More on Functions

In this section you will learn about issues related to function use, including more on function invocation and parameter passing.

Calling Value-Returning Functions

Calls to value-returning functions can be used anywhere that a function's return value is appropriate,

```
result = max(num_list) * 100
```

Here, we apply built-in function `max` to a list of integers, `num_list`. Examples of additional allowable forms of function calls are given below.

```
(a)    result = max(num_list1) * max(num_list2)
(b)    result = abs(max(num_list))
(c)    if max(num_list) , 10:...
(d)    print('Largest value in num_list is ', max(num_list))
```

The examples demonstrate that an expression may contain multiple function calls, as in (a); a function call may contain function calls as arguments, as in (b); conditional expressions may contain function calls, as in (c); and the arguments in print function calls may contain function calls, as in (d).

What if a function is to return more than one value, such as function `maxmin` to return *both* the maximum and minimum values of a list of integers? In Python, we can do this by returning the two values as a single tuple,

function definition

```
def maxmin(num_list):
return (max(num_list), min(num_list))
```

function use

```
weekly_temps = [45, 30, 52, 58, 62, 48, 49]
```

(a) `highlow_temps = maxmin(weekly_temps)`
(b) `high, low = maxmin(weekly_temps)`

In (a) above, the returned tuple is assigned to a single variable, `highlow_temps`. Thus, `highlow_ temps[0]` contains the maximum temperature, and `highlow_temps[1]` contains the minimum temperature. In (b), however, a *tuple assignment* is used. In this case, variables `high` and `low` are each assigned a value of the tuple based on the order that they appear. Thus, `high` is assigned to the tuple value at index 0, and `low` the tuple value at index 1 of the returned tuple.

Note that it does not make sense for a call to a value-returning function to be used as a statement, for example,

```
max(num_list)
```

Such a function call does not have any utility because the expression would evaluate to a value that is never used and thus is effectively “thrown away.”

Finally, we can design value-returning functions that do not take any arguments, as we saw in the `getConvertTo` function of the previous temperature conversion program. Empty parentheses are used in both the function header and the function call. This is needed to distinguish the identifier as denoting a function name and not a variable.

Your Turn

Enter the definitions of functions `avg` and `minmax` given above. Then enter the following function calls and observe the results.

```
>>> avg(10,25,40)
???

>>>avg(10,25,40) + 10
???

>>>if avg(10,25,240) ,0:
       print 'Invalid avg'
???

>>> avg(avg(2,4,6),8,12)
???

>>>avg(1,2,3) * avg(4,5,6)
???
```

```
>>>num_list = [10,20,30]

>>>max_min = maxmin(num_list)
>>>max_min[0]
???

>>> max_min[1]
???

>>>max, min = maxmin(num_list)
>>>max
???

>>>min
???
```

Part II - Calling Non-Value-Returning Functions

As we have seen, non-value-returning functions are called for their side effects, and not for a returned function value. Thus, such function calls are statements, and therefore can be used anywhere that an executable statement is allowed. Consider such a function call to `display-Welcome`

```
displayWelcome()
```

It would not make sense to treat this function call as an expression, since no meaningful value is returned (only the default return value `None`). Thus, for example, the following assignment statement would not serve any purpose,

```
welcome_displayed = displayWelcome()
```

Finally, as demonstrated by function `displayWelcome()`, functions called for their side effects can be designed to take no arguments, the same as we saw for value-returning functions. Parentheses are still included in the function call to indicate that identifier `displayWelcome` is a function name, and not a variable.

Your Turn

Enter the definition of function `hello` given below, then enter the following function calls and observe the results.

```
>>> def sayHello():
      print('Hello!')

>>>sayHello()
???

>>>t = sayHello()
???

>>> t = = None
???
```

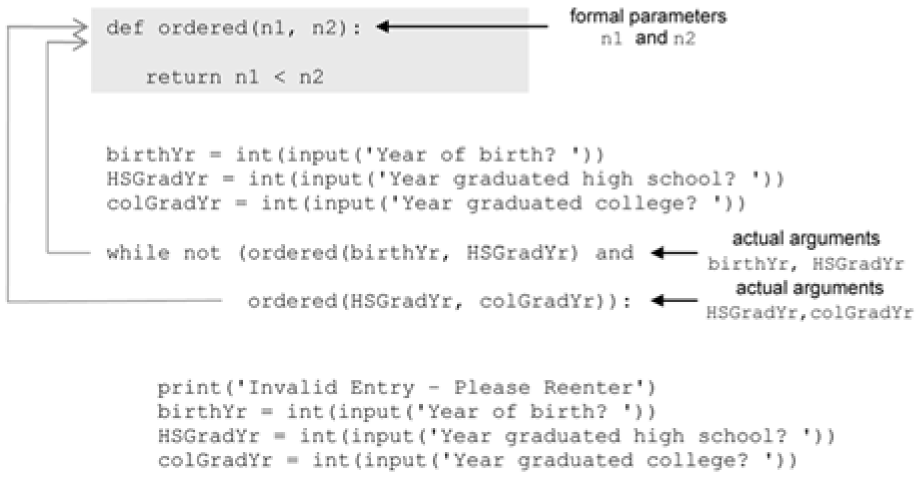
```
>>>def buildHello(name):
        return 'Hello' + name + '!'

>>>greeting = buildHello('Charles')
>>>print (greeting)
???

>>>buildHello ('Charles')
???

>>>buildHello ()
???
```

Part III - Parameter Passing

Now that we have discussed how functions are called, we take a closer look at the passing of arguments to functions.

