LECTURE NOTES ON

PROGRAMMING WITH PYTHON

PREPARED BY :

Miss. PRIYANKA MAHARANA

MODULE-1

INTRODUCTION:

Python is a high-level, interpreted, and general-purpose programming language created by **Guido van Rossum** and first released in 1991. It is widely celebrated for its clean, English-like syntax, which prioritizes readability and allows developers to accomplish tasks with fewer lines of code than languages like Java or C++.

Key Features of Python:

- **Simple & Readable Syntax:** Python uses a clean, intuitive syntax that mimics natural English. It uses **indentation** instead of curly brackets to define code blocks, enforcing a consistent visual structure.
- **Interpreted Language:** Code is executed line-by-line, which simplifies debugging and allows for rapid prototyping as there is no separate compilation step.
- **Dynamically Typed:** Variables do not require explicit type declarations; their types are determined at runtime based on the assigned value.
- **Multi-Paradigm Support:** Python supports various programming styles, including **ObjectOriented(OOP)**, **Procedural**,and **Functional** programmig.
- **Extensive Standard Library:** It comes with a vast collection of built-in modules for tasks like file I/O, system calls, and internet protocols.
- **Cross-Platform Compatibility:** Python code can run on major operating systems like Windows, macOS, and Linux without modification.
- **Free and Open Source:** The language is freely available on the official Python website and is supported by a massive global community.

Applications:

- **Data Science & Machine Learning:** Python is the industry standard for data analysis, visualization, and AI, supported by libraries like NumPy, Pandas, TensorFlow, and PyTorch.
- **Web Development:** Developers use frameworks like Django and Flask to build scalable backend systems for platforms like Instagram and Pinterest.
- **Automation & Scripting:** It is widely used to automate repetitive tasks, such as file management, web scraping (using BeautifulSoup), and system administration.
- **Software Testing:** Python aids in automated testing and bug tracking through tools like Pytest and Unittest.
- **Desktop GUI Applications:** Libraries such as Tkinter, PyQt, and Kivy enable the creation of cross-platform desktop software.
- **Scientific Computing:** Research organizations like NASA and CERN use Python for complex simulations and data processing with tools like SciPy.

- **FinTech:** Used for algorithmic trading, risk management, and financial modeling.

Setting up a Python environment involves installing the Python interpreter, choosing an Integrated Development Environment (IDE), and configuring virtual environments to manage project dependencies.

Setting Up the Python Environment (Python Installation, IDEs):

1. Python Installation

- **Windows**: Download the executable installer from the official Python website. During installation, it is crucial to check the box **"Add Python to PATH"** to enable running Python from the command line. Alternatively, you can install it via the Microsoft Store for an easier setup.
- **macOS**: While macOS often comes with a pre-installed version, it is recommended to install the latest version using the Homebrew package manager by running brew install python in the terminal.
- **Linux**: Most distributions include Python. If not, use the package manager for your distribution (e.g., sudo apt install python3 for Ubuntu/Debian).
- **Verification**: Confirm the installation by opening a terminal or command prompt and typing python --version (or python3 --version)

2. Popular IDEs and Editors

- While you can write code in a simple text editor, specialized IDEs offer features like debugging and code completion.
- **Visual Studio Code (VS Code)**: A lightweight, highly customizable editor. To use it for Python, download VS Code and install the **Python extension** from the Marketplace.
- **PyCharm**: A powerful, full-featured IDE specifically for Python. The **Community Edition** is free and available for download at JetBrains.
- **IDLE**: The "Integrated Development and Learning Environment" that comes bundled with the standard Python installer, ideal for beginners.
- **Jupyter Notebook**: Popular for data science and interactive programming, allowing you to run code in individual cells.

3. Setting Up Virtual Environments

Virtual environments are essential for isolating project-specific dependencies and avoiding version conflicts.

- **Create**: Navigate to your project folder in the terminal and run:
    - Windows: python -m venv env_name.
    - macOS/Linux: python3 -m venv env_name.
- **Activate**:

- Windows: env_name\Scripts\activate.
- macOS/Linux: source env_name/bin/activate.

- **Deactivate**: Simply type deactivate to exit the virtual environment.

Variables:

In Python, variables are used to store data that can be referenced and manipulated during program execution. A variable is essentially a name that is assigned to a value.

- Unlike Java and many other languages, Python variables do not require explicit declaration of type.
- The type of the variable is inferred based on the value assigned.

```
x = 5
name = "Samantha"
print(x)
print(name)
```

Output:

```
5
Samantha
```

Rules for Naming Variables

To use variables effectively, we must follow Python's naming rules:

1. Variable names can only contain letters, digits and underscores (_).
2. A variable name cannot start with a digit.
3. Variable names are case-sensitive like myVar and myvar are different.
4. Avoid using Python keywords like if, else, for as variable names.

Below listed variable names are valid:

```
age = 21
_colour = "lilac"
total_score = 90
```

Below listed variables names are invalid:

```
1name = "Error"  # Starts with a digit
class = 10       # 'class' is a reserved keyword
user-name = "Doe"  # Contains a hyphen
```

Assigning Values to Variables

Basic Assignment: Variables in Python are assigned values using the = operator.

```
x = 5

y = 3.14

z = "Hi"
```

Dynamic Typing: Python variables are dynamically typed, meaning the same variable can hold different types of values during execution.

```
x = 10

x = "Now a string"
```

Multiple Assignments

Assigning Same Value: Python allows assigning the same value to multiple variables in a single line, which can be useful for initializing variables with the same value.

```
a = b = c = 100

print(a, b, c)
```

Assigning Different Values: We can assign different values to multiple variables simultaneously, making the code concise and easier to read.

```
x, y, z = 1, 2.5, "Python"

print(x, y, z)
```

Type and Casting a Variable

- We can determine the type of a variable using the type() function. This built-in function returns the type of the object passed to it.
- Type casting refers to the process of converting the value of one data type into another. Python provides several built-in functions to facilitate casting, including int(), float() and str() among others. For example, int() converts compatible values to an integer, float() to a floating point number, and str() to a string,

Deleting a Variable

We can remove a variable from the namespace using the del keyword. This deletes the variable and frees up the memory it was using.

```
x = 10

del x

print(x)
```

DATA TYPES

In Python, data types are used to classify and categorize data items. Python is **dynamically typed**, meaning you do not need to explicitly declare a variable's type; it is determined at runtime based on the assigned value.

