Py

Q) Explain the characteristics and provide scenarios where different data structures of python would be most appropriate for usage.

Answer :

Data Structure	Ordered	Mutable	Allows Duplicates	Use Case Example
List	Yes	Yes	Yes	Task lists, stacks, queues
Tuple	Yes	No	Yes	Coordinates, fixed data
Set	No	Yes	No	Unique items, set operations
Dictionary	Yes	Yes	Keys: No	Mappings like ID to data
String	Yes	No	Yes	Text processing
Deque	Yes	Yes	Yes	Efficient queue and stack operations

List :

Characteristics:

- Ordered collection of items.
- Mutable (can be changed after creation).
- Allows duplicate elements.
- Supports indexing and slicing.

Usage Scenarios:

- Storing a sequence of items where order matters.
- Implementing stacks or queues (with extra methods).
- Example: Managing a to-do list or playlist.

```
tasks = ["email", "meeting", "coding", "review"]
```

2. Tuple

Characteristics:

- Ordered collection of items.
- Immutable (cannot be changed after creation).
- Allows duplicate elements.
- Faster than lists for fixed data.

Usage Scenarios:

- Storing fixed collections of values (e.g., coordinates).
- Can be used as dictionary keys (if elements are immutable).
- Example: Representing a point in 2D space.

```
point = (10, 20)
print(point)
```

---

3. Set

Characteristics:

- Unordered collection.
- Mutable, but only holds **unique** elements.
- Does not allow indexing or duplicates.

Usage Scenarios:

- Removing duplicates from data.
- Performing set operations like union, intersection.
- Example: Getting unique visitors to a website.

```
unique_visitors = {"alice", "bob", "charlie"}
```

---

4. Dictionary

Characteristics:

- Unordered collection of key-value pairs (ordered since Python 3.7).
- Keys must be unique and immutable.
- Values can be of any type.

Usage Scenarios:

- Storing data associated with labels (like a phone book).
- Fast lookup by keys.
- Example: Representing a student's record.

```
student = {"name": "John", "age": 20, "grade": "A"}
```

---

1. List Methods

Method	Description
`append(x)`	Adds element `x` to the end of the list
`extend(iter)`	Adds all elements from another iterable
`insert(i, x)`	Inserts `x` at index `i`
`remove(x)`	Removes first occurrence of `x`
`pop([i])`	Removes and returns item at index `i`
`index(x)`	Returns index of first occurrence of `x`
`count(x)`	Counts occurrences of `x`
`sort()`	Sorts the list in ascending order
`reverse()`	Reverses the list in place
`clear()`	Removes all elements from the list

---

2. Tuple Methods

Method	Description
`count(x)`	Returns number of times `x` appears
`index(x)`	Returns index of first occurrence of `x`
`max(x)`	Returns max value from the tuple `x`
`min(x)`	Returns min value from the tuple `x`
`len(x)`	Returns length of tuple `x`

> Tuples are immutable; hence, they support very few methods.

---

3. Set Methods

Method	Description
`add(x)`	Adds element `x`
`update(iter)`	Adds multiple elements
`remove(x)`	Removes `x`; raises error if not found
`discard(x)`	Removes `x`; no error if not found
`pop()`	Removes and returns an arbitrary element
`clear()`	Removes all elements
`union(set)`	Returns union of sets
`intersection(set)`	Returns common elements
`difference(set)`	Elements in current set but not in another
`issubset(set)`	Checks if set is subset

Q) Discuss the usage of date and time functionalities in Python

1. Date & Time Module

Common Classes and Methods:

- `datetime.datetime.now()` – Current local date and time.
- `datetime.date.today()` – Current local date.
- `datetime.datetime.strptime()` – Convert string to datetime.
- `datetime.datetime.strftime()` – Convert datetime to string.
- `datetime.timedelta()` – Represents a duration (difference between dates/times).

Usage Examples:

```
from datetime import datetime, date, timedelta
now = datetime.now()
print("Now:", now)

dob = date(2000, 5, 1)

future_date = now + timedelta(days=10)

formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted:", formatted)

parsed = datetime.strptime("2025-05-01", "%Y-%m-%d")
```

2. Time Module

Common Functions:

- `time.time()` – Current time in seconds since epoch.
- `time.sleep(seconds)` – Pause execution.
- `time.ctime()` – Converts timestamp to a readable string.

