Functions

Functions (1)

- We have developed a number of programs so far
- While the code is useful, it is limited in its general use
- If we want to reuse the code, we have to copy the code, possibly edit it to change variable names, constants, etc
- We would like to be able to use our code again and in more complex programs
- Functions allow us **to name** segments of code and this can really help to make our programs **easier to read** and hence to understand.
- This is also an excellent reason for using functions even if we only use them once in a program.

Functions (2)

- For example, consider our program to calculate the pay of an employee
- If we wanted to have a program to also calculate the tax, to be paid, we would have to copy the pay calculation code, possibly edit the variable names, include the extra code to calculate the tax and paste it to where we want it
- The more code a program contains, the bigger the chance there is of an error occurring and the harder it is to maintain the program
- For example, say that we discovered that our pay calculation program had an error
- After fixing the error, we would then have to repeat that fix in the tax calculation program

Functions in Python

- We have already used a number of built-in functions:
  - `range()`
  - `len()`
  - `print()`
  - `...`
- The facility for programmers to define and subsequently use their own functions marks “a qualitative leap forward in convenience”

Defining functions in Python

- In Python, each function definition is of the following form:

  def *name of function* **(***list of formal parameters***):**

  statement(s)

- **def** is a reserved keyword that introduces the definition of the function
- The function name is simply an identifier (a name) that is used to refer to the function

```
def max(a, b):
    if a > b:
        return a
    else:
        return b
```

Formal and actual parameters

- The formal parameters of the function are the sequence of names within the parentheses after the function name (`(a, b)` in our example)
- When the function is used, ie at function invocation or function call, the formal parameters are bound to the values of the actual parameters or arguments
- This binding is similar to the binding that takes place in an assignment statement
- For example, the function call `max(2, 5)` binds `a` to `2` and `b` to `5`

- `2` and `5` are the **arguments** to `max`
- `a` and `b` are the **parameters** of `max`

Example

```
#f1.py: show use of functions
def max(a, b):
    if a > b:
        return a
    else:
        return b
x = 4
y = 10
m = max( x, y)
print(f"Maximum = {m} \n")
i = int(input("\nEnter an integer: "))
j = int(input("\nEnter an integer: "))
m = max (i, j)
print(f"Maximum = {m} \n")
```

Formal and actual parameters

```
Running f1.py:


Maximum = 10

Enter an integer: 2

Enter an integer: 6
Maximum = 6
```

Formal and actual parameters

```
#f2.py: show use of functions
def max(a, b):
    if a > b:
        return a
    else:
        return b
x = 4
y = 10
print(f"Maximum = {max( x, y)} \n")
i = int(input("\nEnter an integer: "))
j = int(input("\nEnter an integer: "))
print(f"Maximum = {max(i,j)} \n")
```

Function body

- The **function body** may be any Python code.
- The body of `max` is:

```
if a > b:
    return a
else:
    return b
```

- You can have as many statements as you wish in the function body but a good rule of thumb for function length is that you should be able to read the whole function on one screen i.e. about 24 lines.
- If a function body is very long then you may be able to break it into two or more separate functions.

Function body

- The function body, when executed, carries out the actions of the function
- A function invocation is an expression and, like all expressions, it has a value
- That value is the value returned by the invoked function
- There is a special statement, **`return`**, that can only be used within the body of a function
- This statement returns the value from the function to the expression in which it was invoked

The `return` statement

- For example, the value of the expression `max(2, 5)` is 5
- The value of the expression `max(4, 3) * max(2, 5)` is 20 (the first invocation of `max` returns 4 and the second invocation returns 5)
- Execution of a `return` statement terminates that invocation of the function
- If there is no value specified after the `return` statement, the special value **`None`** is returned
- If there is no `return` statement, the invocation of the function continues until there are no more statements to execute
- In this case, the value `None` is returned

```
# menu.py: Display a menu of options

def display_menu():
    print '''

        0    Quit

        1    Calculate area of triangle

        2    Calculate area of circle

        3    Calculate area of rectangle

        'n\n\n

        '''

     return

display_menu()
```