Basic Data Types

- **Integer (int)**: Whole numbers without decimals.
    - Example: x = 10.
- **Float (float)**: Real numbers with a decimal point.
    - Example: y = 20.5.
- **String (str)**: Sequences of characters enclosed in single ('), double ("), or triple (''') quotes.
    - Example: name = "Alice".
- **Boolean (bool)**: Represents truth values, either True or False.
    - Example: is_active = True.
- **NoneType (None)**: Represents the absence of a value.
    - Example: z = None

Collection Data Types

- **List (list)**: An ordered, changeable collection that allows duplicate members.
    - Example: fruits = ["apple", "banana", "cherry"].
- **Tuple (tuple)**: An ordered, **immutable** (unchangeable) collection.
    - Example: point = (10, 20).
- **Dictionary (dict)**: A collection of key-value pairs; it is ordered and changeable.
    - Example: user = {"name": "John", "age": 36}.
- **Set (set)**: An unordered collection of unique items with no duplicates.
    - Example: unique_ids = {101, 102, 103}.

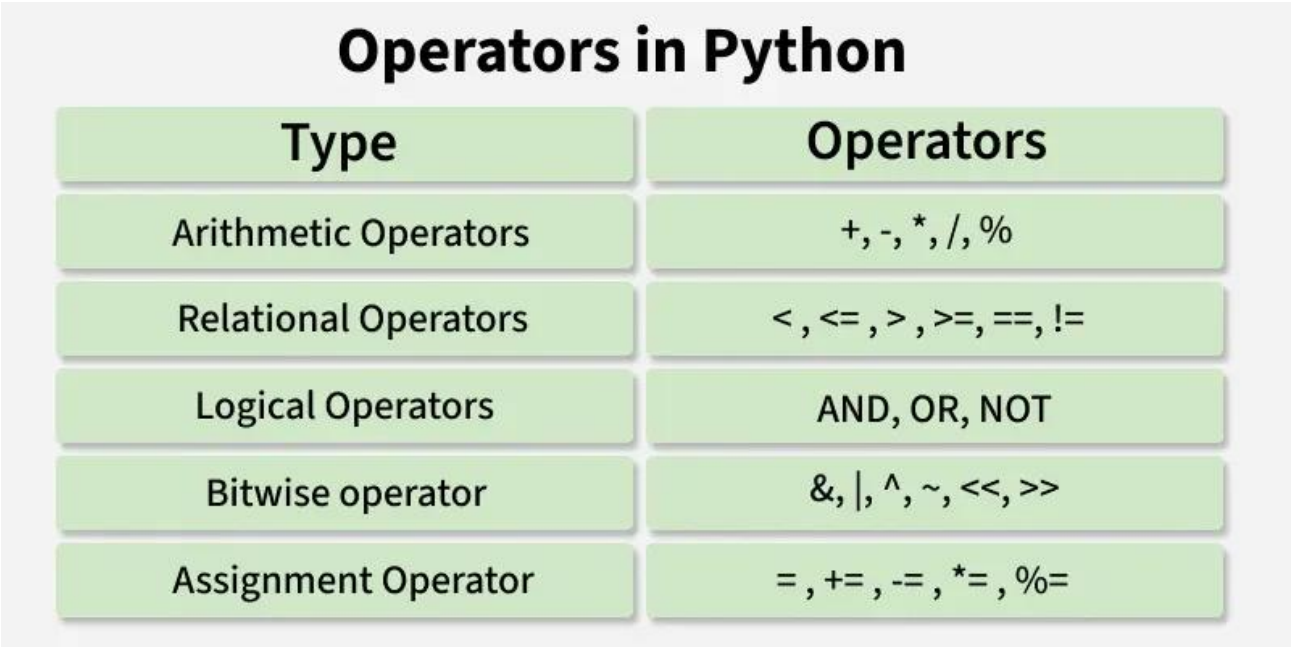
Checking and Converting Types

- **Check Type:** Use the type() function to determine a variable's data type.
    - Syntax: print(type(x))
- **Type Casting:** Manually convert one type to another using constructor functions.
    - Examples: int("10"), str(25), float(5).

In Python programming, Operators in general are used to perform operations on values and variables.

- **Operators:** Special symbols like -, + , * , /, etc.
- **Operands:** Value on which the operator is applied.

Types of Operators in Python



Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication and division.

```
# Variables

a = 15

b = 4


# Addition

print("Addition:", a + b)


# Subtraction

print("Subtraction:", a - b)


# Multiplication

print("Multiplication:", a * b)


# Division

print("Division:", a / b)
```

```
# Floor Division
print("Floor Division:", a // b)

# Modulus
print("Modulus:", a % b)

# Exponentiation
print("Exponentiation:", a ** b)
```

Output

```
Addition: 19
Subtraction: 11
Multiplication: 60
Division: 3.75
Floor Division: 3
Modulus: 3
Exponentiation: 50625
```

Comparison Operators

Comparison (or Relational) operators compares values. It either returns True or False according to the condition.

```
a = 13
b = 33

print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
print(a <= b)
```

Output

False

True

False

True

False

True

Logical Operators

Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations. It is used to combine conditional statements.

The precedence of Logical Operators in Python is as follows:

1. Logical not
2. logical and
3. logical or

Example:

```
a = True
b = False
print(a and b)
print(a or b)
print(not a)
```

Output

False

True

False

Bitwise Operators

Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

Bitwise Operators in Python are as follows:

1. Bitwise NOT
2. Bitwise Shift

3. Bitwise AND
4. Bitwise XOR
5. Bitwise OR

Example:

```
a = 10
b = 4
print(a & b)
print(a | b)
print(~a)
print(a ^ b)
print(a >> 2)
print(a << 2)
```

Output

```
0
14
-11
14
2
40
```

Assignment Operators

Assignment operators are used to assign values to the variables. This operator is used to assign the value of the right side of the expression to the left side operand.

