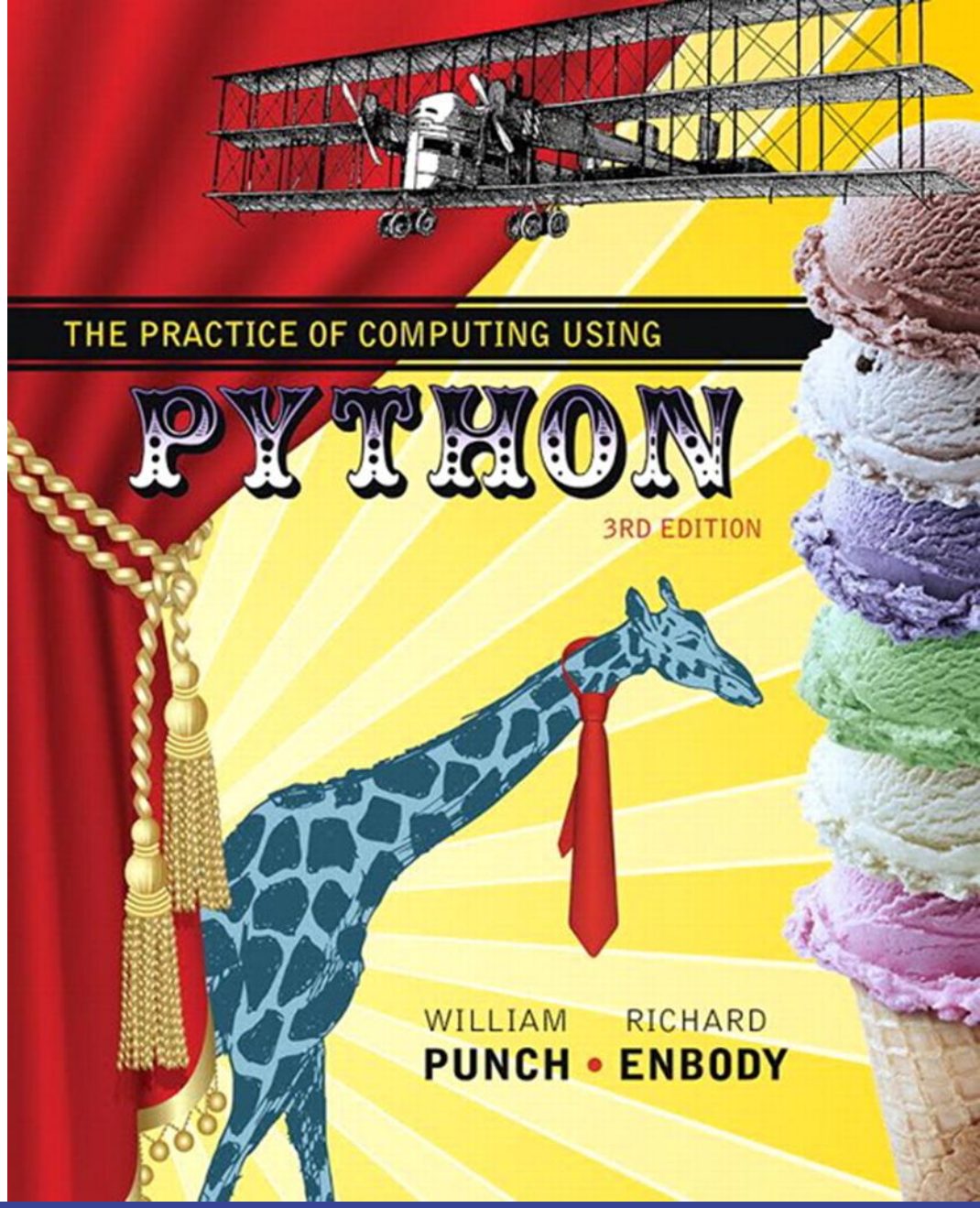
chapter 8

More On Functions



Scope

Scope

- definition: the set of program statements over which a variable exists, i.e., can be accessed
- it is about understanding, for any variable, what its associated value is
- the problem is that multiple namespaces might be involved

