SCOPE

SCOPE

- Scope is: "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
  - A 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 to a variable that is created within the function
  - a variable that gets its value when the function is called
    - That is , the parameters of the function

PASSING ARGUMENTS TO PARAMETERS

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

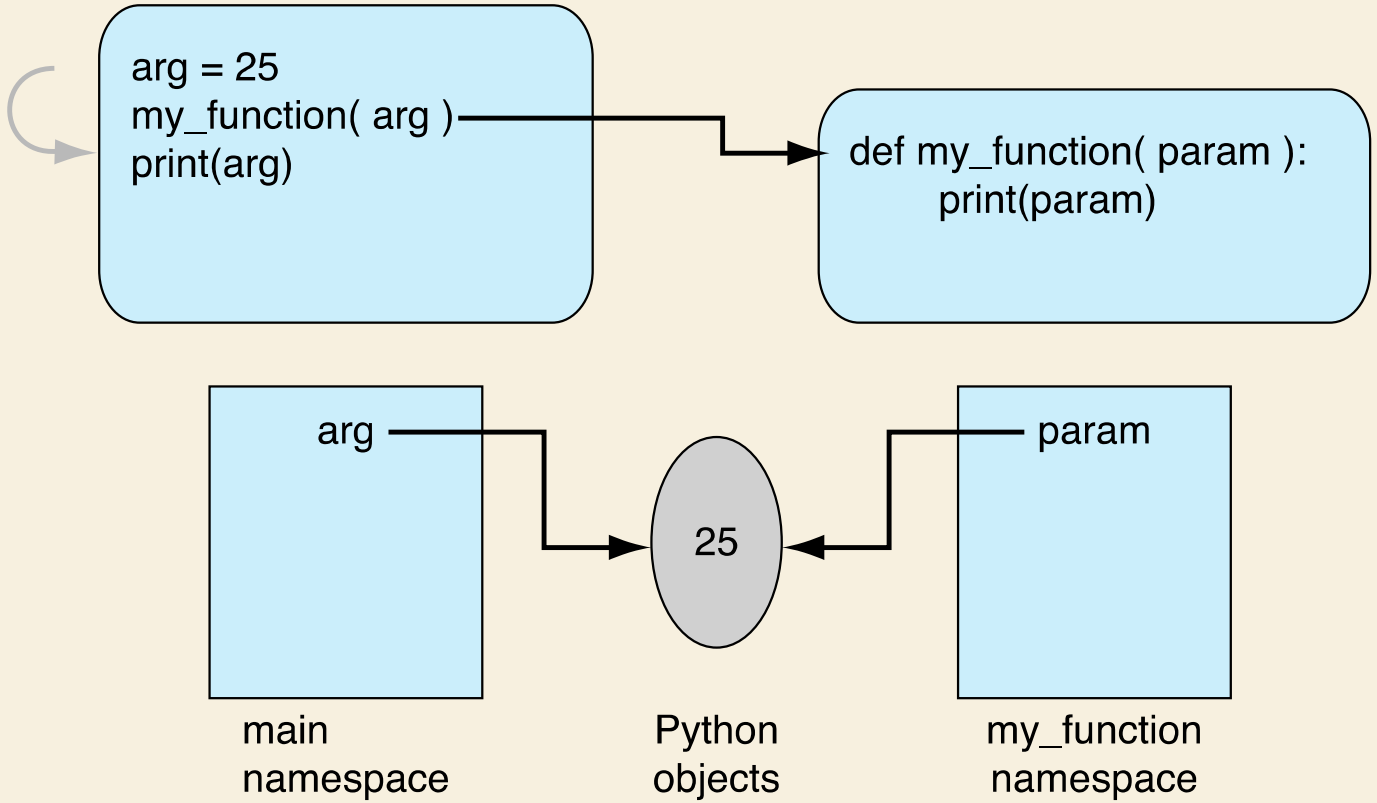
The variables a and b are local to the function sum. But they will not get a value until the function is called

```
1    def sum(a, b):
2        return a + b
3
4
5    number1 = 3
6    number2 = 4
7
8    result = sum(number1, number2)
```