Example:

```
a = 10
b = a
print(b)
b += a
print(b)
b -= a
print(b)
```

```
b *= a

print(b)

b <<= a

print(b)
```

Output

```
10

20

10

100

102400
```

Identity Operators

"**is**" and "**is not**" are the identity operators and both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

is True if the operands are identical
is not True if the operands are not identical

```
a = 10

b = 20

c = a


print(a is not b)

print(a is c)
```

Output

```
True

True
```

Membership Operators

"**in**" and "**not in**" are the membership operators that are used to test whether a value or variable is in a sequence.

in True if value is found in the sequence
not in True if value is not found in the sequence

```
x = 24
```

```
y = 20
list = [10, 20, 30, 40, 50]


if (x not in list):
   print("x is NOT present in given list")
else:
   print("x is present in given list")


if (y in list):
   print("y is present in given list")
else:
   print("y is NOT present in given list")
```

Output

```
x is NOT present in given list
y is present in given list
```

Ternary Operator

Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It simply allows testing a condition in a **single line** replacing the multiline if-else, making the code compact.

```
a, b = 10, 20
min = a if a < b else b
print(min)
```

Output

```
10
```

Precedence and Associativity of Operators

Operator precedence and associativity determine the priorities of the operator.

Operator Precedence

This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

```
expr = 10 + 20 * 30
print(expr)
name = "Alex"
age = 0

if name == "Alex" or name == "John" and age >= 2:
   print("Hello! Welcome.")
else:
   print("Good Bye!!")
```

Output

```
610
Hello! Welcome.
```

Operator Associativity

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

```
print(100 / 10 * 10)
print(5 - 2 + 3)
print(5 - (2 + 3))
print(2 ** 3 ** 2)
```

Output

```
100.0
6
0
512
```

Executing, and Debugging Python Scripts:

1. Executing Python Scripts

- **Command Line:** The most direct way to run a script is via the terminal or command prompt:
    - Use python script_name.py or python3 script_name.py.

    - If the script requires arguments, append them after the filename: python script_name.py arg1 arg2.

- **IDEs and Editors:**
    - **VS Code:** Click the **Play** button in the top-right corner or use the Python extension to run the file in an integrated terminal.
    - **PyCharm:** Right-click inside the editor and select **Run 'script_name'** or use the green play icon in the toolbar.
- **Interactive REPL:** For testing small snippets, start the Python interpreter by typing python in your terminal to enter a Read-Eval-Print Loop (REPL).

2. Debugging Python Scripts

Debugging allows you to pause execution, inspect variables, and step through code line-by-line.

Built-in Tools

- **pdb (Python Debugger):** A command-line debugger included in the standard library.
    - **Launch from terminal:** python -m pdb script_name.py.
    - **Hardcode a breakpoint:** Insert import pdb; pdb.set_trace() at the desired line.
    - **Modern Alternative:** In Python 3.7+, use the built-in breakpoint() function to trigger the debugger without manual imports.
- **Essential pdb Commands:**
    - n (next): Move to the next line in the current function.
    - s (step): Step into a function call.
    - c (continue): Continue execution until the next breakpoint.
    - p variable: Print the value of a variable.
    - q (quit): Exit the debugger.

Executing and debugging Python scripts can be performed through the command line, built-in modules, or sophisticated Integrated Development Environments (IDEs).

IDE Debuggers

Modern IDEs provide a visual interface for the debugging process.

- Visual Studio Code**:**
    1. Set a breakpoint by clicking to the left of a line number (red dot).
    2. Press **F5** or click the **Run and Debug** icon.

3. Use the **Variables Pane** to inspect data and the **Debug Toolbar** to control execution (Step Over, Step Into, etc.).

- PyCharm**:**
    1. Set breakpoints in the gutter (next to line numbers).
    2. Click the **Bug** icon to start the session.
    3. Features include **Inline Debugging** (variable values shown next to code) and a **Watch** window for specific expressions.

Basic Techniques

- **Print Statements:** The simplest method involves using print(variable) to track values manually.
- **Logging:** For more complex apps, use the logging module to record events with severity levels (DEBUG, INFO, ERROR).

MODULE-2

Control Structures and Functions:

In Python, conditional statements help control the flow of a program by executing different blocks of code based on whether a condition is true or false. These statements allow decision-making in code. The main types of conditional statements are:

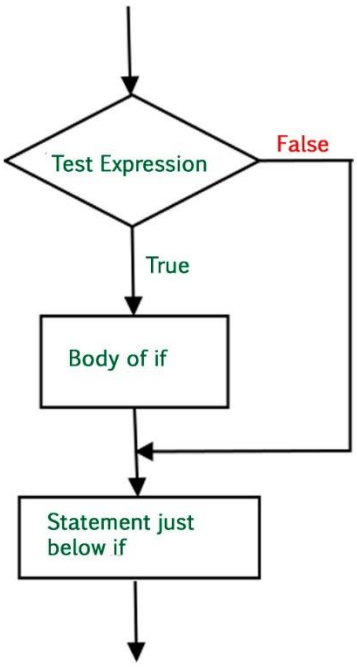
- if
- if-else
- nested if
- if-elif

if Statement

An if statement is used when a block of code needs to be executed only if a specific condition evaluates to True.