Running menu.py:

```
        0         Quit

        1         Calculate area of triangle

        2         Calculate area of circle

        3         Calculate area of rectangle
```

Note use of ‘’’ in print to display a multi line string

```
# menu.py: Display a menu of options

import math
pi = math.pi

def display_menu():
    print ('''
        0           Quit
        1           Calculate area of triangle
        2           Calculate area of circle
        3           Calculate area of rectangle
        \n\n\n
        ''')
    return

def get_user_option():
    option = int(input('\n\nEnter option:  '))
    return (option)

def area_of_circle( radius ):
    area = pi * radius ** 2
    return area
```

```
def get_radius():
    radius = float(input('Enter radius: '))
    return radius

def get_base():
    base = float(input('Enter base: '))
    return base

def get_height():
    height = float(input('Enter height: '))
    return height

def get_breadth():
    breadth = float(input('Enter breadth: '))
    return breadth

def get_length():
    length = float(input('Enter length: '))
    return length

def area_of_triangle( base, height ):
    area = (base * 0.5) * height
    return area
def area_of_rectangle( length, breadth  ):
    area = length * breadth
    return area
```

```
display_menu()

option =  get_user_option()

while option != 0:
    if option == 1:
        b = get_base()
        h = get_height()
        print (f'Triangle area: {area_of_triangle(b,h)} \n\n')
    elif option == 2:
        r = get_radius()
        print (f'Circle area: {area_of_circle( r ):.2f} \n\n')
    elif option == 3:
        l = get_length()
        b = get_breadth()
        print (f'Rectangle area:  {area_of_rectangle(l,b)} \n\n')

    display_menu()
    option = get_user_option()

print('\n\n Finished Areas program\n\n')
```

We can shorten the loop body above by calling the functions inside the print function but which version is easier to read ?

```
if option == 1:
   print (f'Triangle area: {area_of_triangle(get_base(), get_height())} \n\n’)

elif option == 2:
        print (f'Circle area: {area_of_circle( get_radius() ):.2f} \n\n’)

elif option == 3:
        print (f'Rectangle area:  {area_of_rectangle(get_length(),
get_breadth())} \n\n’)



print('\n\n Finished Areas program\n\n')
```

We can also shorten the function definitions:

Change

```
def get_radius():
    radius = float(input('Enter radius: '))
    return radius
```

To

```
def get_radius():
    return float(input('Enter radius: '))
```

Again, which version is easier to read

Exercise: Write a function **`process_user_option(option)`** so the program below will function as the previous version

```
# menu3.py

display_menu()

option =  get_user_option()

while option != 0:
    process_user_option( option )
    display_menu()
    option = get_user_option()

print('\n\n Finished Areas program\n\n’)
```

Note how simple the main loop is in our program, when we write it using functions.

```
def process_user_option( option ):
    if option == 1:
      print (f'Triangle area: {area_of_triangle(get_base(),get_height())}\n\n')
    elif option == 2:
      print (f'Circle area: {area_of_circle( get_radius() ):.2f} \n\n')
    elif option == 3:
     print (f'Rectangle area:  {area_of_rectangle(get_length(), get_breadth())} \n\n')
    return

option =  get_user_option()

while option != 0:
    process_user_option( option )
    display_menu()
    option = get_user_option()

print('\n\n Finished Areas program\n\n')
```

Modules

- So far, all our programs have been in a single file
- This is fine as long as programs are small
- However, as programs get larger, it is more convenient to store **different parts of them in different files**
- This allows you create files with useful functions that can be used in many different programs, without have to define them in each program
- **Python modules** allow us to easily construct a program from code in multiple files

- A Python module is a `.py` file containing Python definitions and statements

- For example, we could create a **module** (file) **`circle.py`** containing the following:

```
pi = 3.1415927

def area(radius):
    return pi * (radius ** 2)

def circumference(radius):
    return 2 * pi * radius

def sphereVolume(radius):
    return (4.0 / 3.0) * pi * (radius ** 3)
```

