NAME: - Dhvanit

Subject: - Python

Introduction to Python – Theory

1. Introduction to Python and its Features

Python is a **high-level, interpreted, general-purpose programming language** known for its simplicity and readability.
It was designed with an emphasis on code readability, using indentation instead of braces or keywords for block delimiters.

Key Features:

- **Simple & Easy to Learn** – The syntax is close to English, making it beginner-friendly.
- **High-Level Language** – Abstracts low-level details like memory management.
- **Interpreted** – No need to compile; code runs line-by-line via the Python interpreter.
- **Portable** – Works on multiple platforms (Windows, Mac, Linux) without modification.
- **Object-Oriented** – Supports object-oriented programming with classes and objects.
- **Extensive Libraries** – Includes a vast standard library and supports third-party modules.
- **Dynamic Typing** – No need to declare variable types explicitly.
- **Open Source** – Freely available to use and distribute.

2. History and Evolution of Python

- **Late 1980s** – Python was conceived by **Guido van Rossum** at the Centrum Wiskunde & Informatica (CWI) in the Netherlands.
- **1991** – Python 0.9.0 was released (included exception handling, functions, and core data types).
- **2000** – Python 2.0 introduced features like list comprehensions and garbage collection.
- **2008** – Python 3.0 was released with improvements and removal of older legacy features (not backward-compatible with Python 2).
- **Present** – Python is maintained by the Python Software Foundation (PSF) and is one of the most popular languages in data science, AI, web development, automation, and more.

3. Advantages of Using Python Over Other Languages

- **Readability** – Clean syntax makes it easy to write and understand.
- **Versatility** – Supports multiple programming paradigms (procedural, OOP, functional).
- **Large Community** – Active community and rich documentation.
- **Cross-Platform** – Code runs on different OS without modification.
- **Vast Libraries** – Thousands of pre-built modules for tasks like web scraping, machine learning, and automation.
- **Rapid Development** – Faster coding and prototyping compared to languages like C++ or Java.
- **Integration** – Easily integrates with C, C++, Java, and other languages.

4. Installing Python and Setting Up the Development Environment

You can install Python in multiple ways:

Option 1: Official Python Installation

1. Visit python.org
2. Download the latest stable version.
3. Install and check installation via:
4. python --version

Option 2: Anaconda (Recommended for Data Science)

1. Download Anaconda from anaconda.com.
2. Install and use **Spyder** or **Jupyter Notebook**.

Option 3: IDEs and Editors

- **PyCharm** – Full-featured professional IDE.
- **VS Code** – Lightweight, with Python extension.
- **IDLE** – Comes pre-installed with Python.

5. Writing and Executing Your First Python Program

Example Program:

```
# This is a simple Python program

print("Hello, World!")
```

Steps to Run:

1. Open your editor or IDE.

2. Save the file with .py extension (e.g., hello.py).
3. Run in terminal/command prompt:

python hello.py

Hello, World!

1. Understanding Python's PEP 8 Guidelines

PEP 8 stands for **Python Enhancement Proposal 8**, which is the official style guide for writing Python code.
Its purpose is to make code **consistent, readable, and maintainable** across different projects.

Key Points from PEP 8:

- **Indentation** – Use 4 spaces per indentation level (never use tabs).
- **Maximum Line Length** – Keep lines under **79 characters**.
- **Blank Lines** – Use blank lines to separate functions, classes, and code sections.
- **Imports** – Keep imports at the top of the file and group them logically.
- **Spaces Around Operators** – Use spaces around =, +, -, etc. (e.g., x = y + z).
- **Naming Conventions** – Follow consistent naming styles (explained below).

2. Indentation, Comments, and Naming Conventions in Python

Indentation

Python uses **indentation to define code blocks** (instead of {} braces like in C/C++ or Java).

```
if True:
    print("Indented with 4 spaces")
```

If indentation is wrong, Python will raise an **IndentationError**.

Comments

- **Single-line comment** – starts with #

```
# This is a comment
```

- **Multi-line comment** – use triple quotes ''' or """ (though technically these are string literals)

```
"""
This is a
multi-line comment
"""
```

Naming Conventions

- **Variables & functions** – lowercase with underscores (my_variable, calculate_area)
- **Constants** – uppercase with underscores (PI = 3.14)
- **Classes** – PascalCase (StudentDetails)
- **Private variables** – prefix with _ (_hidden_value)

3. Writing Readable and Maintainable Code

To make code easy for others (and yourself) to read later:

- Use **meaningful variable and function names** (calculate_salary() is better than cs()).
- Follow **consistent formatting** (indentation, spacing, naming).
- Add **comments** to explain complex logic, but avoid unnecessary comments for obvious code.
- Break long functions into smaller functions for **modularity**.
- Avoid deep nesting; use **early returns** to simplify.
- Write **docstrings** for functions and classes to explain usage:

```
def add(a, b):
    """Return the sum of two numbers."""
    return a + b
```

1. Understanding Data Types

Python has several built-in data types, classified as **mutable** (changeable) or **immutable** (unchangeable):

Basic Data Types

- **Integers (int)** – Whole numbers, positive or negative.
- age = 25
- **Floating Point (float)** – Numbers with decimal points.
- pi = 3.14
- **Strings (str)** – Sequence of characters enclosed in quotes.
- name = "Python"

Collection Data Types

- **List (list)** – Ordered, mutable sequence of items.
- fruits = ["apple", "banana", "cherry"]
- **Tuple (tuple)** – Ordered, immutable sequence of items.