Syntax

if condition:
 # Statements to execute if condition is true



Example:

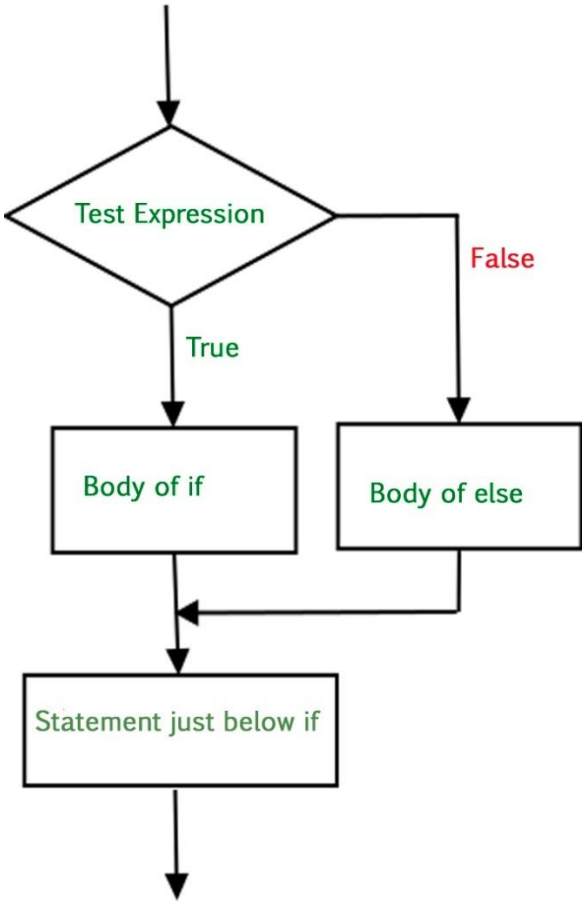
```
if 10 > 5:
    print("10 greater than 5")
print("Program ended")
```

if else Statement

In conditional if Statement the additional block of code is merged as else statement which is performed when if condition is false.

Syntax

if (condition):
 # Executes this block if condition is true
else:
 # Executes this block if condition is false



Example -1:

```
x = 3
if x == 4:
    print("Yes")
else:
    print("No")
```

Example -2:

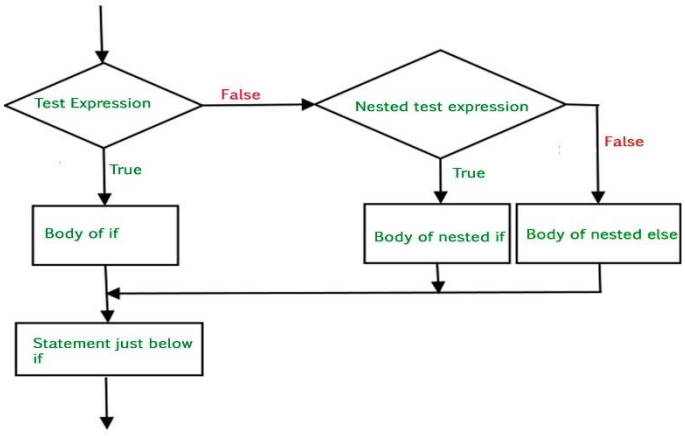
```
letter = "A"
if letter == "B":
    print("letter is B")
else:
    if letter == "C":
        print("letter is C")
    else:
        if letter == "A":
            print("letter is A")
        else:
            print("letter isn't A, B and C")
```

Nested if Statement

if statement can also be checked inside other if statement. This conditional statement is called a nested if statement. This means inner if condition will only be checked if the outer if condition is True.

Syntax

```
if (condition1):
# Executes when condition1 is true
if (condition2):
# Executes when condition2 is true
```



Example

```
a = 10
if a > 5:
    print("Bigger than 5")
    if a <= 15:
```

```
    print("Between 5 and 15")
```

if-elif Statement

The if-elif statement is a shortcut for chaining multiple if-else conditions. While using if-elif statement at the end else block is added which is performed if none of the above if-elif statement is true.

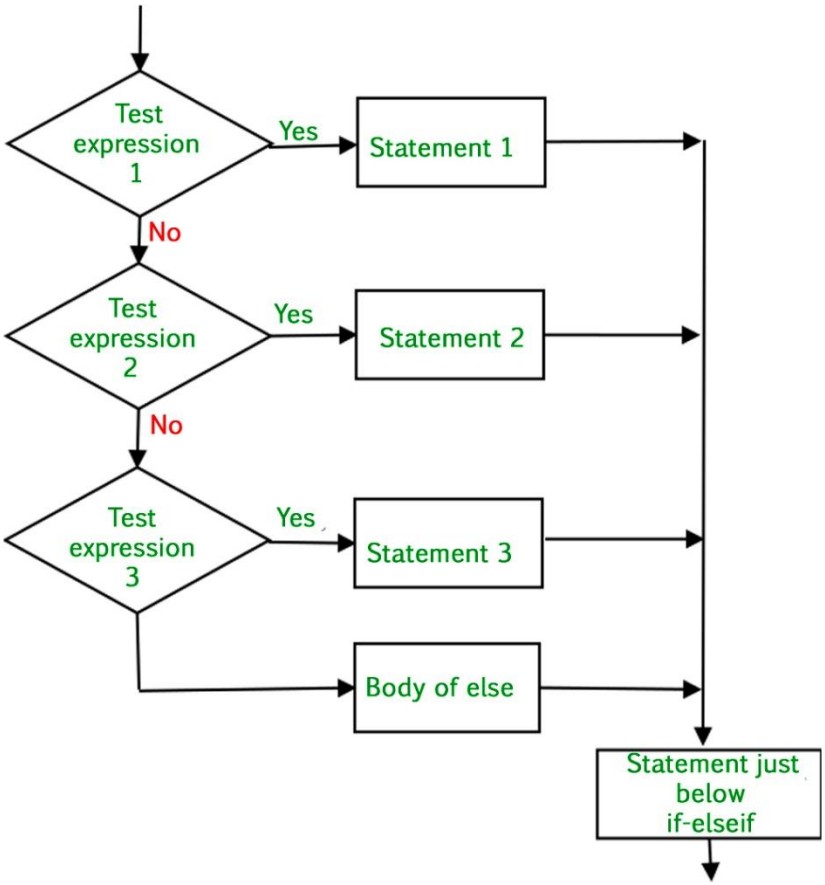
Syntax

if (condition):
statement
elif (condition):
statement
else:
statement

Example:

```
letter = "A"

if letter == "B":
   print("letter is B")
elif letter == "C":
   print("letter is C")
elif letter == "A":
   print("letter is A")
```

else: print("letter isn't A, B or C")



