User defined function:

A function is a named piece of code. We write a sequence of Python code and then give a name to it. We can then refer and run the whole code by using its name.

We are already familiar with this concept. We have been using functions like **`len`**, **`math.sqrt`**, **`input`** and so on.

Why do we use this concept of function? When do we use this?

- We have limitation of thinking about number of minute details at once.
  We would like to 'divide and conquer' - the philosophy Britishers used to gain control over India.
  Dijkstra, one of the greatest Computer Scientists ever says that a person may remember 7 + or – 2 things at a time.

- **A function is expected to do one and only thing**. We call such a function **cohesive**. Because it has a clearly defined aim, the code will be small, neat and clean.

- Similar functionality is normally required at number of places. If we write a function, then we can call or invoke the function without rewriting or copying the code. **It makes the user's program more readable.**

- If the code of the function has to be changed for whatever reason – make it more efficient, make it more flexible – we have to do at only one place. Thus **it helps in maintenance**.

- Once tested, the function can be used with total assurance. There is no necessity of debugging the code repeatedly.

Function Definition and Invocation:

Function Definition:

A function has two parts – leader and suite.

- The leader starts with the keyword **def**
- then the function name
  - is an identifier : start with a letter of english or _ and then followed by any number of letter of english or _ or digit
- then a pair of parentheses
- then a colon
- then the suite follows – suite can have any valid statement of Python including another function definition!
- Within the parentheses, we may have parameters – we shall discuss them at length later.

This is an example of a function definition.
Run the program under Python tutor to understand how the function definition and function call are processed.

```
# file: 1_function.py
# example of function definition and invocation
def foo() :
      print("I am foo")

print("one")
foo()
print("two")
```

Processing of Function Definition:

Name of the function followed by parentheses causes a function call – this results in transfer of control to the leader of the function and then the suite is executed – after that the control comes back to the point after the function call in the user's code.
A few more things happen when the function call is mapped to the function leader. We shall discuss them later.

In our example, leader of function foo is processed first and the function entity with the name foo is created. The suite of foo is not processed at this point. `print("one")` is called displaying the string one. Then foo is called – transferring control to the leader of foo – then the suite of foo is executed resulting in display of the string I am foo. Then the control is returned. Then print is called displaying the string two.

Note: Processing of function definition demo with http://pythontutor.com/

Function Definition and Function Name – internals:

Let us examine the program 2_function.py to understand two things.

- **What does the function name stand for?**
- **What happens when the function is defined?**

```
# fle: 2_function.py
# function definition and function name: internals
# foo : is a function; therefore callable

def foo() :
      print("I am foo")

print("one")
foo # no function call

print("two")
print(foo) # <function foo at 0x________>

bar = foo # bar also becomes callable
```

```
print(bar) # <function foo at 0x________>

# both give the same output
foo()
bar()

# remove foo
del foo
bar() # still works!
```

When the function is defined, the function name becomes an interface for us to refer to the function. The function entity is stored with the name used in the definition along with the suite. Whatever is stored with the function entity remains unchanged until no name refers to it. Each entity in Python has a reference count. It is equally true of the function entity.

```
foo # no function call
```

This above state does not result in a function call. The function name like a variable name is an expression. Any expression on a line is also a statement. But it does not cause a function call. **To invoke a function, we do require the function call operator - a pair of parentheses.**

```
bar = foo
```

This is like a variable assignment. Both foo and bar refer to the function entity called foo (and not bar!). The reference count of the function entity goes up by 1 and in this case, becomes 2.

If we display either bar or foo, we get the same output as both refer to the same function entity foo.

