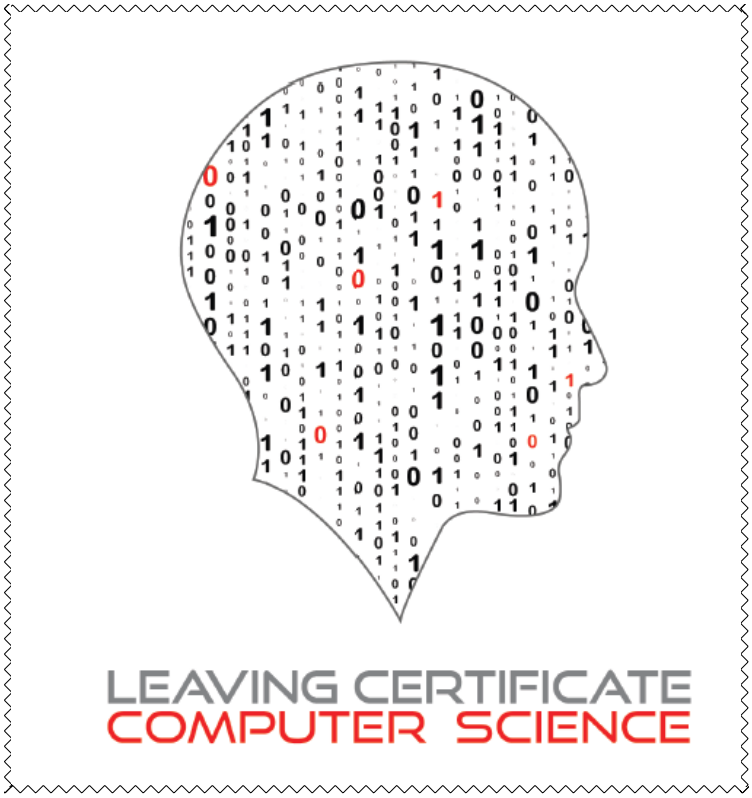
LC CS Python

Student Manual



Section 6

Functions

Name: _____

Introduction

Functions are the building blocks of programs. They allow programmers to organise their code into logically related sections.

In order to understand where functions fit into the overall scheme of things it is useful to have some understanding of the architecture of a Python program. This is depicted in the following illustration

