Python Fundamentals

THEORY ASSIGNMENT

1. Introduction to Python

1. Introduction to Python and its Features (simple, high-level, interpreted language).

Introduction to Python

Python is a **simple, high-level, interpreted programming language** created by **Guido van Rossum (1991)**. It is popular for its **readability, flexibility, and wide applications** like web development, data science, AI, and automation.

Features of Python

- Simple and easy to learn
- High-level language (no need to manage memory)
- Interpreted (executes line by line)
- Cross-platform and portable
- Free and open-source
- Large standard libraries
- Supports OOP and procedural styles
- Dynamically typed
- Strong community support

2. History and evolution of Python.

History and Evolution of Python

- 1980s – Conceived by Guido van Rossum at CWI, Netherlands.
- 1991 – First release (Python 0.9.0) with basic features.
- 1994 – Python 1.0 launched.

- 2000 – Python 2.0 (added list comprehensions, garbage collection).
- 2008 – Python 3.0 released (simpler, modern, not backward compatible).
- Today – One of the most popular languages for AI, data science, web, and automation.

3. Advantages of using Python over other programming languages.

Advantages of Python

1. Simple & Easy to Learn – Syntax is close to English, easier than C, C++ or Java.
2. Cross-Platform – Runs on Windows, macOS, Linux without modification.
3. Large Standard Libraries – Ready-to-use modules for math, web, data, AI, etc.
4. Rapid Development – Faster coding compared to C++ or Java.
5. Interpreted Language – Executes line by line, easy to debug.
6. Supports Multiple Paradigms – Object-oriented, functional, and procedural programming.
7. Dynamic Typing – No need to declare variable types.
8. Strong Community Support – Huge global community and resources.
9. Integration Friendly – Easily integrates with C, C++, Java, and databases.
10. Wide Applications – Used in AI, Machine Learning, Data Science, Web Development, Automation, etc.

4. Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

Installing Python & Setting up Environment

1. Install Python
    - Download from python.org.

- Install and check using python --version in terminal/command prompt.

2. Using Anaconda
    - Download from anaconda.com.
    - Comes with Python, Jupyter Notebook, and many libraries (best for Data Science/AI).
3. Using IDEs
    - PyCharm → Full-featured IDE for Python projects.
    - VS Code → Lightweight editor; install *Python extension* for coding/debugging.

5. Writing and executing your first Python program.

Open any editor/IDE (IDLE, VS Code, PyCharm, or Jupyter Notebook).

Write the code:

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

Save the file with .py extension (e.g., first.py).

Run the program:

- In terminal → python first.py
- Or directly run inside your IDE.

2. Programming Style

1. Understanding Python's PEP 8 guidelines.

PEP 8 Guidelines (in short)

- Use 4 spaces for indentation.
- Keep lines ≤ 79 characters.
- Imports at the top.

- Naming: snake_case (functions/variables), CamelCase (classes), ALL_CAPS (constants).
- Spaces around operators (a = b + c).
- No extra spaces inside brackets.
- Write clear comments/docstrings.

2. Indentation, comments, and naming conventions in Python.

- **Indentation:** 4 spaces to define code blocks.
- **Comments:**
  - Single-line → #
  - Multi-line → """ """
- **Naming Conventions:**
  - Variables/functions → snake_case
  - Classes → CamelCase
  - Constants → ALL_CAPS

3. Writing readable and maintainable code.

Readable & Maintainable Code (Short)

- Follow PEP 8 (indentation, naming, spacing)
- Use meaningful names
- Add comments/docstrings
- Keep code simple and modular
- Be consistent and avoid hardcoding.

3. Core Python Concepts

1. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

Python Data Types (Short)

- **int** → Whole numbers (x = 10)
- **float** → Decimal numbers (y = 3.14)
- **str** → Text (name = "Python")
- **list** → Ordered, mutable ([1,2,3])
- **tuple** → Ordered, immutable ((1,2,3))
- **dict** → Key-value pairs ({"a":1})
- **set** → Unordered, unique ({1,2,3})

2. Python variables and memory allocation.

Python Variables and Memory Allocation

- Variable: A name that stores data/value.
- x = 10
- name = "Python"
- Dynamic Typing: Python automatically detects the type; no need to declare.
- Memory Allocation: Python stores objects in memory and variables reference them.
    - Same value can be shared by multiple variables.
    - Memory is managed automatically by garbage collector.

3. Python operators: arithmetic, comparison, logical, bitwise.

Python Operators (Short)

- Arithmetic: + - * / // % ** → math operations
- Comparison: == != > < >= <= → compare values
- Logical: and or not → combine conditions
- Bitwise: & | ^ ~ << >> → operate on bits.

