UNIT - 2

FUNCTIONS & MODULES:

FUNCTIONS IN PYTHON

- A function is a block of organized, reusable code that is used to perform a single, related action.
- Functions provide better modularity for your application and a high degree of code reusing.
- Python gives you many built-in functions like print(), etc. but you can also create your own functions.
- These functions are called user-defined functions.

DEFINING A FUNCTION IN PYTHON

- You can define functions to provide the required functionality.
- Simple rules to define a function in Python.
  - Function blocks begin with the keyword def followed by the function name and parentheses (( )).
  - Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
  - The first statement of a function can be an optional statement - the documentation string of the function or docstring.
  - The code block within every function starts with a colon (:) and is indented.
  - The statement return [expression] exits a function, optionally passing back an expression to the caller.
  - A return statement with no arguments is the same as return None.

- **SYNTAX ::**

```
def functionname( parameters ):
        "function_docstring"
         function_suite
         return [expression]
```

- By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.
- **EXAMPLE ::**

```
def printme( str ):
        "This prints a passed string into this function"
        print (str)
        return
```

CALLING A FUNCTION

- Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.
- Following is an example to call the printme() function.

```
# Function definition
def printme( str ):
        "This prints a passed string into this function"
        print (str)
        return

# call to printme function
printme("This is first call to the user defined function!")
printme("Again second call to the same function")
```

PASS BY REFERENCE VS VALUE

- All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.
- For example-

```
# Function definition
def changeme( mylist ):
        "This changes a passed list into this function"
        print ("Values inside the function before change: ", mylist)
        mylist[2] = 50
        print ("Values inside the function after change: ", mylist)
        return
# Call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

- Example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition
def changeme( mylist ):
        "This changes a passed list into this function"
        mylist = [1,2,3,4] # This would assi new reference in mylist
        print ("Values inside the function: ", mylist)
        return
# call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

FUNCTION ARGUMENTS

- A function can be called using the following types of formal arguments-
  - Positional arguments / Required arguments
  - Keyword arguments
  - Default arguments
  - Variable-length argument

REQUIRED/POSITIONAL ARGUMENTS

- Required arguments are the arguments passed to a function in correct positional order.
- Here, the number of arguments in the function call should match exactly with the function definition.
- These arguments are passed to the function in a specific order based on their position. The order of the arguments is essential because the function interprets the arguments based on their position.

```
def add_numbers(a, b):
        return a + b

result = add_numbers(5, 10)
print(result)
```

KEYWORD ARGUMENTS

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.
- These arguments are useful when we have many arguments, and we want to avoid confusion between the arguments' positions.
- function in the following ways-

```
def student(name, age):
    print('Student Details:', name, age)


student('Naman', 21)


student(name='Vaibhav', age=22)


student('Vibhor', age=23)
```

- The following example gives a clearer picture. Note that the order of parameters does not matter.

```
# Function definition is here
def printinfo( name, age ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return
# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

DEFAULT ARGUMENTS

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
- The following example gives an idea on default arguments, it prints default age if it is not passed.
- # Function definition is here

```
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print ("Name: ", name)
    print ("Age ", age)
    return

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )
```

VARIABLE-LENGTH ARGUMENTS

- Variable-length arguments are used when we do not know the number of arguments that will be passed to the function.
- In Python, we can use two types of variable-length arguments:
  - *args
  - **kwargs
- When in a function we require to pass a variable of positional arguments we use the args syntax. We can also pass multiple arguments to the function using args.
- The *args argument is treated as a tuple of arguments inside the function.

```
def my_function(*args):
    for a in args:
        print(a)

my_function(1, 2, 3)
```

- **The** kwargs syntax is used to pass a variable number of keyword arguments to a function.
- The **kwargs argument is treated as a dictionary of key-value pairs inside the function.**
- **Here is an example of how to use** kwargs:

```
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(key, value)

my_function(name="Vikas", age=20)
```

LOCAL, NONLOCAL AND GLOBAL VARIABLES

- **Global Variables ::**
- The variables that are declared outside the scope of a function are defined as global variables in python.
- Alternatively, the python global variables are considered declared in a global scope. As a result, a user can access a global variable inside or outside the function.
- The following example illustrates how a global variable can be created in python:

```
x = "global"
def foo():
    print("x inside:", x)
foo()
print("x outside:", x)
```

- To modify a global variable from within a function, you must use the global keyword before the variable name. This tells Python that you are referring to the global variable instead of creating a new local one. For example :

```
global_var = 10
def modify_global_var():
        global global_var
        global_var += 5

print(global_var)  # Output: 10
modify_global_var()
print(global_var)  # Output: 15
```