Usage Examples:

```
import time

timestamp = time.time()
print("Timestamp:", timestamp)

time.sleep(2)
print("Readable:", time.ctime(timestamp))
```

Q). Concept of Modules and Their Significance in Python

What is a Module?

A **module** in Python is simply a file containing **Python code** (functions, classes, or variables) that you can **import and reuse** in other programs.

- Any `.py` file is a module.
- Python also provides a large **standard library of modules** (like `math`, `os`, `random`, etc.).

How to Use a Module:

```
import math
print(math.sqrt(16))

from math import sqrt
print(sqrt(25))
```

Creating Your Own Module:

If you have a file `mytools.py`:

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

`# 2nd file`

```
import mytools
print(mytools.greet("Alice"))
```

``

Significance of Modules:

Benefit	Explanation
Code Reusability	Write once, use anywhere.
Organization	Keeps code organized and manageable.
Namespace Management	Avoids name conflicts using module names.
Simplifies Maintenance	Easier to update or debug small, focused modules.
Supports Modularity	Encourages breaking down programs into components.

Q). Defining Functions in Python and Their Advantages

What is a Function?

A **function** is a named block of code designed to **perform a specific task**, and can be called multiple times in a program.

Defining a Function:

```
def greet():
    return "Hello World "
greet()
```

Function with Default Argument:

```
def greet(name="Guest"):
```

```
    return f"Hello, {name}!"
```

``

Function with Multiple Arguments:

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

Advantages of Using Functions:

Advantage	Explanation
Code Reusability	Write once, call multiple times.
Modularity	Breaks program into smaller, manageable parts.
Improves Readability	Easy to understand and maintain.
Avoids Repetition	Reduces duplication of code.
Debugging Made Easy	Problems can be isolated within functions.
Scalability	Functions can grow or change without affecting other code parts.

Q) Discuss the purpose and usage of the exit function in Python.

What is `exit()` in Python?

The `exit()` function is used to **terminate** or **exit** a Python program **before it reaches the end of the script**.

- It is most commonly used in **interactive sessions**, **testing**, or when you want to stop program execution based on a certain condition.
- It is part of the `sys` **module** (`sys.exit()`) or the built-in `exit()` function provided by the `site` module.

Common Variants:

1. `exit()` **or** `quit()`
    - These are **built-in helpers** mainly for the **interactive shell** (like IDLE or Jupyter).

- They are not recommended in scripts, as they raise `SystemExit` and may not work in all environments.

2. `sys.exit()`
   - The **recommended way** to exit a program in Python scripts.
   - Requires importing the `sys` module.
3. You can pass a status code:

- `sys.exit(0)` → Successful exit.
- `sys.exit(1)` → Exit with an error (non-zero usually means error).

```
import sys

age = int(input("Enter your age: "))
if age < 18:
    print("You are not eligible.")
    sys.exit()

print("Welcome to the application!")

# if Calling `exit()` or `sys.exit()` raises the `SystemExit` exception then
use try except block

try:
    sys.exit()
except SystemExit:v
    print("Exit was intercepted!")
```

Scenario	Use
User input validation fails	`sys.exit()` to stop execution
Fatal error in script	`sys.exit(1)` to indicate error
Interactive debugging/testing	Use `exit()` or `quit()`
Early termination from loops or conditions	`sys.exit()`

Q) Explain the concept of default arguments in Python functions with examples.

Default Arguments in Python Functions

Default arguments are function parameters that assume a default value if no argument is provided during the function call.

Default arguments must be defined **after** non-default arguments.

Why Use Default Arguments?

1. To make function parameters optional.
2. To simplify function calls.
3. To provide sensible fallbacks.
4. To avoid overloading (Python doesn't support method overloading like some languages).

Important Points

1. **Ordering of Parameters:**
   - Non-default parameters must come **before** default parameters.
   - Invalid: `def func(a=1, b):` → This causes a syntax error.
   - Valid: `def func(a, b=1):`
2. **Default Values Are Evaluated Once:**
   - Default values are evaluated only **once** when the function is defined, not each time it is called.

Syntax

```
def function_name(para1=defalut_value):
    return val # or function body
```

Examples

```
def greet(name="Guest"):
    return "Hello, " + name

print(greet("Alice"))
print(greet())

def add(a=0, b=0):
    return a + b

print(add(5, 3))
print(add(5))
print(add())

def power(base, exponent=2):
    return base ** exponent

print(power(3))
print(power(2, 5))
```

Q) Define objects and classes in Python and discuss their relationship.

