AIMERZ.ai
Aim.Act.Achieve

NOTES

Python Namespace

Namespaces and Scope of Variables in Python: Understanding Variable Visibility

In this submodule, we'll cover the foundational concepts of **namespaces**, **variable scope**, and how these two concepts affect the visibility and lifetime of variables in Python. This section will provide learners with a clear understanding of how Python handles variables and where they can be accessed or modified.

1. What is a Namespace?

A **namespace** is a system that allows Python to keep track of all the names (variables, functions, objects, etc.) that are currently being used in a program. Think of a namespace as a container or a **mapping between names and their corresponding objects**.

There are different types of namespaces:

- **Built-in Namespace**: Contains all the built-in functions and objects in Python like `print()`, `int()`, etc.
- **Global Namespace**: Contains names defined at the top level of a script or module.
- **Local Namespace**: Contains names defined inside a function or block of code.

Example:

```python
x = 10  # Global Namespace

def my_function():
    y = 5  # Local Namespace
    print(y)

my_function()
print(x)
```

Output:

```
Python
5
10
```

Here, x is in the **global namespace** (accessible anywhere in the program), and y is in the **local namespace** of the function my_function() (only accessible within the function). Namespaces organise variables and objects so they don't clash with each other.

2. Scope of Variables

The **scope** of a variable defines the **region of the code** where the variable can be accessed. Python has four levels of scope, commonly abbreviated as **LEGB**:

- **Local**: Variables created inside a function (accessible only within the function).
- **Enclosed**: Variables in the enclosing function (functions inside functions).
- **Global**: Variables declared at the top of the script or outside all functions (accessible throughout the entire program).
- **Built-in**: Special names that are part of Python's core (like len() or print()).

Examples of Scope:

Local Scope:

```
Python
def my_func():
    a = 10  # Local scope
    print(a)

my_func()
print(a)  # Error: 'a' is not defined outside the function
```

a is local to my_func() and is not accessible outside the function.

Global Scope:

```python
b = 20  # Global scope

def another_func():
    print(b)

another_func()  # Prints 20, as 'b' is a global variable
```

`b` is global, so it can be accessed from both inside and outside the function.

Enclosed Scope (Nested Functions):

```python
def outer_func():
    c = 30  # Enclosing scope: 'c' is defined in the outer function

    def inner_func():
        print(c)  # Inner function can access 'c' from the enclosing scope
    inner_func()  # Call the inner function to execute its code

outer_func()  # Prints 30, Call the outer function to start the process
```

`c` is in the enclosing scope and accessible to `inner_func()`.

Built-in Scope:

```python
print(len([1, 2, 3]))  # Print 3,Using the built-in 'len()' function
```

Python provides built-in functions that are available everywhere unless overridden.

3. Variable Visibility and Namespace Hierarchy

Python uses the **LEGB rule** to determine the visibility and accessibility of variables. When you try to access a variable, Python searches in this order:

1. **Local scope** (inside the current function or block of code).
2. **Enclosing scope** (if there's a nested function).
3. **Global scope** (at the top of the script or module).
4. **Built-in scope** (Python's built-in functions and constants).

Example of LEGB Rule:

```python
x = 'global'  # Global scope variable

def outer():
    x = 'enclosing'  # Enclosing scope variable

    def inner():
        x = 'local'  # Local scope variable
        print(x)  # Prints the local variable 'x'

    inner()  # Call the inner function

outer()  # Call the outer function
print(x)  # Prints the global variable 'x'
```

In this example, the `inner()` function will print `'local'` because it follows the LEGB rule, starting with the local scope first. The outer function won't affect the local variable.

Output:

```python
local
global
```

4. Changing Global Variables in Functions

By default, Python will not allow you to change a global variable inside a function unless you explicitly tell it to do so using the **global** keyword.

Example of Using global:

```
Python
x = 10  # Global variable

def modify_global():
    global x
    x = 20  # Modify the global variable

modify_global()
print(x)  # Output: 20
```

Without the global keyword, Python would treat x inside modify_global() as a local variable, creating a new one instead of modifying the global x.

5. global() and locals() Functions

- **globals()**: This function returns a dictionary of the current global symbol table.
- **locals()**: This function returns a dictionary of the current local symbol table.

Examples:

```
Python
def test():
    a = 10
    print(locals())  # Shows local variables

test()
print(globals())  # Shows all global variables
```

6. nonlocal Keyword

- **Purpose**: Used to modify a variable in the enclosing (non-global) scope.

Example:

```
Python
def outer():
    x = 10
    def inner():
        nonlocal x
        x = 20
    inner()
    print(x)  # Output: 20
outer()
```

7. Shadowing and Variable Overlap

- **Variable Shadowing**: When a local variable has the same name as a global variable, the local one "shadows" the global one within its scope.

Example:

```
Python
x = 5  # Global

def my_function():
    x = 10  # Local (shadows global)
    print(x)  # Output: 10

my_function()
print(x)  # Output: 5
```

8. Best Practices

Avoid Overusing Global Variables: It's generally a good idea to limit the use of global variables, as they can make debugging more difficult.

Use Descriptive Variable Names: Avoid conflicts in namespace by giving meaningful, unique names to your variables, especially in larger projects.

Understand Variable Scope: Be mindful of where you declare variables to avoid scope-related bugs (e.g., trying to use a local variable outside its function).

Don't Override Built-in Functions: Avoid naming your variables the same as Python's built-in functions (e.g., `len`, `list`, etc.).

THANK YOU

AIMERZ.ai

Aim.Act.Achieve