Using elif Inside Nested if Statements

Using elif within nested if statements in Python allows for more complex decision structures within a branch of another decision.

```
x = 10

y = 5

if x > 5:

    if y > 5:

        print("x is greater than 5")

    elif y==5:

         print("x is greater than 5 and y is 5")

    else:

        print("x is greater than 5 and y is less than 5")
```

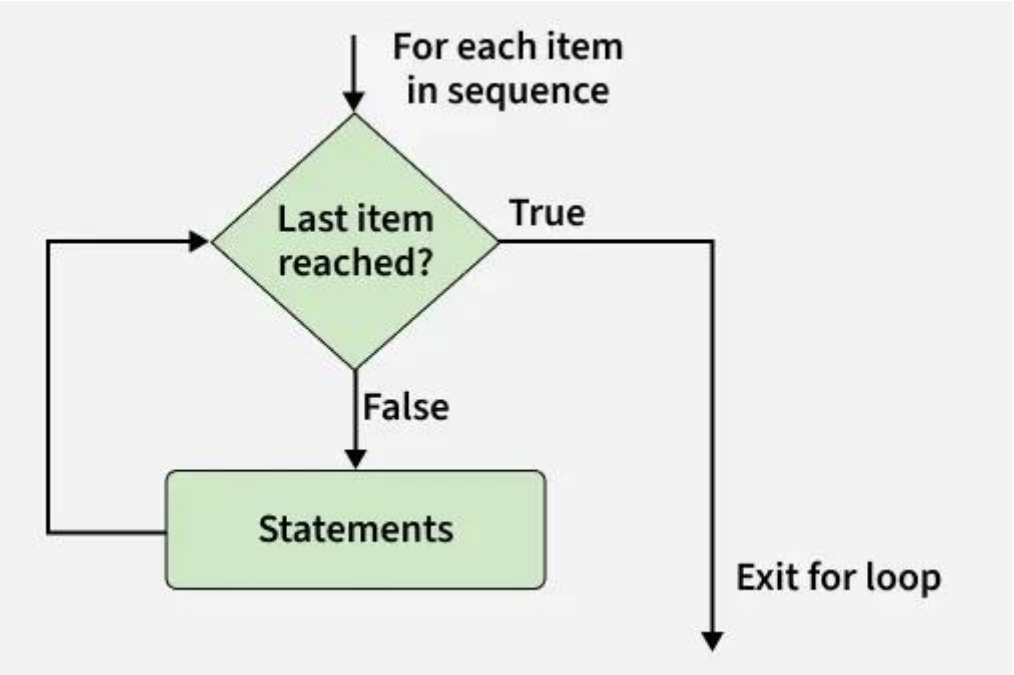
<u>LOOPS:</u>

Loops in Python are used to repeat actions efficiently. The main types are For loops (counting through items) and While loops (based on conditions).

For Loop

For loops is used to iterate over a sequence such as a list, tuple, string or range. It allow to execute a block of code repeatedly, once for each item in the sequence.

```
n = 4

for i in range(0, n):
    print(i)
```



Example:

```
li = ["geeks", "for", "geeks"]

for x in li:
    print(x)


tup = ("geeks", "for", "geeks")

for x in tup:
    print(x)


s = "abc"

for x in s:
    print(x)


d = dict({'x':123, 'y':354})
```

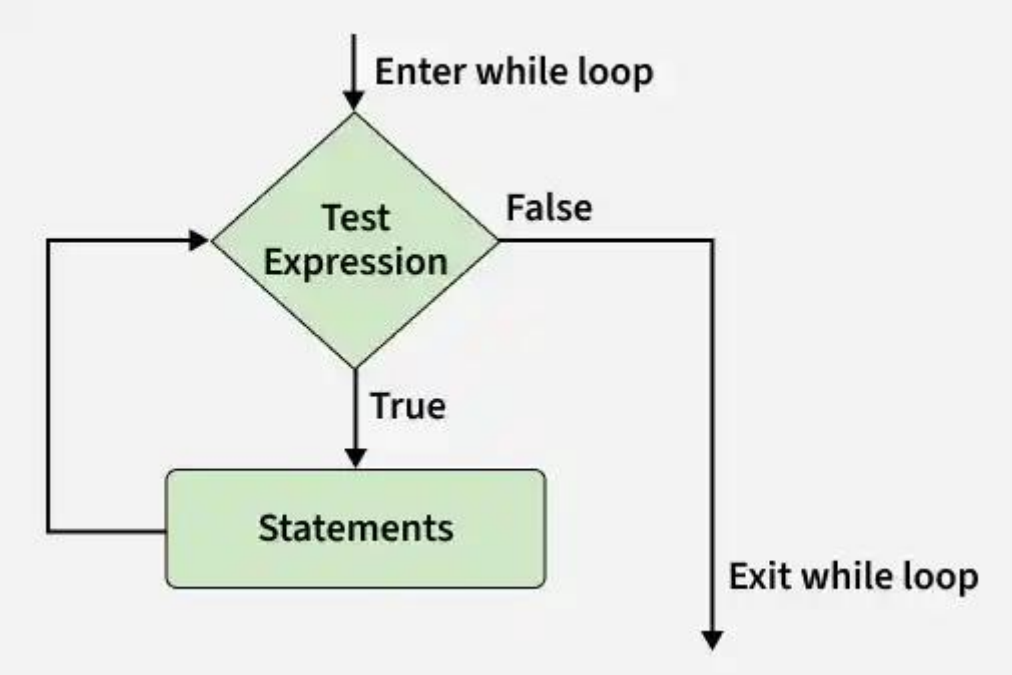
```
for x in d:
    print("%s  %d" % (x, d[x]))


set1 = {10, 30, 20}
for x in set1:
    print(x),
```

While Loop

while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.

