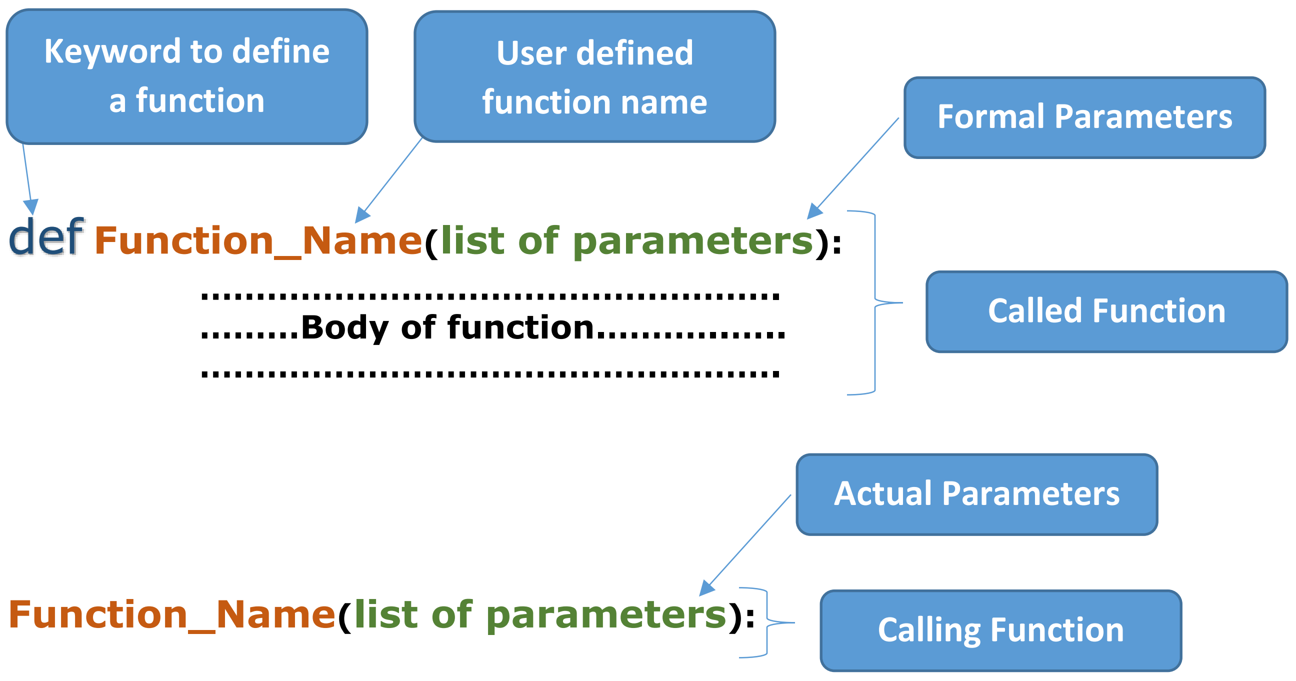
Python User Defined Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Prototype/ definition of a Function



```
def Function_Name(list of parameters):
                ..................................................
                .........Body of function................
                ..................................................

Function_Name(list of parameters):
```

Creating a Function

In Python a function is defined using the `def.` keyword:

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")

my_function() # Calling statement of my_function()
```

Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

Actual Parameter: Used inside the Calling function.

Formal Parameter: Used inside the Called function/ function definition

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_name(fname):
  print(fname + " Birla")

my_name ("Ram")
my_name ("Shyam")
my_name ("Gopal")
```

Output:

```
Ram Birla
Shyam Birla
Gopal Birla
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

```
def my_country(country = "Norway"):
     print("I am from " + country)

my_country ("Sweden")
my_country ("India")
my_country ()
my_country ("Brazil")
```

Output:

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

```
def my_food(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_food(fruits)
```

Output:

```
apple

banana

cherry
```

Return Values

To let a function return a value, use the `return` statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
n=my_function(9)
print(n)
Output:
```

```
15

25

45
```

Keyword Arguments

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

Output: The youngest child is Linus


Note: The phrase Keyword Arguments are often shortened to
kwargs in Python documentations
```

Arbitrary Arguments

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

If the number of arguments are unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

```
Output: The youngest child is Linus
```

Global / Local Scope Vs Local/ Private Scope

The scope of a variable is the range of program where that variable can access. A variable is visible within its scope and invisible or hidden outside it.

Private Scope: if the variable declared inside a function then it is private / local scope. This variable can access only inside the function in which it is declared. It cannot access outside thee function

Public Scope: if the variable declared outside the function then it is public / global scope. This variable can access anywhere in whole program or it can access in all functions.

Visibility of Variable: A variable is visible in its scope and invisible out of its scope.

Note: if both public and private scoped variables are in a function / block then the private scope variable will be accessed and public scope variable will not access. This is because, the private scope variable has higher visibility priority than public scope variable. In other words the private scope hides the visibility of public scope variable.

```
Num=1000    # public scope variable

def First_function():
   Num=10              # private scope variable
   print("Num inside First fun(): ", Num)
```

```
Output:  10
```

```
def Second_function():
    Num=20   # private scope variable
    print("Num inside Second fun(): ",Num)
```

Output: 20

```
def Third_function():
    print("Num inside Third fun(): ",Num)
```

Output: 1000

```
First_function()
Second_function()
Third_function()
print("Num outside functions: ",Num)
```

Output: 1000

The pass Statement

`Function` definitions cannot be empty, but if you for some reason have a `function` definition with no content, put in the `pass` statement to avoid getting an error.

```
def myfunction:
  pass
```

# having an empty function definition like this, would raise an error without the pass statement

Recursion

Python also accepts function recursion. Recursion is a common mathematical and programming concept, which means a defined function can call itself. This has the benefit of meaning that you can loop through data to reach a result.

Recursion can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

```
def tri_recursion(k):
  if(k>0):
    result = k+tri_recursion(k-1)
    print(result)
  else:
    result = 0
  return result
print("\n\nRecursion Example Results")
tri_recursion(6)
```

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (`-1`) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

Output: Recursion Example Results

```
1
3
6
10
15
21
```

```
# Sum of n numbers by using recursion.

def sum_of_n_numbers(n):
            if n==1:
                  return n
         else:
                  return n+sum_of_n_numbers(n-1)

sum=sum_of_n_numbers(4)
print(sum)



# Factorial using recursion

def fact_rec(n):
   if(n==0):
      return 1
   else:
      return n*fact_rec(n-1)

num=5
f=fact_rec(num)
print("Factorial of %d is %d"%(num,f))
```

```
# sum of list elements by using recursion

def sum_of_list(p):
   if(len(p)==0):
      return 0
   else:
      return p[0]+sum_of_list(p[1:])

list=[2,4,5,7,9,1]
t=sum_of_list(list)
print(t)
```