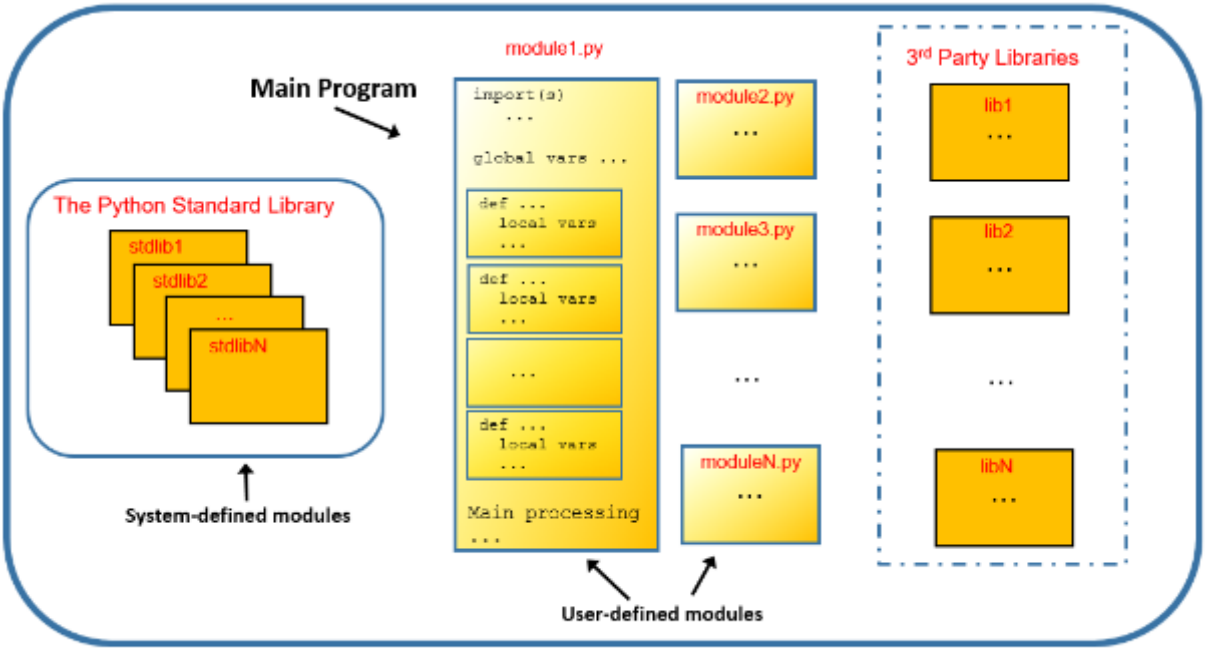


Figure: Python Program Architecture

As can be seen a Python program is typically made up of many components. One of the key components is called a *module*. Modules can be thought of as individual Python script files (i.e. a file with a `.py` extension) made up mostly of functions. It is useful to think of a Python program as a collection of modules.

At runtime, modules and function co-operate with one another to achieve some desired result. Program execution starts in one special module called the *top level module* which is also commonly referred to as the 'main' program.

Modules can be classified into the following three groups:

1. **User defined modules:** These are modules that are written by the programmer as part of the program which is being developed (aka the current development)

2. **Standard library modules:** These are modules that come pre-installed as part of Python. The Python standard library is designed to save programmers from having to come up with their own solutions to common programming problems. As such, it includes a full suite of off-the-shelf, ready to go solutions in the form of built-in functions.

   Some examples of Python libraries we have already come across are `math`, `random`, `statistics`, and `turtle`. See https://docs.python.org/3/library/index.html for a complete reference.

3. **3rd party modules:** These are modules that are developed by an external source either for commercial purposes or as open source (available free). There are literally thousands – some examples include `tkinter`, `numpy`, `plotly`, `scipy`, `Django`, and `flask`. The Python Package Index (`PyPI`) is a repository of software for the Python programming language. See https://pypi.org/ for more information.

KEY POINT: A Python program consists of one or more modules and modules are made up of functions. Each individual module is a Python script or `.py` file.

So, modules and functions are constructs used by programmers to organise their code into separate units (or chunks). A package is another such construct – it is used to group a number of related modules into a single entity. You can think of a package as a set of modules that reside on the file system in the same folder/directory.

Programmers can use the Python `import` statement to access the functionality of external modules. These external modules can be individual modules or multiple modules that have been grouped together into a package.

For the purpose of Leaving Certificate Computer Science (LCCS), a typical Python program might be made up of a single file that draws on and exploits the functionality made available by the standard library and, possibly, some other third party package(s).

Most Python files are organised into three sections

1. the import statements (typically at the top)
2. the function definitions (by far the longest and most important section)
3. the main code i.e. the section from where the program execution begins.

What are functions?

The 'art of computer programming' can be seen as a process of designing and creating individual functions and combining them together into larger units of code called modules. Over time, programmers combine these modules to produce the final program.

We have already learned that functions provide a means for programmers to organise their code, but what exactly is a function and why are they important?

KEY POINT: A *function* is a short piece of re-usable code that carries out a specific task.

Each individual Python module is (mostly) made up of functions.

User-defined vs Built-in functions

Functions are very useful because they typically provide solutions to common programming problems e.g. display some text on the screen, read data from the end-user, send a tweet, process a cash withdrawal etc.

- In certain cases, the programming problems are so common that Python provides a built-in function to do the job. Such functions are called *built-in functions*.

- In other cases, the problem to be solved is so specific to the program being developed that the programmer needs to design and write the code themselves. Such functions are called *user-defined functions*.