Using Modules

- A program gets access to a module through an **`import`** statement
- To use a function from a module, you put **the `module name.` before the `function` name as**

`module_name.function()`

```
# mod.py: program to call functions from module
import circle          # do not  add .py

print(circle.pi)
print(circle.area(3))
print(circle.circumference(3))
print(circle.sphereVolume(3))
```

Running `mod.py` produces the following:

```
3.1415927
28.2743343
18.8495562
113.0973372
```

Create a file called `mf.py` with the definitions of the functions we used in menu programs shown earlier.

```
# mf.py: code of functions for our menu3.py program

import math
pi = math.pi

def get_radius():..
def get_base():..
def get_height() :..
def get_breadth():..
def get_length():..
def area_of_triangle( base, height ): ..
def area_of_rectangle( length, breadth  ):..
def area_of_circle( radius :..
def get_user_option():..
def display_menu():..
def process_user_option( option ):..
```

We can now use the functions from `mf.py` in any program by importing the module into the program where we wish to use them. For example, we can create a new program in a separate file menu4.py with the following code:

```
# menu4.py: Display a menu of options and use functions from
# the module mf.py

import mf

mf.display_menu()

option =  mf.get_user_option()
while option != 0:
    mf.process_user_option( option )
    mf.display_menu()
    option = mf.get_user_option()

print('\n\n Finished Areas program\n\n')
```

Another Example of a function to compute the factorial of a number

Factorial 5 written as 5! = 5 * 4 * 3 * 2 * 1

Factorial written as n! = n * (n-1) * (n-2) * (n-3) … * 1

```
# fact.py: Calculate the factorial of a number

def fact(x):
    result = 1
    for i in range(1, x+1):
        result = result * i
    return result


# Prompt the user for an integer

number = int(input('Enter a number (an int >= 0): '))


if number >= 0:
    print(f'\nThe factorial of {number} is {fact(number)}' )
else:
    print(f'\nCannot compute factorial of neg number {number} \n')
```

```
# fact2.py: Calculate the factorial of numbers
# until user enters a negative number

def fact(x):
    result = 1
    for i in range(1, x+1):
        result = result * i
    return result


# Prompt the user for an integer

number = int(input('\nEnter a number (negative to quit): '))

while number >= 0:
      print(f'\nThe factorial of {number} is {fact(number)}' )
      number = int(input('\nEnter a positive number (negative to quit ): '))
```

Running `fact2.py`:

```
% python3 fact2.py
Enter a number (negative to quit): 4

The factorial of 4 is 24

Enter a positive number (negative to quit ): 8

The factorial of 8 is 40320

Enter a positive number (negative to quit ): 10


The factorial of 10 is 3628800

Enter a positive number (negative to quit ): -1

%
```

Scope

Consider the following example:

```
# Program to illustrate scoping in Python

def f(x):
    print ('In function f: ')
    x = x + 1
    y = 1
    print('x is', x)
    print('y is', y)
    print('z is', z)
    return x

x, y, z = 5, 10, 15

print('Before function f:')
print('x is', x)
print('y is', y)
print('z is', z)

z = f(x)

print('After function f:')
print('x is', x)
print('y is', y)
print('z is', z)
```

Scoping

This program produces the following output:

```
Before function f:
x is 5
y is 10
z is 15

In function f:
x is 6
y is 1
z is 15

After function f:
x is 5
y is 10
z is 6
```

The **scope of a variable** is the block of code in the entire program where the variable is declared, used, and can be modified. In the function f, the variable y is **local** to f. Any changes made to y in the function f, do NOT affect a variable y declared outside the function f

```
def f(x):
    print ('In function f: ')
    x = x + 1
    y = 1
    print('x is', x)
    print('y is', y)
    print('z is',z)
    return x
```

Thus variable y still has the value 10 after f has been called. The variable y inside f is NOT the same as the variable y outside f.

In the same way, the function does not change the value of the argument **x outside of the function.**