Overview: Function Return Values

```
def connect(dev_IP, username, password):
    ...
    return session
```

value to return to caller

```
...
session = connect('10.0.0.1','cisco','cisco')
```

returned value

Returning Values

- **Returning single items**

```
return session
session = connect(...)    # return session object
```

Returning multiple items

```
return [dev1,dev2,dev3]      # return list
return (username, password) # return tuple
return username, password    # return two values
```


Returning values to callers is one of the more important attributes of functions. In Python, your function can return values to the caller using the 'return' statement.

The returned value is received by the caller using a simple assignment statement:

```
def connect(dev_IP, username, password):
   ...
   return session


...
session = connect('10.0.0.1','cisco','cisco')
```

Your function can return simple values or it can return more complex data such as lists, dictionaries, and so on. It is even possible to return another function.

In the example, the connect function wishes to return the Pexpect 'session' object to the caller, so that more operations can be done with that session. In the function itself, this is done using the return statement, 'return session'.
In the caller, the output of the connect function call is the value of the session, which is assigned to the variable called 'session' in the calling code. The calling code could have assigned the return value to any variable – it could be called 'p_session' or 'my_session' or 'connection' or any name you desire.

```
connection = connect('10.0.0.1','cisco','cisco')
```

Recall also about a variable's scope. The variable 'session' inside the function itself is entirely separate and distinct from the outer, global variable 'session' that received the return value in the original example.

What values can be returned by a function? In Python there are many options:

It is possible to return single items, as in the following examples:

```
def connect(...):
   return session
...
session = connect(...
```

In the example above, the function returns Pexpect 'session' object, received from Pexpect after the `connect()` function performed the required connection and login steps.

```
def get_ip_address(...):
   return ip_address
...
ip = get_ip_address(...)
```

In the example above, the function returns a string holding information regarding the IP address requested by the caller.

```
def get_route(...):
   return route
...
new_route = get_route(...)
```

In the example above, the get_route function calculates and returns the route.

It is also possible to return multiple or more complex items:

```
def get_devs(...):
   return [dev1,dev2,dev3]
...
dev_list = get_devs(...)
```

In the example above, the get_devs function returns a Python list of devices.

```
def get_credentials(...):
   return (username, password)
...
cred = get_credentials(...)
```

In the example above, the get_credentials function returns a Python tuple containing the username and password credentials for a specific device.

Parameters are Passed by Object Reference

For software developers, an important question to ask is "how are the parameters passed to functions?" This is important because if the answer is "passed by value," that means that the function can change the parameter, and it will have no effect on the calling program – that parameter passed by value, from the perspective of the calling program, will not have changed at all.

On the other hand, parameters that are "passed by reference," can actually change; in other words, if the function changes the value of that parameter, then from the calling application, the value has also changed.

Python passes everything by reference, since "everything is an object," which means that, for mutable objects, if you change them in your function, the calling application will see those changes. Specifically, if the parameter passed is mutable such as a list or dictionary, and you append or add an item, that change will be visible to the calling code.

For immutable data types such as strings or integers, what happens is the same thing that happens everywhere in Python – a new object is allocated with the new value. Hence, your changes within your function are not seen by the calling program, because its reference still points at the original piece of data.

Gathering Optional Parameters

You may see this type of code in Python applications you inherit, in which the parameters that are listed in the function definition have one or two asterisks ('*' or '**') preceding them. This indicates that the code is making use of a Python feature allowing the caller to specify a variable number of parameters when calling a function.

Note that this is entirely different from default values for parameters. With default values, Python knows exactly how many parameters are defined for the function; the 'optional' part means that the caller can specify fewer than the total number of parameters, and the ones that are not provided will be assigned default values.

In this situation, the function definition does not know how many parameters will be passed. It could be none, or one, or very many.

```
Positional:
def function_name(*args):
    tuple_input = args
```

```
Keyword:
def function_name(**kwargs):
    dict_input = kwargs
```

As mentioned, there are two types of optional parameter models, 'positional', and 'keyword', identified by whether the function definition specifies one asterisk (positional), or two asterisks (keyword). The name of the parameter in the function definition does not matter, but by convention, for positional parameters, 'arg' is used, and for keyword parameters, 'kwargs' is used.

For positional parameters passed in this way, Python wraps them up into a tuple, and your function is able to access and use the items in the tuple .

```
def connect(*args):
   connect_tuple = args
   ...
...
session=connect(ip,user,pw)
```

The example above shows what Python would do if the 'connect' function used a positional optional parameter, rather than three discreet values. In the example, the first item in the tuple would be the device IP address; the second would be the username; and the third would be the password.

For keyword parameters, passed in this way, Python wraps them up into a dictionary (with the keyword part becoming the key, and the actual value becoming the value for the item in the dictionary.

```
def connect(**kwargs):
   connect_dict = kwargs
   ...
...
session=connect(ip='10.0.0.1',
                user='cisco',
                pw='cisco')
```

The example above shows what Python would do if the 'connect' function used a keyword optional parameter, rather than discreet keyword values. In the example, the connect function would receive a dictionary, with keys being 'ip', 'user', and 'pw', with their accompanying values.