

Chapter 5 Functions

Up until this point, we have viewed a computer program as a single series of instructions. Most programs, however, consist of distinct groups of instructions, each of which accomplishes a specific task. Such a group of instructions is referred to as a “routine.” Program routines, called “functions” in Python, are fundamental building blocks in software development. We take our first look at functions in this chapter.

Motivation

So far, we have limited ourselves to using only the most fundamental features of Python— variables, expressions, control structures, input/print, and lists. In theory, these are the only instructions needed to write any program. From a *practical* point-of-view, however, these instructions alone are not enough.

The problem is one of complexity. Some smart phones, for example, contain over 10 million lines of code. In order to manage such complexity, programs are divided into manageable pieces called program routines (or simply *routines*). Doing so is a form of *abstraction* in which a more general, less detailed view of a system can be achieved. In addition, program routines provide the opportunity for code reuse, so that systems do not have to be created from scratch. Routines, therefore, are a fundamental building block in software development.

We look at the definition and use of program routines in Python.

Term	Number of Lines of Code (LOC)	Equivalent Storage
KLOC	1,000	Application Programs
MLOC	1,000,000	Operating Systems / Smart Phones
GLOC	1,000,000,000	Number of lines of code in existence for various programming languages

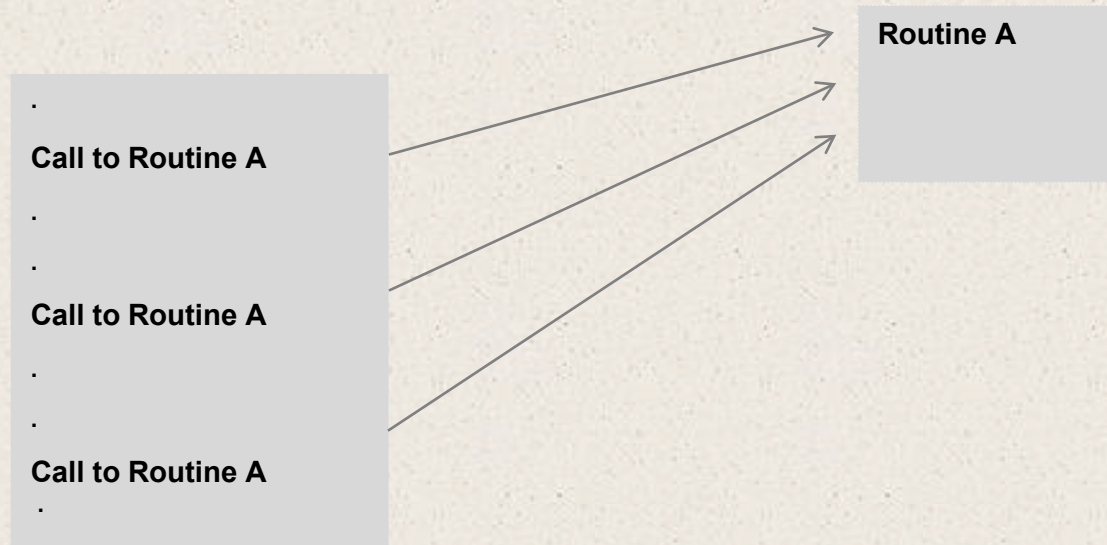
Measures of Lines of Program Code

Program Routines

We first introduce the notion of a program routine. We then look at program routines in Python, called functions. We have already been using Python's built-in functions such as `len`, `range`, and others. We now look more closely at how functions are used in Python, as well as how to define our own.

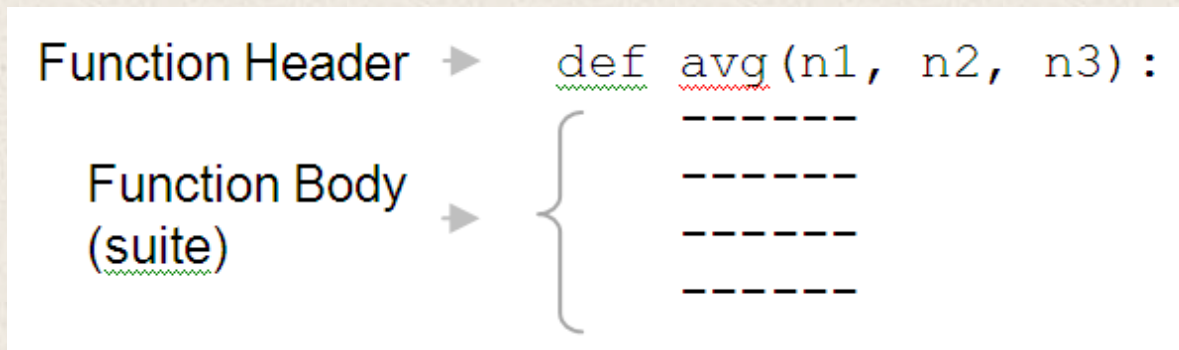
What is a Function Routine?

A **routine** is a named group of instructions performing **some task**. A routine can be **invoked** (called) as many times as needed in a given program



When a routine terminates, execution automatically returns to the point from which it was called. Such routines may be predefined in the programming language, or designed and implemented by the programmer. A **function** is Python's version of program routine.

Defining Functions



- A **function header** starts with the keyword **def**, followed by an identifier (`avg`), which is the function's name.
- The function name is followed by a comma-separated (possibly empty) list of identifiers (`n1, n2, n3`) called **formal parameters**, or simply "**parameters**." Following the **parameter list** is a colon (`:`).
- Following the function header is the **function body**, a **suite** (program block) containing the function's instructions. As with all suites, the statements must be indented at the same level, relative to the function header.