Class

A **class** is a blueprint for creating objects. It defines attributes (data) and methods (functions) that determine the behavior of the objects.

Syntax

```
class ClassName:
    def __init__(self, param1, param2):
        self.param1 = param1
        self.param2 = param2

    def method_name(self):
        return self.param1
```

Object

An **object** is an instance of a class. It contains actual data and uses the methods defined in the class.

```
obj = ClassName(value1, value2)
```

Example:

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def get_info(self):
        return self.brand + " " + str(self.year)

car1 = Car("Toyota", 2020)
car2 = Car("Honda", 2022)

print(car1.get_info())
```

```
print(car2.get_info())
```

Relationship Between Class and Object

Feature	Class	Object
Definition	Blueprint or template	Actual instance created from a class
Purpose	Defines structure and behavior	Holds data and uses methods
Creation	Defined once	Can be created many times from a class
Memory	Not stored until instantiated	Stored in memory with unique data

Q) Extra Example for class and object

```
import math

class Rectangle:
    def __init__(a, length, width):
        a.l = length
        a.w = width

    def area(a):
        return a.l * a.w

class Square:
    def __init__(s, side):
        s.s = side

    def area(s):
        return s.s * s.s

class Circle:
    def __init__(c, radius):
        c.r = radius

    def area(c):
        return math.pi * c.r * c.r

r = Rectangle(10, 5)
s = Square(4)
c = Circle(7)
```

```
print("Area of Rectangle:", r.area())
print("Area of Square:", s.area())
print("Area of Circle:", c.area())
```

Q) Explain the concept of inheritance in Python

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called the **child** or **derived** class) to inherit attributes and methods from another class (called the **parent** or **base** class). This promotes code reusability and establishes a hierarchical relationship between classes.

Syntax :

```
class BaseClass:
    # parent class definition

class DerivedClass(ParentClass):
    # child class inherits from ParentClass
```

Example

```
# Single Level Inheritence

class Animal:
    def sound(self):
        return "Some generic sound"

class Dog(Animal):
    def bark(self):
        return "Bark!"

d = Dog()
print(d.sound())
print(d.bark())

# Multiple Inhertance

class Father:
    def skills(self):
        return " Rust Is The Best Programming Language Fuck So Called
Python"

class Mother:
    def skills(self):
```

```
        return " Art"

class Child(Father, Mother):
    def skills(self):
        return super().skills() + ", Sports"

c = Child()
print(c.skills())
```

Type	Description
Single Inheritance	A child class inherits from one parent class.
Multiple Inheritance	A child class inherits from multiple parent classes.
Multilevel Inheritance	A class inherits from a child class, making it a grandchild of the base.
Hierarchical Inheritance	Multiple child classes inherit from a single parent class.
Hybrid Inheritance	A combination of two or more types of inheritance.

Q) Discuss the importance of regular expressions in Python and provide examples.

What is a Regular Expression?

A **Regular Expression (RegEx)** is a sequence of characters that defines a search pattern. It is primarily used for string pattern matching and manipulation.

Furthermore It's Use For Following :

- **Pattern Matching**: Identify specific patterns within strings, such as email addresses, phone numbers, or dates.
- **Data Validation**: Ensure that inputs conform to expected formats.
- **Data Cleaning**: Remove or replace unwanted characters or substrings.
- **Text Parsing**: Extract specific information from large text datasets.
- **Efficiency**: Perform complex string operations efficiently

In Python **The `re` Module in Python**

Python provides the built-in `re` module to work with regular expressions.

You Can Use This For Implementing The Regex Module In Python

```
import re
```

Widely Used Functions In Re Module Of Python

Commonly Used Functions

- `re.match(pattern, string)`: Checks for a match only at the beginning of the string.
- `re.search(pattern, string)`: Searches the string for a match anywhere.
- `re.findall(pattern, string)`: Returns a list of all non-overlapping matches.
- `re.sub(pattern, repl, string)`: Replaces matches with a specified string

Example :