Namespaces

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

Function 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 Arguments

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

Passing Immutable Objects

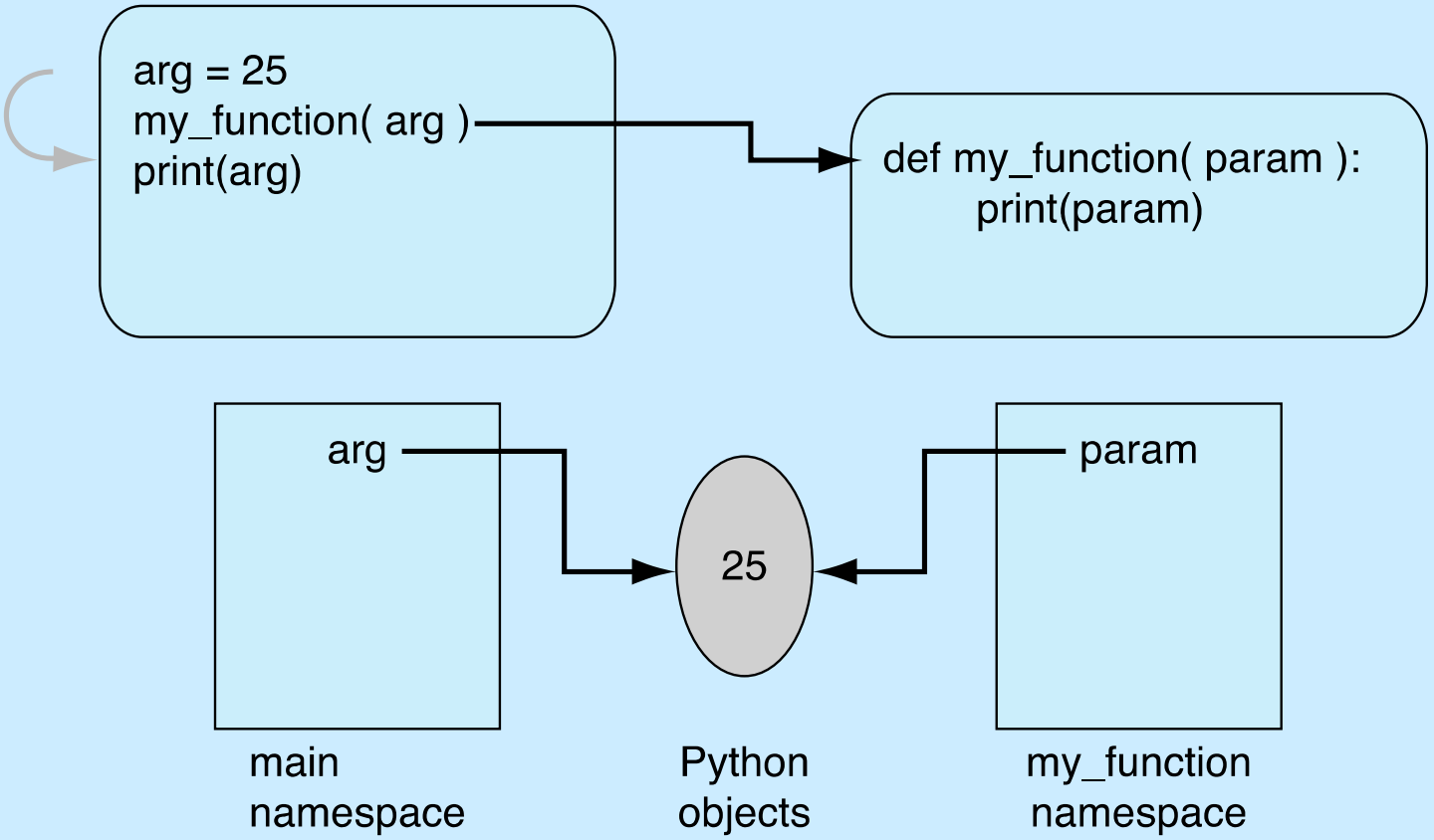


FIGURE 8.1 Function namespace: at function start.

What Does “pass” Mean?

- the previous 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
- “passing” therefore 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

