Modules in Python

Like many other programming languages, Python supports **modularity**. That is, you can break large code into smaller and more manageable pieces. And through modularity, Python supports **code reuse**. You can import modules in Python into your programs and reuse the code therein as many times as you want.

What are Python Modules?

Modules provide us with a way to share reusable functions. A module is simply a “Python file” which contains code we can reuse in multiple Python programs. A module may contain functions, classes, lists, etc.

Modules in Python can be of two types:

- Built-in Modules.
- User-defined Modules.

Built-in Modules in Python

One of the many superpowers of Python is that it comes with a “rich standard library”. This rich standard library contains lots of built-in modules. Hence, it provides a lot of reusable code.

To name a few, Python contains modules like “os”, “sys”, “datetime”, “random”.

You can import and use any of the built-in modules whenever you like in your program.

User-Defined Modules in Python

Another superpower of Python is that it lets you take things in your own hands. You can create your own functions and classes, put them inside modules and voila! You can now include hundreds of lines of code into any program just by writing a simple import statement.
To create a module, just put the code inside a .py file.
Let's create one.

```
# my Python module
def greeting(x):
print("Hello,", x)
```

Write this code in a file and save the file with the name mypymodule.py. Now we have created our own module.

Importing Modules in Python

We use the import keyword to import both built-in and user-defined modules in Python.

Let's import our user-defined module from the previous section into our Python shell:

```
>>> import mypymodule
```

To call the greeting **function of** mypymodule, we simply need to use the dot notation:

```
>>> mypymodule.greeting("Techvidvan")
```

Let's now import a built-in module into our Python shell:

```
>>> import random
```

To call the randint **function of** random, we simply need to use the dot notation:

```
>>> random.randint(20, 100)
```

Using import…as statement (Renaming a module)

This lets you give a shorter name to a module while using it in your program.

```
>>> import random as r

>>> r.randint(20, 100)
```

Using from…import statement

You can import a specific function, class, or attribute from a module rather than importing the entire module. Follow the syntax below,

from <modulename> import <**function**>

```
>>> from random import randint
>>> randint(20, 100)
69
```

You can also import multiple attributes and functions from a module:

```
>>> from math import pi, sqrt

>>> print(3 * pi)
9.42477796076938

>>> print(sqrt(100))
10.0
>>>
```

Note that while importing from a module in this way, we don’t need to use the dot operator while calling the function or using the attribute.

Importing everything from Python module

If we need to import everything from a module and we don't want to use the dot operator, do this:

```
>>> from math import *

>>> print(3 * pi)

9.42477796076938

>>> print(sqrt(100))

10.0
>>>
```

Python dir() function

The dir() function will return the names of all the properties and methods present in a module.