The number of items in a parameter list indicates the number of values that must be passed to the function, called **actual arguments** (or simply “arguments”), such as variables `num1`, `num2`, and `num3` below.

```
>>> num1 = 10
>>> num2 = 25
>>> num3 = 16

>>> avg(num1, num2, num3)
```

Functions are generally defined at the top of a program. However, **every function must be defined before it is called.**

Value-Returning Functions

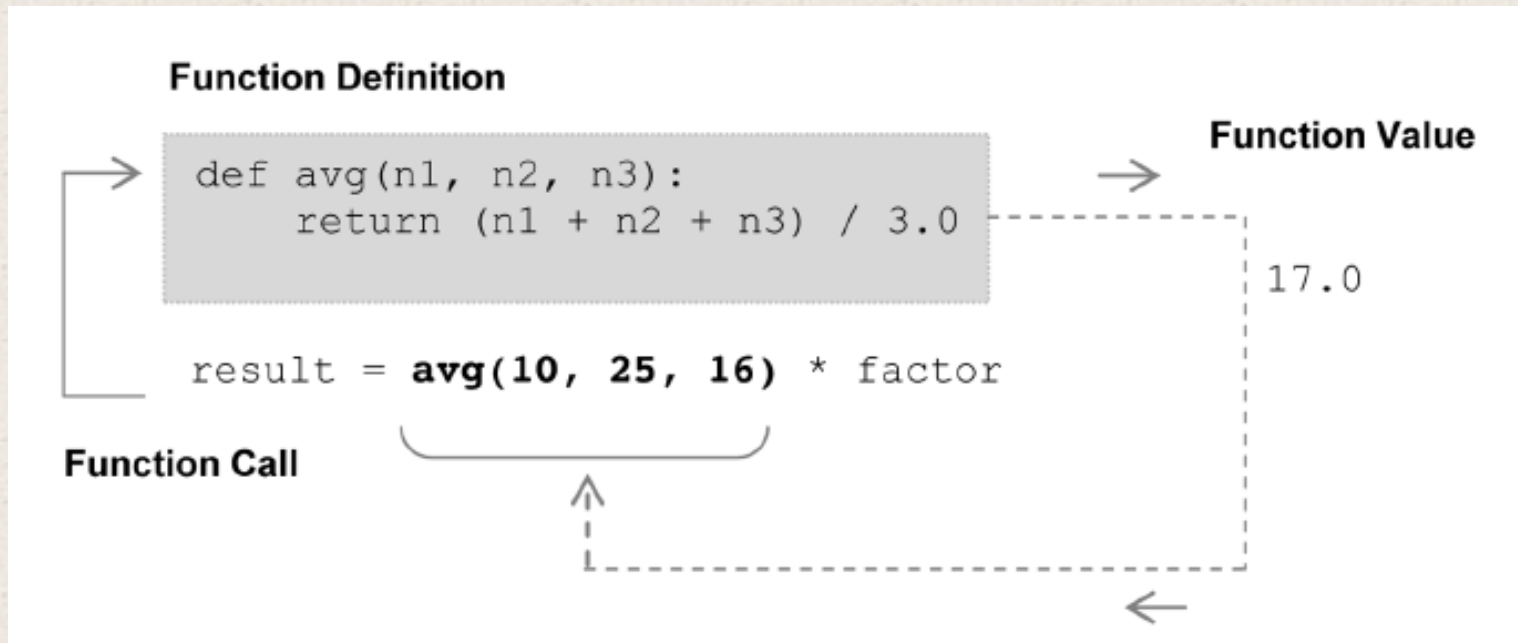
A **value-returning function** is a program routine called for its return value, and is therefore similar to a mathematical function, e.g.,

$$f(x) = 2x$$

In this notation, “x” stands for any numeric value that function f may be applied to,

$$f(2) = 2 \times 2 = 4$$

Program functions are similarly used.



Function **avg** takes three arguments (**n1**, **n2**, and **n3**) and returns the average of the three. The function call **avg(10, 25, 16)**, therefore, is an expression that evaluates to the returned function value. This is indicated in the function's return statement of the form **return expr**, where **expr** may be any expression.

Let's Try It

From the Python Shell, first enter the following function, making sure to indent the code as given. Hit return twice after the last line of the function is entered. Then enter the following function calls and observe the results.

```
>>> def avg(n1, n2, n3):  
    return (n1 + n2 + n3) / 3.0
```

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


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

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

Non-Value-Returning Functions

A **non-value-returning function** is called not for a returned value, but for its *side effects*. A **side effect** is an action other than returning a function value, such as displaying output on the screen.

Function Definition



```
def displayWelcome():  
    print('This program will convert between Fahrenheit and Celsius')  
    print('Enter (F) to convert Fahrenheit to Celsius')  
    print('Enter (C) to convert Celsius to Fahrenheit')
```

```
# main
```

```
.  
displayWelcome()
```


Let's Try It

From the Python Shell, first enter the following function, making sure to indent the code as given. Then enter the following function calls and observe the results.

```
>>> def hello(name):  
    print('Hello', name + '!')
```

```
>>> name = 'John'  
>>> hello(name)  
???
```

Function Calls Overview

- A call to a value-returning function is an expression. It evaluates to the value returned by the function call. Thus, calls to value-returning functions are made part of a larger expression or instruction,

```
result = avg(10, 25, 16) * factor
```

- A call to a non-value-returning function is a statement. Thus, calls to non-value-returning functions are written as a statement (instruction) on its own,

```
displayWelcome()
```

- Technically, all functions in Python are value-returning, since functions that do not return a value return special value `None`.

Let's Apply It

Temperature Conversion Program

The following is a program that allows a user to convert a range of values from Fahrenheit to Celsius, or Celsius to Fahrenheit, as presented in Chapter 3. In this version, however, the program is designed with the use of functions. This program utilizes the following programming features.

- value-returning functions
- non-value-returning functions

```
This program will convert a range of temperatures
Enter (F) to convert Fahrenheit to Celsius
Enter (C) to convert Celsius to Fahrenheit
```

```
Enter selection: F
Enter starting temperature to convert: 75
Enter ending temperature to convert: 90
```