```
import re

email = "example@domain.com"
pattern = r'^\w+@\w+\.\w+$'

if re.match(pattern, email):
    print("Valid email address")
else:
    print("Invalid email address")

text = "Contact us at 123-456-7890 or 987-654-3210"
pattern = r'\d{3}-\d{3}-\d{4}'

phone_numbers = re.findall(pattern, text)
print(phone_numbers)

text1 = "This   is  a    sentence with   irregular spacing."
new_text = re.sub(r'\s+', ' ', text1)
print(new_text)

text2 = "Words, separated; by: various! punctuation?"
words = re.split(r'\W+', text2)
print(words)
```

Metacharacter	Description
`.`	Matches any character except newline
`^`	Matches the beginning of the string
`$`	Matches the end of the string
`*`	Matches 0 or more repetitions
`+`	Matches 1 or more repetitions

Metacharacter	Description
`?`	Matches 0 or 1 repetition
`{n}`	Matches exactly n repetitions
`{n,}`	Matches n or more repetitions
`{n,m}`	Matches between n and m repetitions
`[]`	Matches any character in the set
`()`	Captures the matched subexpression

Sequence	Description
`\d`	Matches any digit; equivalent to `[0-9]`
`\D`	Matches any non-digit character
`\w`	Matches any alphanumeric character
`\W`	Matches any non-alphanumeric character
`\s`	Matches any whitespace character
`\S`	Matches any non-whitespace character
`\b`	Matches the empty string at the beginning or end of a word
`\B`	Matches the empty string not at the beginning or end of a word

Q) Explain event-driven programming and its relevance in Python.

Event-driven programming is a programming paradigm where the program flow is determined by events such as:

- User actions (mouse clicks, key presses)
- Sensor outputs
- Messages from other programs or threads
- System-generated notifications

Instead of executing code linearly, event-driven programs wait for events and respond with event handlers—functions that are triggered by specific events.

Term	Description
Event	An action or occurrence (e.g., mouse click, key press, signal)
Event Handler	A function or method that responds to a specific event
Event Loop	A control structure that waits for and dispatches events or messages

Term	Description
Callback	A function passed as an argument to another function, executed later

Python supports event-driven programming through several libraries and frameworks. It's especially relevant in:

- **GUI Programming** (e.g., with Tkinter, PyQt)
- **Asynchronous Programming** (e.g., `asyncio`)
- **Web Servers** (e.g., Flask with SocketIO, Tornado)
- **Game Development** (e.g., Pygame)
- **Networking** (e.g., Twisted)

Example

```
import tkinter as tk

def on_click():
    print("Button clicked!")

root = tk.Tk()
button = tk.Button(root, text="Click Me", command=on_click)
button.pack()
root.mainloop()
```

Advantages of Event-Driven Programming

- **Responsive interfaces**
- **Efficient I/O handling**
- **Better structure for real-time applications**
- **Improved modularity** (via handlers and callbacks)

Disadvantages

- **Complex debugging**
- **Harder to trace flow**
- **Callback hell** in complex systems without async/await

Use Cases in Python

Use Case	Framework/Library
GUI Applications	Tkinter, PyQt, Kivy
Asynchronous Web Apps	asyncio, FastAPI
Game Development	Pygame
Network Servers	Twisted, Tornado
Real-time Systems	SocketIO, websockets

Q) Describe the basics of GUI programming in Python and its libraries.

GUI (Graphical User Interface) programming involves building visual interfaces that allow users to interact with software using elements like:

- Buttons
- Textboxes
- Menus
- Labels
- Windows

Instead of typing commands, users interact with applications via **events** like mouse clicks or key presses.

1. Windows (Tk)

The main window is created using the `Tk()` class.

Feature	Description
`Tk()`	Initializes the main window
`title()`	Sets the window title
`mainloop()`	Starts the event loop

2. Labels

Used to display **text or images**.

Option	Description
`text`	Text to display
`font`	Font style and size
`bg` / `fg`	Background / foreground color

Option	Description
`image`	Display an image

Buttons

Used to perform an **action** when clicked.

Option	Description
`text`	Button label
`command`	Function to call on click
`state`	`NORMAL` or `DISABLED`
`bg`, `fg`	Background / text color

4. Textboxes

Used for **single-line text input**.

Option	Description
`width`	Width of the entry field
`show='*'`	For password-like input
`.get()`	Retrieves entered value

Common Layout Managers

Layout	Description
`pack()`	Places widgets vertically or horizontally
`grid()`	Places widgets in a table-like structure
`place()`	Places widgets using absolute positioning

Example :