4. Conditional Statements

1. Introduction to conditional statements: if, else, elif.

Python Conditional Statements

- if → Executes block if condition is True
- elif → Checks another condition if previous if was False
- else → Executes block if all conditions are False

2. Nested if-else conditions.

Nested if-else in Python

- Definition: An if-else statement inside another if or else block.
- Purpose: Handle multiple levels of conditions.

5. Looping (For, While)

1. Introduction to for and while loops.

Python Loops

1. **for loop – Repeats a block of code for each item in a sequence (list, string, range).**

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

2. **while loop – Repeats a block of code while a condition is True.**

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

2. How loops work in Python.

How Loops Work in Python

- Loops repeat a block of code multiple times.
- for loop: Iterates over each item in a sequence (list, string, range).
- while loop: Repeats as long as the condition is True.
- Control statements:
    - break → exit loop
    - continue → skip current iteration
    - pass → do nothing.

3. Using loops with collections (lists, tuples, etc.).

Using Loops with Collections

- for loop is commonly used to iterate over collections: lists, tuples, sets, dictionaries.

Examples:

```
# List
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Tuple
nums = (1, 2, 3)
for num in nums:
    print(num)
# Dictionary
student = {"name": "John", "age": 20}
for key, value in student.items():
    print(key, value)
```

6. Generators and Iterators

1. Understanding how generators work in Python.

Generators In Python

- Special functions that use yield instead of return.
- Produce values one at a time (lazy evaluation).
- Memory-efficient, good for large data.

How They Work

1. Call generator → returns generator object (not values immediately).
2. Use next() or for loop → values come one by one.
3. Stops with StopIteration when done.

2. Difference between yield and return.

return → Ends the function, gives back one final value.

yield → Pauses the function, gives back one value at a time, and can continue to produce more values later.

3. Understanding iterators and creating custom iterators.

An **object** that allows you to traverse through elements **one at a time**.

Must implement two methods:

- __iter__() → returns the iterator object.
- __next__() → returns the next value, raises StopIteration when finished.

7. Functions and Methods

1. Defining and calling functions in Python.

Use def keyword:

def greet(name):

```
    return "Hello " + name
```

Just use the function name with arguments:

```
print(greet("Harshad"))
```

2. Function arguments (positional, keyword, default).

1. **Positional Arguments** → Passed in correct order.

```
def add(a, b):
   return a + b
print(add(5, 3))   # 8
```

2. **Keyword Arguments** → Specify name while calling.

```
def greet(name, msg):
   print(msg, name)

greet(name="Harshad", msg="Hello")
```

3. **Default Arguments** → Predefined value if not given.

```
def greet(name, msg="Hello"):
   print(msg, name)

greet("Harshad")   # Hello Harshad
```

3. Scope of variables in Python.

1. **Local Scope** → Variable defined inside a function (accessible only there).

```
def func():
   x = 10  # local
```

```
    print(x)
```

2. **Global Scope** → Variable defined outside functions (accessible everywhere).

```
x = 20  # global

def func():
    print(x)
```

3. **Nonlocal Scope** → Used in nested functions (refers to variable in outer but not global scope).

```
def outer():
    x = 5
    def inner():
        nonlocal x
        x += 1
        print(x)
    inner()
```

4. Built-in methods for strings, lists, etc.

◇ **Common Built-in Methods**

👉 **Strings (str)**

- upper() → "hello".upper() → "HELLO"
- lower() → "HI".lower() → "hi"
- strip() → " hi ".strip() → "hi"
- split() → "a,b,c".split(",") → ['a','b','c']
- replace() → "hello".replace("h","y") → "yello"

👉 **Lists (list)**

- append() → [1,2].append(3) → [1,2,3]

- extend() → [1,2].extend([3,4]) → [1,2,3,4]
- insert() → [1,2].insert(1,10) → [1,10,2]
- remove() → [1,2,3].remove(2) → [1,3]
- pop() → [1,2,3].pop() → 3
- sort() → [3,1,2].sort() → [1,2,3]

👉 **Tuples (tuple)**

- count() → (1,2,2,3).count(2) → 2
- index() → (10,20,30).index(20) → 1

👉 **Dictionaries (dict)**

- keys() → {"a":1,"b":2}.keys() → ['a','b']
- values() → {"a":1,"b":2}.values() → [1,2]
- items() → {"a":1,"b":2}.items() → [('a',1),('b',2)]
- get() → d.get("x",0) → returns value or default

8. Control Statements (Break, Continue, Pass)

1. Control Statements (Break, Continue, Pass).

Here's a **short explanation of Control Statements in Python**:

- **break** → Immediately exits the loop (stops the iteration fully).
- **continue** → Skips the current iteration and moves to the next loop iteration.
- **pass** → Does nothing (a placeholder statement, used when code is syntactically required but no action is needed).