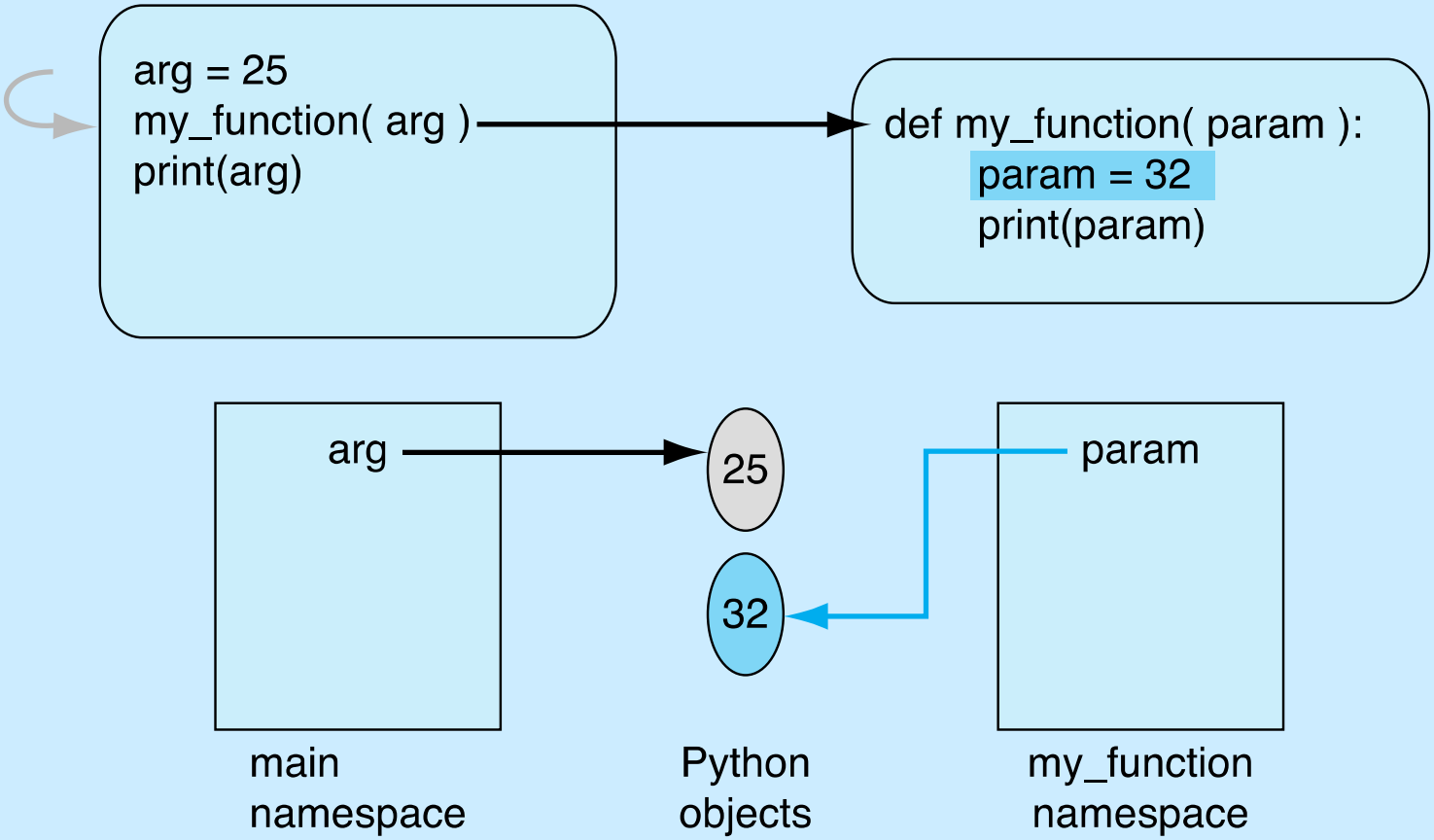


FIGURE 8.2 Function namespace modified.