- coordinates = (10, 20)
- **Dictionary (dict)** – Key-value pairs, unordered, mutable.
- student = {"name": "Jeel", "age": 21}
- **Set (set)** – Unordered collection of unique items.
- unique_nums = {1, 2, 3}

2. Python Variables and Memory Allocation

- **Variable** – A name that refers to a value stored in memory.
- Python variables **do not need explicit type declaration**; they are assigned when a value is given.
- x = 5 # int
- y = "Hi" # str
- **Dynamic Typing** – Variable type can change during runtime.
- x = 10
- x = "Python" # Now x is a string
- **Memory Allocation** – Python uses a system called **reference counting and garbage collection** to manage memory.
  When you assign a value to a variable, Python stores the value in memory and the variable name points (references) to it.

3. Python Operators

Arithmetic Operators

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- // (floor division)
- % (modulus)
- ** (exponentiation)

```
a, b = 5, 2

print(a + b)  # 7

print(a ** b) # 25
```

Comparison Operators (Return True or False)

- == (equal to)
- != (not equal to)
- > (greater than)
- < (less than)
- >= (greater or equal)
- <= (less or equal)

Logical Operators

- and – True if both conditions are true
- or – True if at least one condition is true
- not – Reverses the logical state

Bitwise Operators (Work on binary representations)

- & (AND)
- | (OR)
- ^ (XOR)
- ~ (NOT)
- << (left shift)
- >> (right shift)

1. Introduction to Conditional Statements (if, else, elif)

Conditional statements are used to **make decisions in code**.
They allow a program to execute different blocks of code depending on whether a condition is **True** or **False**.

Basic Syntax

```
if condition:
    # code block executed if condition is True
elif another_condition:
    # code block executed if the first condition is False but this one is True
else:
    # code block executed if all conditions are False
```

Example

```
age = 18
```

```
if age > 18:
    print("Adult")
elif age == 18:
    print("Just turned adult")
else:
    print("Minor")
```

Output:

Just turned adult

Notes:

- Conditions must evaluate to **True** or **False**.
- You can have multiple elif blocks but only one else block.

2. Nested if-else Conditions

Nested conditional statements mean having an if statement **inside another if statement**.

Example

```
age = 20
citizen = True

if age >= 18:
    if citizen:
        print("Eligible to vote")
    else:
        print("Not a citizen, cannot vote")
else:
    print("Underage, cannot vote")
```

Output:

Eligible to vote

When to Use Nested if-else:

- When you need to check **multiple related conditions**.
- Useful for decision-making that depends on more than one factor.

Looping (For, While) – Theory

1. Introduction to for and while Loops

Loops allow you to **execute a block of code repeatedly** until a certain condition is met.

for Loop

- Used to iterate over a sequence (like a list, tuple, string, or range of numbers).
- Automatically stops when the sequence ends.

Example:

```
for i in range(5):
    print(i)
```

Output:

0

1

2

3

4

while Loop

- Repeats a block of code **as long as the condition is True**.
- Be careful to avoid infinite loops by updating variables inside the loop.

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Output:

0

1

2

3

4

2. How Loops Work in Python

- **Initialization** – Set a starting value (e.g., i = 0).
- **Condition Check** – Before each iteration, Python checks if the condition is True.
- **Execution** – If True, run the loop body.
- **Update** – Change variables to eventually stop the loop.
- Loops stop when the condition becomes False or the sequence is exhausted.

Control Statements in Loops:

- break – Stops the loop immediately.
- continue – Skips the current iteration and goes to the next.
- pass – Does nothing (used as a placeholder).

3. Using Loops with Collections

Loops are powerful when working with collections like **lists, tuples, sets, and dictionaries**.

Looping through a list

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Looping through a tuple

```
numbers = (1, 2, 3)

for num in numbers:
    print(num)
```

Looping through a dictionary

```
student = {"name": "Jeel", "age": 21}

for key, value in student.items():
    print(key, ":", value)
```

1. Understanding How Generators Work in Python

- **Generators** are a special type of function in Python that **yield** values one at a time instead of returning them all at once.
- They are **memory-efficient** because they do not store all values in memory; they generate them **on the fly**.
- A generator is created using a function with the yield statement instead of return.

Example:

```
def my_generator():
    yield 1
    yield 2
    yield 3


for value in my_generator():
    print(value)
```

Output:

1

2

3

How It Works:

- When called, the generator function returns a **generator object**.
- Each call to next() resumes execution until the next yield statement.