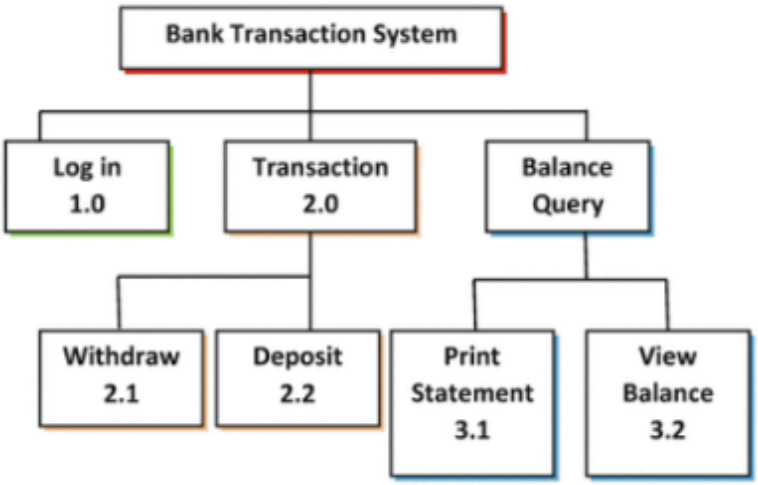
All functions, regardless of whether they are built-in or user-defined, are given a name as part of their definition. The function name can then be used in a program to invoke the code contained in the function.

We'll examine the syntax and semantics of defining and invoking functions shortly but first let's take a closer look at the two main reasons why functions are considered important:

1. they lead to modular systems
2. they can be used to maximise code reuse and minimise code redundancy

Large scale software systems are very often developed by breaking big problems down into smaller problems (decomposition). While designers and programmers are working on the detail of one part of the system, they can ignore the rest of the system (abstraction).

Consider the functional decomposition diagram of an ATM/Bank Transaction system show below. The diagram shows the system broken down into a number of smaller sub-systems. The functionality of each sub-system can be implemented using functions. This piece-by-piece approach to developing systems is sometimes called *divide and conquer*.



Functions are a programming construct that support this divide and conquer approach to software development. They lend themselves to *modular systems* which are easier and less costly to maintain than their non-modular counterparts.

A second reason why functions are so important is that they can be used to minimise (and even avoid) code duplication. Consider the following example.

Example 1 – Maximising code reuse and minimising redundancy

Study the short program shown below that displays a slightly adapted version of the Dr Seuss poem, *Green Eggs and Ham*

```
# A program to display Green Eggs and Ham
1. print("I do not like green eggs and ham.")
2. print("I do not like them Sam-I-am.")
3. print() # display a blank line
4. print("I do not like them here or there.")
5. print("I do not like them anywhere.")
6. print("I do not like them in a house")
7. print("I do not like them with  a mouse")
8. print() # display a blank line
9. print("I do not like green eggs and ham.")
10.print("I do not like them Sam-I-am.")
11.print() # display a blank line
12.print("I do not like them in a box")
13.print("I do not like them with a fox")
14.print("I will not eat them in the rain.")
15.print("I will not eat them on a train")
16.print() # display a blank line
17.print("I do not like green eggs and ham.")
18.print("I do not like them Spam-I-am.")
```

- **In workbook, do Task 1**

The above code is considered poor design because it contains duplication. Lines 1,2 are duplicated in three different places. This is an example of redundant code. To eliminate the redundancy, we write a *function* to display the chorus. The program below uses a function to eliminate the duplication referred to earlier.

```
# A program to display Green Eggs and Ham (v2)
1.  def displayChorus():
2.      print()
3.      print("I do not like green eggs and ham.")
4.      print("I do not like them Sam-I-am.")
5.      print()
6.
7.  displayChorus()
8.  print("I do not like them here or there.")
9.  print("I do not like them anywhere.")
10. print("I do not like them in a house")
11. print("I do not like them with  a mouse")
12. displayChorus()
13. print("I do not like them in a box")
14. print("I do not like them with a fox")
15. print("I will not eat them in the rain.")
16. print("I will not eat them on a train")
17. displayChorus()
```

This program displays the same output as the program on the previous page – the only difference is that this program uses a function called `displayChorus` to eliminate duplication of code.