Degrees Fahrenheit	Degrees Celsius
75.0	23.9
76.0	24.4
77.0	25.0
78.0	25.6
79.0	26.1
80.0	26.7
81.0	27.2
82.0	27.8
83.0	28.3
84.0	28.9
85.0	29.4
86.0	30.0
87.0	30.6
88.0	31.1
89.0	31.7
90.0	32.2


```

1 # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3 def displayWelcome():
4
5     print('This program will convert a range of temperatures')
6     print('Enter (F) to convert Fahrenheit to Celsius')
7     print('Enter (C) to convert Celsius to Fahrenheit\n')
8
9 def getConvertTo():
10
11     which = input('Enter selection: ')
12     while which != 'F' and which != 'C':
13         which = input('Enter selection: ')
14
15     return which
16
17 def displayFahrenToCelsius(start, end):
18
19     print('\n Degrees', ' Degrees')
20     print('Fahrenheit', 'Celsius')
21
22     for temp in range(start, end + 1):
23         converted_temp = (temp - 32) * 5/9
24         print(' ', format(temp, '4.1f'), ' ', format(converted_temp, '4.1f'))
25
26 def displayCelsiusToFahren(start, end):
27
28     print('\n Degrees', ' Degrees')
29     print(' Celsius', 'Fahrenheit')
30
31     for temp in range(start, end + 1):
32         converted_temp = (9/5 * temp) + 32
33         print(' ', format(temp, '4.1f'), ' ', format(converted_temp, '4.1f'))
34
35 # ---- main
36
37 # Display program welcome
38 displayWelcome()
39
40 # Get which conversion from user
41 which = getConvertTo()
42
43 # Get range of temperatures to convert
44 temp_start = int(input('Enter starting temperature to convert: '))
45 temp_end = int(input('Enter ending temperature to convert: '))
46
47 # Display range of converted temperatures
48 if which == 'F':
49     displayFahrenToCelsius(temp_start, temp_end)
50 else:
51     displayCelsiusToFahren(temp_start, temp_end)

```

In **lines 3–33** are defined functions `displayWelcome`, `getConvertTo`, `displayFahrenToCelsius`, and `displayCelsiusToFahren`. The functions are directly called from the main module of the program in **lines 35–51**.

Two non-value-returning functions are defined, used to display converted temperatures: `displayFahrenToCelsius` (**line 17**) and `displayCelsiusToFahren` (**line 26**). Each is passed two arguments, `temp_start` and `temp_end`, which indicate the range of temperature values to be converted.

Note that (value-returning) function `getConvertTo` (**line 9**) does not take any arguments. This is because the value returned strictly depends on input from the user.

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS:

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS 1. (b),

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS: 1. (b) 2. (a)

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS: 1. (b), 2. (a), 3. TRUE

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS: 1. (b), 2. (a), 3. TRUE 4. (b),

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS: 1. (b), 2. (a), 3. TRUE, 4. (b), 5. (a),

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS: 1. (b), 2. (a), 3. TRUE, 4. (b), 5. (a), 6. (a)

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS: 1. (b), 2. (a), 3. TRUE, 4. (b), 5. (a), 6. (a), 7. (b)

Self-Test Questions

1. The values passed in a given function call in Python are called,
(a) formal parameters
(b) actual arguments
2. The identifiers of a given Python function providing names for the values passed to it are called,
(a) formal parameters
(b) actual arguments
3. Functions can be called as many times as needed in a given program. (TRUE/FALSE)
4. When a given function is called, it is said to be,
(a) subrogated (b) invoked (c) activated
5. Which of the following types of functions must contain a return statement,
(a) value-returning functions (b) non-value-returning functions
6. Value-returning function calls are,
(a) expressions (b) statements
7. Non-value-returning function calls are,
(a) expressions (b) statements
8. Which of the following types of routines is meant to produce side effects?
(a) value-returning functions (b) non-value-returning functions

ANSWERS: 1. (b), 2. (a), 3. TRUE, 4. (b), 5. (a), 6. (a), 7. (b), 8. (b)

More on Functions

We further discuss 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
```


Other examples of calls to value-returning functions.

```
(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))
```

- (a) Expressions containing multiple function calls
- (b) Function call as an argument to another function call
- (c) Function call as conditional expression
- (d) Function call as part of calls to built-in print function

A value-returning function may return more than one value by returning the values as a 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)
```

(a) Assigns the returned tuple to variable **highlow_temps**

(b) Uses **tuple assignment** to assign both variables **high** and **low**

Let's Try It

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,-40) < 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
???
```

Calling Non-Value-Returning Functions

Calls to non-value-returning functions are for the *side-effects* produced, and not for a returned function value, such as displaying output on the screen,

```
displayWelcome()
```

It does not make any sense to treat this function call as an expression,

```
welcome_displayed = displayWelcome()
```


Functions Designed to Take No Arguments

As shown in the previous examples, both value-returning and non-value-returning functions can be designed to take no arguments.

Let's Try It

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

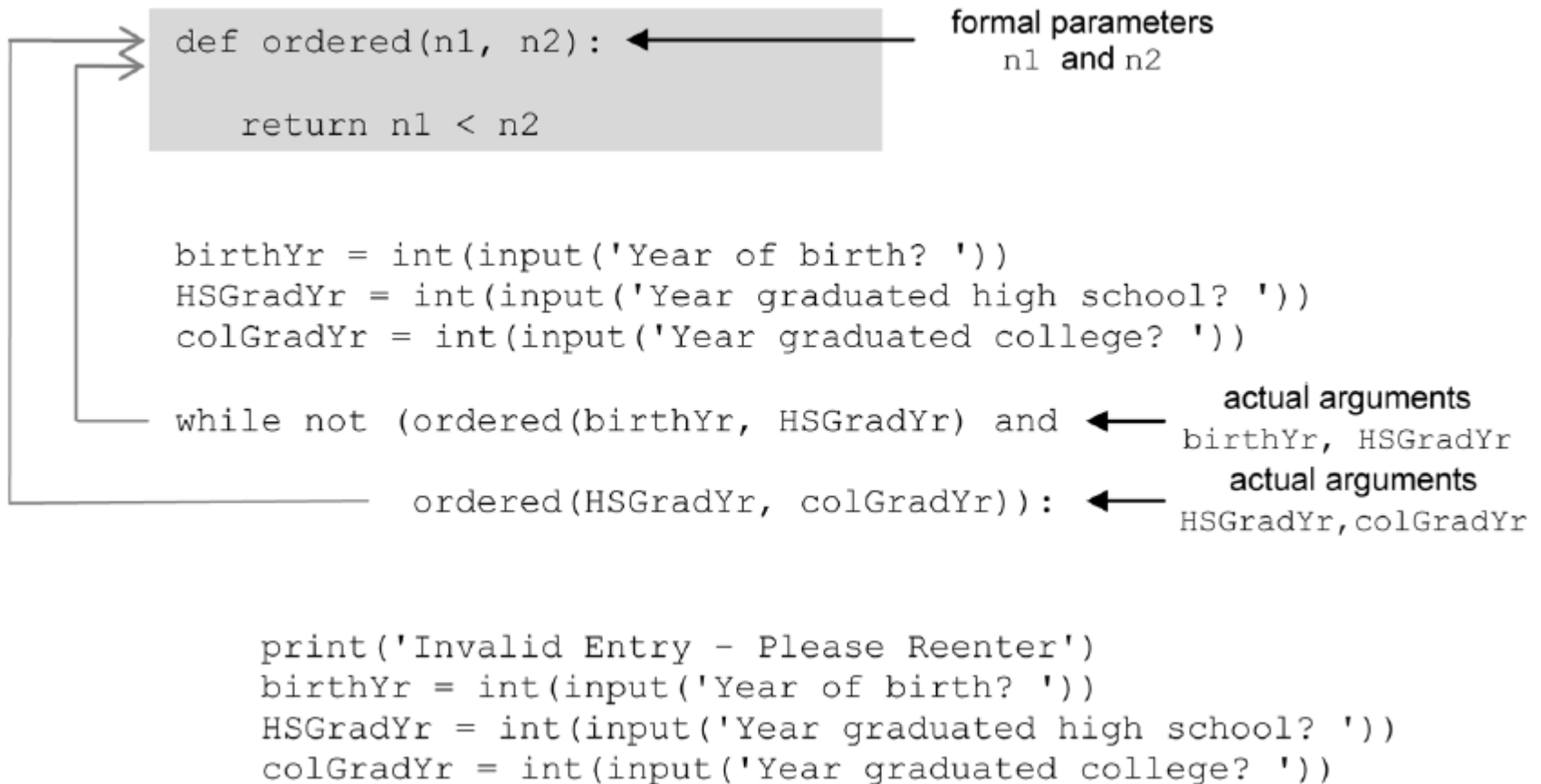
```
>>> def sayHello():
        print('Hello!')
>>> sayHello()
???
>>> t = sayHello()
???
>>> t
???
>>> t == None
???
```

