Python

UNIT 1 — Introduction to Python

1. Define Python

Definition :
Python is a high-level, interpreted, general-purpose programming language that emphasizes readable code and rapid development. It uses dynamic typing, automatic memory management (garbage collection), and a large standard library. Python supports multiple paradigms (procedural, object-oriented, functional).

---

2. What is mutability?

Definition :
Mutability describes whether the contents of an object can be changed after it is created. Mutable objects can change in place (e.g., lists, dictionaries); immutable objects cannot be changed (e.g., tuples, strings, integers).
Example:

```
# list is mutable
lst = [1, 2, 3]
lst[0] = 100  # modifies same object -> [100, 2, 3]

# tuple is immutable
t = (1, 2, 3)
# t[0] = 100  # TypeError: 'tuple' object does not support item assignment
```

---

3. What is recursion?

Definition :
Recursion is a technique where a function calls itself to solve a smaller instance of the same problem. It needs a base case to stop recursion and reduce problem size each call. Useful for divide-and-conquer tasks and tree traversal.

Syntax / Example (factorial):

```
def factorial(n):
    if n <= 1:     # base case
        return 1
    return n * factorial(n - 1)  # recursive call
print(factorial(5))  # 120
```

---

4. Give an example of a tuple

Definition :
A tuple is an ordered, immutable collection in Python. Use parentheses `()` or just commas. Good for fixed collections and can be used as dictionary keys if elements are hashable.
Example:

```
person = ("Alice", 25, "Student")
print(person[0])  # "Alice"
```

---

5. What is scoping in Python?

Definition :
Scoping determines where a name is visible (accessible). Python follows the **LEGB** rule: **L**ocal → **E**nclosing (nonlocal) → **G**lobal → **B**uilt-in. `global` and `nonlocal` keywords alter scope binding.
Example:

```
x = 10  # global

def outer():
    x = 5  # enclosing
    def inner():
        nonlocal x
        x = 3  # modifies enclosing x
    inner()
    print(x)  # 3

outer()
print(x)  # 10  (global unchanged)
```

---

6. Write syntax for defining a function in Python

Definition :
A function is defined using `def` followed by name and parameters, a colon, and an indented block as body. Optional `return` to give back a value.
Example:

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

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

2-mark: Differentiate between list and tuple

Definition (detail):

- **List:** mutable, uses `[]`, many modifying methods (`append`,`pop`,`extend`), good for collections that change.
- **Tuple:** immutable, uses `()`, fewer methods, slightly faster, safe to use as dict keys if elements are hashable.
  Comparison table (short):
- Syntax: `list = [1,2]` vs `tuple = (1,2)`
- Mutability: list = mutable, tuple = immutable
- Use-case: list for variable data, tuple for fixed records
  Example:

```
lst = [1, 2]; lst.append(3)  # [1,2,3]

tup = (1, 2) # cannot modify in-place
```

2-mark: Explain iteration with an example

Definition :
Iteration is repeating over elements of a collection using loops. Python supports `for` loops (iterate over iterable objects) and `while` loops (conditional). Iterators (`iter()`, `next()`) and generator are also used.
Example (for / while / iterator):

```
# for loop
for i in [10,20,30]:
    print(i)

# while loop
i = 0
while i < 3:
    print(i)
    i += 1

# iterator example
it = iter([1, 2, 3])
print(next(it))  # 1
```

5-mark: Explain different types of functions in Python

Definition (detail):
Python functions can be classified by how they are created/used:

- **User-defined functions:** created with `def` or `lambda`.
- **Recursive functions:** functions that call themselves to solve a subproblem; must include base case.
- **Anonymous (lambda) functions:** small one-line functions using `lambda`.
- **Higher-order functions:** functions that can accept other functions as arguments or return functions (functions are first-class objects).
- **Built-in functions:** e.g., `len()`, `sum()`.

Syntax & examples:

```
# user-defined
def greet(name):
    return f"Hello, {name}"

# lambda (anonymous)
square = lambda x: x * x
print(square(4))  # 16

# recursive
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

# higher-order: function as object
def apply(func, value):
    return func(value)

print(apply(square, 5))  # 25

# returning function
def make_multiplier(n):
    def multiply(x):
        return x * n
    return multiply

double = make_multiplier(2)
print(double(10))  # 20
```

Notes to mention in exam: point out base case for recursion, potential recursion depth issues, and advantages of first-class functions for functional-style coding (`map`, `filter`, callbacks).