- **<u>Local Variables</u>** ::
- Local variables in python are those variables that are declared inside the function.
- Alternatively, they are said to defined within a local scope. A user can only access a local variable inside the function but never outside it.
- Let us take the help of a simple example to understand this.

```
def sum(x,y):
        sum = x + y
        return sum
print(sum(5, 10))
```

- Now let us see how a program will respond should we happen to assign the same name to a global variable and local variable within it.

```
x = 5
def foo():
   x = 10
   print("local x:", x)  # 10
foo()
print(“global x:", x) #5
```

- Here we assigned the same name “x” to both the global variable and the local variable. However, when we went ahead with printing it, we obtained two different values.
- This is due to the fact that even though the names of the global variable and local variable is the same, they have been declared in two different scopes.
- When declared inside the function foo(), the scope is local whereas when it was declared outside that function, the scope became global.

- **<u>Nonlocal Variables ::</u>**
- In python, nonlocal variables refer to all those variables that are declared within nested functions.
- The local scope of a nonlocal variable is not defined. This essentially means that the variable exists neither in the local scope nor in the global scope.
- In order to create a nonlocal variable in pyhton, one needs to use the nonlocal keyword. Take a look at the following example to understand how:

```
def outer():
    x = “local”
    def inner():
            nonlocal x
            x = “nonlocal”
            print(“inner:”, x)
   inner()
   print(“outer:”, x)
```

- Here, the inner() function is nested in nature.
- The nonlocal keyword has been used to create a nonlocal variable. As we can see, the inner() function in itself is defined within the scope of another function i.e., outer().
- An important point to be noted here is that if one were to modify the value of a nonlocal variable, the changes carried out would manifest themselves in the local variable as well.

Comparison Basis	Global Variable	Local Variable
Definition	declared outside the functions	declared within the functions
Lifetime	They are created the execution of the program begins and are lost when the program is ended	They are created when the function starts its execution and are lost when the function ends
Data Sharing	Offers Data Sharing	It doesn't offers Data Sharing
Scope	Can be access throughout the code	Can access only inside the function
Parameters needed	parameter passing is not necessary	parameter passing is necessary
Storage	A fixed location selected by the compiler	They are kept on the stack
Value	Once the value changes it is reflected throughout the code	once changed the variable don't affect other functions of the program

ASSIGN FUNCTION TO A VARIABLE IN PYTHON

- In Python, we can assign a function to a variable. And using that variable we can call the function as many as times we want. Thereby, increasing code reusability.
- Simply assign a function to the desired variable but without () i.e. just with the name of the function. If the variable is assigned with function along with the brackets (), None will be returned.

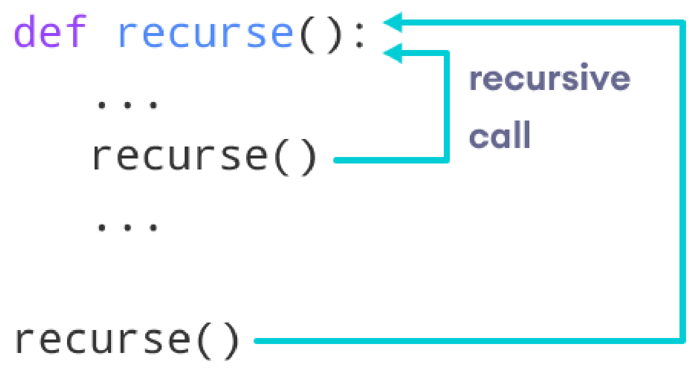
```
def a():
        print("GFG")
# assigning function to a variable
var=a
# calling the variable
var()
```

```
# defined function
x = 123
def sum():
        x = 98
        print(x)
        print(globals()['x'])
# drivercode
print(x)
# assigning function
z = sum
# invoke function
z()
z()
```

```
# function defined
def even_num(a):
        if a % 2 == 0:
                print("even number")
        else:
                print("odd number")
# drivercode
# assigning function
z = even_num
# invoke function with argument
z(67)
z(10)
z(7)
```

RECURSIVE FUNCTION

- In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.
- The following image shows the working of a recursive function called recurse.



```
def recurse():
    ...
    recurse()
    ...

recurse()
```

- Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
- The developer should be very careful with recursion as it 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.

- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError.
- Let's look at one such condition.

```
def recursor():
        recursor()
recursor()
```

- **Advantages of Recursion :**
  - Recursive functions make the code look clean and elegant.
  - A complex task can be broken down into simpler sub-problems using recursion.
  - Sequence generation is easier with recursion than using some nested iteration.
- **Disadvantages of Recursion :**
  - Sometimes the logic behind recursion is hard to follow through.
  - Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
  - Recursive functions are hard to debug.

```
def factorial(n):
    if n==0:
        result=1
    else:
        result = n * factorial(n-1)
    return result
n=int(input("Enter Value : "))
print("Factorial is :  ",factorial(n))
```

LAMBDA EXPRESSIONS / LAMBDA FUNCTIONS

- A function without a name is called ‘anonymous function’.
- These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

- Syntax : The syntax of lambda function contains only a single statement, which is as follows

```
lambda [arg1 [,arg2,.... argn]]:expression
```

- Example to show how lambda form of function works-

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2
# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

- Because a lambda functions is always represented by a function, we can pass a lambda function to another function.
- It means we are passing a function (i.e. lambda) to another function. This makes processing the data very easy.
- For example, lambda functions are generally used with functions like filter(), map() or reduce().

USING LAMBDAS WITH FILTER() FUNCTION

- The filter() function is useful to filter out the elements of a sequence depending on the result of a function.
- We should supply a function and a sequence to the filter() function as:

  filter(function, sequence)

- Here, the 'function' represents a function name that may return either True or False; and 'sequence' represents a list, string or tuple.
- The 'function' is applied to every element of the 'sequence' and when the function returns True, the element is extracted otherwise it is ignored.

- **Example**: A python program using filter() to filter out even numbers from a list

```
# filter() function that returns even numbers from a list
def is_even(x) :
if x%2 == 0 :
return True
else:
return False
# let us take a list of numbers
lst = [10, 23, 45, 46, 70, 99]
# call filter() with is_even() and lst
lst1 = list(filter(is_even, lst))
print(lst1)
```

- Example: A python program using lambda that returns even numbers from a list.

```
# a lambda function that returns even numbers from a list
lst = [10, 23, 45, 46, 70, 99]
lst1 = list(filter(lambda x: (x%2 == 0), lst))
print(lst1)
```

USING LAMBDAS WITH MAP() FUNCTION

- The map() function is similar to filter() function but it acts on each element of the sequence and perhaps changes the elements. The format of the map() function is:

  map(function, sequence)

- The 'function' performs a specified operation on all the elements of the sequence and the modified elements are returned which can be stored in another sequence.
- Example: A python program to find squares of elements in a list.

```
# map() function that gives squares
def squares(x):
return x*x
# let us take a list of numbers
lst = [1,2,3,4,5]
# call map() with squares() and lst
lst1 = list(map(squares, lst))
print(lst1)
```

- Example: A python program using lambda that returns squares of elements in a list.

```
# Lambda that returns squares
lst = [1,2,3,4,5]
lst1 = list( map( lambda x: x*x, lst ) )
print(lst1)
```

- Example: A python program to find the products of elements of 2 different lists using lambda function.

```
# Lambda that returns products of elements of two lists
lst1 = [1,2,3,4,5]
lst2 = [10,20,30,40,50]
lst3 = list( map( lambda x, y : x*y , lst1 , lst2 ) )
print(lst3)
```

USING LAMBDAS WITH REDUCE() FUNCTION

- The reduce() function reduces a sequence of elements to a single value by processing the elements according to a function supplied.
- The format of the reduce() function is:

  reduce(function, sequence)

- Example: A lambda function to calculate products of elements of a list.

```
# Lambda that returns products of elements of a list
from functools import *
lst = [1,2,3,4,5]
result = reduce(lambda x, y : x*y , lst)
print (result)
```

- reduce() function belongs to functools module in Python, we had imported it.
- Here, the reduce() function reduces the list to a final value as indicated by the lambda function. The lambda function is taking two arguments and returning their product.
- Hence, starting from the 0 th element of the list 'lst', the first two elements are multiplied and the product is obtained. Then this product is multiplied with the third element and the product is obtained.
- Again this product is multiplied with the fourth element and so on. The final product value is returned

- Example: A lambda function to calculate the sum of numbers from 1 to 50.

```
# Lambda that returns products of elements of a list
from functools import *
result = reduce(lambda x, y : x+y , range(1,51))
print (result)
```

- **filter()** is used to filter elements based on a condition.
- **map()** is used to apply a function to each element in an iterable.
- **reduce()** is used to accumulate the results of a binary function over the elements of an iterable.