```
>>> def buildHello(name):
        return 'Hello' + name + '!'
>>> greeting = buildHello('Charles')
>>> print(greeting)
???
>>> buildHello('Charles')
???
>>> buildHello()
???
```

Parameter Passing


Parameter passing is the **process of passing arguments to a function**.

Recall that **actual arguments** are the values passed to a function's **formal parameters** to be operated on.



The correspondence of actual arguments and formal parameters is determined by the *order of the arguments passed*, and not their names.

The following parameter passing is perfectly valid.



```
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?')
```

The diagram consists of two vertical lines on the left. The leftmost line has an arrow pointing to the `ordered` function definition. The second line has two arrows: one pointing to the `ordered(num1, num2)` call in the `if` statement, and another pointing to the `ordered(num2, num1)` call in the `else` block.

It is fine to pass actual arguments `num1` and `num2` to function `ordered` as shown (either `num1` as the first argument, or `num2` as the first)

Let's Try It

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]
>>> nums_2 = [8,4,6,1]
```

```
>>> ordered(max(nums_1), max(nums_2))
>>> ???
```

```
>>> ordered(min(nums_1), max(nums_2))
???
```

Mutable vs. Unmutable Arguments

There is an issue related to parameter passing not yet considered.

If a function changes the value of any of its formal parameters, does that change the value of the corresponding actual argument passed?

When literals are passed as arguments, there is no issue.

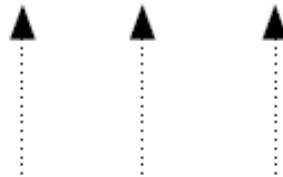
```
def avg(n1, n2, n3):  
    ↑      ↑      ↑  
    ···    ···    ···  
    avg(10, 25, 40)
```

It is when the actual arguments are *variables* that this must be considered.

```
def avg(n1, n2, n3):  
    ↑      ↑      ↑  
    ···    ···    ···  
    avg(num1, num2, num3)
```


Since function `avg` does not change the value of its parameters, the corresponding actual parameters **num1**, **num2** and **num3** will not be altered.

```
def avg(n1, n2, n3):
```



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

Consider, however, the following function.

```
def countdown(n):  
    while n >= 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 it, you will see that variable `num_tics` is left unchanged.

Now consider the following function.

```
def sumPos(nums):  
    for k in range(0, len(nums)):  
        if nums[k] < 0:  
            nums[k] = 0  
  
    return sum(nums)
```

```
>>> nums_1 = [5, -2, 9, 4, -6, 1]  
>>> total = sumPos(nums_1)  
>>> total  
19  
>>> nums_1  
[5, 0, 9, 4, 0, 1]
```

Function **sumPos** returns the sum of only the positive numbers in the provided argument (**list**). It does this by first replacing all negative values in parameter **nums** with 0, then summing the list using built-in function `sum`.

We see 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** but there was in list argument **nums_1** has to do with their types.

Immutable

Numeric Types
(integers and floats)

Boolean Type

String Type]

Tuples

Immutable

Lists

Dictionaries
(not yet introduced)

Let's Try It

Enter the following and observe the results.

```
>>> num = 10
>>> def incr(n):
    n = n + 1
>>> incr(num)
>>> num
???
```

```
>>> nums_1 = [1,2,3]
>>> def update(nums):
    nums[1] = nums[1] + 1
>>> update(nums_1)
>>> nums_1
???
```

```
>>> nums_2 = (1,2,3)
>>> update(nums_2)
>>> ???
```

Keyword Arguments in Python

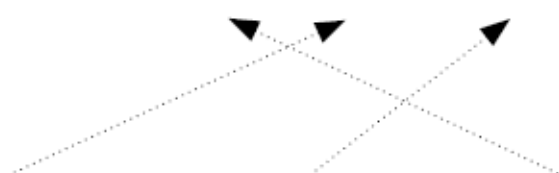
The functions we have looked at so far were called with a fixed number of *positional arguments*. A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list,

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

Python provides the option of calling any function by the use of keyword arguments. A keyword argument is an argument that is specified by parameter name, rather than as a positional argument.

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

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


A diagram illustrating keyword argument mapping. Three dotted lines with arrowheads at the top point from the keyword arguments in the function call to the corresponding parameters in the function definition. One line connects 'rate=0.06' to 'rate', another connects 'term=20' to 'term', and a third connects 'amount=350000' to 'amount'. The lines cross, showing that the order of keyword arguments does not matter.

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.

Let's Try It

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

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