5-mark: Discuss data structures in Python (lists, tuples, dictionaries)

Definition :

- **List:** ordered, mutable sequence. Methods: `append`, `insert`, `pop`, `remove`, slicing. Good for dynamic collections.
- **Tuple:** ordered, immutable sequence. Use for fixed collections, heterogeneous records (like rows).
- **Dictionary:** unordered (as abstract concept; insertion-ordered since Python 3.7), mutable mapping of key → value. Keys must be hashable. Methods: `get`, `keys`, `values`, `items`, `update`.

Syntax & examples:

```
# List
fruits = ["apple", "banana"]
fruits.append("mango")
print(fruits[0])   # "apple"

# Tuple
point = (10, 20)
# point[0] = 5  # not allowed

# Dictionary
student = {"roll": 1, "name": "Alice", "marks": 85}
print(student["name"])   # "Alice"
student["marks"] = 90    # update

# iterate
for k, v in student.items():
    print(k, v)
```

When to use which: Use lists for changeable sequences; tuples for fixed-grouping of values (safer); dictionaries for key-based lookup and representing objects/records.

UNIT 2 — OOP using Python

1. Define inheritance

Definition :
Inheritance is an OOP mechanism where a new class (derived/child) inherits attributes and methods from an existing class (base/parent). It promotes code reuse and enables polymorphism.
Syntax / Example:

```
class Animal:
    def speak(self):
        print("some sound")

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        print("bark")

d = Dog(); d.speak()  # "bark"
```

---

2. What is encapsulation?

Definition :
Encapsulation bundles data (attributes) and methods acting on that data inside a class and restricts direct access from outside. In Python, name conventions `_protected` and `__private` indicate intent; properties and getter/setter methods control access.

Example:

```
class BankAccount:
    def __init__(self, balance=0):
        self.__balance = balance  # private by name mangling

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance
```

---

3. Write syntax of try-except block

Definition (detail):
`try`/`except` handles exceptions to prevent program crash. Optional `else` executes when no exception; `finally` runs always.
Syntax & Example:

```
try:
    # risky code
    x = 1 / 0
except ZeroDivisionError as e:
    print("Division by zero:", e)
except Exception as e:
    print("Other error:", e)
else:
    print("No error")
finally:
    print("Always runs")
```

---

4. What is an assertion in Python?

Definition :
An `assert` statement tests an expression and raises `AssertionError` if expression is false. Used for debugging and verifying assumptions. Assertions can be disabled with `-O` interpreter flag.
Syntax / Example:

```
x = 5
assert x > 0, "x must be positive"
```

2-mark: Differentiate between abstract class and normal class

Definition :

- **Normal class:** can be instantiated and may implement all methods.
- **Abstract class:** defines a common interface with one or more abstract methods (no implementation). Cannot instantiate; concrete subclasses must implement abstract methods. In Python use `abc.ABC` and `@abstractmethod`.
  Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, r): self.r = r
    def area(self):
        return 3.14 * self.r * self.r

c = Circle(5)
print(c.area())
# Shape()  # TypeError: can't instantiate abstract class
```

2-mark: Give an example of exception handling

Definition (detail):
Show typical `try/except/finally` with specific exception handling and fallback.
Example:

```
try:
    n = int(input("Enter a number: "))
    print(10 // n)
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("Done.")
```

5-mark: Explain types of inheritance in Python with examples

Definition (detail):
Common inheritance types:

1. Single Inheritance

📖 *Definition:* In single inheritance, a child class derives from only one parent class.

```
# Single Inheritance Example

class Parent:

    def display(self):

        print("This is the Parent class.")


class Child(Parent):  # Child inherits from Parent

    def show(self):

        print("This is the Child class.")


# Object of Child

c = Child()

c.display()

c.show()
```

---

2. Multiple Inheritance

📖 *Definition:* In multiple inheritance, a child class inherits features from **two or more parent classes**.