2. Difference Between yield and return

Feature	**return**	**yield**
Stops function execution immediately	✅	❌ (pauses execution)
Returns a single value	✅	❌ (can produce multiple values over time)
Memory usage	Can be high if returning large data	Very low (values generated on demand)
Use case	Return final result	Stream values lazily

Example using yield:

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1
```

```
for num in count_up_to(5):
    print(num)
```

3. Understanding Iterators and Creating Custom Iterators

- An **iterator** is an object that implements **two methods**:
  - __iter__() → Returns the iterator object itself.
  - __next__() → Returns the next value, raises StopIteration when no more items.

Example: Creating a custom iterator

```
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

for num in Counter(1, 3):
    print(num)
```

Output:

1

2

3

Key Difference:

- **Iterator**: Any object that can be looped over using iter() and next().

- **Generator**: A special type of iterator created using a function with yield.

1. Defining and Calling Functions in Python

A **function** is a block of reusable code that performs a specific task.

Syntax:

```
def function_name(parameters):
    """Optional docstring explaining the function."""
    # code block
    return value  # optional
```

Example:

```
def greet(name):
    return f"Hello, {name}!"


print(greet("Jeel"))
```

Output:

Hello, Jeel!

Calling a function – use the function name followed by parentheses () and pass arguments if required.

2. Function Arguments

Python supports different types of function arguments:

Positional Arguments

Values are assigned to parameters in the same order they are passed.

```
def add(a, b):
    return a + b


print(add(2, 3))  # 5
```

Keyword Arguments

Specify the parameter name when calling the function.

```
def greet(name, message):
    print(f"{message}, {name}")
```

```
greet(message="Good Morning", name="Jeel")
```

Default Arguments

Provide a default value if no argument is passed.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")


greet()  # Hello, Guest!
```

Variable-Length Arguments

- *args – Accepts multiple positional arguments as a tuple.
- **kwargs – Accepts multiple keyword arguments as a dictionary.

3. Scope of Variables in Python

The **scope** determines where a variable is accessible.

- **Local Scope** – Defined inside a function, accessible only there.
- **Global Scope** – Defined outside all functions, accessible everywhere.
- **Enclosing Scope** – Variables in nested functions.
- **Built-in Scope** – Python's reserved names.

Example:

```
x = 10  # global


def func():
    y = 5  # local
    print(x, y)


func()
print(x)  # works
# print(y)  # error: y is local
```

4. Built-in Methods for Strings, Lists, etc.

Python provides many built-in methods for different data types.

String Methods

```
text = "python"
print(text.upper())   # PYTHON
print(text.capitalize()) # Python
print(text.replace("p", "j")) # jython
```

List Methods

```
fruits = ["apple", "banana"]
fruits.append("cherry")  # add element
fruits.remove("banana")  # remove element
print(fruits)  # ['apple', 'cherry']
```

Dictionary Methods

```
student = {"name": "Jeel", "age": 21}
print(student.keys())   # dict_keys(['name', 'age'])
print(student.get("age"))  # 21
```

Control Statements (Break, Continue, Pass) – Theory

Control statements change the normal flow of a loop in Python.
They are mainly used inside for and while loops to alter execution.

1. break Statement

- **Purpose:** Immediately exits the loop, even if the loop condition is still True.
- Useful when you want to stop the loop based on a certain condition.

Example:

```
for num in range(1, 10):
   if num == 5:
      break
   print(num)
```

Output:

1

2

3

4

2. continue Statement

- **Purpose:** Skips the rest of the code in the current iteration and moves to the next iteration.
- Useful when you want to **skip certain cases** without stopping the loop.

Example:

```
for num in range(1, 6):
    if num == 3:
        continue
    print(num)
```

Output:

1

2

4

5

3. pass Statement

- **Purpose:** Does nothing — it's a placeholder when a statement is syntactically required but you don't want any action.
- Useful when writing code stubs for future implementation.

Example:

```
for num in range(1, 4):
    if num == 2:
        pass  # do nothing
    print(num)
```

Output:

1

2

3

String Manipulation – Theory

Strings in Python are sequences of characters enclosed within single ('), double ("), or triple quotes (''' or """).
They are **immutable**, meaning once created, their contents cannot be changed — but you can create new strings based on existing ones.

1. Accessing and Manipulating Strings

- **Indexing:** Access individual characters using square brackets.
- s = "Python"
- print(s[0]) # P (first character)
- print(s[-1]) # n (last character)

2. Basic Operations

(a) Concatenation

Joining strings using the + operator:

first = "Hello"

second = "World"

result = first + " " + second

print(result) # Hello World

(b) Repetition

Repeating a string using *:

print("Hi! " * 3) # Hi! Hi! Hi!

(c) String Methods

Commonly used built-in string methods:

text = " Python Programming "

print(text.upper()) # ' PYTHON PROGRAMMING '

print(text.lower()) # ' python programming '

print(text.strip()) # 'Python Programming'

print(text.replace("Python", "Java")) # ' Java Programming '

print(text.split()) # ['Python', 'Programming']

3. String Slicing

Slicing allows extracting substrings:

```
s = "Python"
print(s[0:4])   # Pyth (index 0 to 3)
print(s[:3])    # Pyt (from start to index 2)
print(s[2:])    # thon (from index 2 to end)
print(s[::-1])  # nohtyP (reversed string)
```