Namespaces and scopes (global & local variables)

- `Namespace` is nothing but context.
- `Namespace` is a naming system to make names unique to avoid ambiguity.
- Our everyday examples - naming family members. Within `Namespace` of family members we use unique name to identify each family member.
- `Namespaces` are maintained as dictionary objects in Python's internal implementation.

Namespaces

- Global (names defined at module level)
- Local (names defined within function/ class definition etc.)
- Built-in (names defined in programming language)

Scope

The code block within which certain namespace is directly accessible.

During program execution following is the order of scope that is followed -

- Innermost scope is searched first.
- Outmost scope is searched at last.

Inner-most scope is usually function definition. Next to it could be module level scope. And finally, the outer-most scope could be built-in names.

So in case of functions `func1` defined under file `file1.py`, the order of namespace look-up goes like following -

`func1` (function) → `file1.py` (module) → `built-in` names

`Global` keyword

What is `global` variable?

- Variables defined outside function definition (or class definition) and referenced inside function definition can be called as global variables.
- We use `global` keyword to be able to use a variable inside function definition from outer scope of function definition.

If a variable is being accessed inside a function definition from outer scope using `global`keyword then that variable is referred as `global` variable.

What is `local` variable?

- If a variable is only defined anywhere under function body (or a class body) it is called as `local` variable.

```
def change(a):
    a = 90
    print(f"Inside of the change function {a}")


a = 9
print(f"Before the function call {a}")
change(a)
print(f"After the function call {a}")
```

```
def change(b):
    global a
```

```
    a = 90
    print("inside change function", a)


a = 9
print("Before the function call ", a)
change(a)
print("After the function call ", a)
```

When variable is declared before function call namespace look-up will look within scope of function definition first. If variable is not defined there, it will look-up into module level scope.

```
def func():
    print(statement)

statement = "I love Pune for it's food"
func()

# OUTPUT
# I love Pune for it's food
```

When variable is referenced before “declaration” (assignment) we get exception as `unbondLocalError`.

```
def func():
    print(statement)
    statement = "I love Pune!"
    print(statement)


statement = "I love Mumbai!"
func()

#  ** ERROR **
# UnboundLocalError: local variable 'statement' referenced before assignment
```

`nonlocal` keyword

- When there is a function inside a function, the inner function can access the variables declared in outer function using `nonlocal` keyword.
- In short, `nonlocal` **keyword** is used to reference a variable in the nearest scope.

```
def foo():
    name = "Pune"  # Our local variable

    def bar():
        nonlocal name  # Reference name in the upper scope
        name = 'Mumbai'  # Overwrite this variable
        print(name)

    # Calling inner function
    bar()

    # Printing local variable
    print(name)


foo()
```