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

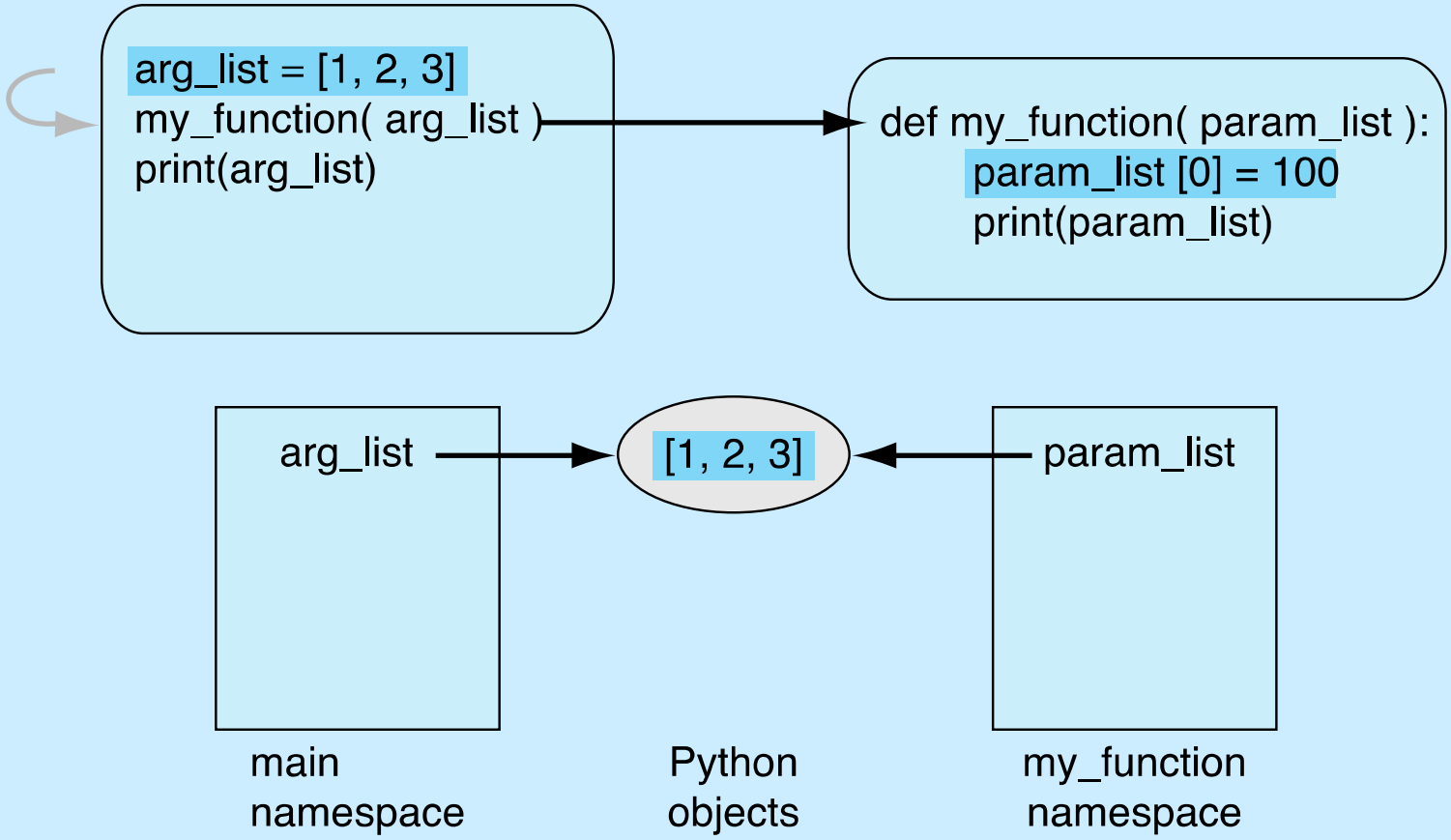


FIGURE 8.3 Function namespace with mutable objects: at function start.