BASICS OF MODULE

- A **library** refers to a collection of modules that together cater to a specific type of needs or application.
- **Module** is a file(.py file) containing variables, class definitions statements, and functions related to a particular task.
- Python modules that come preloaded with Python are called standard library modules.
- Using Modules increase **code reusability** and helps in **easy maintenance** of programs.
- In simple terms, we can consider a module to be the same as a code library or a file that contains a set of functions that you want to include in your application.
- The Python standard library contains well over 200 modules, although the exact number varies between distributions.
- There are a lot of built-in modules in Python. Some of the important ones are - collections, datetime, logging, math, numpy, os, pip, sys, and time

CREATING A MODULE

- We will be creating a module named tempConversion.py that converts values from F to C and vice-versa.

```
# function to convert F to C
def to_centigrade(x):
        return 5 * (x - 32) / 9.0

# function to convert C to F
def to_fahrenheit(x):
        return 9 * x / 5.0 + 32

# constants
# water freezing temperature(in Celsius)
FREEZING_C = 0.0
# water freezing temperature(in Fahrenheit)
FREEZING_F = 32.0
```

- Now save this python file and the module is created. This module can be used in other programs after importing it.

IMPORTING A MODULE

- In python, in order to use a module, it has to be imported. Python provides multiple ways to import modules in a program :
- To import the entire module :

```
import module_name
```

- To import only a certain portion of the module :

```
from module_name import object_name
```

- To import all the objects of the module :

```
from module_name import *
```

- We can rename the module while importing it using the as keyword.

```
import module_name as New_name
```

USING AN IMPORTED MODULE

- After importing the module, we can use any function/definition of the imported module as per the following syntax:

```
module_name.function_name()
```

- This way of referring to the module's object is called dot notation.
- If we import a function using from, there is no need to mention the module name and the dot notation to use that function.
- **Example 1 :** Importing the whole module :

```
# importing the module
import tempConversion

# using a function of the module
print(tempConversion.to_centigrade(12))

# fetching an object of the module
print(tempConversion.FREEZING_F)
```

- **Example 2 :** Importing particular components of the module :

```
# importing the to_fahrenheit() method
from tempConversion import to_fahrenheit

# using the imported method
print(to_fahrenheit(20))

# importing the FREEZING_C object
from tempConversion import FREEZING_C

# printing the imported variable
print(FREEZING_C)
```

THIRD PARTY MODULE

- A third party module is any code that has been written by a third party (neither you nor the python writers (PSF)).
- You can use them to add functionality to your code without having to write it yourself.
- Some commonly used third party modules are:
  - Pyperclip
  - Emoji
  - Howdoi
  - Wikipedia
  - Antigravity
  - Trutle

INSTALLING THIRD PARTY PYTHON PACKAGES

- For this purpose, download the compressed (tar.gz or .zip) file for the required package.
- Extract the compressed file (to extract tar.gz file in linux use, tar -xvzf package_name.tar.gz).
- Now, go to the extracted package directory using

```
cd package_name
```

- Some sources to download third party libraries are:

  https://github.com

  https://pypi.python.org/pypi

INSTALLING THIRD PARTY PYTHON PACKAGES

- The executable file for the pip tool is called *pip* on Windows and *pip3* on macOS and Linux.
- From the command line, you pass it the command install followed by the name of the module you want to install.
- For example, on Windows you would enter pip install --user *MODULE*, where *MODULE* is the name of the module.
- After installing a module, you can test that it installed successfully by running import *ModuleName* in the interactive shell.
- If no error messages are displayed, you can assume the module was installed successfully.
- If you already have the module installed but would like to upgrade it to the latest version available on PyPI, run pip install --user -U *MODULE*

LIBRARIES

- A library is an umbrella term that comprises a reusable set of Python code/instructions.
- A Python library is typically a collection of similar modules grouped together under a single name.
- Developers commonly utilize it to share reusable code with the community. This eliminates the need to write Python code from scratch.
- Developers and community researchers can construct their own set of useful functions in the same domain.
- When programmers and developers install the Python interpreter on their machines, standard libraries are included.
- Python libraries include matplotlib, Pygame, Pytorch, Requests, Beautifulsoup, and others.