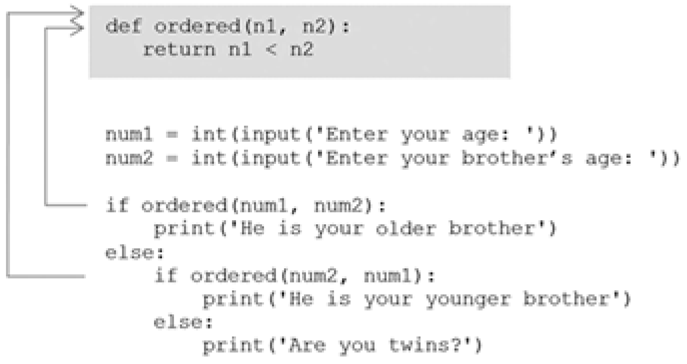
Actual Arguments vs. Formal Parameters

Parameter passing is the process of passing arguments to a function. As we have seen, actual arguments are the values passed to a function's formal parameters to be operated on. This is illustrated below:


```
def ordered(n1, n2):

   return n1 < n2


birthYr = int(input('Year of birth? '))
HSGradYr = int(input('Year graduated high school? '))
colGradYr = int(input('Year graduated college? '))

while not (ordered(birthYr, HSGradYr) and
           ordered(HSGradYr, colGradYr)):

    print('Invalid Entry - Please Reenter')
    birthYr = int(input('Year of birth? '))
    HSGradYr = int(input('Year graduated high school? '))
    colGradYr = int(input('Year graduated college? '))
```

Here, the values of `birthYr` (the user's year of birth) and `HSGradYr` (the user's year of high school graduation) are passed as the actual arguments to formal parameters n1 and n2. Each call is part of the same Boolean expression `ordered(birthYr, HSGradYr) and ordered(HSGradYr, colGradYr)`. In the second function call of the expression, a different set of values `HSGradYr` and `colGradYr` are passed. Formal parameter names n1 and n2, however, remain the same.

Note that the correspondence of actual arguments and formal parameters is determined by the order of the arguments passed, and not their names. Thus, for example, it is perfectly fine to pass an actual argument named `num2` to formal parameter n1, and actual argument `num1` to formal parameter n2, as shown below:.


```
def ordered(n1, n2):
   return n1 < n2


num1 = int(input('Enter your age: '))
num2 = int(input('Enter your brother's age: '))

if ordered(num1, num2):
    print('He is your older brother')
else:
    if ordered(num2, num1):
        print('He is your younger brother')
    else:
        print('Are you twins?')
```

In this example, function `ordered` is called once with arguments `num1`, `num2` and a second time with arguments `num2`, `num1`. Each is a proper function call and each is what is logically needed in this instance.

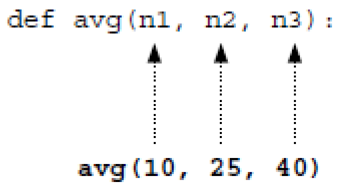
Your Turn

Enter the definition of function `ordered` given above into the Python Shell. Then enter the following and observe the results.

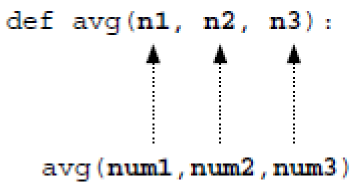
```
>>> nums_1 = [5,2,9,3]          >>>ordered(max(nums_1), max(nums_2))
>>> nums_2 = [8,4,6,1]          ???
                                >>>ordered(min(nums_1), min(nums_2))
```

Part IV - Mutable vs. Immutable Arguments

There is an issue related to parameter passing that we have yet to address. We know that when a function is called, the current values of the arguments passed become the initial values of their corresponding formal parameters,



In this case, literal values are passed as the arguments to function `avg`. When variables are passed as actual arguments, however, as shown below,



there is the question as to whether any changes to formal parameters `n1`, `n2`, and `n3` in the function result in changes to the corresponding actual arguments `num1`, `num2`, and `num3`. In this case, function `avg` doesn't assign values to its formal parameters, so there is no possibility of the actual arguments being changed. Consider, however, the following function,

```
def countDown(n):
    while n >5 0:
       if (n != 0):
            print(n, '..', end='')
        else:
            print(n)
       n = n -1
```