We can invoke the function either by using the name foo or bar. We say that both are callable. A callable is the name of a defned function or a variable holding the function name.

```
del foo
```

This statement causes the name foo to be removed and the reference count of the function entity foo to be decremented. As bar still refers to the same function entity, the reference count would not be zero. Therefore the function entity foo remains and can be called using the name bar.

```
bar() # still works!
```

Function call: arguments and parameters:

A function does some task for us. It requires some values to operator on. We provide them when calling a function. We put these within parentheses in the function call. **We call them arguments**. **The arguments are always expressions. They should have some value.**

In the function definition, we specify variables which receive these arguments. **These are called parameters. The parameters are always variables.**

When the function call is made, the control is transferred to the function definition. The arguments are evaluated and copied to the corresponding parameters. This is called parameter passing by value or call by value.

```
# name: 3_ function.py
import math
# a, b : parameters; always variables
def hyp(a, b) :
      print("in hyp function")
      # can create variables within the function; local variables
      h = math.sqrt(a * a + b * b)
      print("hyp of triangle with sides : ", a , 'and', b, 'is ', h)

print("one")
hyp(5, 12) #5, 12 : arguments : are expressions; should have some value
hyp(3, 4)

# processing:
#     executes the leader of the function
#     skips the suite
#     continues further
#     When the function is called, start executing from the leader
#     copy the arguments to the parameters
#     there parameters and local variables form activation record
#     at the end of the function execution, if this activation record is not required,
#     it is removed

# Number of arguments should match the number of parameters
#hyp(3, 4, 5) # error
#hyp(3) # error
```

The above example shows how to compute hypotenuse given the two limbs of a right angled triangle. In this example the arguments are constants – they can be any type of expression in general. The corresponding parameters are always variables.

In Python, parameters do not have any fixed type. So, we say that the types are generic.

When the function call is made, an **activation record i**s created which will have

- **Parameters**
- **local variables**
  Variables created within the suite of the function
- **return address**
  Location to which the control of the program should be transferred once the function terminates
- **temporary variables**

unnamed variables required by the translator

- **return value**

  value to be passed back to the caller

In this example, we have not used the return mechanism.
One very important point to note is that the activation record is created for every call of the function. At the end of the function, if no other callable can refer to the activation record, it will be removed. We will discuss this point again later.

```
#hyp(3, 4, 5) # error
#hyp(3) # error
```

Observe that the last statements will give errors. The number of arguments should match the number of parameters. We copy the argument to the corresponding parameter on a function call.

Function call and return mechanism:

```
# name: 4_function.py

import math
# a, b : parameters; always variables
# A function by default returns nothing - called None in Python

def hyp(a, b) :
      h = math.sqrt(a * a + b * b) #local to the fn

res = hyp(3, 4)
#print("h : ", h) # not available here
print(res, type(res)) # None of NoneType

def hyp1(a, b) :
      h = math.sqrt(a * a + b * b) #local to the fn
      return h # returns the value of the expression

res = hyp1(3, 4)
print(res, type(res))
```

A called function(Callee) returns the control to the caller when

- **the end of the function body is reached**
- **the return statement is executed.**

The return statement is also used to return a value to the caller. The function does not specify a return type in its definition.

There is no default return mechanism. In the case of the function hyp, there is no return statement in the suite of the function. So, the caller is returned a notional value None of NoneType. This is similar to void functions of ‘C’.

In the second case, the function hyp1 returns h as the result. The value of this expression(in this case, a variable) is returned to the caller.

A function can return a value of any type. There is no restriction in Python.

A couple of small examples of user defined functions.

```
# name: 5_function.py
# given two strings, find common letters.

def find_common(s1, s2):
      set1 = set(s1);
      set2 = set(s2)
      res = ""
      for ch in set1 & set2:
            res += ch
      return res

s1 = "cattle"
s2 = "concat"
# expected output : c a t
print("common letters : ", find_common(s1, s2))

s3 = "horse"
print("common letters : ", find_common(s1, s3))

s4 = "zzzzz"
print("common letters : ", find_common(s1, s4))

s5 = "abcde"
s6 = "acdeb"
print("common letters : ", find_common(s5, s6))


# name: 6_function.py
# output length of words in a string
def disp_count(s):
      wordlist = s.split()
      for word in wordlist:
            print("{} = {}".format(word,len(word)))

disp_count("we are the world")
we = 2
are = 3
the = 3
world = 5
```