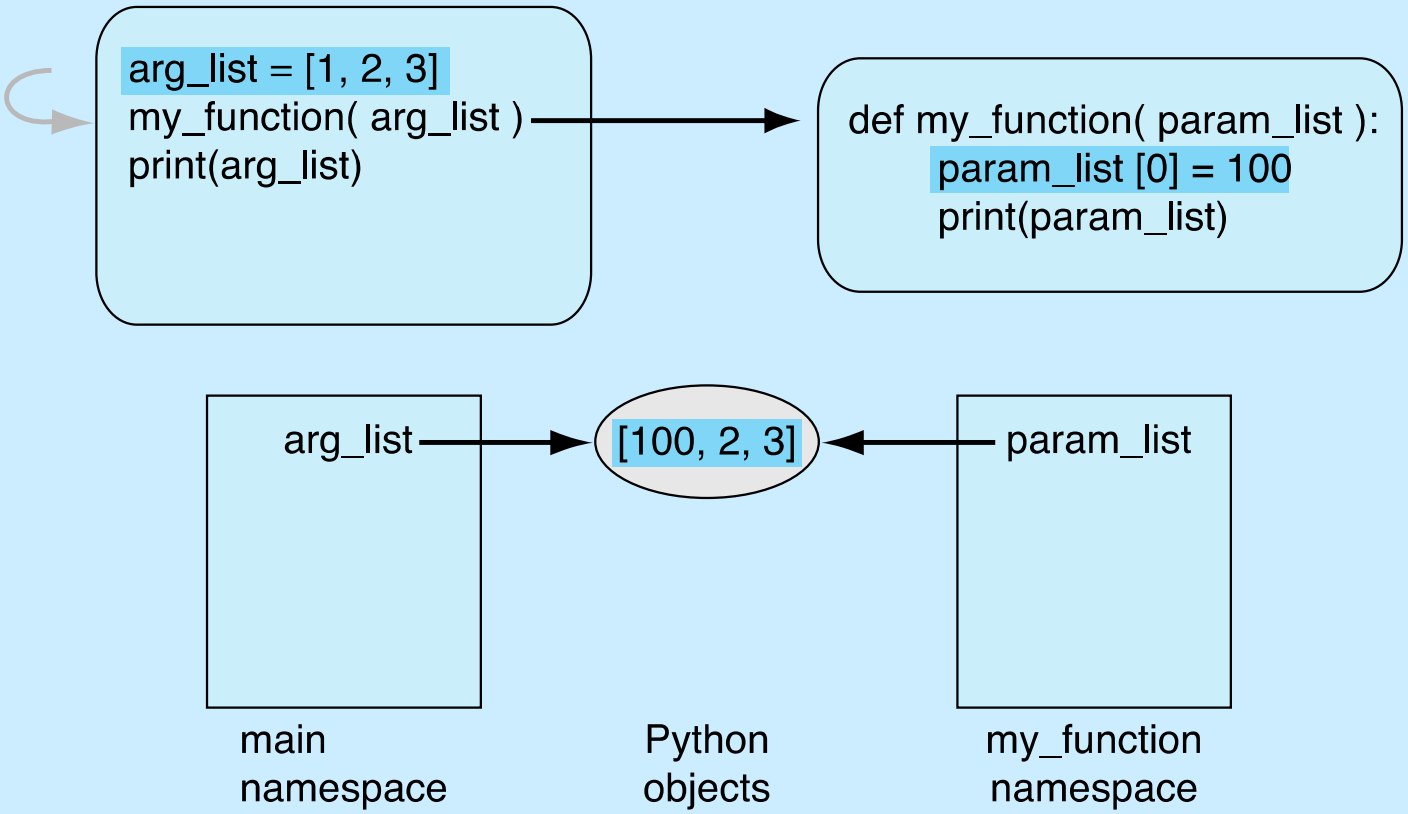


FIGURE 8.4 Function namespace with mutable objects after `param_list[0]=100`.

More on Functions

Function Returns

- functions return one thing, but it can be a ‘chunky’ thing (e.g., 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

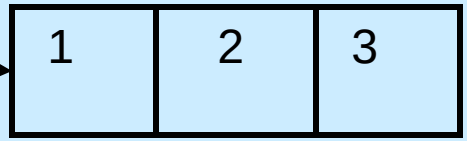
Example

```
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`	

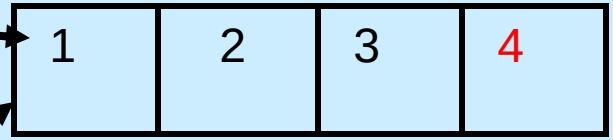


my_fun Namespace

Name	value
`param`	

Main Namespace

Name	value
`my_list`	



my_fun Namespace

Name	value
`param`	

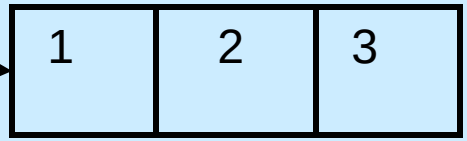
Example

```
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`	

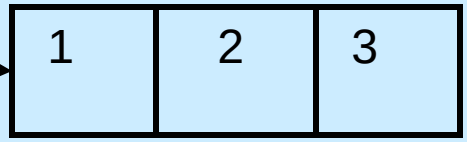


my_fun Namespace

Name	value
`param`	

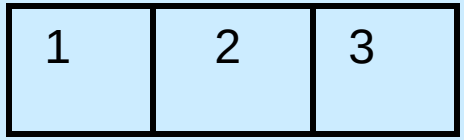
Main Namespace

Name	value
`my_list`	



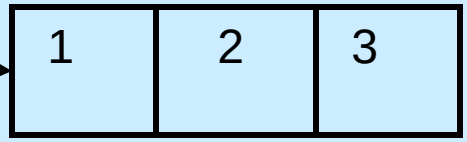
my_fun Namespace

Name	value
`param`	



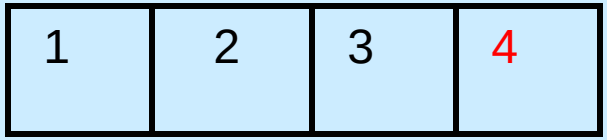
Main Namespace

Name	value
`my_list`	



my_fun Namespace

Name	value
`param`	



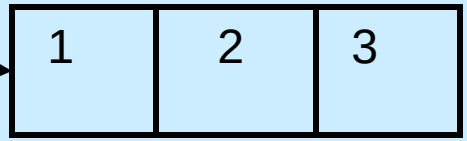
Example

```
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)
```