Every function must have a name (assigned by the programmer). In this case, the name of the function is `displayChorus` and it is defined (or made known to Python) on lines 1–5 inclusive.

The lines from line 7 onwards are executed in sequence. Lines 7, 12 and 17 *call* the function `displayChorus`. Every time the function is called the lines 2-5 are executed. Even though these lines only appear once in the program they are used on three different occasions.

KEY POINT: When a function is *called*, the flow of control jumps to the first line of the function and execution continues from that point to the last line of the function. Once the last line of the function has been executed the flow of control jumps back to the point from which the call to the function was initially made.

STUDENT TIP

Avoid duplicating blocks of code. Code duplication is considered to be symptomatic of 'poor code'.

If you find that you need to re-use a number of lines of code to do something specific use a function instead of duplicating the code.

- **In workbook, do Task 2**

Example 2 – Modular Code

Let's continue with the same example to show how functions develop modular code.

```
# A program to display Green Eggs and Ham (v3)
1. def displayChorus():
2.     print()
3.     print("I do not like green eggs and ham.")
4.     print("I do not like them Sam-I-am.")
5.     print()
6.
7. def displayVerse1():
8.     print("I do not like them here or there.")
9.     print("I do not like them anywhere.")
10.    print("I do not like them in a house")
11.    print("I do not like them with  a mouse")
12.
13. def displayVerse2():
14.    print("I do not like them in a box")
15.    print("I do not like them with a fox")
16.    print("I will not eat them in the rain.")
17.    print("I will not eat them on a train")
18.
19. displayChorus()
20. displayVerse1()
21. displayChorus()
22. displayVerse2()
23. displayChorus()
```

Consider the differences between the program shown here and the two programs shown earlier in this section.

Even though the three programs are different they all do the same thing.

Notice the different level of indentation inside each function body.

In this program, the code to display each part of the poem is '*factored*' into separate functions. The program is considered better than the two previous versions because it is more *modular*. Modular code is the result of good design and is both easier and less costly to maintain than non-modular code.

The above listing defines three functions as follows:

- The function `displayChorus` is defined on lines 1-5
- The function `displayVerse1` is defined on lines 7-11
- The function `displayVerse2` is defined on lines 13-17

Lines 19-23 are executed in sequence and cause the poem to be displayed.

- **In workbook, do Task 3**

Summary

- A Python program is made up of one or more modules - each module is a `.py` file
- Modules are made up mostly of functions.
- Functions are the building blocks of modules (and by extension, programs)
- A function is a short piece of re-usable code that carries out a specific task.
- All functions have a name which must be used to invoke the function's code
- Python comes with a set of pre-installed modules - called the *standard library*
- Logically related modules can be grouped together into packages. A package is made up of multiple modules.
- The functionality of individual modules and entire packages can be made accessible to other Python files by using the `import` statement
- Functions that do not require the `import` statement in order to be accessible are known as *built-in functions* e.g. `print`, `input`. Python 3.6.x comes with 67 built-in functions; these are listed in the appendix – more details can be found by browsing to https://docs.python.org/3/library/functions.html

- **In workbook, do Tasks 4 & 5**

Basic Function Syntax

Once the 'need' for a function in your program has been recognised (the difficult bit!) the next step is to write the code. For this, we need to understand some syntax (the easy bit!).

Functions are made known to Python by writing a *function definition*. In Python, the function definition is made up of two parts

1. the function **header** (aka the function *signature* or *prototype*) is always the first line of the function definition
2. the function **body** contains the Python statements which carry out the work of the function.