```
import tkinter as tk

def show_message():
    name = entry.get()
    label_output.config(text=f"Hello, {name}!") # .config is for dynamically
change the text or anyhting you want to update

root = tk.Tk()
```

```
root.title("Simple GUI Example")
root.geometry("300x200") # Make a window of the 300 by 200

label_in = tk.Label(root, text="Enter your name:")
label_in.pack(pady=5)

entry = tk.Entry(root, width=25) # TEXT BOX
entry.pack(pady=5) # Padding y-axis

button = tk.Button(root, text="Submit", command=show_message) # BUTTON
button.pack(pady=10)

label_output = tk.Label(root, text="")
label_output.pack(pady=5)

root.mainloop()
```

Key Concepts in GUI Programming

Concept	Description
Widgets	UI elements (e.g., Button, Label, Entry)
Event Loop	Continuously checks for user actions and triggers responses
Geometry Manager	Controls widget layout (e.g., `pack`, `grid`, `place`)
Callback/Handler	Function triggered by an event (e.g., a button click)

Popular Python GUI Libraries

Library	Description
Tkinter	Built-in standard GUI library in Python. Simple and lightweight
PyQt	Python bindings for the Qt framework. Rich features and cross-platform
Kivy	Designed for multi-touch applications and mobile platforms
wxPython	Native look on multiple platforms using wxWidgets
PySide	Official Qt bindings (alternative to PyQt)

Library	Best For
Tkinter	Beginners, quick prototypes
PyQt/PySide	Desktop applications with rich UIs

Library	Best For
Kivy	Mobile apps, multi-touch interfaces
wxPython	Native desktop applications

2-marks questions

1. Explain different methods for accepting user input in Python

Python provides multiple techniques to accept input from users, depending on the nature and source of the input. These methods help collect data from the keyboard, files, command-line arguments, and standard input streams.

1. **Using `input()` Function**

```python
variable = input("Prompt")
```

The `input()` function is the most common method used for taking input from the keyboard.

It always returns a string, so type conversion is required when working with numbers.

```python
name = input("Enter your name: ")
print("Hello,", name)
```

2. **Reading Multiple Inputs in One Line**

You can use `.split()` along with `map()` to take multiple inputs from one line.

```python
numbers = input("Enter numbers: ").split()
print("Numbers:", numbers)
```

```

3. **Using `sys.stdin.read()`**

```python
import sys

data = sys.stdin.read()

print("You entered:", data)
```

4. **Using `sys.stdin.readline()`**

`python import sys line = sys.stdin.readline().strip() print("You entered:", line)`

5. **Using `fileinput.input()`**

`python import fileinput for line in fileinput.input(): print(line.strip())`

6. **Using Command-line Arguments (`sys.argv`)**

`python import sys print("Argument passed:", sys.argv[1])`

2. Discuss various output formatting techniques in Python.

1. **Using `print()` with Commas**

- Automatically inserts spaces between items.

- Converts data types automatically.

```python
name = "Alice"

age = 25

print("Name:", name, "Age:", age)
```

2. **String Concatenation**

- Uses `+` to join strings.
```

- Requires manual type conversion for non-strings.

```python
name = "Alice"
age = 25
print("Name: " + name + ", Age: " + str(age))
```

3. String Formatting using `%` Operator (Old Style)

```python
name = "Alice"
age = 25
print("Name: %s, Age: %d" % (name, age))
```

4. Using `str.format()` Method

- More readable and flexible.
- Supports positional and keyword arguments.

```python
name = "Alice"
age = 25
print("Name: {}, Age: {}".format(name, age))
```

5. Using f-Strings (Python 3.6+)

- Cleanest and most modern method.
- Embeds expressions directly inside string literals.

```python
name = "Alice"
age = 25
```

```
print(f"Name: {name}, Age: {age}")
```

3. Difference Between `while` Loop and `for` Loop in Python

Python provides two main types of loops: `while` and `for`. Both are used for iteration, but they are suited to different situations.

VS `while` Loop vs `for` Loop

Feature	`while` Loop	`for` Loop
Usage	Used when the number of iterations is unknown	Used when the number of iterations is known
Condition-based	Runs as long as a condition is True	Iterates over a sequence (list, tuple, etc.)
Loop Control	Must manually update loop variable	Automatically handles iteration
Best For	Infinite loops, conditional processing	Iterating over collections or ranges
Risk	Risk of infinite loop if condition isn't updated	Less risk of infinite loop

Example: Using a `while` Loop