It is at this point the variables a and b (which are local to the function sum) get values. The variable **a** will get the value of number1 and **b** will get the value of number 2

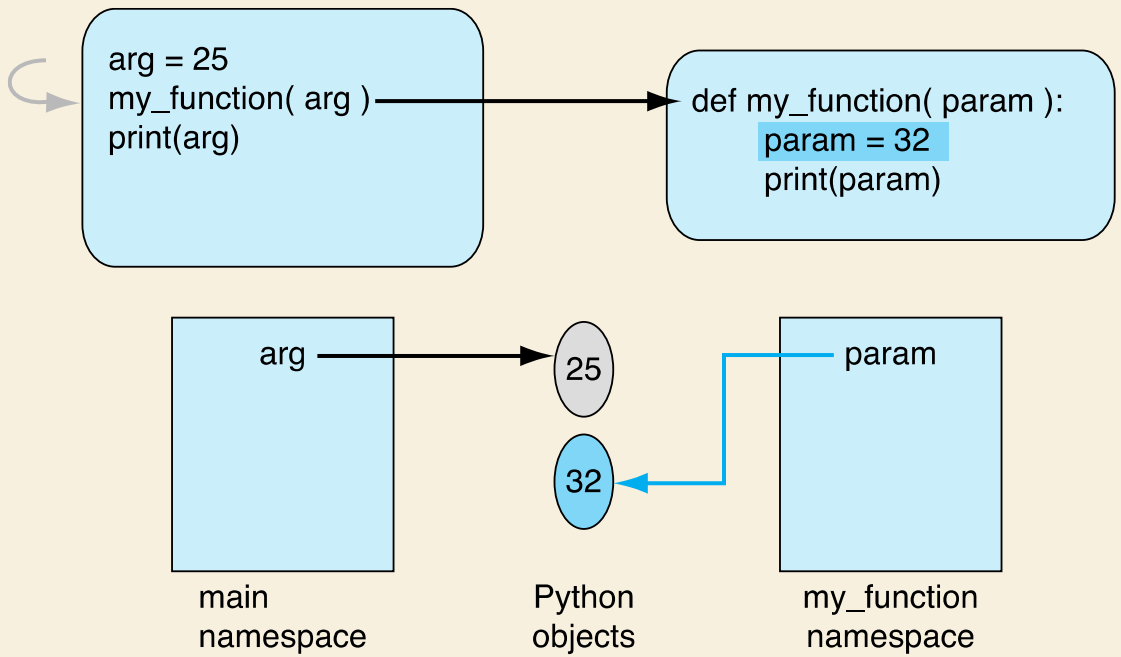
PASSING IMMUTABLE OBJECT TO FUNCTIONS

- This 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



PASSING IMMUTABLE OBJECT TO FUNCTIONS

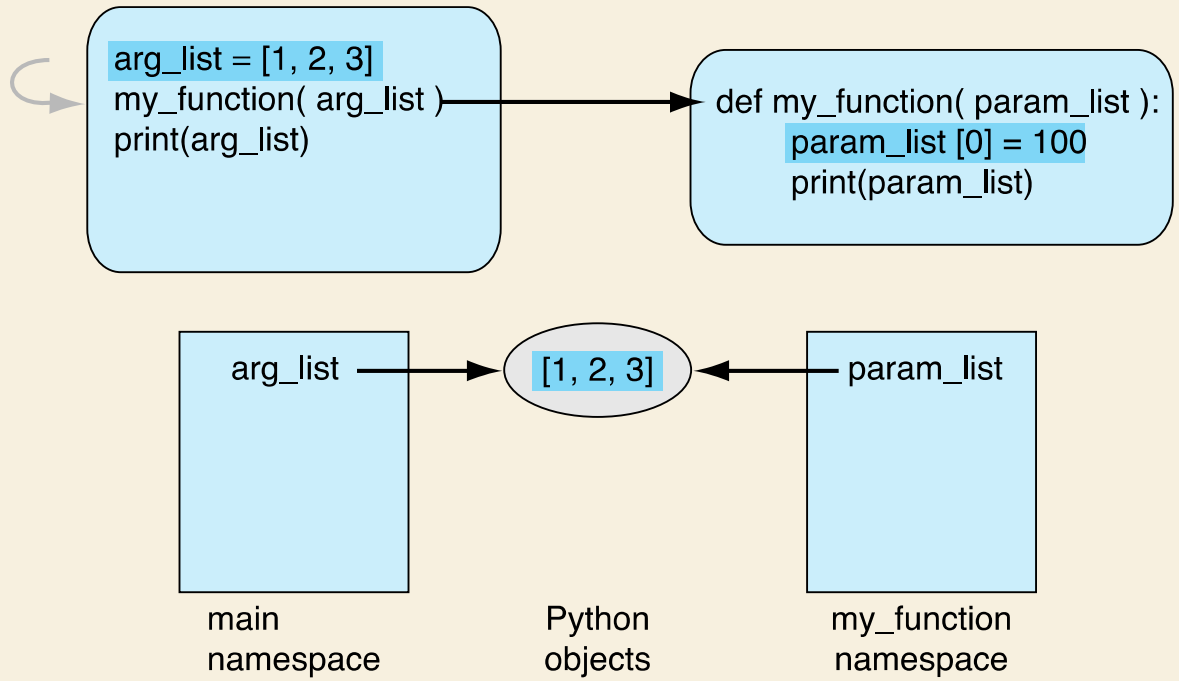