The code below illustrates the definition of a function called `displayPoem`. The function header is on line 1 and the function body runs from line 2 to 5 inclusive.

```
1. def displayPoem():
2.     print("One fine day in the middle of the night,")
3.     print("Two dead men got up to fight,")
4.     print("Back to back they faced each other,")
5.     print("Drew their swords and shot each other.")
```

Function definition for `displayPoem`

The function header

Every function header is composed of four separate parts:

1. The word **`def`**: This is a Python keyword which tells Python to create a new function object. **Every function must start with the `def` statement.**
2. The function name (in this case *displayPoem*):
   It is up to the programmer to decide what name to give a function. The rules for naming functions are the same as those for naming variables.
3. Brackets: These can be used to pass information into functions. In this example, no information is being passed into the function and therefore the contents of the brackets is empty.
4. Colon: The colon is the end of the function header. If the colon is missing, Python will display a syntax error.

The function body

The body of every Python function consists of one or more Python statements. These statements carry out the function's task. Although there is no limit to the number of statements that can be in a function body, it is generally considered good practice to keep functions short.

Notice how the four lines of code that make up the function body (lines 2 – 5) are indented. In Python, the statements inside a function body must always be indented. The function body ends when the indentation ends i.e. when the next statement appears at the same level of indentation as the def statement.

Calling a function

It is important to realise that the code inside a function body will not be executed unless the programmer explicitly asks for it to be executed. The term for such a request is a *function call*.

KEY POINT: A *function* call causes the code inside the function body to be executed.

Functions can be called (or invoked) at runtime by writing the name of the function followed by brackets. The code to call the function `displayPoem` is shown below.

```
displayPoem()
```

This line calls the function `displayPoem`

Note, the brackets are needed but `def` and colon are not

When the above call is made the four lines in the function body are executed thus causing the following four lines of text to be displayed on the output console.

One fine day in the middle of the night,
Two dead men got up to fight,
Back to back they faced each other,
Drew their swords and shot each other.

The semantics of a function call are now explained.

Call semantics

Consider the order in which the lines of code in example program below are processed by Python.

```
1. def homework(): # function header
2.     print("Jack loves to do his homework")
3.     print("He never misses a day")
4.     print("He even loves the men in white")
5.     print("Who are taking him away")
6.     # End of function
7.
8. homework() # call the function
9. print("The End!")
```

Python starts at line 1, notices that it is a function definition and skips over all of the lines in the function definition until it finds a line that is no longer included in the function (line 8). On line 8 it notices that it has a function to execute, so it goes back and executes that function – lines 2-6 inclusive. Once all the lines in the function body have been executed, it continues at line 9. The result of the above program is:

Jack loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away
The End!

- **In workbook, do Tasks 6, 7 & 8**

Guidelines and Rules for Naming Functions

We begin with a few simple guidelines as opposed to actual rules. These guidelines help improve program readability (and therefore maintainability).

Function names should be meaningful i.e. they should in some way describe what the function does. Since functions are usually actions, the name should contain at least one verb.

If a function name is only one single word, you should use lowercase; if the name of the function is made up of more than one word, the use of camel case (e.g. calculateArea) or underscore (calculate_area) is considered good practice.

The basic syntax rules for naming functions are:

- A function name cannot be a Python keyword (e.g. `def`, `if`, `while`, etc.)
- Function names must contain only letters, digits, and the underscore character, _.
- Function names cannot have a digit for the first character.

- **In workbook, do Task 9**

Function Parameters and Arguments

Let's return to the homework function we were looking at earlier and ask the question – how can we modify the function to display the verse using names other than Jack?

```
1. def homework(): # function header
2.     print("Jack loves to do his homework")
3.     print("He never misses a day")
4.     print("He even loves the men in white")
5.     print("Who are taking him away")
6.     # End of function
7.
8. homework() # call the function
9. print("The End!")
```

Every time this function is called it always displays precisely the same text. We'd like to move away from the concrete name – Jack – to a more abstract name – anybody.

One solution would be to have a different version of the function for each different name we wanted to have it display, but this would contradict the whole purpose of functions which is to eliminate code duplication. Another solution is to use *parameters*.

What we really want is some way of telling the function what name to display as part of the verse i.e. we need a means to pass information into the function from the code outside the function. This is exactly the purpose of parameters.