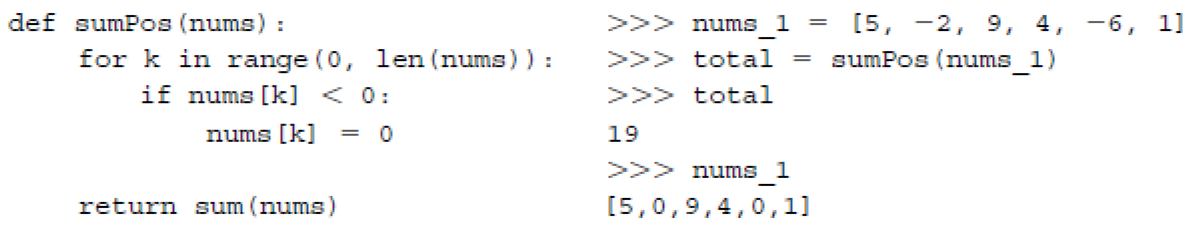
This function simply displays a countdown of the provided integer parameter value. For example, function call `countDown(4)` produces the following output,

```
4 . . 3 . . 2 . . 1 . . 0
```

What if the function call contained a variable as the argument, for example, `countDown(num_tics)`? Since function `countDown` alters the value of formal parameter n, decrementing it until it reaches the value −1, does the corresponding actual argument `num_tics` have value −1 as well?

```
>>> num_tics = 10
>>> countDown(num_tics)
>>> num_tics
???
```

If you try this, you will see that `num_tics` is unchanged. Now consider the following function,



```
def sumPos(nums):                        >>> nums_1 = [5, −2, 9, 4, −6, 1]
    for k in range(0, len(nums)):        >>> total = sumPos(nums_1)
        if nums[k] < 0:                  >>> total
            nums[k] = 0                  19
                                         >>> nums_1
    return sum(nums)                     [5,0,9,4,0,1]
```

Function `sumPos` returns the sum of only the positive numbers in the provided argument. It does this by first replacing all negative values in parameter `nums` with 0, then summing the list using built-in function `sum`. You can see above that the corresponding actual argument `nums_1` has been altered in this case, with all of the original negative values set to 0.

The reason that there was no change in integer argument `num_tics` above but there was in list argument `nums_1` has to do with their types. Lists are mutable. Thus, arguments of type list will be altered if passed to a function that alters its value. Integers, floats, Booleans, strings, and tuples, on the other hand, are immutable. Thus, arguments of these types cannot be altered as a result of any function call.

It is generally better to design functions that do not return results through their arguments. In most cases, the result should be returned as the function's return value. What if a function needs to return more than one function value? The values can be returned in a tuple, as discussed above.

Your Turn

Enter the following and observe the results.

```
>>> num = 10             >>>nums_1 = [1,2,3]          >>>nums_2 = (1,2,3)
>>> def incr(n):         >>>def updates(n):           >>>update(nums_2)
>>> incr(num)            >>>update(nums_1)            >>> ???
>>> num                  >>>nums_1
???                       ???
```

Concepts and Procedures

1. A function call can be made anywhere within a program in which the return type of the function is appropriate. (TRUE/FALSE)

2. An expression may contain more than one function call. (TRUE/FALSE)

3. Function calls may contain arguments that are function calls. (TRUE/FALSE)
4. All value-returning functions must contain at least one parameter. (TRUE/FALSE)

5. Every function must have at least one mutable parameter. (TRUE/FALSE)

Problem Solving

1. Suppose there are nine variables, each holding an integer value as shown below, for which the average of the largest value in each line of variables is to be computed.

```
num1 = 10      num2 = 20      num3 = 25   :   max1 = 25
num4 = 5       num5 = 15      num6 = 35   :   max2 = 35
num7 = 20      num8 = 30      num9 = 25   :   max3 = 30

average = (max1 + max2 + max3) / 3.0
        = (25 + 35 + 30) / 3.0
        = 30.0
```

Using functions `avg` and `max`, give an expression that computes the average as shown above.

2. Assume that there exists a Boolean function named `isLeapYear` that determines if a given year is a leap year or not. Give an appropriate if statement that prints “Year is a Leap Year” if the year passed is a leap year, and “Year is Not a Leap Year” otherwise, for variable year.

3. For the following function definition and associated function calls,

```
def somefunction(n1, n2):
    .
    .
# main
num1 = 10
somefunction(num1, 15)
```

3 (a) List all the formal parameters.

3 (b) List all the actual arguments.

4. For the following function, indicate whether each function call is proper or not. If improper, explain why.

```
def gcd(n1, n2):
```

function gcd calculates the greatest common divisor of n1 and n2, with the requirement that n1 be less than or equal to n2, and n1 and n2 are integer values.

(a)
```
a = 10
b = 20
result = gcd(a, b)
```

(b)
```
a = 10.0
b = 20
result = gcd(a, b)
```

(c)
```
a = 20
b = 10
result = gcd(b, a)
```

(d)
```
a = 10
b = 20
c = 30
result = gcd(gcd(a, b), c)
```

(e)
```
a = 10
b = 20
c = 30
print(gcd(a, gcd(c, b)))
```