DIFFERENCE BETWEEN MODULES AND LIBRARIES

Modules	Libraries
A module is a set of code or functions with the.py extension.	A library is a collection of related modules or packages.
They are used by both programmers and developers.	Libraries are used by community members, developers and researchers.
The use of modules makes it easier to read the code.	Libraries provide no contribution for easy readability.
Modules are logical groups of functionality that programmers can import in order to reuse their code or set of statements.	Libraries enable users of programming languages, developers, and other researchers to reuse collections of logically related code.
When a Python programmer imports a module, the interpreter searches various locations for the module's definition or body.	Before we can use the libraries' modules or packages, we must first install them in our Python project. We generally use the pip install command.

DIFFERENCE BETWEEN MODULES AND LIBRARIES

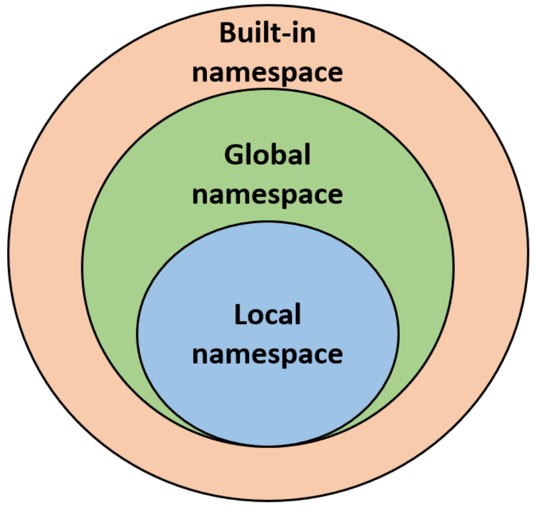
Modules	**Libraries**
Modules are generally written in Python with valid statements or codes.	Libraries, especially standard libraries, are usually developed in C or Python.
The basic goal of creating a module is to prevent DRY i.e, Don't Repeat Yourself.	Libraries do not have such a goal.
To return a sorted list of strings containing the function names specified within a module, we can use Python's built-in dir() function.	There is no explicit function that returns the number of modules in a library. Even so, programmers can utilize the help() function to get some information.
Popular built-in Python modules include os, sys, math, random, and so on.	Popular built-in Python libraries include Pygame, Pytorch, matplotlib, and more.

WHAT IS NAMESPACE:

- A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method.
- Python itself maintains a namespace in the form of a Python dictionary.
- Real-time example, the role of a namespace is like a surname. One might not find a single "Alice" in the class there might be multiple "Alice" but when you particularly ask for "Alice Lee" or "Alice Clark" (with a surname), there will be only one (time being don't think of both first name and surname are same for multiple students).
- On similar lines, the Python interpreter understands what exact method or variable one is trying to point to in the code, depending upon the namespace.
- So, the division of the word itself gives a little more information. Its **Name** (which means name, a unique identifier) + **Space**(which talks something related to scope).
- Here, a name might be of any Python method or variable and space depends upon the location from where is trying to access a variable or a method.

TYPES OF NAMESPACES :

- The **built-in namespace** contains the names of all of Python's built-in objects. These are available at all times when Python is running. Some functions like print(), id() are always present, these are built-in namespaces.
- The **global namespace** contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.
- The interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