A *parameter* is a special kind of variable which appears as part of the function header and can be used inside the function body. Take a look at this!

```
1. def homework(personName): # function header
2.     print(personName, "loves to do his homework")
3.     print("He never misses a day")
4.     print("He even loves the men in white")
5.     print("Who are taking him away")
6.     # End of function
7.
8. homework("David") # call the function
```

The function `homework` modified to use a parameter

David loves to do his homework
He never misses a day
He even loves the men in white
Who are taking him away

Program output

STUDENT TIP
When you want to pass information into a function use parameter(s)

KEY POINT: A function parameter is a variable which gets it value from the argument passed in. When a function is called value of the argument is assigned to the parameter.

Notice `personName` appears between the brackets in the function header? This is a function **parameter**. Parameters are received by functions.

Notice also the text *David* between brackets in the function call (line 8)? This is a function **argument**. Arguments are passed into functions.

- **In workbook, do Task 10**

Functions can be defined to accept multiple arguments.

```
# Functions can have multiple parameters
1. def displayMessage(msg1, msg2):
2.     print(msg1)
3.     print(msg2)
4.     # End of function
5.
6. displayMessage("Hello world", "How are you today?")
```

The function `displayGreeting` has two parameters

```
Hello World
How are you today?
```

Program output

Notice the use of a comma to separate the parameters in the function header (line 1) and the arguments in the function call (line 6).

When the function `displayGreeting` is called Python performs two assignments:

- the value of the first argument (i.e. *Hello world*) is assigned to the parameter msg1.
- the value of the second argument (i.e. *How are you today?*) is assigned to the parameter msg2.

Parameters are received into a function in the same order as the arguments provided. Check what happens if the arguments were switched around like this in the function call.

```
6.  displayMessage("How are you today?", "Hello world")
```

Thus far in this section the arguments used in the example programs have all been string literals. However, arguments can be literals of any datatype (e.g. numeric, Boolean etc.); arguments can also be expressions made up of variables and/or literal values together. Be careful though - as a general guide the number of arguments provide should match the number of parameters provided for in the function header.

- **In workbook, do Tasks 11, 12 & 13**

KEY POINT: The advantage of using parameters and arguments is that they make functions much more flexible and provide for more general solutions to problems.

The runtime behaviour of a function can be altered by passing different arguments into it. This is useful, and a very common way, of achieving *abstraction*.

Function Return Values

Functions can be thought of little machines (black box) that accept input(s) and produce output(s) – like functions in Maths do!

In the previous section we learned that data can be passed into functions through the use of arguments (at the function call) and parameters (as part of the function header). In this section we explore the use of the `return` statement as a means to pass data back out of a function.

Consider the function shown below to add the first `n` non-negative integers.

```
# Add all the numbers from 0 to n
1. def sumOfN(n):
2.     total = 0
3.
4.     for i in range(n+1):
5.         total = total + i
6.
7.     return total
```

`range(n)` generates a list of integers from 0 to n-1 e.g. `range(10)` [0, 1, 2, 3, … 9]

The `for` loop iterates over each integer with the value of the next item in the sequence being assigned to the loop variable `i`

The function works by maintaining a running total of all the numbers from 0 to n+1 in the variable `total`. At the start of each loop iteration, the loop variable (`i`) is assigned the next value in the sequence. On each iteration of the loop, the value of the loop variable is added to `total` and the result is used to update `total` with the new running total. The loop ends after the last value in the sequence has been processed.

Line 7 shows the `return` statement being used to pass the value of `total` out of the function.

To test our function – let's say we wanted to add the first 10 integers - would add the line `sumOfN(10)` to call the function.

- **In workbook, do Task 14**

The reason nothing appears to happen in **Task 14**, is that although the function is called and it does calculate and return the sum of the first 10 non-negative integers, the calling code takes no action to save, process or even just print the result.

KEY POINT: The return value of a function can be saved for further processing by making the function call part of an assignment statement.