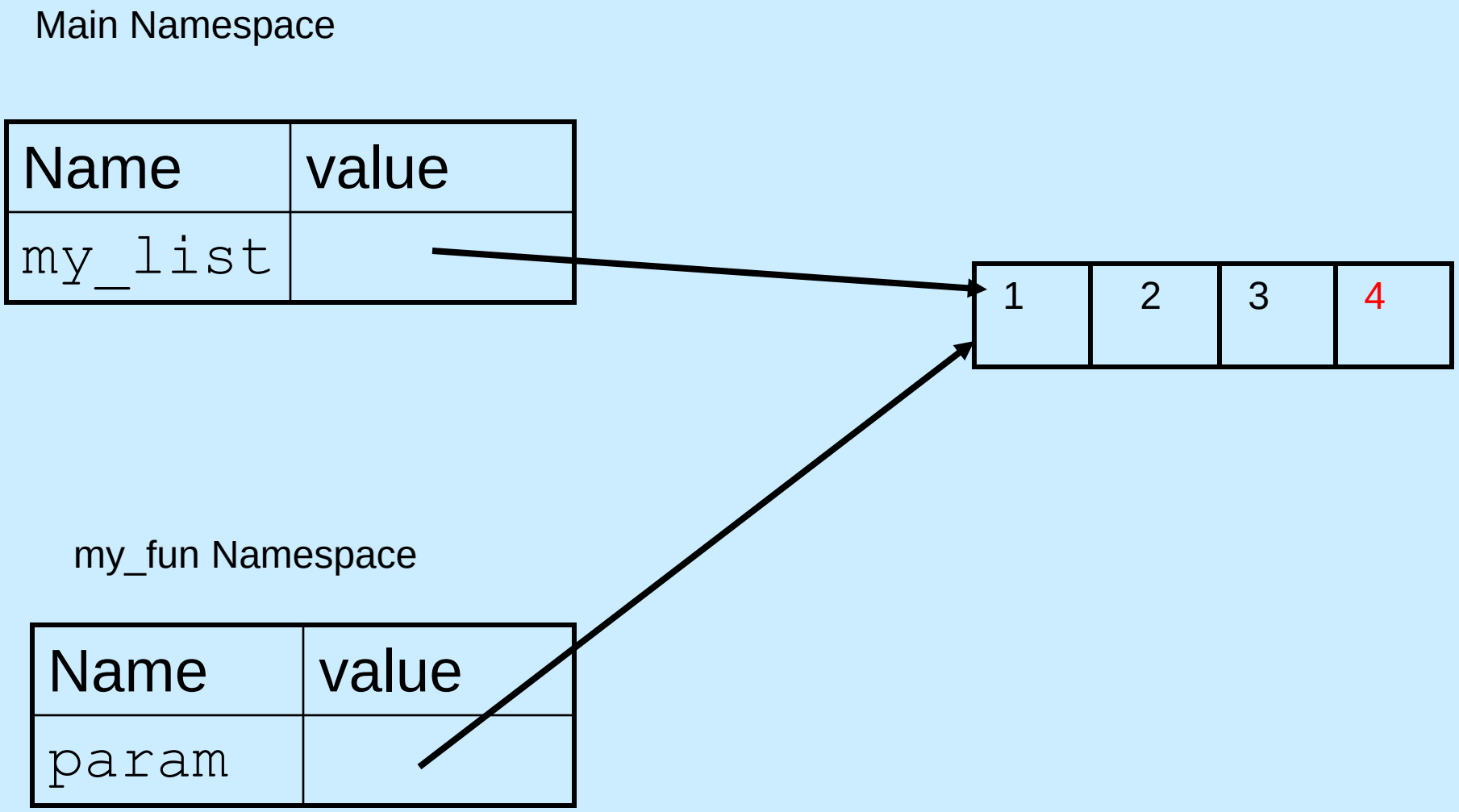
Main Namespace

Name	value
`my_list`	



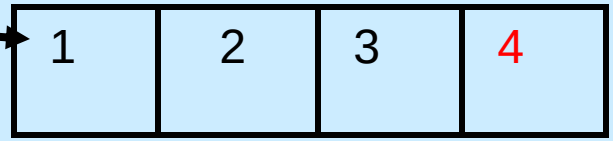
my_fun Namespace

Name	value
`param`	



Main Namespace

Name	value
`my_list`	



my_fun Namespace

Name	value
`param`	`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, it cannot be referenced 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 Parameters

```
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 Arguments and Mutables

- one of the problem with default args occurs with mutables
  - 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

Unusual Results

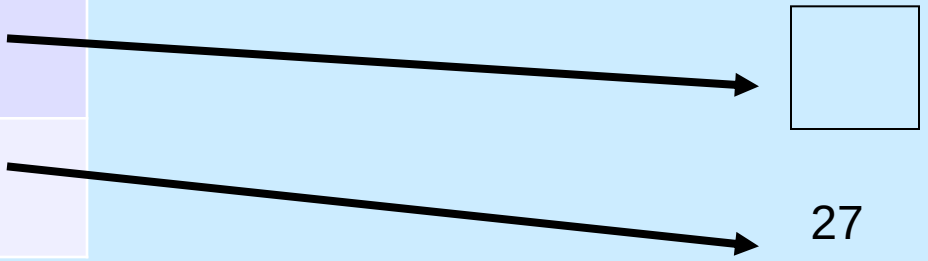
```
def fn1 (arg1=[], arg2=27):
  arg1.append(arg2)
  return arg1


