🚀 Session 11 : Advanced Functions in Python

In this session, we will cover:

- Arbitrary arguments (`*args`, `**kwargs`)
- Scope and lifetime of variables (LEGB Rule)
- Lambda functions
- Higher-order functions (`map`, `filter`)

🔷 1. Arbitrary Arguments (`*args`, `**kwargs`)

📌 Sometimes we don't know in advance how many arguments a function should take.

- `*args` → collects **positional arguments** into a tuple
- `**kwargs` → collects **keyword arguments** into a dictionary

```
# Example with *args
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))       # 6
print(add_numbers(10, 20, 30, 40)) # 100
```

```
# Example with **kwargs
def print_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} = {value}")

print_details(name="Vipin", age=22, country="India")
```

```
# Using both *args and **kwargs together
def mix_example(*args, **kwargs):
    print("Args:", args)
    print("Kwargs:", kwargs)

mix_example(1, 2, 3, name="Alice", job="Engineer")
```

🔷 2. Scope and Lifetime of Variables (LEGB Rule)

Python searches variables in the following order:

1. **Local (L)** – inside the current function
2. **Enclosing (E)** – inside nested functions
3. **Global (G)** – defined at the top level
4. **Built-in (B)** – Python's reserved names (`len`, `sum`, etc.)

🔷 1. Local (L)

📌 Local variables are defined **inside a function** and can only be accessed within that function.

```
def local_example():
    x = 10   # Local variable
    print("Inside function:", x)

local_example()

# print(x)  # ❌ Uncommenting this will throw an error (x not defined outside)
```

🔷 2. Enclosing (E)

📌 In **nested functions**, the outer function's variables are in the enclosing scope.
Inner functions can access them, but need `nonlocal` to modify them.

```
def outer():
    x = "enclosing"

    def inner():
        print("Inner sees:", x)  # Enclosing scope variable

    inner()

outer()
```

```
# Example with nonlocal
def outer():
    x = "old"

    def inner():
        nonlocal x
        x = "new"  # modifies enclosing variable
        print("Inner changed x to:", x)

    inner()
    print("Outer now sees:", x)

outer()
```

🔷 3. Global (G)

📌 Variables declared at the **top level** of a script are global.
They can be accessed anywhere in the file.
To modify them inside a function, use the `global` keyword.

```
x = "global"

def show_global():
    print("Inside function:", x)  # Uses global variable

show_global()
print("Outside function:", x)
```

```
# Example with global keyword
x = "global"

def change_global():
    global x
    x = "changed globally"

change_global()
print(x)  # changed globally
```

🔷 2. Enclosing (E)

📌 In **nested functions**, the outer function's variables are in the enclosing scope.
Inner functions can access them, but need `nonlocal` to modify them.

```
# LEGB Example
x = "global"

def outer():
    x = "enclosing"

    def inner():
        x = "local"
        print("Inner:", x)   # Local

    inner()
    print("Outer:", x)       # Enclosing

outer()
```

```
print("Global:", x)          # Global
```

🔷 4. Built-in (B)

📌 Python provides many built-in names (`len`, `sum`, `min`, etc.).
Avoid overriding them, or it can cause errors.

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

# Overriding built-in (⚠️ not recommended)
len = 100
# print(len([1,2,3]))  # ❌ This will cause an error if run
```

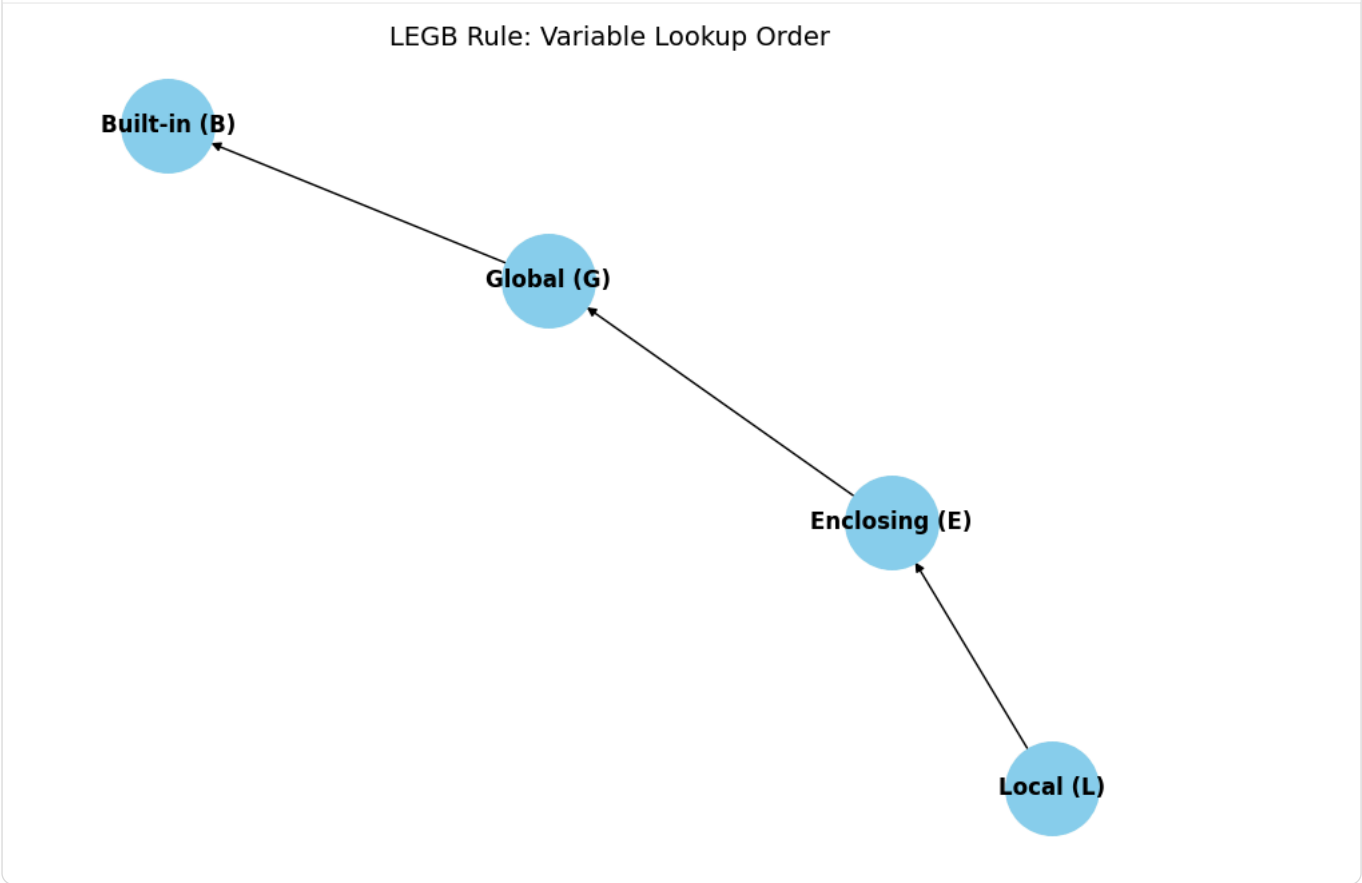
```
import matplotlib.pyplot as plt
import networkx as nx