- if a parameter is assigned to a new value, then just like any other assignment, a new association is created
- This assignment does not affect the object associated with the argument, as a new association was made with the parameter



PASSING MUTABLE OBJECTS

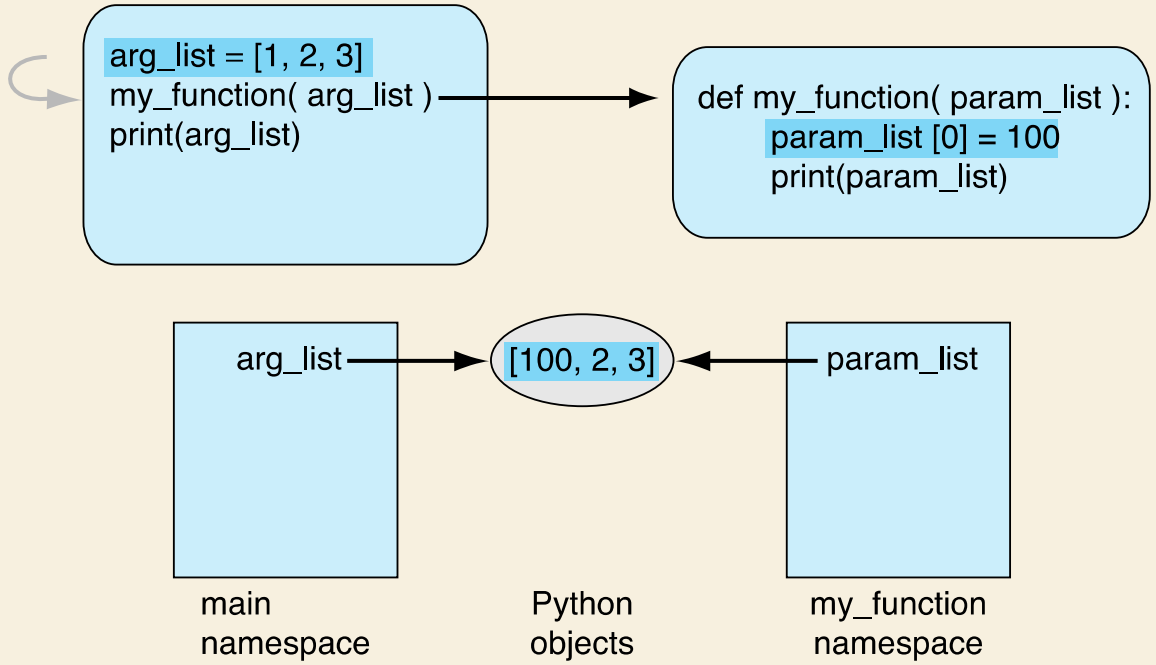
PASSING MUTABLE OBJECTS TO FUNCTIONS

- When passing mutable data structures, 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.



