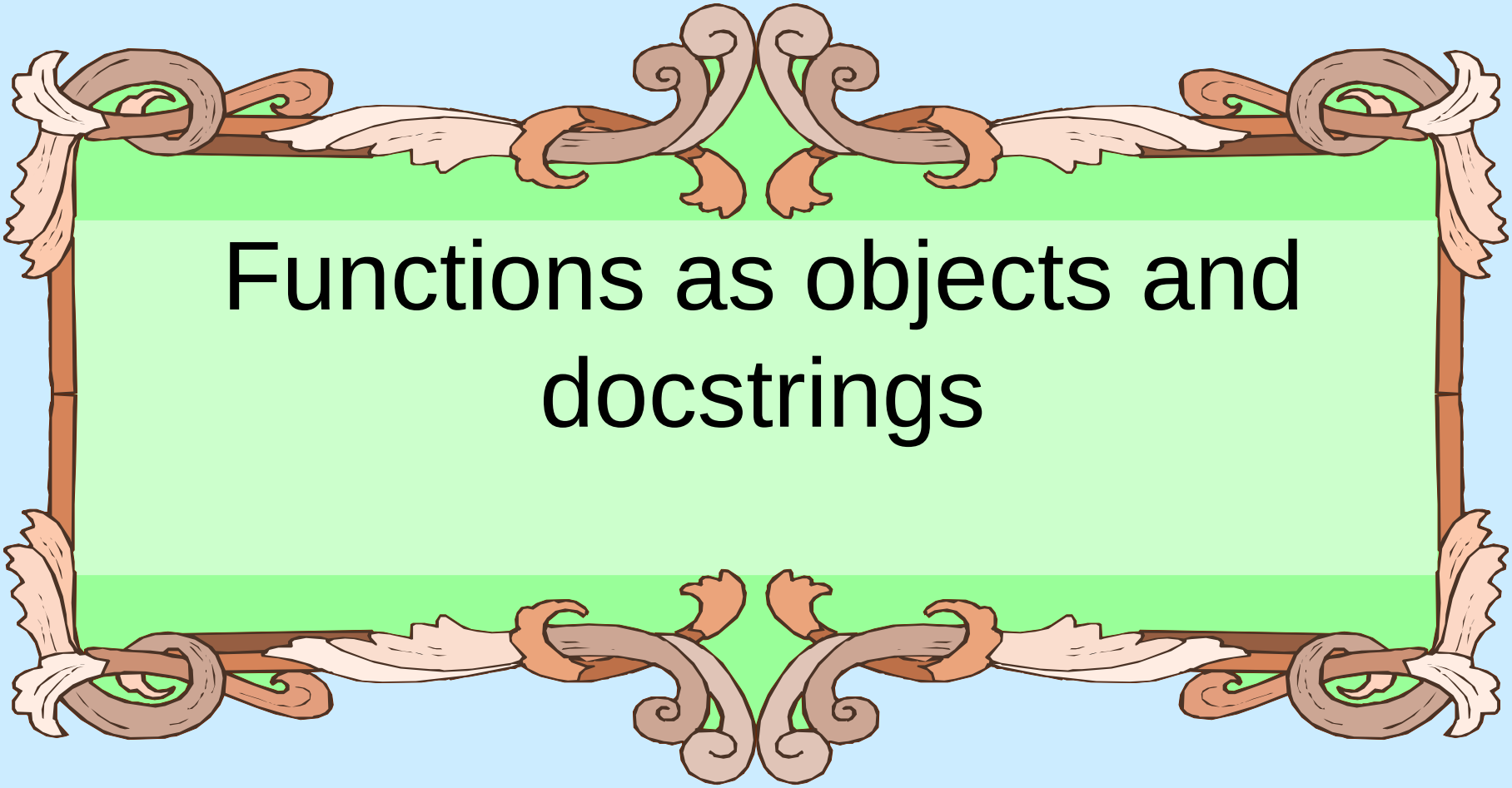


Now the function default, a mutable, is updated and will remain so for the next call