# Create a directed graph
G = nx.DiGraph()

# Add nodes for LEGB
scopes = ["Built-in (B)", "Global (G)", "Enclosing (E)", "Local (L)"]
for scope in scopes:
    G.add_node(scope)

# Add edges to represent search order
G.add_edges_from([
    ("Local (L)", "Enclosing (E)"),
    ("Enclosing (E)", "Global (G)"),
    ("Global (G)", "Built-in (B)")
])

# Draw the graph
plt.figure(figsize=(8,6))
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color="skyblue", node_size=2500, font_size=12, font_weight="bold", arrows=T
plt.title("LEGB Rule: Variable Lookup Order", fontsize=14)
plt.show()
```



🔷 3. Lambda Functions

📌 Lambda functions are **anonymous (nameless) functions** written in a single line.

Syntax:

```
lambda arguments: expression
```

```
```python
# Normal function
def square(x):
    return x**2

print(square(5))  # 25

# Lambda function
square_lambda = lambda x: x**2
print(square_lambda(5))  # 25
```

```
# Lambda with multiple arguments
add = lambda a, b: a + b
print(add(3, 7))  # 10
```

🔷 Higher-Order Functions in Python – `map()` and `filter()`

In Python, **higher-order functions** are functions that can take other functions as arguments or return them as results.

Two very useful built-in higher-order functions are:

1. **`map(function, iterable)`**
   - Applies a given `function` to **every element** of an `iterable` (like list, tuple).
   - Returns a `map` object (which can be converted to `list`, `tuple`, etc.).
2. **`filter(function, iterable)`**
   - Filters elements of an `iterable` using a `function` that returns `True` or `False`.
   - Returns only those elements for which the function is `True`.

```
# Example 1: Square numbers using map
nums = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, nums))
print("Squared:", squared)
```

```
# Example 2: Convert Celsius to Fahrenheit
celsius = [0, 10, 20, 30, 40]
fahrenheit = list(map(lambda c: (9/5)*c + 32, celsius))
print("Fahrenheit:", fahrenheit)
```

```
# Example 3: Map with multiple iterables
a = [1, 2, 3]
b = [4, 5, 6]
sum_ab = list(map(lambda x, y: x + y, a, b))
print("Sum of lists:", sum_ab)
```

🔷 2. `filter()` Function

Syntax:

```
filter(function, iterable)
```

```
# Example 1: Get even numbers using filter
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
evens = list(filter(lambda x: x % 2 == 0, nums))
print("Even numbers:", evens)
```
```

```
# Example 2: Filter words longer than 3 letters
words = ["hi", "hello", "sun", "python", "AI"]
long_words = list(filter(lambda w: len(w) > 3, words))
print("Words longer than 3 letters:", long_words)
```


```
# Example 2: Filter words longer than 3 letters
words = ["hi", "hello", "sun", "python", "AI"]
long_words = list(filter(lambda w: len(w) > 3, words))
print("Words longer than 3 letters:", long_words)
```


```
# Example 3: Filter positive numbers
numbers = [-5, -2, 0, 1, 7, -3, 8]
positives = list(filter(lambda n: n > 0, numbers))
print("Positive numbers:", positives)
```

🔷 Key Difference Between `map()` and `filter()`

Feature	`map()`	`filter()`
Purpose	Transforms elements	Selects elements
Function Type	Returns transformed value	Returns `True` / `False`
Output Size	Same as input size	Smaller or equal to input size
Example	`map(lambda x: x*2, [1,2,3]) → [2,4,6]`	`filter(lambda x: x>2, [1,2,3]) → [3]`

🎯 Practice Tasks

1. Write a function that accepts any number of numbers using `*args` and returns their average.
2. Write a function that accepts personal details using `**kwargs` and prints them.
3. Demonstrate the **LEGB rule** with your own nested function example.
4. Use a **lambda function** to:
    - Cube a number
    - Check if a number is even
5. Use `map` with lambda to convert a list of temperatures in Celsius to Fahrenheit.
6. Use `filter` with lambda to get only numbers divisible by 3 from a list.