PASSING MUTABLE OBJECTS TO FUNCTIONS

- In this diagram we can see that the list passed to the function is modified within the function
  - Since lists are mutable it is changed directly and since both the variables arg_list and parm_list reference the same object they will both reflect the change



MORE ON FUNCTIONS

FUNCTIONS RETURN ONE THING!

- Functions can only return one thing,
  - but that one thing can be a thing. That hold lots of data
- For example, a function can return a single tuple or a single list
  - A single tuple and a single list can hold multiple values

This function returns a single list. Even though the list contains multiple values. The function is still just returning a single value

This function returns a single tuple. Even though the tuple contains multiple values. The function is still just returning a single value

```
1  def get_list():
2      return [1, 2, 3, 4, 5]
3
4  def get_tuple():
5      return 1, 2, 3, 4, 5
```

ASSIGNMENT IN A FUNCTION

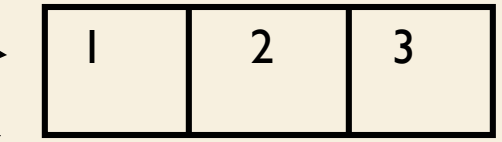
- If you assign a value in a function, that name becomes part of the local namespace of the function
- It can have some odd effects

EXAMPLE

```
1  def my_fun (param):
2      param.append(4)
3      return param
4
5
6  my_list = [1,2,3]
7  new_list = my_fun(my_list)
8
9  print(my_list, new_list)
```

Main Namespace

Name	value
my_list	



When the function is called the variable param will get a reference to the list. The same list in memory that the variable my_list is referencing

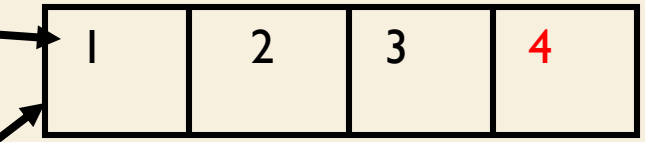
my_fun Namespace

Name	value
param	

```
1  def my_fun (param):
2      param.append(4)
3      return param
4
5
6  my_list = [1,2,3]
7  new_list = my_fun(my_list)
8
9  print(my_list, new_list)
```

Main Namespace

Name	value
my_list	



When the append method is called on the param variable we can see that the value is appended to the list. Because the variables param and my_list reference the same list in memory they both reflect that change!

my_fun Namespace

Name	value
param	

```
1  def my_fun (param):
2      param.append(4)
3      return param
4
5
6  my_list = [1,2,3]
7  new_list = my_fun(my_list)
8
9  print(my_list, new_list)
```

EXAMPLE

- Let's see what happens if we change the function a little bit

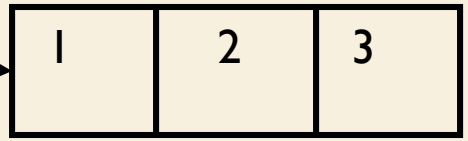
Here the function makes the variable param reference a new list in memory

```
1   def my_fun (param):
2       param=[1,2,3]
3       param.append(4)
4       return param
5
6   my_list = [1,2,3]
7   new_list = my_fun(my_list)
8
9   print(my_list,new_list)
```

- Assigning the parameter a new value
  - An assignment creates a local variable
  - Changes to a local variable affects only the local context, even if it is made on a mutable parameter

Main Namespace

Name	value
my_list	



When the function is called the variable param will get a reference to the list. The same list in memory that the variable my_list is referencing. This is exactly the same as in the previous example.