```
>>> addup(1,10)  
???
```

```
>>> addup(first=1, last=10)  
???
```

```
>>> addup(last=10, first=1)  
???
```

Default Arguments in Python

Python provides for *default arguments*. A **default argument** is an argument that can be optionally provided.

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

                                ↑      ↑
                                ·      ·
monthly_payment = mortgage_rate(35000, 0.62)
```

Parameter `term` is assigned a default value, 20, and therefore is optionally provided when calling function `mortgage_rate`. **All positional arguments must come before any default arguments in a function definition.**

Let's Try It

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

```
>>> def addup(first, last, incr=1):
```

```
    if first > last:
```

```
        sum = -1
```

```
    else:
```

```
        sum = 0
```

```
        for i in range(first, last+1, incr):
```

```
            sum = sum + i
```

```
    return sum
```

```
>>> addup(1,10)
```

```
???
```

```
>>> addup(1,10,2)
```

```
???
```

```
>>> addup(first=1, last=10)
```

```
???
```

```
>>> addup(incr=2, first=1,  
          last=10)
```

```
???
```


Variable Scope

Variable scope has to do with the parts a program that a given variable is accessible.

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.

```
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**? The execution of **func2**, however, shows that the value of **n** remains unchanged.

Now consider the following.

```
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, when **func2** is called, we get an error that **variable n** is not defined within **func1**. This is because variable **n** defined in **func2** is inaccessible from **func1**.

Variable Lifetime

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.**

Let's Try It

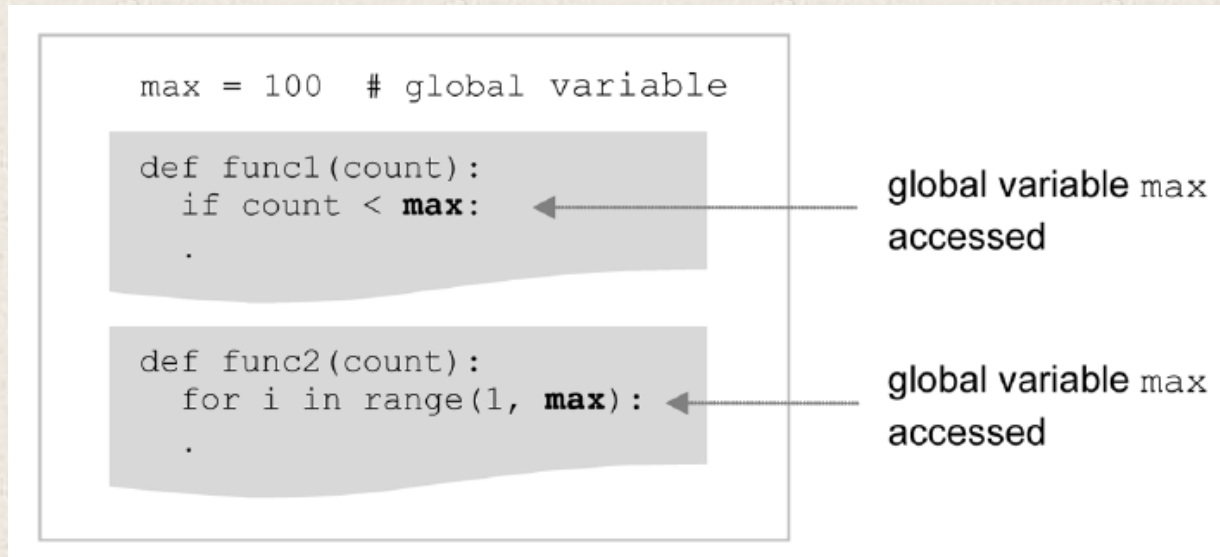
Enter the following function definition in the Python shell. Execute the statements below and observe the results.

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

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

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**.



Variable **max** is defined outside **func1** and **func2**, and therefore “global” to each. Thus, it is referred to as a **global variable**. As a result, it is directly accessible by both functions.

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

Let's Apply It

GPA Calculation Program

The following program computes a semester GPA and new cumulative GPA for a given student. This program utilizes the following programming features:

► **tuple assignment**

This program calculates semester and cumulative GPAs

Enter total number of earned credits: 30

Enter your current cumulative GPA: 3.25

Enter grade (hit Enter if done): A

Enter number of credits: 4

Enter grade (hit Enter if done): A

Enter number of credits: 3

Enter grade (hit Enter if done): B

Enter number of credits: 3

Enter grade (hit Enter if done): B

Enter number of credits: 3

Enter grade (hit Enter if done): A

Enter number of credits: 3

Enter grade (hit Enter if done):

Your semester GPA is 3.62

Your new cumulative GPA is 3.38

>>>

```

1 # Semester GPA Calculation
2
3 def convertGrade(grade):
4     if grade == 'F':
5         return 0
6     else:
7         return 4 - (ord(grade) - ord('A'))
8
9 def getGrades():
10     semester_info = []
11     more_grades = True
12     empty_str = ''
13
14     while more_grades:
15         course_grade = input('Enter grade (hit Enter if done): ')
16         while course_grade not in ('A', 'B', 'C', 'D', 'F', empty_str):
17             course_grade = input('Enter letter grade received: ')
18             if course_grade == empty_str:
19                 more_grades = False
20         else:
21             num_credits = int(input('Enter number of credits: '))
22             semester_info.append([num_credits, course_grade])
23
24 def calculateGPA(sem_grades_info, cumm_gpa_info):
25     sem_quality_pts = 0
26     sem_credits = 0
27     current_cumm_gpa, total_credits = cumm_gpa_info
28
29     for k in range(len(sem_grades_info)):
30         num_credits, letter_grade = sem_grades_info[k]
31
32         sem_quality_pts = sem_quality_pts + \
33             num_credits * convertGrade(letter_grade)
34
35         sem_credits = sem_credits + num_credits
36
37     sem_gpa = sem_quality_pts / sem_credits
38     new_cumm_gpa = (current_cumm_gpa * total_credits + sem_gpa * \
39                     sem_credits) / (total_credits + sem_credits)
40
41     return (sem_gpa, new_cumm_gpa)

```

Function **convertGrade** (lines 3-7) is passed a letter grade, and returns its numerical value.