```
# Multiple Inheritance Example

class Father:

    def quality(self):

        print("Father: Hardworking")


class Mother:

    def skill(self):

        print("Mother: Caring")


class Child(Father, Mother):  # Inherits from both

    def show(self):

        print("Child: Combination of parents")


c = Child()

c.quality()

c.skill()

c.show()
```

---

3. Multilevel Inheritance

📖 *Definition:* In multilevel inheritance, a class is derived from a child class, which is already derived from another class. (Grandparent → Parent → Child)

```
# Multilevel Inheritance Example

class Grandparent:

    def feature1(self):

        print("Grandparent: Wise")


class Parent(Grandparent):

    def feature2(self):

        print("Parent: Responsible")


class Child(Parent):

    def feature3(self):

        print("Child: Learning")


c = Child()

c.feature1()

c.feature2()

c.feature3()
```

---

4. Hierarchical Inheritance

📖 *Definition:* In hierarchical inheritance, multiple child classes inherit from the **same parent** class.

```
# Hierarchical Inheritance Example

class Parent:

    def message(self):

        print("Parent: Common property")


class Child1(Parent):

    def feature1(self):

        print("Child1: Feature A")


class Child2(Parent):

    def feature2(self):

        print("Child2: Feature B")


c1 = Child1()

c1.message()

c1.feature1()


c2 = Child2()

c2.message()

c2.feature2()
```

---

5. Hybrid Inheritance

📖 *Definition:* Hybrid inheritance is a **combination** of different types of inheritance (like multiple + hierarchical).

```
# Hybrid Inheritance Example

class A:

    def featureA(self):

        print("Class A: Base class")


class B(A):  # Single inheritance from A

    def featureB(self):

        print("Class B inherits from A")


class C(A):  # Another child of A (hierarchical)

    def featureC(self):

        print("Class C inherits from A")
```

```
class D(B, C):  # Multiple inheritance (B + C)
    def featureD(self):
        print("Class D inherits from B and C")

d = D()
d.featureA()
d.featureB()
d.featureC()
d.featureD()
```

5-mark: Explain sorting algorithms (bubble, insertion, quick sort) in Python

Definition (detail) & complexity summary:

- **Bubble sort:** repeatedly swap adjacent elements if out of order. Simple but O(n²) average/worst.
- **Insertion sort:** build sorted portion by inserting each element into the correct position. O(n²) average/worst, O(n) best when nearly sorted.
- **Quick sort:** divide-and-conquer: choose pivot, partition, sort partitions recursively. Average O(n log n), worst O(n²) if poor pivot.

Code examples:

```
# Bubble sort
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Insertion sort
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key

# Quick sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    mid  = [x for x in arr if x == pivot]
    right= [x for x in arr if x > pivot]
    return quick_sort(left) + mid + quick_sort(right)
```

Exam tip: Mention stability (insertion and bubble are stable; naive quicksort above is not guaranteed stable). Also note practical usage: use built-in `sorted()` or `list.sort()` for real tasks (Timsort, O(n log n)).

UNIT 3 — Plotting using PyLab

> Note: Many courses call matplotlib's MATLAB-like interface **PyLab**. Recommended modern usage is `matplotlib.pyplot` (often imported as `plt`). I'll show both notations.

1. What is PyLab?

Definition :
PyLab is a module that mixes `matplotlib` plotting functionality with `numpy` into a MATLAB-like namespace for interactive plotting. Practically, you'll use `matplotlib.pyplot` (`plt`) and `numpy` to plot graphs in Python.

Example import:

```
# Recommended
import matplotlib.pyplot as plt
import numpy as np

# Old style (not recommended for scripts but used in interactive notebooks)
from pylab import *   # imports many names into namespace
```

2. Write a command to plot a simple graph

Definition (detail):
Use `plot()` to draw lines connecting data points and `show()` to display the figure.
Example:

```
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [2, 4, 6]
plt.plot(x, y)   # draw line
plt.xlabel("x-axis")
plt.ylabel("y-axis")
plt.title("Simple plot")
plt.show()
```

---

3. What is `xlabel()` used for?

Definition (detail):
`xlabel()` sets the label for the x-axis of the current plot — improves readability and documentation of plots.
Example:

```
plt.plot([0,1], [0,1])
plt.xlabel("Time (s)")
plt.show()
```

2-mark: Explain difference between `plot()` and `bar()`

Definition (detail):

- `plot()` draws lines (and markers) connecting numeric x-y points — used for continuous data/trends.
- `bar()` draws rectangular bars whose heights correspond to values — used for categorical or grouped comparisons.
  Examples:

```
# line plot
plt.plot([1,2,3], [2,4,6])

# bar chart
plt.bar(['A','B','C'], [5,7,3])
```

2-mark: Write code to plot a sine wave

Example (sine):

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 200)
y = np.sin(x)
plt.plot(x, y)
plt.xlabel('x (radians)')
plt.ylabel('sin(x)')
plt.title('Sine Wave')
plt.grid(True)
plt.show()
```

Explain plotting in Python with PyLab functions and examples

Definition (Detail):

- **PyLab** is a module that combines **NumPy (numerical computing)** with **matplotlib.pyplot (plotting)** into a single environment.
- It provides functions to create **2D plots, charts, and graphs** easily.
- PyLab allows visualization of mathematical functions, data analysis results, and scientific computations.

Key Functions in PyLab:

1. **plot(x, y)** → Draws a simple 2D line graph.
2. **xlabel("label")** → Adds label to X-axis.
3. **ylabel("label")** → Adds label to Y-axis.
4. **title("graph title")** → Adds a title to the graph.
5. **grid(True)** → Shows grid for readability.
6. **legend()** → Adds legend to identify multiple plots.
7. **bar(x, height)** → Creates bar charts.
8. **hist(data, bins)** → Creates histogram.

Explanation of Steps in Plotting:

- **Import PyLab / matplotlib**:

  ```
  import matplotlib.pyplot as plt
  ```

  Prepare data (x and y values):
  For example, x = [1,2,3,4], y = [2,4,6,8].

  Call plotting function:

  ```
  plt.plot(x, y)
  ```

Add labels and title:

```
plt.xlabel("X-axis")
```

- `plt.ylabel("Y-axis")`
- `plt.title("Simple Line Graph")`

- **Display graph**:

```
plt.show()
```

Exampl:

- `import matplotlib.pyplot as plt`
-
- `x = [1, 2, 3, 4, 5]`
- `y = [2, 4, 6, 8, 10]`
-
- `plt.plot(x, y)  # Line graph`
- `plt.xlabel("X values")`
- `plt.ylabel("Y values")`
- `plt.title("Basic PyLab Plot")`
- `plt.grid(True)`
- `plt.show()`

Summary:

- PyLab is a convenient plotting module in Python.
- It provides functions for **line plots, bar charts, scatter plots, histograms** etc.
- Steps: Import → Prepare Data → Plot → Customize → Show.

Types of Plots in PyLab

In Python, **PyLab/Matplotlib** provides many functions to visualize data in the form of graphs.
The most important plotting types are:

---

1. Line Plot (`plot(x,y)`)

- Shows relationship between two variables using a line.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 50)
y = np.sin(x)
plt.plot(x, y, label="sin(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Line Plot Example")
plt.legend()
plt.show()
```

---

2. Scatter Plot (`scatter(x,y)`)

- Displays individual data points, useful for pattern/relationship detection.

```
x = [5,7,8,7,6,9,5,6,7,8]
y = [99,86,87,88,100,86,103,87,94,78]
plt.scatter(x, y)
plt.title("Scatter Plot")
plt.show()
```

---

3. Bar Chart (`bar(x, heights)`)

- Represents data using rectangular bars.

```
x = ['A','B','C','D']
y = [3,7,2,5]
plt.bar(x, y)
plt.title("Bar Chart")
plt.show()
```

---

4. Histogram (`hist(data, bins)`)

- Represents distribution of data (frequency).

```
data = np.random.randn(1000)
plt.hist(data, bins=20, color='g')
plt.title("Histogram")
plt.show()
```

---

5. Multiple Plots (`subplot()`)

- Used to show multiple graphs in one figure.

```
x = np.linspace(0, 2*np.pi, 100)
plt.subplot(1,2,1)
plt.plot(x, np.sin(x))
plt.title("Sine")

plt.subplot(1,2,2)
plt.plot(x, np.cos(x))
plt.title("Cosine")
plt.show()
```

Extra Useful Functions

- `xlabel(), ylabel(), title()` → Labels and title
- `legend()` → Show labels
- `grid(True)` → Show gridlines
- `savefig("file.png")` → Save plot to file