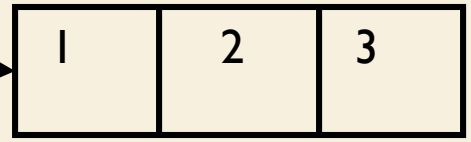
my_fun Namespace

Name	value
param	

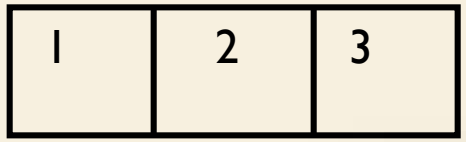
```
1   def my_fun (param):
2       param=[1,2,3]
3       param.append(4)
4       return param
5
6   my_list = [1,2,3]
7   new_list = my_fun(my_list)
8
9   print(my_list,new_list)
```

Main Namespace

Name	value
my_list	



Next the function will make the variable param reference a completly new list in memory. This happens in line two of the code example



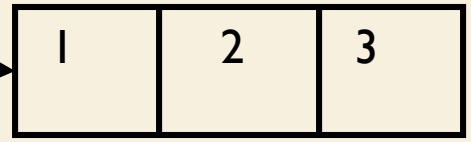
my_fun Namespace

Name	value
param	

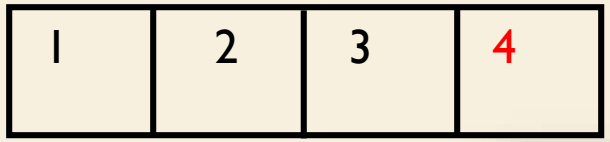
```
1   def my_fun (param):
2       param=[1,2,3]
3       param.append(4)
4       return param
5
6   my_list = [1,2,3]
7   new_list = my_fun(my_list)
8
9   print(my_list,new_list)
```

Main Namespace

Name	value
my_list	



Now we can see that when we call the append method on the variable param, the value 4 will only be appended to the list that param is currently referencing



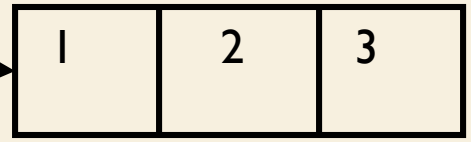
my_fun Namespace

Name	value
param	

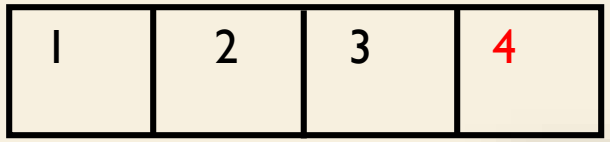
```
1   def my_fun (param):
2       param=[1,2,3]
3       param.append(4)
4       return param
5
6   my_list = [1,2,3]
7   new_list = my_fun(my_list)
8
9   print(my_list,new_list)
```

Main Namespace

Name	value
my_list	



This means that when the function call is finished a reference to the newly created list will be returned



my_fun Namespace

Name	value
param	

```
1   def my_fun (param):
2       param=[1,2,3]
3       param.append(4)
4       return param
5
6   my_list = [1,2,3]
7   new_list = my_fun(my_list)
8
9   print(my_list,new_list)
```

This line will now print [1, 2, 3] [1, 2, 3, 4]. Two completely different lists.

DEFAULT AND NAMED PARAMETERS

- We can give parameters in a function default parameters
  - This parameter assignment means two things:
  - If the caller does not provide a value, the default is the parameter's assigned value
  - You can get around the order of parameters by using the name of the parameter

If this function is called and no arguments are provided these values will be used

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

DEFAULT AND NAMED PARAMETERS

Here now arguments are supplied so all the default values are used

Here only one argument is supplied so the first parameter will get that value. The other parameters will use the default values

Here only one named argument is supplied so the parameter with the name length will get the given value. The other paramters will use the default values

```
1  def box(height=10,width=10,length=10):
2      print(height, width, length)
3
4  box() # prints: 10 10 10
5
6  box(30) # prints: 30 10 10
7
8  box(length=20) # prints: 10 10 20
```

Do note that the order of arguments does not matter when named arguments are used. But the names provided in the arguments must match the parameter names!

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

CAN ASK FOR DOCSTRING

- We have seen docstrings before
  - Every object (including functions) can have a docstring. It is stored as an attribute of the function (the __doc__ attribute)

This will print the docstring of the function listAverage

```
1  def listAverage(lis):
2      '''Takes a list of integers, returns the average of the list.'''
3      return sum(lis) / len(lis)
4
5
6  print(listAverage.__doc__)
```

- **Other programs can use the docstring to report to the user**