Function **getGrades** (lines 9-12) gets the semester grades from the user. It returns a list of sublists. Each containing a letter grade and the number of credits, [['A', 3], ['B', 4], ['A', 3], ['C', 3]].

Function **calculateGPA** (lines 24-41) is given a current GPA (and number credits based on), and grades and credits for the current semester, and calculates both the current semester GPA and new cumulative GPA. (returned as a tuple).

```

43 # ---- main
44
45 # program greeting
46 print('This program calculates new semester and cummulative GPAs\n')
47
48 # get current GPA info
49 total_credits = int(input('Enter total number of earned credits: '))
50 cumm_gpa = float(input('Enter your current cummulative GPA: '))
51 cumm_gpa_info = (cumm_gpa, total_credits)
52
53 # get current semester grade info
54 print()
55 semester_grades = getGrades()
56
57 # calculate semester gpa and new cummulative gpa
58 semester_gpa, cumm_gpa = calculateGPA(semester_grades, cumm_gpa_info)
59
60 # display semester gpa and new cummulative gpa
61 print('\nYour semester GPA is', format(semester_gpa, '.2f'))
62 print('Your new cummulative GPA is', format(cumm_gpa, '.2f'))

```

The program greeting is on **line 46**. **Lines 49–50** get the number of earned credits and current cumulative GPA from the user. These two variables are bundled into a tuple named `cumm_gpa_info` on **line 51**. Since they are always used together, bundling these variables allows them to be passed to functions as one parameter rather than as separate parameters.

Function `getGrades` is called on **line 55**, which gets the semester grades from the user and assigns it to variable `semester_grades`. On **line 58**, function `calculateGPA` is called with arguments `semester_grades` and `cumm_gpa_info`. Finally, the results are displayed on **lines 61-62**.

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True,

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True,

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True,

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True, 4. False, 5. False, 6. (c), 7. False, 8. False

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True, 4. False, 5. False,

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True, 4. False, 5. False, 6. (c),

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True, 4. False, 5. False, 6. (c), 7. True, 8. False

Self-Test Questions

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)
6. 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
7. A global variable is a variable that is defined outside of any function definition. (TRUE/FALSE)
8. The use of global variables is a good way to allow different functions to access and modify the same variables. (TRUE/FALSE)

ANSWERS: 1. True, 2. True, 3. True, 4. False, 5. False, 6. (c), 7. True, 8. False

Credit Card Calculation Program

We look at the problem of determining the length of time needed to pay off a credit card balance, as well as the total interest paid.

Credit Card Calculation

The Problem

The problem is to generate a table showing the decreasing balance and accumulating interest paid on a credit card account for a given credit card balance, interest rate, and monthly payment.

Year	Balance	Interest Paid
1	330.18	5.17
	315.13	10.13
	299.85	14.85
	284.35	19.35
	268.62	23.62
	252.65	27.65
	236.44	31.44
	219.98	34.98
	203.28	38.28
	186.33	41.33
	169.13	44.13
	151.66	46.66
	133.94	48.94
	115.95	50.95
2	97.69	52.69
	79.15	54.15
	60.34	55.34
	41.25	56.25
	21.86	56.86
	2.19	57.19
	0.00	57.22

Credit Card Calculation

Problem Analysis

The factors that determine how quickly a loan is paid off are the **amount of the loan**, the **interest rate** charged, and the **monthly payments** made. For a fixed-rate home mortgage, the monthly payments are predetermined so that the loan is paid off within a specific number of years.

For a credit card, there is only a minimum payment required each month. The minimum payment for a credit card is usually around 2–3% of the outstanding loan amount each month, and no less than twenty dollars. Thus, calculating this allows us to project the amount of time that it would take before the account balance becomes zero, as well as the total interest paid.

Calendar Year

Program Design

- **Meeting the Program Requirements**
- **Data Description**
- **Algorithmic Approach**

Meeting the Program Requirements

No particular format is specified for how the output is to be displayed. All that is required is that the user be able to enter the relevant information and that the length of time to pay off the loan and the total interest paid is displayed. The user will also be given the choice of assuming the monthly payment to be the required minimum payment, or a larger specified amount.

Data Description

All that needs to be represented in this program are numerical values for the **loan amount**, the **interest rate**, and the **monthly payment** made. There is no need to create a data structure as the table of payments can be generated as it is displayed.

Algorithmic Approach

The only algorithm needed for this problem is the calculation of the required **minimum payment**. The minimum payment is usually calculated at 2% or 3% of the outstanding balance, with a lower limit of around \$20. Therefore, we will assume a worst case scenario of a minimum payment calculated at 2%, with a minimum payment of \$20.

Program Greeting



Get Card Balance,
Annual Interest Rate,
and Monthly Payment
from User



Calculate and Display
Payoff of Loan and
Total Interest Paid



The Overall Steps of the Program

Credit Card Calculation

Program Implementation

Stage 1— Developing the Overall Program Structure

- **Prompts user** for current credit card balance, interest rate, and monthly payment they wish to have calculated.
- **Function displayPayments** is designed to be passed the current balance, interest rate, and monthly payment and display the month-by-month balance and interest accrued. In this version, the function is implemented to simply display on the screen the values of the parameters passed.

```

1  # Credit Card Calculation Program (Stage 1)
2
3  def displayWelcome():
4      print('\n.... Entering function display welcome')
5
6  def displayPayments(balance, int_rate, monthly_payment):
7      print('\n.... Entering function displayPayments')
8      print('parameter balance =', balance)
9      print('parameter int_rate =', int_rate)
10     print('parameter monthly_payment =', monthly_payment)
11
12     # ---- main
13
14     # display welcome screen
15     displayWelcome()
16
17     # get current balance and APR
18     balance = int(input('\nEnter the balance on your credit card: '))
19     apr = int(input('Enter the interest rate (APR) on the card: '))
20
21     monthly_int_rate = apr/1200
22
23     # determine monthly payment
24     response = input('Use the minimum monthly payment? (y/n): ')
25
26     if response in ('y','Y'):
27         print('Minimum payment selected')
28         monthly_payment = 20
29     else:
30         print('User-entered monthly payments selected')
31         monthly_payment = input('Enter monthly payment: ')
32
33     # display monthly payoff
34     displayPayments(balance, monthly_int_rate, monthly_payment)

