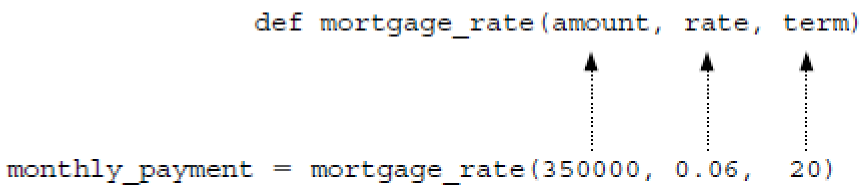
Still More Functions

Keyword Arguments in Python

The functions we have looked at so far were called with a fixed number of positional arguments. A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list, as illustrated below.

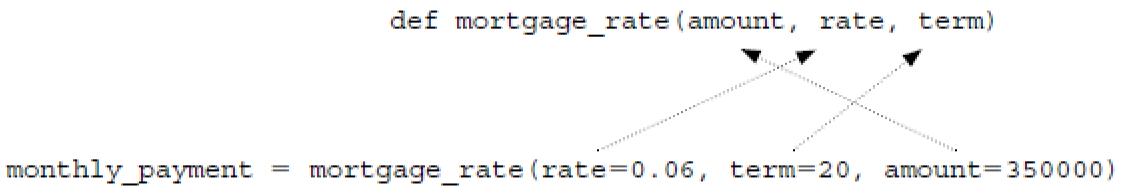


```
def mortgage_rate(amount, rate, term)

monthly_payment = mortgage_rate(350000, 0.06,  20)
```

This function computes and returns the monthly mortgage payment for a given loan amount (amount), interest rate (rate), and number of years of the loan (term).

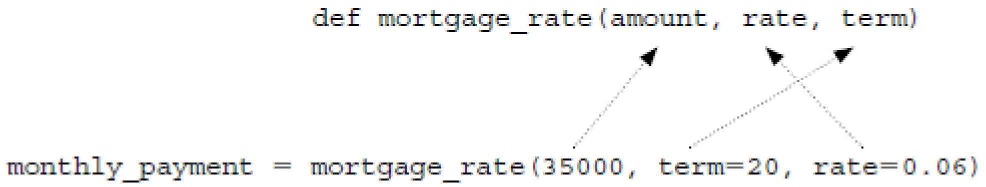
Python provides the option of calling any function by the use of keyword arguments. A **key-word argument** is an argument that is specified by parameter name, rather than as a positional argument as shown below (note that keyword arguments, by convention, do not have a space before or after the equal sign),



```
def mortgage_rate(amount, rate, term)

monthly_payment = mortgage_rate(rate=0.06, term=20, amount=350000)
```

This can be a useful way of calling a function if it is easier to remember the parameter names than it is to remember their order. It is possible to call a function with the use of both positional and keyword arguments. However, all positional arguments must come before all keyword arguments in the function call, as shown below.



```
def mortgage_rate(amount, rate, term)

monthly_payment = mortgage_rate(35000, term=20, rate=0.06)
```

This form of function call might be useful, for example, if you remember that the first argument is the loan amount, but you are not sure of the order of the last two arguments `rate` and `term`.

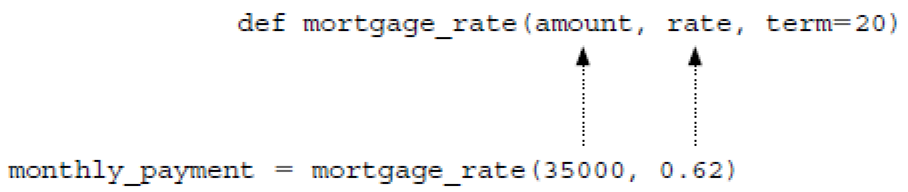
Your Turn

Enter the following function definition in the Python Shell. Execute the statements below and observe the results.

```
>>> def addup(first, last):                  >>>addup(1,10)
                                               ???
    if first > last:
        sum  = -1
    else:                                    >>>addup(first=1, last=10)
        sum = 0                               ???
        for i in range(first, last+1):
            sum = sum + i                   >>>addup(last=10, first =1)
     return sum                              ???
```

Part II - Default Arguments in Python

Python also provides the ability to assign a default value to any function parameter allowing for the use of default arguments. A **default argument** is an argument that can be optionally provided, as shown here



```
def mortgage_rate(amount, rate, term=20)

monthly_payment = mortgage_rate(35000, 0.62)
```

In this case, the third argument in calls to function mortgage_rate is optional. If omitted, parameter term will default to the value 20 (years) as shown. If, on the other hand, a third argument is provided, the value passed replaces the default parameter value. All positional arguments must come before any default arguments in a function definition.