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 argument or to the function default for the argument

fn1 Namespace

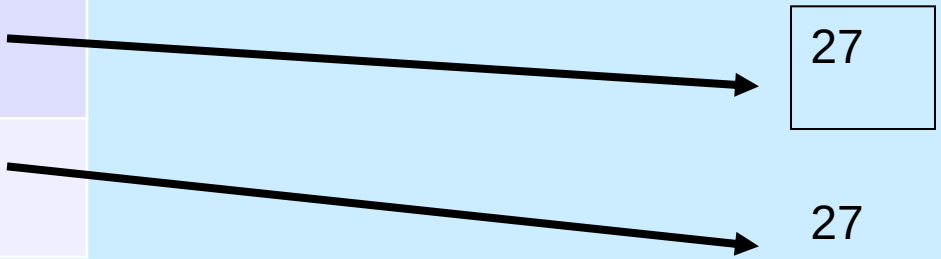
Name	Value
arg1	
arg2	



- 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 "->" 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

docstring

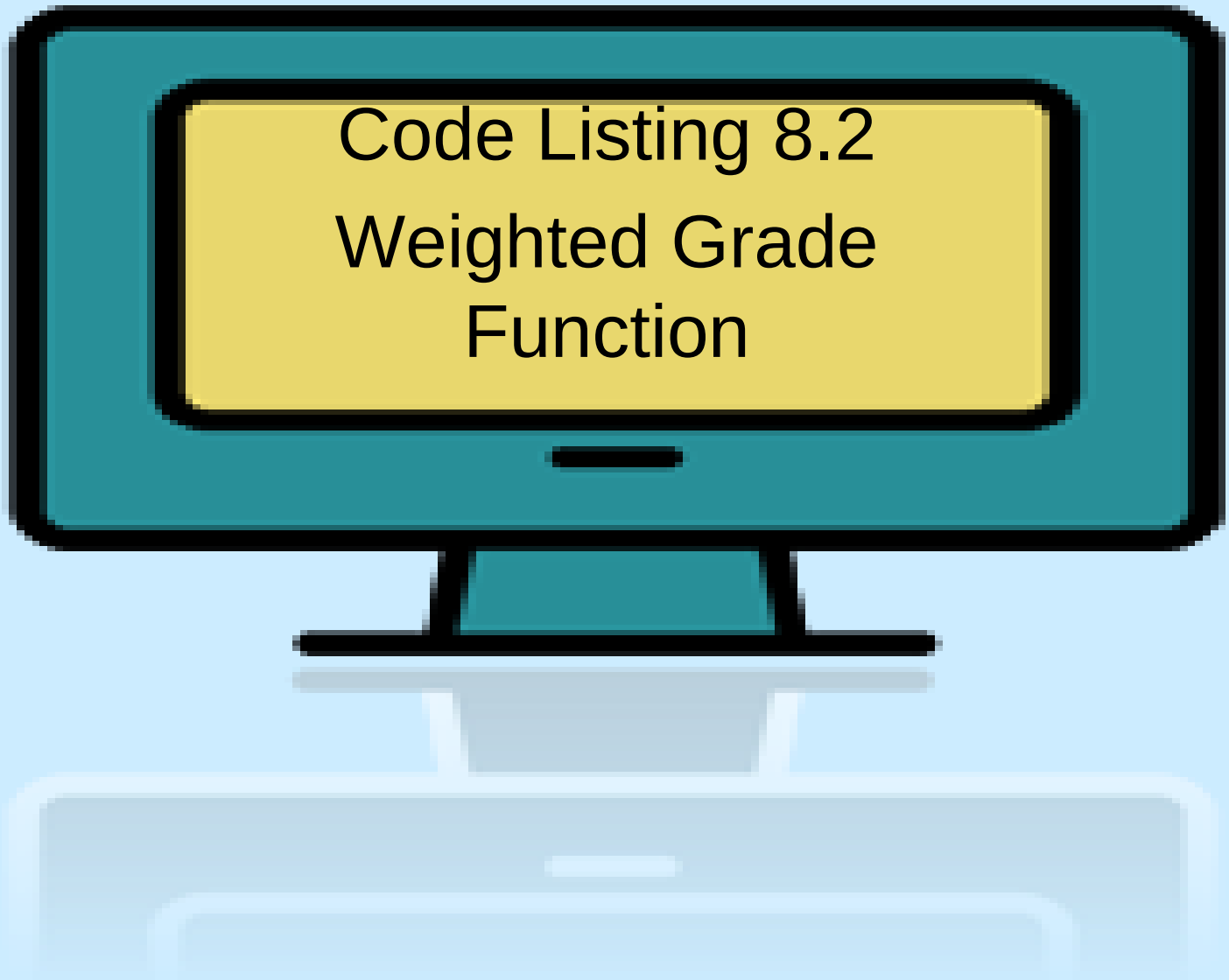
- every object (function, etc.) can have a docstring
- it is stored as an attribute of the function (the **`__doc__`** attribute)
- **`listMean.__doc__`**

```
'Takes a list of integers, returns
the average of the list.'
```