```


Stage 1 Testing

From the test results, we see that the appropriate values are being input and passed to function **displayPayments**.

```
.... Entering function display welcome
```

```
Enter the balance on your credit card: 1500  
Enter the interest rate (APR) on the card: 18  
Use the minimum monthly payment? (y/n): n  
User-entered monthly payments selected  
Enter monthly payment: 100
```

```
.... Entering function displayPayments  
parameter balance = 1500  
parameter int_rate = 0.015  
parameter monthly_payment = 100  
>>>
```

```
.... Entering function display welcome
```

```
Enter the balance on your credit card: 1500  
Enter the interest rate (APR) on the card: 18  
Use the minimum monthly payment? (y/n): y  
Minimum payment selected
```

```
.... Entering function displayPayments  
parameter balance = 1500  
parameter int_rate = 0.015  
parameter monthly_payment = 20  
>>>
```

Program Implementation

Stage 2 – Generating an Unformatted Display of Payments

In this stage of the program, **function `displayPayments`** is implemented to display the new monthly balance and interest accrued, without any concern of formatting the output at this point.

```

1  # Credit Card Calculation Program (Stage 2)
2
3  def displayWelcome():
4      print('This program will determine the time to pay off a credit')
5      print('card and the interest paid based on the current balance,')
6      print('the interest rate, and the monthly payments made.')
7
8  def displayPayments(balance, int_rate, monthly_payment):
9
10     # init
11     num_months = 0
12     total_int_paid = 0
13
14     # display loan info
15     print('PAYOFF SCHEDULE')
16     print('\nCredit card balance: $' + format(balance, '.2f'))
17     print('Annual interest rate:', str(1200 * int_rate) + '%')
18     print('Monthly payment: $', format(monthly_payment, '.2f'))
19
20     # display year-by-year account status
21     while balance > 0:
22         monthly_int = balance * int_rate
23         total_int_paid = total_int_paid + monthly_int
24         balance = balance + monthly_int - monthly_payment
25
26         year = (num_months // 12) + 1
27         print(year, format(balance, '.2f'), format(total_int_paid, '.2f'))
28
29         num_months = num_months + 1
30
31     # ---- main
32
33     # display welcome screen
34     displayWelcome()
35
36     # determine current balance and APR
37     balance = int(input('\nEnter the balance on your credit card: '))
38     apr = int(input('Enter the interest rate (APR) on the card: '))
39
40     monthly_int_rate = apr/1200
41
42     # determine monthly payment
43     response = input('Use the minimum monthly payment? (y/n): ')
44
45     if response in ('y', 'Y'):
46         if balance < 1000:
47             monthly_payment = 20
48         else:
49             monthly_payment = balance * .02
50     else:
51         monthly_payment = input('Enter monthly payment: ')
52
53     # display monthly payoff
54     displayPayments(balance, monthly_int_rate, monthly_payment)

```


Stage 2 Testing

We test this program once for the **minimum monthly payment**, and once for a **specified monthly payment** amount .

Testing for Minimum Payment

This program will determine the time to pay off a credit card and the interest paid based on the current balance, the interest rate, and the monthly payments made.

```
Enter the balance on your credit card: 350
Enter the annual interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): y
PAYOFF SCHEDULE
```

```
Credit card balance: $350.00
Annual interest rate: 18.0%
Monthly payment: $ 20.00
```

```
1 335.25 5.25
1 320.28 10.28
1 305.08 15.08
1 289.66 19.66
1 274.00 24.00
1 258.11 28.11
1 241.99 31.99
1 225.62 35.62
1 209.00 39.00
1 192.13 42.13
1 175.02 45.02
1 157.64 47.64
2 140.01 50.01
2 122.11 52.11
2 103.94 53.94
2 85.50 55.50
2 66.78 56.78
2 47.78 57.78
2 28.50 58.50
2 8.93 58.93
2 -10.94 59.06
>>>
```

Balance and
Accrued Interest
Calculated
Correctly for
Minimum Payment

Testing for Payment Specified by User

```
This program will determine the time to pay off a credit
card and the interest paid based on the current balance,
the interest rate, and the monthly payments made.
```

```
Enter the balance on your credit card: 1500
Enter the annual interest rate (APR) on the card: 18
Use the minimum monthly payment? (y/n): n
Enter monthly payment: 100
PAYOFF SCHEDULE
```

```
Credit card balance: $1500.00
```

```
Annual interest rate: 18.0%
```

```
Traceback (most recent call last):
```

```
  File "C:\My Python Programs\CreditCardCalc-Stage2 with
error.py", line 53, in <module>
```

```
    displayPayments(balance, monthly_int_rate, monthly_payment)
```

```
  File "C:\My Python Programs\CreditCardCalc-Stage2 with
error.py", line 18, in displayPayments
```

```
    print('Monthly payment: $', format(monthly_payment, '.2f'))
```

```
ValueError: Unknown format code 'f' for object of type 'str'
```

```
>>>
```

Clearly, there is something wrong with this version of the program. The `ValueError` generated indicates that the format specifier `.2f` is an unknown format code for a string type value, referring to line 18. Thus, this must be referring to variable `monthly_payment`. But that should be a numeric value, and not a string value! How could it have become a string type?

Since the problem only occurred when the user entered the monthly payment (as opposed to the minimum payment option), we try to determine what differences there are in the program related to the assignment of variable `monthly_payment`.

```
# determine monthly payment
response = input('Use the minimum monthly payment? (y/n): ')
if response in ('y', 'Y'):
    if balance < 1000:
        monthly_payment = 20
    else:
        monthly_payment = balance * .02
else:
    monthly_payment = input('Enter monthly payment: ')
```

Since the variable `monthly_payment` is not a local variable, we can display its value directly from the Python shell,

```
>>> monthly_payment
'140'
```

We immediately realize that variable `monthly_payment` is input as a string type! We fix this problem by replacing the line with the following,

```
monthly_payment = int(input('Enter monthly payment: '))
```