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

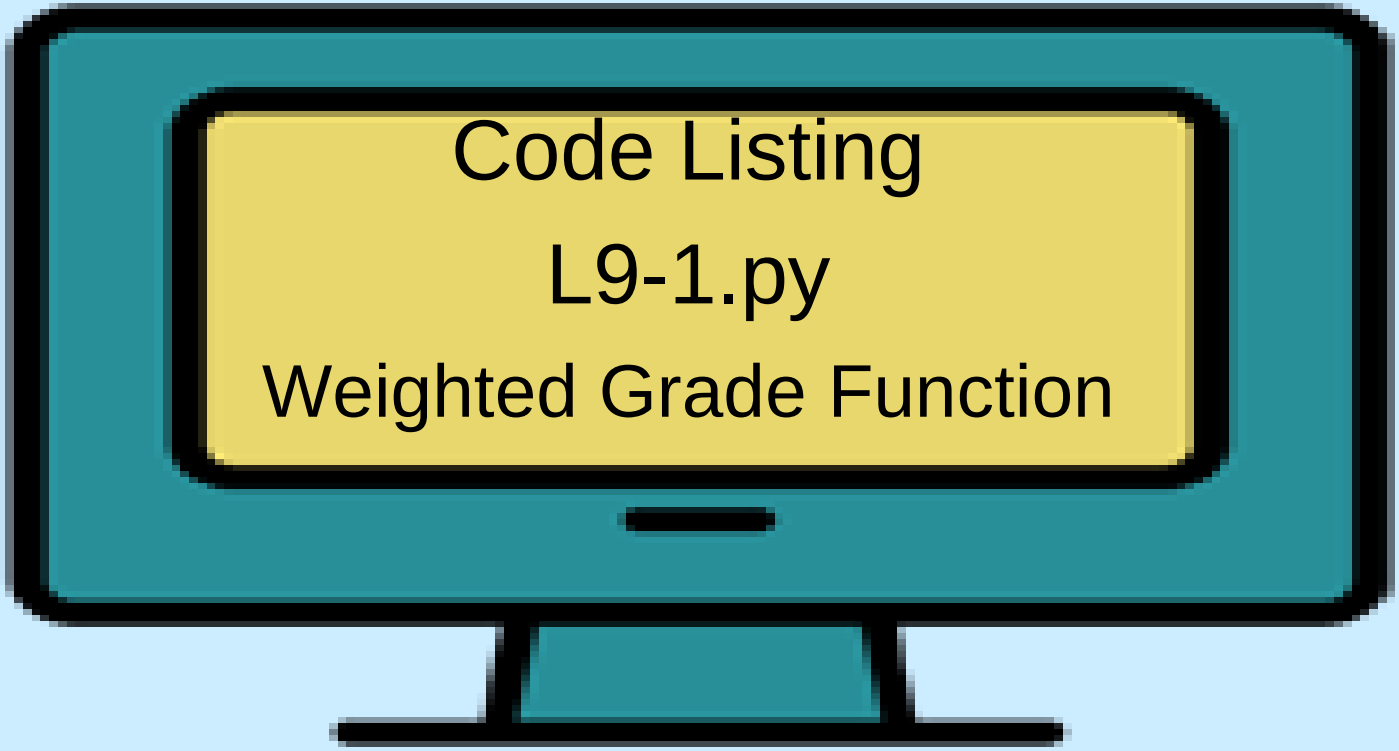
- to be used by the reader or user the colon ":" indicates the parameter annotation
- the "->" the annotation is associated with the return value
- stored in dictionary name_fn.__annotations__

```
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'>}
>>>
```

Docstring

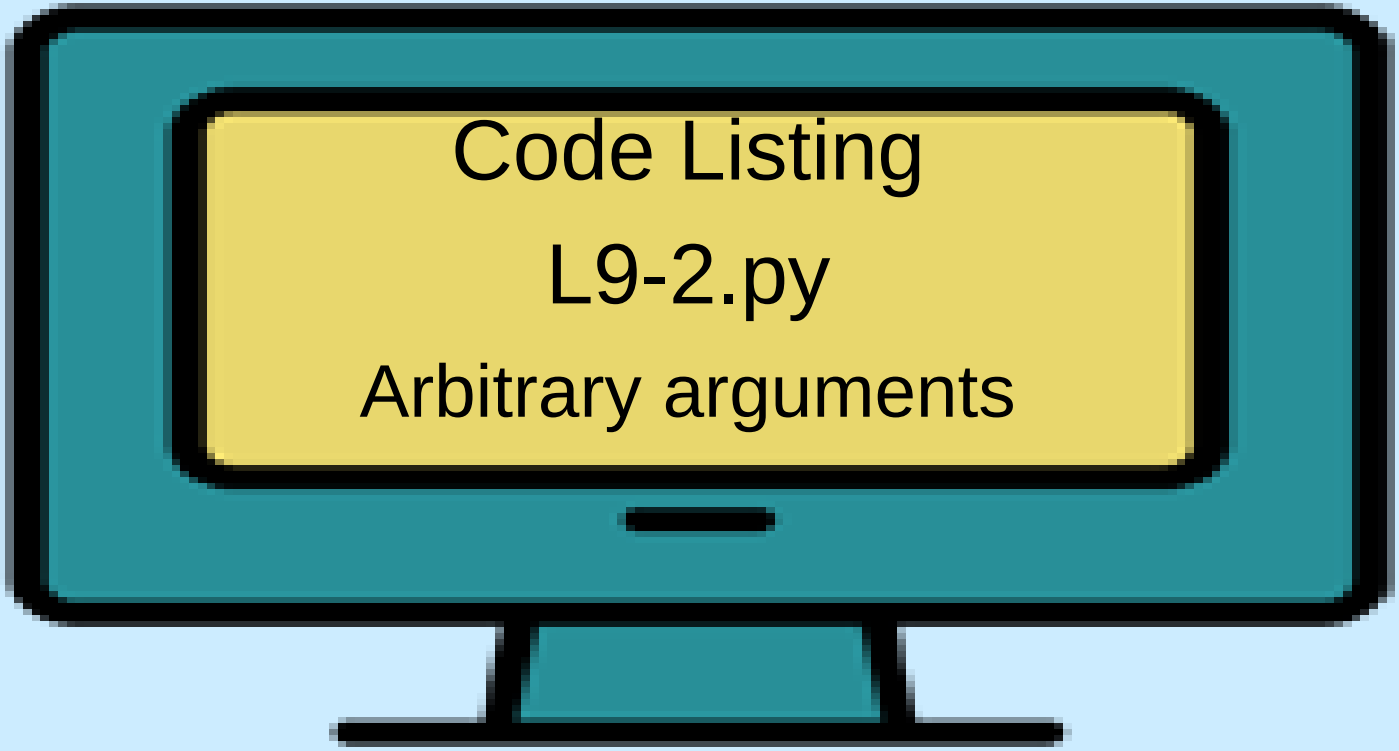
- If the first item after the def is a string, then that string is specially stored as the docstring of the function
- This string describes the function and is what is shown if you do a help on a function
- Usually triple quoted since it is multi-lined


Code Listing
L9-1.py
Weighted Grade Function

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

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 preceded by a * (which is not part of the param name, its part of the language)
- positional arguments only


Code Listing
L9-2.py
Arbitrary arguments

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=4
tuple=()
```

```
aFunc(tupleParam=(1,2,3),fixedParam=1)
```

Error!