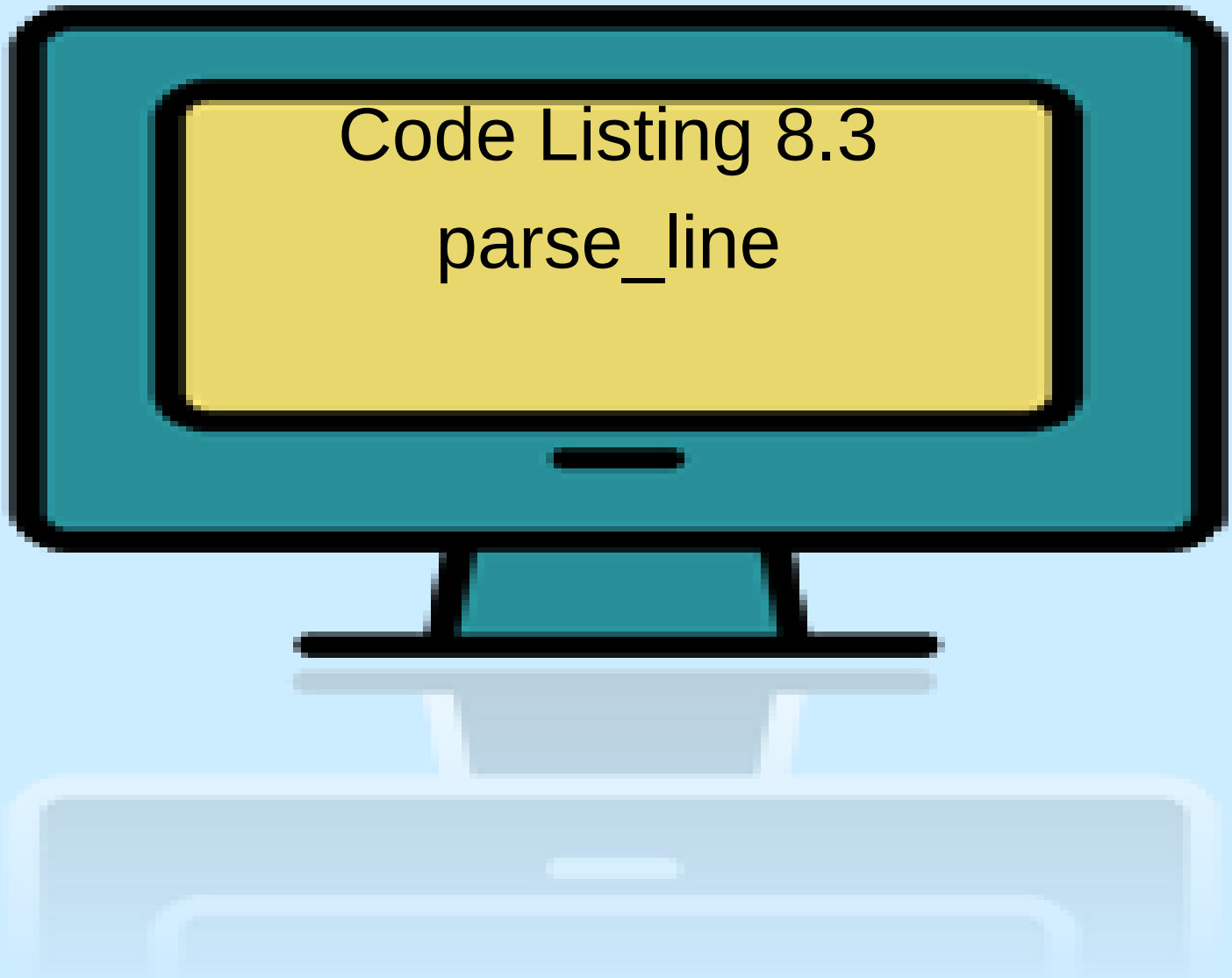
- other programs can use the docstring to report to the user (for example, Spyder)

Example: Final Grade Program

- the following code shows how you can read in a file of grades
- each line of the file contains five comma-separated fields:
  - last name
  - first name
  - exam1, exam2, final_exam
- print name and final grade



```
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
```



```
def parse_line(line_str):
    ''' Expects a line of form last, first, exam1, exam2, final.
    returns a tuple containing first+last and list of scores. '''

    field_list = line_str.strip().split(',')
    name_str = field_list[1] + ' ' + field_list[0]

    score_list = []
    # gather the scores, now strings, as a list of ints
    for element in field_list[2:]:
        score_list.append(int(element))

    return name_str,score_list
```

```
def main ():
    ''' Get a line_str from the file,
       print the final grade nicely. '''

    file_name = input('Open what file:')
    grade_file = open(file_name, 'r')

    print('{:>13s}  {:>15s}'.format('Name','Grade'))
    print('-'*30)

    for line_str in grade_file:
        name_str,score_list = parse_line(line_str)

        grade_float = weighted_grade(score_list)

        print('{:>15s} {:14.2f} '.format(name_str, 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 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=4**
tuple=()

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

Error!

lambda Functions

- lambda expressions are short functions that are defined in-line
  - can only contain a single expression
  - cannot contain statements
  - result is automatically returned

```
list.sort (key = lambda x: float(x[2]))
```

Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also executes on a computer.
3. The best way to improve 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.