Program Implementation

Stage 3 – Formatting the Displayed Output

In this final stage of the program, input error checking is added. The program is also modified to allow the user to continue to enter various monthly payments for recalculating a given balance payoff. Output formatting is added to make the displayed information more readable. Finally, we correct the display of a negative balance at the end of the payoff schedule.

```

1 # Credit Card Calculation Program (Final Version)
2
3 def displayWelcome():
4     print('This program will determine the time to pay off a credit')
5     print('card and the interest paid based on the current balance,')
6     print('the interest rate, and the monthly payments made.')
7
8 def displayPayments(balance, int_rate, monthly_payment):
9
10     # init
11     num_months = 0
12     total_int_paid = 0
13     payment_num = 1
14
15     empty_year_field = format(' ', '8')
16
17     # display heading
18     print('\n', format('PAYOFF SCHEDULE', '>20'))
19     print(format('Year', '>10') + format('Balance', '>10') +
20           format('Payment Num', '>14') + format('Interest Paid', '>16'))
21
22     # display year-by-year account status
23     while balance > 0:
24         monthly_int = balance * int_rate
25         total_int_paid = total_int_paid + monthly_int
26
27         balance = balance + monthly_int - monthly_payment
28
29         if balance < 0:
30             balance = 0
31
32         if num_months % 12 == 0:
33             year_field = format(num_months // 12 + 1, '>8')
34         else:
35             year_field = empty_year_field
36
37         print(year_field + format(balance, '>12,.2f') +
38               format(payment_num, '>9') +
39               format(total_int_paid, '>17,.2f'))
40
41         payment_num = payment_num + 1
42         num_months = num_months + 1
43

```

```

44 # ---- main
45
46 # display welcome screen
47 displayWelcome()
48
49 # get current balance and APR
50 balance = int(input('\nEnter the balance on your credit card: '))
51 apr = int(input('Enter the interest rate (APR) on the card: '))
52
53 monthly_int_rate = apr/1200
54
55 yes_response = ('y','Y')
56 no_response = ('n','N')
57
58 calc = True
59 while calc:
60
61     # calc minimum monthly payment
62     if balance < 1000:
63         min_monthly_payment = 20
64     else:
65         min_monthly_payment = balance * .02
66
67     # get monthly payment
68     print('\nAssuming a minimum payment of 2% of the balance ($20 min)')
69     print('Your minimum payment would be',
70         format(min_monthly_payment, '.2f'), '\n')
71
72     response = input('Use the minimum monthly payment? (y/n): ')
73     while response not in yes_response + no_response:
74         response = input('Use the minimum monthly payment? (y/n): ')
75
76     if response in yes_response:
77         monthly_payment = min_monthly_payment
78     else:
79         acceptable_payment = False
80
81         while not acceptable_payment:
82             monthly_payment = int(input('\nEnter monthly payment: '))
83
84             if monthly_payment < balance * .02:
85                 print('Minimum payment of 2% of balance required ($' +
86                     str(balance * .02) + ')')
87
88             elif monthly_payment < 20:
89                 print('Minimum payment of $20 required')
90             else:
91                 acceptable_payment = True
92

```

```

93     # check if single payment pays off balance
94     if monthly_payment >= balance:
95         print('* This payment amount would pay off your balance *')
96     else:
97         # display month-by-month balance payoff
98         displayPayments(balance, monthly_int_rate, monthly_payment)
99
100        # calculate again with another monthly payment?
101        again = input('\nRecalculate with another payment? (y/n): ')
102        while again not in yes_response + no_response:
103            again = input('Recalculate with another payment? (y/n): ')
104
105        if again in yes_response:
106            calc = True    # continue program
107            print('\n\nFor your current balance of $' + str(balance))
108        else:
109            calc = False   # terminate program

```


Stage 3 Testing

We give example output of this version of the program for both a payoff using the required minimum monthly payment, and for a user-entered monthly payment. The following depicts a portion of the output for the sake of space.

Testing for Minimum Payment

This program will determine the time to pay off a credit card and the interest paid based on the current balance, the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500

Enter the interest rate (APR) on the card: 18

Assuming a minimum payment of 2% of the balance (\$20 min)
Your minimum payment would be 30.00

Use the minimum monthly payment? (y/n): y

PAYOFF SCHEDULE

Year	Balance	Payment Num	Interest Paid
1	1,492.50	1	22.50
	1,484.89	2	44.89
	1,477.16	3	67.16
	1,469.32	4	89.32
	.	.	.
	.	.	.
7	517.51	73	1,207.51
	495.28	74	1,215.28
	472.70	75	1,222.70
	449.79	76	1,229.79
	.	.	.
	.	.	.
8	227.51	85	1,277.51
	200.92	86	1,280.92
	173.94	87	1,283.94
	146.55	88	1,286.55
	118.74	89	1,288.74
	90.53	90	1,290.53
	61.88	91	1,291.88
	32.81	92	1,292.81
	3.30	93	1,293.30
	0.00	94	1,293.35

Recalculate with another payment? (y/n): n

>>>

Testing for Payment Specified by User

This program will determine the time to pay off a credit card and the interest paid based on the current balance, the interest rate, and the monthly payments made.

Enter the balance on your credit card: 1500

Enter the interest rate (APR) on the card: 18

Assuming a minimum payment of 2% of the balance (\$20 min)

Your minimum payment would be 30.00

Use the minimum monthly payment? (y/n): n

Enter monthly payment: 100

PAYOFF SCHEDULE

Year	Balance	Payment Num	Interest Paid
1	1,422.50	1	22.50
	1,343.84	2	43.84
	1,264.00	3	64.00
	1,182.95	4	82.95
	1,100.70	5	100.70
	1,017.21	6	117.21
	932.47	7	132.47
	846.45	8	146.45
	759.15	9	159.15
	670.54	10	170.54
	580.60	11	180.60
	489.31	12	189.31
2	396.65	13	196.65
	302.60	14	202.60
	207.13	15	207.13
	110.24	16	210.24
	11.89	17	211.89
	0.00	18	212.07

Recalculate with another payment? (y/n): n

>>>

Results of Testing of Final Stage

Payoff Information			Expected Results		Actual Results		Evaluation
Balance	Interest Rate	Monthly Payment	Num Months	Interest Paid	Num Months	Interest Paid	
250	18%	\$20 (min)	14	28.93	14	28.93	Passed
600	14%	\$20 (min)	38	142.80	38	142.80	Passed
12,000	20%	\$240 (min)	109	14,016.23	109	14,016.23	Passed
250	18%	\$40	7	14.54	7	14.54	Passed
600	14%	\$50	14	50.15	14	50.15	Passed
12,000	20%	\$400	42	4,773.98	42	4,773.98	Passed