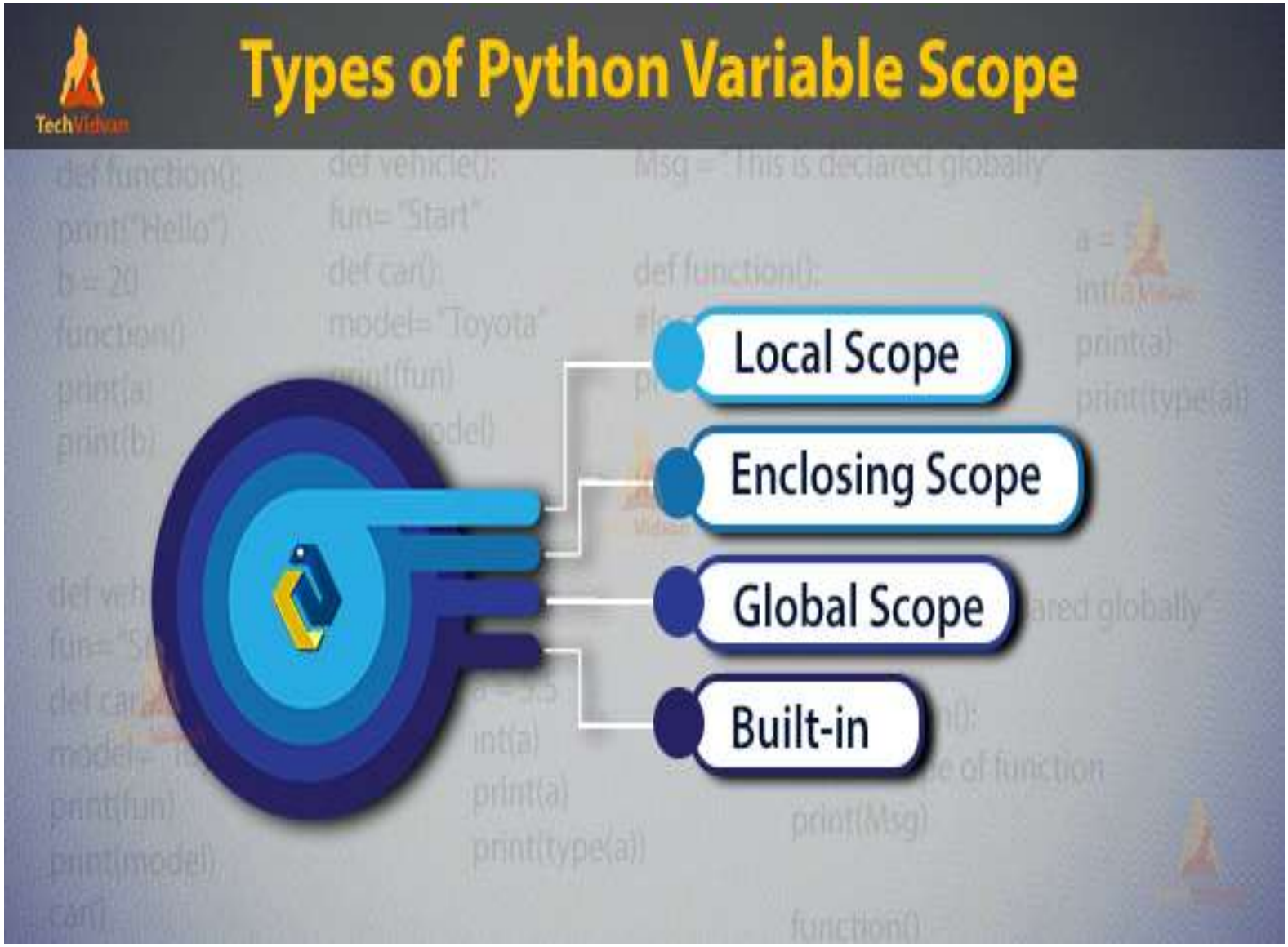
```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_Sequence', '_Set',
'__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_accumulate',
'_acos', '_bisect', '_ceil', '_cos', '_e', '_exp', '_inst', '_log', '_os', '_pi',
'_random', '_repeat', '_sha512', '_sin', '_sqrt', '_test',
'_test_generator', '_urandom', '_warn', 'betavariate', 'choice',
'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits',
'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randint',
'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular',
'uniform', 'vonmisesvariate', 'weibullvariate‘]
```

Python Variable Scope

What is Variable Scope in Python?

In programming languages, variables need to be **defined** before using them. These variables can only be accessed in the area where they are defined, this is called **scope**. You can think of this as a block where you can access variables.

Types of Python Variable Scope



There are four types of variable scope in Python, let's go through each of them.

1. Local Scope

Local scope variables can only be accessed within its **block**.

Let's see it with an example.

```
a = 10

def function():
print("Hello")
b = 20
function()
print(a)
print(b)
```

Output:

```
Hello
10
Traceback (most recent call last):
  File “main.py”, line 7, in <module>
    print(b)
NameError: name ‘b’ is not defined
```

In the above example, we see that Python prints the value of variable a but it cannot find variable b. This is because b was defined as a **local scope** in the function so, we cannot access the variable outside the function. This is the nature of the local scope.

Global Scope

The variables that are declared in the global scope can be accessed from anywhere in the program. Global variables can be used **inside** any functions. We can also change the global variable value.

```
Msg = "This is declared globally"
def function():
#local scope of function
print(Msg)
function()
```

Output:

This is declared globally

But, what would happen if you declare a local variable with the **same name** as a global variable inside a function?

```
msg = “global variable”
def function():
#local scope of function
msg = “local variable”
print(msg)
function()
print(msg)
```

Output:

```
local variable
global variable
```

As you can see, if we declare a local variable with the same name as a **global variable** then the local scope will use the **local variable**.
If you want to use the global variable inside local scope then you will need to use the **“global”** keyword.

Enclosing Scope

A scope that isn’t **local** or **global** comes under enclosing scope.

```
def vehicle():
fun= “Start”
def car():
model= “Toyota”
print(fun)
print(model)
car()
vehicle()
```

Output:

```
Start
Toyota
```

In the example code, the variable fun is used inside the **car() function**. In that case, it is neither a local scope nor a global scope. This is called the enclosing scope.

Built-in Scope

This is the **widest scope** in Python. All the reserved names in Python **built-in modules** have a **built-in scope**.

When the Python doesn't find an identifier in it's local, enclosing or global scope, it then looks in the built-in scope to see if it's defined there.

```
a = 5.5
int(a)
print(a)
print(type(a))
```

Python would see in the local scope first to see which of the variables are defined in the local scope, then it will look in the enclosing scope and then global scope.

If the identifier is not found anywhere then, at last, it will check the built-in scope.

Here the functions **int()**, **print()**, **type()** does not need to be defined because they are already defined in the built-in scope of Python.