**Sach Educational Support**

**Data Science Internship Task**

Question 1) What is Scope in Python and Scope Resolution in Python ?

Answer: **Python Scope:** A variable is only available from inside the region it is created. This is called **scope** .The concept of **scope** rules how [variables](https://realpython.com/python-variables/) and names are looked up in your code. It determines the visibility of a variable within the code. The scope of a name or variable depends on the place in your code where you create that variable.

The **scope** of a name defines the area of a program in which we can unambiguously access that name, such as variables, functions, objects, and so on. A name will only be visible to and accessible by the code in its scope. Several programming languages take advantage of scope for avoiding name collisions and unpredictable behaviours.

There are 2 Types of Scope:

1. **Local scope:** The names that you define in this scope are only available or visible to the code within the scope. A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

**Example:** A variable created inside a function is available inside that function:

def func\_name():  
  x = 300  
  print(x)

1. **Global scope:**  The names that we define in this scope are available to all your code. A variable created in the main body of the Python code is a global variable and belongs to the global scope .Global variables are available from within any scope, global and local.

**Example:** A variable created outside of a function is global and can be used by anyone.

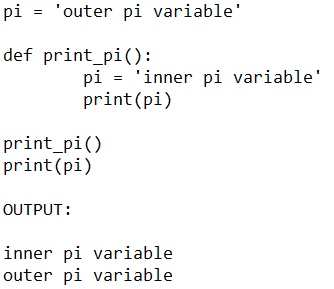
x = 300  
  
def func\_name():  
  print(x)  
  
func\_name()  
  
print(x)

**Names and Scopes in Python**

Since Python is a [dynamically-typed](https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language) language, variables in Python come into existence when you first assign them a value. On the other hand, functions and classes are available after you define them using [def](https://docs.python.org/3/reference/compound_stmts.html#def) or [class](https://docs.python.org/3/reference/compound_stmts.html#class-definitions), respectively. Finally, [modules](https://realpython.com/python-modules-packages/) exist after you import them.

We can use operations like Assignment(=), Import operations, function definitions, argument definitions in the context of functions, class definitions to create or , in case of assignment to update new python names because all of them assign a name to a variable, constant , functions, class, instance, module or other python objects.

**Python Scope Resolution :** It defines the hierarchical order in which we have to search the namespaces for obtaining the mapping o the name-to-object (variables). Variables exist in this context, and from this, they are reference. It also defines the lifetime and accessibility of a variable. Below is an example.



Below we have listed the scopes in terms of their hierarchy (narrowest t broadest):

* Local (L) – We can define it inside the function class.
* Enclosed (E) – We can define it inside the enclosing function (Nested function).
* Global (G) – We can define it at the uppermost level.
* Built-in (B) – These are the reserve name in the built-in modules of Python.

Question 2) What are Decorators in Python ?

Answer: A decorator takes in a function, adds some functionality and returns it. [Decorators](https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/) are very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class. Decorators allow us to wrap another function in order to extend the behaviour of wrapped function, without permanently modifying it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Syntax for Decorator:

@gfg\_decorator

def hello\_decorator():

    print("Gfg")

OR

def hello\_decorator():

    print("Gfg")

hello\_decorator = gfg\_decorator(hello\_decorator)

In the above code, gfg\_decorator is a callable function, will add some code on the top of some another callable function, hello\_decorator function and return the wrapper function.

This is also called metaprogramming because a part of the program tries to modify another part of the program at compile time. Basically, a decorator takes in a function, adds some functionality and returns it.

Generally , we decorate a function and reassign it as,

Ordinary = make\_pretty(ordinary)

This is a common construct and for this reason, Python has a syntax to simplify this. We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated.

Decorating Functions with Parameters - If we had functions that took in parameters like:

Def divide(a,b):

Return a/b

The function has 2 parameters a and b. We know it will give an error if we pass b as 0. Now lets make a decorator to check for this case that will cause an error.

Def decorator\_divide(func):

Def inner(a,b):

If b==0:

Print(“cannot be divided”)

return

return func(a,b)

return inner

@smart\_divide

def divide(a,b):

print(a/b)

This new implementation will return None if the error condition arises.