Chapter 5 Python Functions

Pass-by-object (Binding names to objects)

In [1]:
```
some_guy = 'Fred'
# ...
some_guy = 'George'
```

On line 1, we create a binding between a name, some_guy, and a string object containing 'Fred'. In the context of program execution, the environment is altered; a binding of the name 'some_guy' to a string object is created in the scope of the block where the statement occurred. When we later say some_guy = 'George', the string object containing 'Fred' is unaffected. We've just changed the binding of the name some_guy. We haven't, however, changed either the 'Fred' or 'George' string objects. Let's see another example as follows.

In [2]:
```
some_guy = 'Fred'

first_names = []
first_names.append(some_guy)

another_list_of_names = first_names
another_list_of_names.append('George')
some_guy = 'Bill'

print (some_guy, first_names, another_list_of_names)
```

```
Bill ['Fred', 'George'] ['Fred', 'George']
```

What get's printed in the final line? The binding of some_guy to the string object containing 'Fred' is added to the block's namespace. The name first_names is bound to an empty list object. On line 4, a method is called on the list object first_names is bound to, appending the object some_guy is bound to. At this point, there are still only two objects that exist: the string object and the list object. some_guy and first_names[0] both refer to the same object (Indeed, print(some_guy is first_names[0]) shows this).

Let's continue to break things down. On line 6, a new name is bound: another_list_of_names. Assignment between names does not create a new object. Rather, both names are simply bound to the same object. As a result, the string object and list object are still the only objects that have been created by the interpreter. On line 7, a member function is called on the object another_list_of_names is bound to and it is mutated to contain a reference to a new object: 'George'.

This brings us to an important point: there are actually two kinds of objects in Python. A mutable object exhibits time-varying behavior. Changes to a mutable object are visible through all names bound to it. Python's lists are an example of mutable objects. An immutable object does not exhibit time-varying behavior. The value of immutable objects can not be modified after they are created.

If I call foo(bar), I'm merely creating a binding within the scope of foo to the object the argument bar is bound to when the function is called. If bar refers to a mutable object and foo changes its value, then these changes will be visible outside of the scope of the function.

In [6]:
```
def foo(bar):
    bar.append(42)
    print(bar)
    # >> [42]

answer_list = []
foo(answer_list)
print(answer_list)
# >> [42]
```

```
[42]
[42]
```

On the other hand, if bar refers to an immutable object, the most that foo can do is create a name bar in its local namespace and bind it to some other object.

In [7]:
```
def foo(bar):
    bar = 'new value'
    print (bar)
    # >> 'new value'

answer_list = 'old value'
foo(answer_list)
print(answer_list)
# >> 'old value'
```

```
new value
old value
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways

In [9]:
```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print(str)
   return;

# Now you can call printme function
printme( str = "My string")
```

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

In [11]:
```
# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print("Name: ", name)
   print("Age ", age)
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

```
Name:  miki
Age  50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed.

In [12]:
```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print("Name: ", name)
   print("Age ", age)
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example.

In [13]:
```
# Function definition is here
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print("Output is: ")
   print(arg1)
   for var in vartuple:
      print(var)
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

```
Output is:
10
Output is:
70
60
50
```

The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

1. Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

2. An anonymous function cannot be a direct call to print because lambda requires an expression

3. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

4. Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

The syntax of lambda functions contains only a single statement, which is as follows

lambda [arg1 [,arg2,.....argn]]:expression

In [14]:
```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print("Value of total : ", sum( 10, 20 ))
print("Value of total : ", sum( 20, 20 ))
```

```
Value of total :  30
Value of total :  40
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

In [1]:
```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

Other Use of Lambda Function in Python

We use lambda functions when we require a nameless function for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

In [2]:
```
# Program to filter out only the even items from a list

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

# Output: [4, 6, 8, 12]
print(new_list)
```

```
[4, 6, 8, 12]
```

In [3]:
```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

# Output: [2, 10, 8, 12, 16, 22, 6, 24]
print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

In [ ]:
```

```