Your Turn

Enter the following function definition in the Python Shell. Execute the statements below and observe the results.

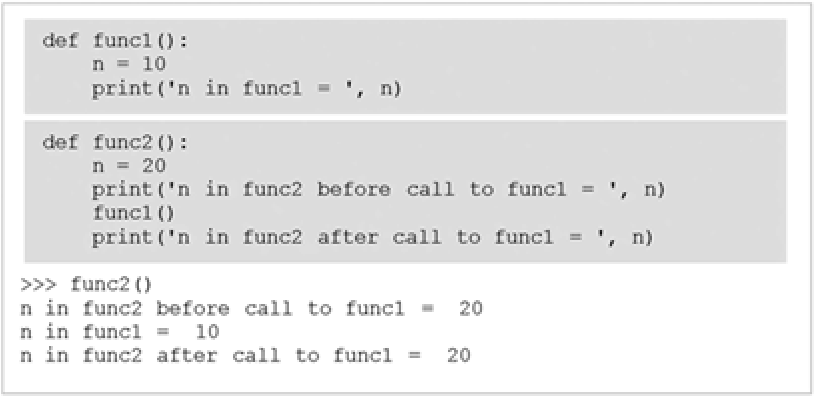
```
>>> def addup(first, last, incr=1):          >>>addup(1,10)
                                               ???
    if first > last:                          >>>addup(1,10,2)
         sum  =-1
    else:                                      >>>addup(first=1, last=10)
         sum =0                                ???
         for i in range(first, last+1, incr):
              sum = sum + i
     return sum                                >>>addup(last=10, first =1)
                                               ???
```

Part III - Variable Scope

Looking back at the temperature conversion program, you can see that functions `display FahrenToCelsius` and `displayCelsiusToFahren` each contain variables named `temp` and `converted_temp`. We ask, “Do these identifiers refer to common entities, or does each function have its own distinct entities?” The answer is based on the concept of identifier scope, which we discuss next.

Local Scope and Local Variables

A local variable is a variable that is only accessible from within a given function. Such variables are said to have local scope. In Python, any variable assigned a value in a function becomes a local variable of the function. Consider the example below:



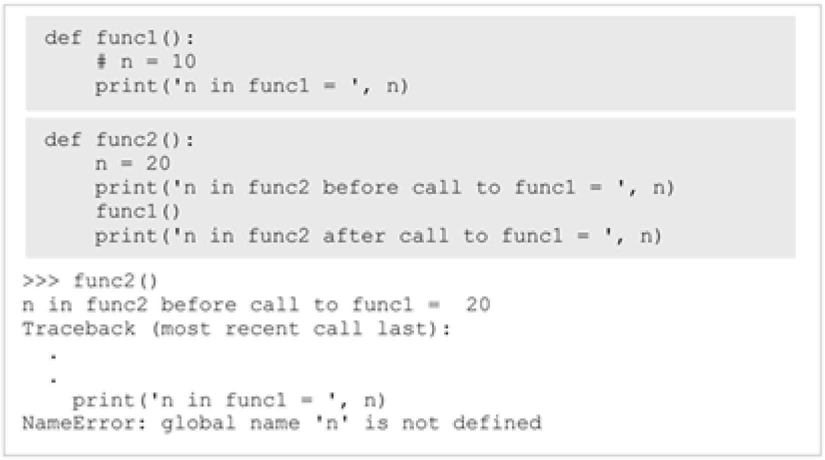
```
def func1():
    n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)

>>> func2()
n in func2 before call to func1 =  20
n in func1 =  10
n in func2 after call to func1 =  20
```

Both `func1` and `func2` contain identifier n. Function `func1` assigns n to 10, while function `func2` assigns n to 20. Both functions display the value of n when called - `func2` displays the value of n both before and after its call to func1. If identifier n represents the same variable, then shouldn't its value change to 10 after the call to `func1`? However, as shown by the output, the value of n remains 20. This is because there are two distinct instances of variable n, each local to the function assigned in and inaccessible from the other.