👉 Example:

```
for i in range(5):
    if i == 2:
```

```
        continue   # skip 2
    if i == 4:
        break      # stop at 4
    print(i)

# Output: 0 1 3
```

9. String Manipulation

1. Understanding how to access and manipulate strings.

Basics

- Strings are sequences of characters and **immutable** (you can't change a character in-place; create a new string instead).
- Get length: len(s)

Access

- Indexing (single char): s[0] (first), s[-1] (last)
- Slicing (substrings): s[1:4] (from index 1 up to 3), s[:3], s[3:], s[::-1] (reverse)

Common operations

- Concatenate: s1 + s2
- Repeat: s * 3
- Membership test: 'a' in s → True/False
- Iterate chars: for ch in s: ...

Useful methods

- Case: s.upper(), s.lower(), s.capitalize()
- Strip whitespace: s.strip(), s.lstrip(), s.rstrip()
- Replace: s.replace('old', 'new')
- Find/index: s.find('sub') (-1 if not found), s.index('sub') (raises)

- Count: s.count('a')
- Split/join: s.split(',') → list; '-'.join(list_of_words) → string
- Padding/format: s.center(10), s.zfill(5); f-strings: f"Hello, {name}!"

2. Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

1. Concatenation → join strings

```
s1 = "Hello"
s2 = "World"
s3 = s1 + " " + s2   # "Hello World"
```

2. Repetition → repeat strings

```
s = "Hi! " * 3       # "Hi! Hi! Hi! "
```

3. Common string methods

- s.upper() → convert to uppercase
- s.lower() → convert to lowercase
- s.capitalize() → first letter uppercase
- s.strip() → remove leading/trailing spaces
- s.replace("old","new") → replace substring
- s.split(",") → split into list
- "-".join(list) → join list into string

3. String slicing.

Syntax:

s[start : end : step]

- start → index to start (inclusive)
- end → index to stop (exclusive)
- step → step size (default 1)

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

1. How functional programming works in Python.

Concept:

- Functional programming (FP) treats computation as evaluation of functions.
- Focuses on pure functions (no side effects) and immutable data.
- Uses functions as first-class objects (can pass them as arguments, return from other functions).

Key features in Python:

1. Higher-order functions → take functions as arguments or return functions

   ```
   def apply(func, x):
       return func(x)
   apply(lambda n: n**2, 5)  # 25
   ```

2. Built-in FP tools
   - map(func, iterable) → apply function to each item
   - filter(func, iterable) → filter items based on condition
   - reduce(func, iterable) → cumulative computation (from functools)

   ```
   from functools import reduce
   nums = [1,2,3,4]
   squared = list(map(lambda x: x**2, nums))      # [1,4,9,16]
   evens = list(filter(lambda x: x%2==0, nums))   # [2,4]
   total = reduce(lambda a,b: a+b, nums)          # 10
   ```

3. Immutability & no side effects → avoid changing original data

In short: FP in Python = use functions to process data, avoid changing it, and use tools like map, filter, reduce.

2. Using map(), reduce(), and filter() functions for processing data.

1. map(func, iterable) → apply a function to every item

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))  # [1, 4, 9, 16]
```

2. filter(func, iterable) → keep items that satisfy a condition

```
nums = [1, 2, 3, 4]
evens = list(filter(lambda x: x % 2 == 0, nums))  # [2, 4]
```

3. reduce(func, iterable) → cumulatively combine items (from functools)

```
from functools import reduce
nums = [1, 2, 3, 4]
total = reduce(lambda a, b: a + b, nums)  # 10
```

Key points:

- map → transforms each element
- filter → selects elements
- reduce → combines elements into a single result

3. Introduction to closures and decorators.

1. Closures

- A **closure** is a function **remembering the environment** in which it was created.
- Happens when a **nested function** uses a variable from its **enclosing scope**.

```
def outer(x):
    def inner(y):
        return x + y
    return inner
```

```
add5 = outer(5)
print(add5(10))  # 15
```

Key: inner remembers x from outer even after outer has finished.

---

2. Decorators

- A **decorator** is a function that **takes another function and extends/modifies its behavior** without changing its code.
- Uses @ syntax.

```
def decorator(func):
    def wrapper():
        print("Before function")
        func()
        print("After function")
    return wrapper

@decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Before function
# Hello!
# After function
```