Line 9 in the code below assigns the result of the function (i.e. `total`) to the variable `answer`.

```
# Add all the numbers from 0 to n
1.  def sumOfN(n):
2.      total = 0
3.
4.      for i in range(n+1):
5.          total = total + i
6.
7.      return total
8.
9.  answer = sumOfN(10) # call the function and save the result
10. print("The sum of the first 10 integers is %d" %answer)
```

The value of `total` is passed out of the function (line 7) and assigned to the variable `answer` (line 9)

STUDENT TIP

When you want to save the return value of a function, the function call should be made as part of an assignment statement :

`<variable-name> = <function-call>`

- **In workbook, do Task 15**

Alternatively, the programmer may decide there is no need to save the result of a function in a variable and process the result as part of the call. This type of *inline processing* is shown on line 10 of the code below where the result of the function is passed in as an argument to the `print` built-in function.

```
# Add all the numbers from 0 to n
1.  def sumOfN(n):
2.      total = 0
3.
4.      for i in range(n+1):
5.          total = total + i
6.
7.      return total
8.
9.  # call the function and print the result
10. print("The sum of the first 10 integers is", sumOfN(10))
```

In this example, the result of the function is not caught by the calling code. Rather, it is passed directly as an argument to `print`. This technique is called *function composition*.

In situations when functions are used directly as arguments to other functions, Python starts at the innermost function and works its way out i.e. the inner function is always evaluated first and evaluation continues from right to left.

For example, let's say we had a function called `sub` to subtract two integers (`b` from `a`) as shown below.

```
def sub(a, b):
    answer = a - b
    return answer

print( sub( 2, sub(3, 4)))
print( sub( sub(2, 3), 4))
print( sub( 2, sub(3, sub(4, 5))))
```

When run, the code would display 3, -5, and -2 on separate lines.

As a final note it is worth pointing out that It is not always necessary for a function to return data. A return statement may be omitted entirely or can be used without an expression. In both cases the value returned by Python is `None`.

- **In workbook, do Task 17**

Examples and Exercises

Study each of the examples in the workbook carefully – read the code first, predict what it would do. Then, key the code in and run it to test your prediction(s). In each case you should complete the reflection exercise provided before you finally move on to implement the programming challenges at the end.

- **In workbook, do Tasks 18, 19 and 20 in the *Examples and Exercises* section**

Boolean Functions

Boolean functions are functions that return either `True` or `False` usually to indicate the outcome of some test. By convention the name of a Boolean function starts with the prefix `is` e.g. `isEven` might be a Boolean function that tests the 'evenness' of a number.

```
# A function to determine evenness
def isEven(number):
    if (number % 2 == 0):
        return True
    else:
        return False
```

The function uses the remainder operator (%) to test whether the value passed in as `number` is even or not – if it is the function returns `True`. Otherwise, the function will return `False`.

The code shown here to the right demonstrates how the above function could be used to display all the even numbers between 1 and 100.

```
# display even numbers < 100
for i in range (100):
    if isEven(i):
        print(i)
```

The line `if isEven(i):` is the key. Here, the call to the function appears as part of a conditional statement. This is fine, since conditions evaluate to `True` or `False` and the function `isEven` is guaranteed to return one of these values.

We can exploit our knowledge of even and odd numbers to define the function `isOdd` as shown. The function applies the `not` operator to the result of the call to `isEven` and returns the result to the caller.

```
# A function to determine oddness
def isOdd(number):
    return not isEven(number)
```

This is a good example of abstraction because the implementation of `isOdd` hides the detail.

- **In workbook, do tasks 21 and 22**

A prime example

A prime number is a positive integer that has exactly two factors; itself and 1.

