

```
# **kwargs example
def configure_device(device_name, **settings):
    """Applies configuration settings to a device."""
    print(f"Configuring device: {device_name}")
    # 'settings' is a dictionary
    for key, value in settings.items():
        print(f" - {key}: {value}")

configure_device("Router-R1", ip_address="192.168.1.1", port=80, enable_qos=True)
```

4.3 Scope and Lifetime

Scope refers to the region of a program where a variable is accessible. **Lifetime** is the period during which the variable exists in memory.

Local vs. Global variables

- **Local:** Variables defined inside a function. They exist only while the function is executing and cannot be accessed from outside the function.
- **Global:** Variables defined outside all functions. They are accessible throughout the program, including inside functions (read-only access by default).

Example Program:

```
Python
global_counter = 0 # Global variable

def update_counter():
    local_variable = 10 # Local variable, dies when function ends

    # We can read the global variable
    print(f"Inside function, global_counter is: {global_counter}")

    # ERROR: Cannot directly modify a global variable without 'global' keyword
    # global_counter = 5 # This would create a new local variable named global_counter

update_counter()
# print(local_variable) # ERROR: local_variable is not defined outside the function
print(f"Outside function, global_counter is: {global_counter}")
```

LEGB Rule (Local, Enclosing, Global, Built-in)

Python uses this rule to determine the order in which scopes are searched for a variable name:

1. **Local:** Inside the current function.
2. **Enclosing:** Inside enclosing functions (for nested functions).
3. **Global:** At the top level of the module (script).
4. **Built-in:** Names pre-assigned by Python (e.g., print, len, range).

The global and nonlocal keywords

- **global:** Used inside a function to explicitly indicate that a variable being assigned to is the global variable.

- **nonlocal:** Used in nested functions to refer to a variable in the nearest enclosing scope that is *not* the global scope.

Example Program:

Python

```
count = 10 # Global variable
```

```
def modify_scopes():
```

```
    # 1. Accessing and modifying the Global scope
```

```
    global count
```

```
    count += 10 # Modifies the global 'count'
```

```
x = 5 # Enclosing scope variable
```

```
def nested_func():
```

```
    # 2. Accessing and modifying the Enclosing scope
```

```
    nonlocal x
```

```
    x += 1
```

```
y = 100 # Local scope variable
```

```
nested_func()
```

```
print(f"Enclosing x after modification: {x}") # x is 6
```

```
modify_scopes()
```

```
print(f"Global count after modification: {count}") # count is 20
```

4.4 Functional Programming Concepts (Core)

Functional programming emphasizes functions as primary, avoiding state and mutable data.

Lambda Functions (Anonymous functions)

A small, single-expression function that does not have a formal name. They are defined using the `lambda` keyword.

- **Syntax:** `lambda arguments: expression`

Real-Time Example: Using a lambda function as a key for sorting data.

Python

```
data = [("apple", 150), ("banana", 100), ("cherry", 200)]
```

```
# Sort the list of tuples based on the second element (the weight/number)
```

```
# The lambda function takes one element (x) and returns x[1]
```

```
sorted_data = sorted(data, key=lambda x: x[1])
```

```
print(f"Sorted by weight: {sorted_data}")
```

```
# Output: [('banana', 100), ('apple', 150), ('cherry', 200)]
```

Built-in higher-order functions: `map()`, `filter()`

These functions take other functions (often lambdas) as arguments.

- **map(func, iterable):** Applies a given function to all items in an input list (or other iterable) and returns a map object (which can be converted to a list).
- **filter(func, iterable):** Creates an iterator that filters out elements from an iterable for which a function returns False.

Real-Time Example: Processing a list of prices.

Python

```
prices = [10.50, 25.00, 5.99, 50.00]
```

```
# 1. map(): Apply 10% tax to all prices
# tax_func takes x and returns x * 1.10
taxed_prices = list(map(lambda p: p * 1.10, prices))
print(f"Taxed Prices: {taxed_prices}")
```

```
# 2. filter(): Select only prices greater than $20
# filter_func takes x and returns True only if x > 20
high_prices = list(filter(lambda p: p > 20, prices))
print(f"High Prices (> $20): {high_prices}")
```

Introduction to reduce() (from functools module)

The reduce() function applies a rolling computation to sequential pairs of values in a list. It returns a single final value.

Real-Time Example: Calculating the product of all numbers in a list.

Python

```
from functools import reduce
```

```
numbers = [1, 2, 3, 4, 5]
```

```
# The lambda accumulates the result.
# accumulator starts with the first item (1), current with the second (2).
# Next iteration: accumulator is 1*2, current is 3, and so on.
product = reduce(lambda accumulator, current: accumulator * current, numbers)

print(f"Product of numbers: {product}") # Output: 120 (1*2*3*4*5)
```

Recursion (Basic concept and examples)

Recursion is a technique where a function calls itself, either directly or indirectly. Every recursive function must have two parts:

1. **Base Case:** The condition that stops the recursion (prevents infinite calls).
2. **Recursive Step:** The part where the function calls itself, usually with a smaller input.

Real-Time Example: Calculating the factorial of a number ($n! = n \times (n-1)!$).

Python

```
def factorial(n):
    """Calculates n! using recursion."""

    # Base Case: Stop condition
```