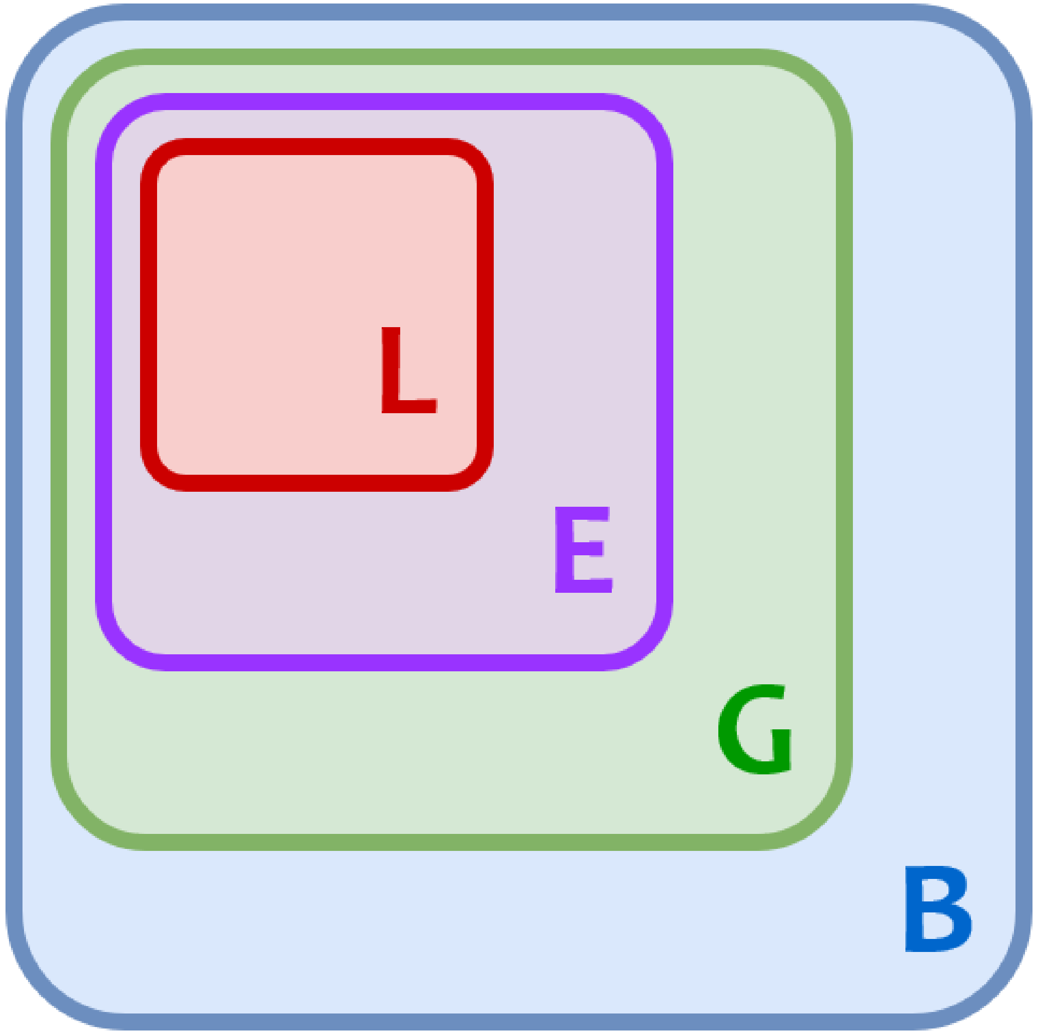


Type of Namespaces

PYTHON VARIABLE SCOPE

- Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.
- A scope is the portion of a program from where a namespace can be accessed directly without any prefix.
- At any given moment, there are at least three nested scopes.
    1. Scope of the current function which has local names
    2. Scope of the module which has global names
    3. Outermost scope which has built-in names
- When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.
- If there is a function inside another function, a new scope is nested inside the local scope.

- If your code refers to the name x, then Python searches for x in the following namespaces in the order shown:
- **Local:** If you refer to x inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
- **Enclosing:** If x isn't in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
- **Global:** If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
- **Built-in:** If it can't find x anywhere else, then the interpreter tries the built-in scope.
- This is **the LEGB rule** as it's commonly called in Python literature (although the term doesn't actually appear in the Python documentation).
- If the interpreter doesn't find the name in any of these locations, then Python raises a NameError exception.


L
E
G
B

EXAMPLE 1: SCOPE AND NAMESPACE IN PYTHON

```
# global_var is in the global namespace
global_var = 10
def outer_function():
    #  outer_var is in the local namespace
    outer_var = 20
    def inner_function():
        #  inner_var is in the nested local namespace
        inner_var = 30
        print(inner_var)
    print(outer_var)
    inner_function()
# print the value of the global variable
print(global_var)
# call the outer function and print local and nested local variables
outer_function()
```

GLOBALS() FUNCTION:

- When the global and local variable names are same, the programmer will face difficulty to differentiate between them inside a function. For example there is a global variable ‘a’ with some value declared above the function.
- The programmer is writing a local variable with the same name ‘a’ with some other value inside the function.
- For example:

```
a = 1 # global variable
def myfunction():
        a = 2 # a is local variable
```

- Now, if the programmer wants to work with global variable, how is it possible? If he uses ‘global’ keyword, then he can access only global variable and the local variable is no more available.
- The globals() function will solve this problem. This is a built in function which returns a table of current global variables in the form of a dictionary.
- Hence, using this function, we can refer to the global variable ‘a’, as: globals()*‘a’+
- Now, this value can be assigned to another variable, say ‘x’ and the programmer can work with that value.

- Example: A python program to get a copy of global variable into a function and work with it.

```
# same name for global and local variables
a = 1 # this is global variable
def myfunction():
    a = 2 # a is local variable
    x = globals()*‘a’+ # get global variable into x
    print(‘global variable a= ‘, x)
    print(‘local variable a= ‘, a)
myfunction()
print(‘global variable a= ‘, a)
```

Thank you!