The Boolean function below `isPrime` determines whether the number passed in is prime or not. The function will return `True` if number is a prime number; `False` otherwise.

```
import math

# A function to test whether a number is prime or not
# Returns True if the number is prime; False otherwise
def isPrime(numToCheck):

    # Any number less than 2 is not prime
    if numToCheck < 2:
        return False

    # see if num is evenly divisible by any number up to num/2
    divisor = 2
    while (divisor < numToCheck/2):
        if (numToCheck % divisor == 0):
            return False
        divisor = divisor+1

    # The number must be prime so return True
    return True
```

The function works by attempting to divide every integer from 2 up to half the number being tested (`numToCheck`) - if the division leaves no remainder, it means the number being checked has factors and is therefore not prime. The following exercises, based on the test harness provided are designed to be used to explore the function `isPrime`.

- **In workbook, do Tasks 23 and 24**

A case study of leap years

Boolean functions provide a useful framework which can be used to determine whether a given year is leap or not. One example is presented as follows:

```
# Determines whether n is leap or not
def isLeap(n):
    if n % 400 == 0:
        return True
    if n % 100 == 0:
        return False
    if n % 4 == 0:
        return True
    else:
        return False

yr = int(input("Enter a year: "))
if isLeap(yr):
   print(yr, "is a leap year")
else:
   print(yr, "is NOT a leap year")
```

- **In workbook, do Task 25**

Now examine the two implementations below – one is correct and one contains a subtle error (i.e. one version reports incorrectly that 2000 is not a leap year.)

```
def isLeapV1(year):
    if year % 4 == 0 and year % 100 != 0:
        return True
    elif year % 100 == 0:
        return False
    elif year % 400 ==0:
        return True
    else:
        return False
```

```
def isLeapV2(year):
    if year % 4 == 0 and year % 100 != 0:
        return True
    elif year % 400 ==0:
        return True
    elif year % 100 == 0:
        return False
    else:
        return False
```

- **In workbook, do Tasks 26, 27 and 28**

STUDENT TIP

Use a Boolean function as a means to organise a block of code that performs any test that leads to one of two possible outcomes - `True` or `False`

Using Functions to Validate Data

Functions provide programmers a convenient way to organise code that perform specific tasks such as data validation. Consider the following scenarios and examples.

1. One common scenario faced by novice programmers is the need to read non-negative integers from the end user. Since the `input` command returns a string we need to write code that ensures the data entered is numeric before converting it to an integer using `int`

   The following four solutions (and there are many more!) are offered for consideration.

```
def readIntegerV1():

    strN = input("Enter a number: ")
    while not strN.isdigit():
        strN = input("Enter a number: ")

    return int(strN)
```

```
def readIntegerV3():

    while True:
        strN = input("Enter a number: ")
        if strN.isdigit():
            break

    return int(strN)
```

```
def readIntegerV2():

    valid = False
    while not valid:
        strN = input("Enter a number: ")
        if strN.isdigit():
            valid = True

    return int(strN)
```

```
def readIntegerV4():

    while True:
        strN = input("Enter a number: ")
        for i in range(0, len(strN)):
            if strN[i] not in '0123456789':
                break
        else:
            break

    return int(strN)
```

Which of the above solutions do you prefer and why?

Would the functions work to read negatives or floating-point values?

2. Building on the previous scenario, it may be that we need to make sure that the number entered is within a specific range.

   This time we offer two versions of a solution

```
def readIntegerInRangeV1(lwr, upr):

    valid = False
    while not valid:
        strN = input("Enter a number: ")
        if strN.isdigit():
            n = int(strN)
            if (n >= lwr) and (n <= upr):
                valid = True

    return n
```

```
def readIntegerInRangeV2(lwr, upr):

    valid = False
    while not valid:
        strN = input("Enter a number: ")
        if strN.isdigit():
            n = int(strN)
            if lwr <= n <= upr:
                valid = True

    return n
```

Compare and contrast the two solutions. What is the main difference? Comment on the use of the flag `valid` in the above code. Under what circumstance is the value of `valid` set to `True`?

3. The function below can be used to ensure a user enters either Y or N

```
def readYesNoResponse():

    response = input("Enter response [Y/N]: ")
    while response != "Y" and response != "N":
         response = input("Enter response [Y/N]: ")

    return response
```

Can you suggest any alternative solution(s)?