Example:

```
cnt = 0
while (cnt < 3):
    cnt = cnt + 1
    print("Hello Geek")
```



Infinite While Loop

To execute a block of code infinite number of times we can use the while loop.

Code given below uses a 'while' loop with the condition "**True**", which means that the loop will run infinitely until we break out of it using "**break"** keyword or some other logic.

Example:

```
while (True):
    print("Hello Geek")
```

Nested Loops

Python programming language allows to use one loop inside another loop which is called nested loop.

Example:

```
from __future__ import print_function
for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

Loop Control Statements

Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Continue Statement

The continue statement in Python returns the control to the beginning of the loop.

Example:

```
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        continue
    print('Current Letter :', letter)
```

Break Statement

The break statement in Python brings control out of the loop.

Example:

```
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        break


print('Current Letter :', letter)
```

Pass Statement

We use pass statement in Python to write empty loops. Pass is also used for empty control statements, functions and classes.

Example:

```
for letter in 'geeksforgeeks':
    pass
print('Last Letter :', letter)
```

Functions: Defining, Calling, and Scope of Variables:

In Python, functions are reusable blocks of code designed to perform specific tasks, while variable scope determines where those variables can be accessed within a program.

1. Defining and Calling Functions

Functions help organize code into modular, manageable chunks, promoting reusability.

- **Defining a Function:** Use the def keyword followed by the function name and parentheses.
    - **Parameters:** Optional inputs specified in the parentheses to provide data for the function's code.
    - **Indentation:** All code within the function must be indented to identify it as part of the function block.
    - **Return Statement:** Optional; used to send a value back to the caller. If omitted, the function returns None by default.
    - A function is defined using the **def** keyword, followed by a unique name, parentheses () for optional parameters, and a colon :. The code inside must be indented.
    - **Syntax:**

      ```
      def function_name(parameter):
          # Function body (indented)
          print(f"Hello, {parameter}!")
      ```

- **Calling a Function:** Invoke a function by writing its name followed by parentheses containing any required arguments.
    - **Arguments:** The actual values passed to the function when it is called.

- **Types of Arguments:** Python supports positional, keyword, default, and arbitrary arguments (*args, **kwargs).
- Defining a function does not execute it; you must **call** it by using its name followed by parentheses containing any required arguments.

  Example:

```
def greet(name):
    print(f"Hello, {name}!")
        greet("Alice")  # Function call with "Alice" as the argument
```

2. Scope of Variables

- Scope defines the visibility and lifetime of a variable. Python follows the **LEGB rule** for resolving variable names in this specific order:
  1. **Local (L):** Variables defined within a function (including parameters) are local to that function. They cannot be accessed from outside.
  2. **Enclosing (E):** In nested functions, the inner function can access variables defined in the outer (enclosing) function.
  3. **Global (G):** Variables defined at the top level of a script or module are global and accessible throughout the code.
  4. **Built-in (B):** Predefined names in Python, such as `print()` and `len()`, are always available.
  5. Scope determines where a variable can be accessed or modified within your program.

  - **Local Scope:** Variables defined *inside* a function are local to that function. They are created when the function is called and destroyed when it exits. They cannot be accessed from outside.
  - **Global Scope:** Variables defined *outside* all functions are global. They are accessible throughout the entire script, including inside functions.
  - **Modifying Global Variables:** To change a global variable's value inside a function, you must use the global keyword.

    Example:

```
total = 10  # Global variable

def update_value(n):
    local_val = 5  # Local variable
    global total
```

```
    total = total + n + local_val  # Modifying global variable
    print("Inside function:", total)

update_value(5)
print("Outside function:", total)
# print(local_val)  # This would cause a NameError
```

3. Modifying Scopes

- By default, functions can read but not modify variables in outer scopes without specific keywords:
    1. **global Keyword:** Used inside a function to indicate that a variable belongs to the global scope, allowing it to be modified.
    2. **nonlocal Keyword:** Used in nested functions to modify a variable from the enclosing (outer) function's scope.

Lambda Functions

Lambda functions are small anonymous functions, meaning they do not have a defined name. These are small, short-lived functions used to pass simple logic to another function.

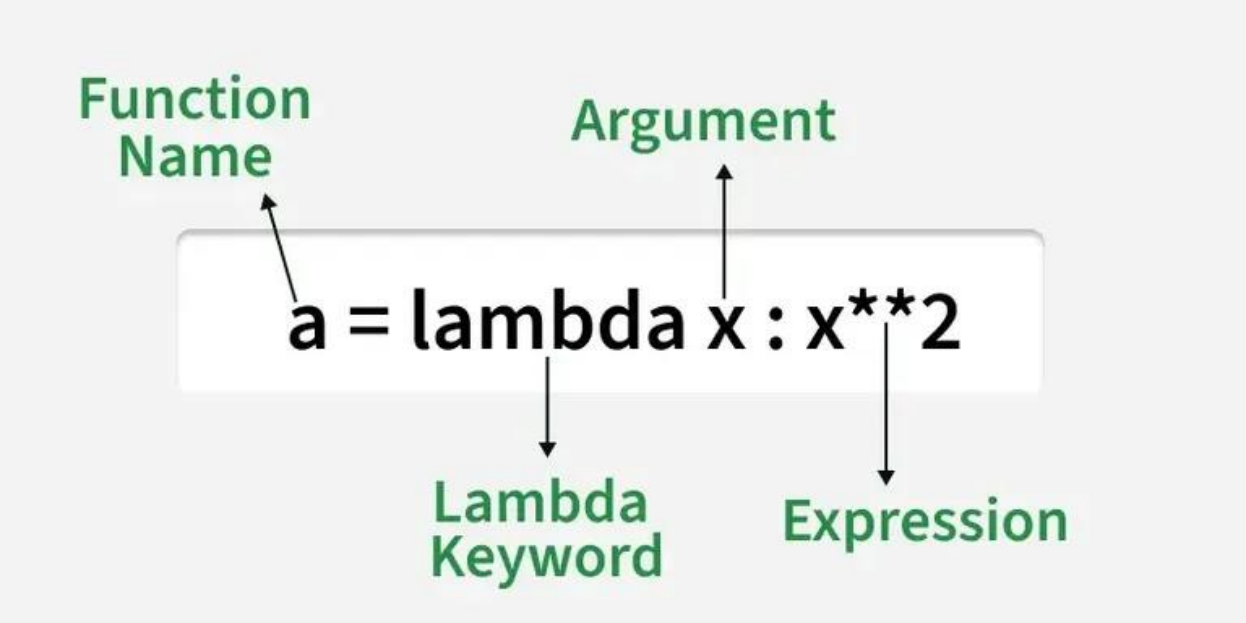
- Contain only one expression.
- Result of that expression is returned automatically (no return keyword needed).
- In this example, a lambda function is defined to convert a string to its upper case using upper().

Example:

```
a = 'GeeksforGeeks'

upper = lambda x: x.upper()

print(upper(a))
```

Syntax:

In Python, lambda functions are created using the lambda keyword. Below is the syntax:



- **Function name (a):** Stores the lambda function so it can be reused later.
- **Lambda keyword (lambda):** Defines an anonymous (inline) function in Python.
- **Argument (x):** The input value passed to the lambda function.
- **Expression (x**2):** The operation performed on the argument and returned as the result.

Key Characteristics

- **Anonymous**: They are not bound to an identifier unless manually assigned to a variable.
- **Single Expression**: They cannot contain multiple statements, loops, or complex blocks like try-except.
- **Implicit Return**: The result of the expression is returned automatically upon execution.
- **Inline Definition**: They are often defined at the exact point where they are needed, such as inside another function call.

Common Use Cases

Lambda functions are most powerful when used as arguments for **higher-order functions** that expect a function as an input:

- **map()**: Applies a transformation to every item in an iterable.
  - *Example*: list(map(lambda x: x * 2, [1, 2, 3])) results in [2, 4, 6].
- **filter()**: Filters items based on a condition.
  - *Example*: list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4])) results in [2, 4].
- **sorted()**: Customizes sorting logic using the key parameter.

- *Example*: sorted(data, key=lambda x: x[1]) sorts a list of tuples by their second element.

- **reduce()**: Cumulatively applies an operation to reduce an iterable to a single value (requires importing from functools).

Functions Recursion:

Function recursion in Python is a technique where a function calls itself to solve a problem by breaking it down into smaller, simpler subproblems. It is particularly effective for tasks with a naturally repetitive or hierarchical structure, such as mathematical calculations and tree traversals.

Core Components

Every well-defined recursive function must have two essential parts to function correctly and avoid infinite execution:

- **Base Case:** A condition that allows the function to return a result directly without further recursion.
- **Recursive Case:** The part where the function calls itself with a modified, typically smaller, argument that moves closer to the base case.
- **Basic Example: Factorial**

```
def factorial(n):
    if n <= 1:  # Base Case
        return 1
    else:       # Recursive Case
        return n * factorial(n - 1)
```

Key Types of Recursion

1. **Direct Recursion:** The function explicitly calls itself within its own body.
2. **Indirect (Mutual) Recursion:** Function A calls Function B, which in turn calls Function A.
3. **Tail Recursion:** The recursive call is the final operation in the function. Note that while some languages optimize this to save memory, Python does not perform automatic tail-call optimization.

MODULE-3

Data Structures in Python:

In Python, lists, tuples, and sets are built-in data structures used to store collections of data. Each has unique characteristics regarding order, mutability, and how they handle duplicate values.

1. **Lists**
   A List is an ordered and changeable (mutable) collection that allows duplicate members. It is defined using square brackets [].

- **Characteristics:** Ordered, mutable, allows duplicates.
- **Best Use Case:** Maintaining a sequence of items that may need frequent modification, such as a shopping cart.
- **Example:**

```
# Creating and modifying a list
fruits = ["apple", "banana", "cherry", "apple"]
fruits.append("orange")    # Adds to the end

fruits[1] = "blueberry"    # Changes the second item
print(fruits)          # Output: ['apple', 'blueberry', 'cherry', 'apple', 'orange']
```

2. **Tuples**
   A **Tuple** is an ordered and unchangeable (immutable) collection that allows duplicate members. It is defined using round brackets ().

- **Characteristics:** Ordered, immutable, allows duplicates.
- **Best Use Case:** Storing fixed data that should not be changed, such as geographic coordinates or database records.

Example:

```
# Creating a tuple

coordinates = (10.0, 20.0)

# coordinates[0] = 15.0   # This would raise a TypeError

print(coordinates[0])     # Output: 10.0
```

3. **Sets**
   A Set is an unordered collection that is unindexed and contains only unique elements. It is defined using curly brackets {}.

- **Characteristics:** Unordered, mutable (can add/remove items), no duplicates.
- **Best Use Case:** Removing duplicates from a collection or performing mathematical operations like unions and intersections.

Example:

```
# Creating a set and adding items
colors = {"red", "green", "blue"}
```

```
colors.add("red")       # Adding a duplicate does nothing
colors.add("yellow")
print(colors)           # Output: {'blue', 'red', 'green', 'yellow'} (Order varies)
```

Dictionaries

In Python, a dictionary is a mutable collection of key-value pairs where keys must be unique and immutable.

Operations

1. Creating a Dictionary

You can create a dictionary using curly braces {} or the dict() constructor.

Example:

```
# Literal syntax
user = {"name": "Alice", "age": 25}

# dict() constructor
user = dict(name="Alice", age=25)
```

2. Accessing Items

Access values using square brackets [] or the safer get() method, which returns None instead of a KeyError if the key is missing.

Example:

```
print(user["name"])        # Output: Alice
print(user.get("email"))   # Output: None
print(user.get("email", "N/A")) # Output: N/A (with default)
```

3. Adding and Updating Items

Assigning a value to a key will either add a new pair or update an existing one. Use the update() method to merge multiple pairs at once.

Example:

```
user["city"] = "New York"   # Adding new key
user["age"] = 26            # Updating existing key
user.update({"role": "Admin", "status": "Active"})
```

4. Removing Items

- **pop(key)**: Removes the specified key and returns its value.
- **popitem()**: Removes and returns the last inserted key-value pair.
- **del**: Deletes a specific item or the entire dictionary.
- **clear()**: Removes all items, leaving an empty dictionary.

5. Iteration and View Objects

- Dictionaries provide "view objects" that stay updated with the original dictionary.
- **keys()**: Returns all keys.

- **values()**: Returns all values.
- **items()**: Returns key-value pairs as tuples.

Example:

```
for key, value in user.items():
    print(f"{key}: {value}")
```

6. Membership and Length

Use in to check if a key exists and len() to get the number of items.

Example:

```
print("name" in user) # Output: True
print(len(user))      # Output: 5
```

7. Modern Operators :

The union operators provide a clean way to merge dictionaries.

Example:

```
d1 = {"a": 1}
d2 = {"b": 2}
merged = d1 | d2    # New dictionary: {'a': 1, 'b': 2}
d1 |= d2            # Updates d1 in-place
```

Applications

1. Database Record Representation

Dictionaries are commonly used to represent single database records or objects. Each key corresponds to a field name, and the value is the data for that field.

Example:

```
user = {
    "id": 101,
    "name": "Alice",
    "email": "alice@example.com",
    "role": "Admin"
}
print(user["name"])  # Output: Alice
```

2. Common Use Case: Frequency Tables

A primary application is counting the occurrences of items in a dataset. Keys represent the items, and values represent their counts.

Example:

```
text = "apple"
frequency = {}
for char in text:
    frequency[char] = frequency.get(char, 0) + 1
```

```
print(frequency)  # Output: {'a': 1, 'p': 2, 'l': 1, 'e': 1}
```

List Comprehensions and Dictionary Comprehensions Working with Strings: Methods and Manipulation:

Python comprehensions provide a concise way to create new collections from existing iterables by applying transformations or filters in a single line of code.

List Comprehensions with Strings:

List comprehensions are used to create a new list by iterating over an existing sequence (like a list of strings or a single string) and applying an expression to each item.

- **Syntax:** [expression for item in iterable if condition].
- **Example (Filtering & Modification):** Create a list of uppercase names for fruits that contain the letter "a".

Example:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
# Uses .upper() and filters for "a"
new_list = [f.upper() for f in fruits if "a" in f]
print(new_list) # Output: ['APPLE', 'BANANA', 'MANGO']
```

Dictionary Comprehensions with Strings:

Dictionary comprehensions generate key-value pairs dynamically. They are particularly useful for mapping strings to their properties, such as length or character counts.

- **Syntax:** {key_expression: value_expression for item in iterable if condition}.
- **Example (Mapping Lengths):** Create a dictionary where the fruit name is the key and its length is the value.

Example:

```
fruits = ["apple", "banana", "cherry"]
# Maps fruit name to its len()
fruit_lengths = {f: len(f) for f in fruits}
print(fruit_lengths) # Output: {'apple': 5, 'banana': 6, 'cherry': 6}
```

String Methods and Manipulation

Strings in Python are immutable, meaning manipulation always results in a new string. Common methods used within comprehensions include:

- **Case Transformation:** Use .upper() for all caps, .lower() for lowercase, or .title() to capitalize each word.
- **Cleaning:** .strip() removes leading and trailing whitespace.
- **Searching & Logic:** .isalpha() checks if a string contains only letters; .startswith() checks for a prefix.
- **Example (Comprehensive Manipulation):** Clean extra spaces from names and keep only those that are alphabetic.

Example:

```
raw_names = [" alice ", "123bob", "  charlie  ", "diana!"]
# strip() removes spaces, isalpha() filters out names with numbers/symbols
clean_names = [n.strip().capitalize() for n in raw_names if n.strip().isalpha()]
print(clean_names) # Output: ['Alice', 'Charlie']
```

Introduction to Python's Collections Module:

The Python collections module provides specialized container data types that serve as alternatives to built-in types like list, dict, set, and tuple. These containers are optimized for specific programming tasks, improving both performance and code readability.

Core Specialized Containers

The following are the most frequently used classes in the collections module:

- **Counter**: A counter is a sub-class of the dictionary. It is used to keep the count of the elements in an iterable in the form of an unordered dictionary where the key represents element in the iterable and value represents the count of that element in the iterable.

Syntax:

class collections.Counter([iterable-or-mapping])

Example:

```
from collections import Counter

counts = Counter(['apple', 'orange', 'apple', 'banana', 'apple'])
print(counts)  # Output: Counter({'apple': 3, 'orange': 1, 'banana': 1})
print(counts.most_common(1))  # Output: [('apple', 3)]
```

- **defaultdict** A DefaultDict is also a sub-class to dictionary. It is used to provide some default values for the key that does not exist and never raises a KeyError.

Syntax:

class collections.defaultdict(default_factory)

default_factory is a function that provides the default value for the dictionary created. If this parameter is absent then the KeyError is raised.

Example:

```
from collections import defaultdict
# Automatically initializes a new key with an empty list
d = defaultdict(list)
d['fruits'].append('apple')
print(d['fruits'])  # Output: ['apple']
```

- **namedtuple**: A NamedTuple is like a regular tuple but with named fields, making data more readable and accessible. Instead of using indexes, you can access elements by name (e.g., student.fname), which improves code clarity and ease of use.

- **Syntax:**
  class collections.namedtuple(typename, field_names)
  Example:
  ```
  from collections import namedtuple
  Point = namedtuple('Point', ['x', 'y'])
  p = Point(10, 20)
  print(p.x, p.y)  # Output: 10 20
  ```
- **deque**: Short for "double-ended queue," this list-like container is optimized for fast appends and pops from both the left and right ends.
  Example:
  ```
  from collections import deque
  d = deque([1, 2, 3])
  d.appendleft(0)
  d.pop()
  print(d)  # Output: deque([0, 1, 2])
  ```