Now consider the example below. In this case, the functions are the same as above except that the assignment to variable n in `func1` is commented out.



```
def func1():
    # n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)
```

```
>>> func2()
n in func2 before call to func1 =  20
Traceback (most recent call last):
  .
  .
    print('n in func1 = ', n)
NameError: global name 'n' is not defined
```

In this case, we get an error indicating that variable n is not defined within `func1`. This is because variable n defined in `func2` is inaccessible from `func1`.

The period of time that a variable exists is called its **lifetime**. Local variables are automatically created (allocated memory) when a function is called, and destroyed (deallocated) when the function terminates. Thus, the lifetime of a local variable is equal to the duration of its function's execution. Consequently, the values of local variables are not retained from one function call to the next.

The concept of a local variable is an important one in programming. It allows variables to be defined in a function without regard to the variable names used in other functions of the program. It also allows previously written functions to be easily incorporated into a program.

Your Turn

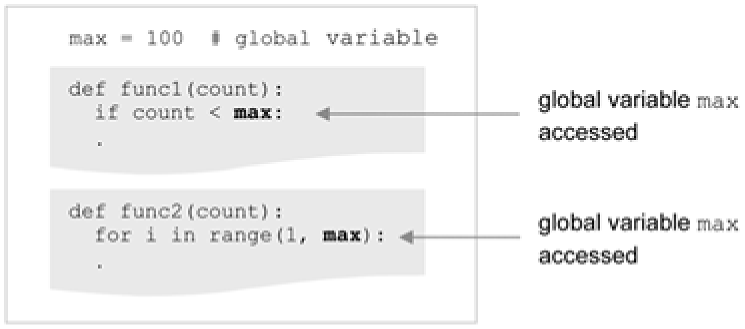
Enter the following function definition in the Python Shell. Execute the statements below and observe the results.

```
>>> def func1():
        some_var = 10
```

```
>>> func1()
>>> some_var
???
```

Part IV - Global Variables and Global Scope

A **global variable** is a variable that is defined outside of any function definition. Such variables are said to have global scope. This is illustrated below:



```
max = 100  # global variable

def func1(count):
  if count < max:
  .

def func2(count):
  for i in range(1, max):
  .
```

Variable `max` is defined outside `func1` and `func2` and therefore “global” to each. As a result, it is directly accessible by both functions. For this reason, *the use of global variables is generally considered to be bad programming style*. Although it provides a convenient way to share values among functions, *all* functions within the scope of a global variable can access and alter it. This may include functions that have no need to access the variable, but none-the-less may unintentionally alter it.

Another reason that the use of global variables is bad practice is related to code reuse. If a function is to be reused in another program, the function will not work properly if it is reliant on the existence of global variables that are nonexistent in the new program. Thus, it is good programming practice to design functions so all data needed for a function (other than its local variables) are explicitly passed as arguments, and not accessed through global variables.

Concepts and Procedures

1. A local variable in Python is a variable that is,
   - (a) defined inside of every function in a given program
   - (b) local to a given program
   - (c) only accessible from within the function it is defined

2. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)

3. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

Problem Solving

1. Write a Python function named `zeroCheck` that is given three integers, and returns true if any of the integers is 0, otherwise it returns false.

2. Write a Python function named `ordered3` that is passed three integers, and returns true if the three integers are in order from smallest to largest, otherwise it returns false.

3. Write a Python function named `modCount` that is given a positive integer, n, and a second positive integer, m <= n, and returns how many numbers between 1 and n are evenly divisible by m.

4. Write a Python function named `helloWorld` that displays "Hello World, my name is name", for any given name passed to the routine.

5. Write a Python function named `printAsterisks` that is passed a positive integer value n, and prints out a line of n asterisks. If n is greater than 75, then only 75 asterisks should be displayed.