```
# Print numbers from 1 to 5 using while loop
i = 1
while i <= 5:
    print(i)
    i += 1
```

Example: Using a `for` Loop

```
# Print numbers from 1 to 5 using for loop
for i in range(1, 6):
```

```
    print(i)
```

Example: Iterating Through a List

Using `for` Loop

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

for fruit in fruits:

    print(fruit)
```

Using `while` Loop

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

i = 0

while i < len(fruits):

    print(fruits[i])

    i += 1
```

4. if...else Conditional Statement in Python

The `if...else` statement is used to perform conditional execution of code blocks in Python.

Basic Syntax

```
if condition:

    # block of code if condition is True

else:

    # block of code if condition is False
```

Example

```
num = 5

if num > 0:

    print("The number is positive.")

else:

    print("The number is not positive.")
```

5. Highlight the differences among break, continue, and pass statements in Python.

Statement	Purpose	Effect
`break`	To exit loop completely	Terminates the loop and transfers control to the statement following the loop
`continue`	To skip current iteration	Skips the rest of the code inside the current iteration and moves to the next iteration
`pass`	To create empty code blocks	Does nothing; acts as a placeholder

Examples:

Break Statement

```
for i in range(10):

    if i == 5:

        break

    print(i)  # Prints 0, 1, 2, 3, 4
```

Continue Statement

```
for i in range(10):

    if i == 5:
```

```
        continue
    print(i)  # Prints 0, 1, 2, 3, 4, 6, 7, 8, 9
```

Pass Statement

```
for i in range(5):
    if i == 3:
        pass  # Does nothing
    print(i)  # Prints 0, 1, 2, 3, 4
```

Python Unit-6 Notes

1. Basic Arithmetic Operations in Python

Python supports all standard arithmetic operations found in mathematics, making it powerful for numerical computations.

- **Addition**: Combines values on either side of the operator
- **Subtraction**: Subtracts right operand from left operand
- **Multiplication**: Multiplies values on either side of the operator
- **Division**: Divides left operand by right operand (returns float)

Syntax:

```
x + y    # Addition
x - y    # Subtraction
x * y    # Multiplication
x / y    # Division (float result)
x // y   # Floor division (integer result)
x % y    # Modulus (remainder)
x ** y   # Exponentiation (power)
```

Example:

```
a = 10
b = 3

print(f"Addition: {a} + {b} = {a + b}")           # Output: 13
print(f"Subtraction: {a} - {b} = {a - b}")        # Output: 7
print(f"Multiplication: {a} * {b} = {a * b}")     # Output: 30
print(f"Division: {a} / {b} = {a / b}")           # Output: 3.3333...
print(f"Floor Division: {a} // {b} = {a // b}")   # Output: 3
print(f"Modulus: {a} % {b} = {a % b}")            # Output: 1
print(f"Exponentiation: {a} ** {b} = {a ** b}")   # Output: 1000
```

2. Precedence of Arithmetic Operators in Python

Operator precedence determines the order in which operations are performed in an expression with multiple operators.

- **Highest to Lowest Precedence**: Python follows PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction)
- **Equal Precedence**: Operations with equal precedence are evaluated from left to right
- **Parentheses**: Can be used to override default precedence order
- **Operator Associativity**: Most operators are left-associative (except exponentiation which is right-associative)

Precedence Order:

```
1. Parentheses ()
2. Exponentiation **
3. Unary plus +x, Unary minus -x
4. Multiplication *, Division /, Floor Division //, Modulus %
5. Addition +, Subtraction -
```

Example:

```
# Without parentheses - follows precedence rules
result1 = 2 + 3 * 4
print(f"2 + 3 * 4 = {result1}")  # Output: 14 (multiplication first)

# With parentheses - overrides precedence
result2 = (2 + 3) * 4
print(f"(2 + 3) * 4 = {result2}")  # Output: 20

# Complex example
```

```
result3 = 2 ** 3 * 4 + 5 // 2
print(f"2 ** 3 * 4 + 5 // 2 = {result3}")  # Output: 34 (2^3=8, 8*4=32, 5//2=2, 32+2=34)
```

3. Significance of Python's Indentation in Code Structure

Unlike many programming languages that use braces or keywords to define blocks of code, Python uses indentation for code structure.

- **Block Definition**: Indentation defines blocks of code belonging to control structures like loops, functions, and conditionals
- **Consistency Requirement**: All statements within the same block must have the same indentation level
- **Error Prevention**: Improper indentation results in IndentationError, enforcing clean code structure
- **Visual Clarity**: Makes code more readable by visually representing its logical structure

Example:

```
# Example showing the importance of indentation
def calculate_total(items):
    total = 0               # Indented - part of function block
    for item in items:      # Indented - part of function block
        total += item       # Double indented - part of loop block
        print(item)         # Double indented - part of loop block
    return total            # Indented - part of function block

prices = [10, 20, 30]
print(calculate_total(prices))  # Not indented - not part of function

# If the return statement was not indented correctly:
# def incorrect_function():
#     if True:
#         print("Inside if block")
# return "Done"  # IndentationError: unexpected unindent
```

4. Python Shell for Code Execution and Experimentation

The Python shell (also called REPL - Read-Evaluate-Print Loop) provides an interactive environment for Python code execution.

- **Immediate Feedback**: Executes code immediately and shows results, allowing quick testing of ideas
- **Variable Persistence**: Variables defined in a session remain available until the shell is closed
- **Exploratory Learning**: Ideal for exploring Python features, testing small code snippets, and learning
- **Documentation Access**: Provides built-in help through functions like `help()` and `dir()`

Ways to Access Python Shell:

1. Command line: Type 'python' or 'python3' in terminal/command prompt
2. IDLE: Python's integrated development environment
3. IPython: Enhanced interactive shell with additional features
4. Jupyter Notebooks: Web-based interactive environment combining code and documentation

Example Session:

```
# A typical Python shell session might look like this:
>>> x = 10
>>> y = 20
>>> x + y
30
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> dir(str)  # Shows all methods available for strings
['__add__', '__class__', ..., 'upper', 'zfill']
>>> help(str.upper)  # Shows documentation for the upper method
```

5. Atoms, Identifiers, and Keywords in Python

These fundamental elements form the basic building blocks of Python syntax.

- **Atoms**: The smallest units in Python that cannot be broken down further (literals, identifiers, operators)
- **Identifiers**: Names given to entities like variables, functions, classes, etc.
- **Keywords**: Reserved words in Python that have special meanings and cannot be used as identifiers

- **Distinction**: While identifiers are named by programmers, keywords are predefined by the language

Python Keywords:

```
# Python 3.9+ has 35 keywords including:
False, None, True, and, as, assert, async, await, break, class, continue,
def, del, elif, else, except, finally, for, from, global, if, import, in,
is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield
```

Example:

```
# Keywords vs Identifiers
import keyword
print(keyword.kwlist)  # Prints all Python keywords

# Valid identifiers
user_name = "John"     # Using underscore
count1 = 10            # Using numbers (not at beginning)
_private = True        # Starting with underscore

# Invalid identifiers (would cause errors if uncommented)
# 2variable = "Invalid"  # Cannot start with a number
# for = "Invalid"        # Cannot use keyword as identifier
# my-var = 5             # Cannot use hyphen in identifier
```

6. Valid Identifiers and Rules for Naming

Identifiers are names used to identify variables, functions, classes, or other objects in Python.

- **Character Requirements**: Must start with a letter (a-z, A-Z) or underscore (_), followed by letters, digits (0-9), or underscores
- **Case Sensitivity**: Python identifiers are case-sensitive (`variable` and `Variable` are different)
- **Reserved Words**: Cannot use Python keywords as identifiers
- **Naming Conventions**: Follow PEP 8 style guide for consistent, readable code

Rules Summary:

```
1. Start with a letter or underscore
2. Cannot start with a digit
3. Can only contain alphanumeric characters and underscores (A-z, 0-9, _)
```

```
4. Cannot use Python keywords
5. Cannot use special symbols (@, #, $, %, etc.)
```

Example:

```
# Valid identifiers with proper naming conventions
student_name = "Alice"      # snake_case for variables (PEP 8
recommendation)
_internal_use = True        # Leading underscore for "private" variables
MAX_SIZE = 100              # ALL_CAPS for constants
class StudentRecord:        # PascalCase for classes
    pass
def calculate_average():    # snake_case for functions
    pass

# Valid but not following conventions (avoid these)
StudentName = "Bob"         # PascalCase should be for classes, not
variables
calculatemean = 3.14        # Missing underscores makes it less readable

# Identifiers showing case sensitivity
age = 25
Age = 30
AGE = 35
print(age, Age, AGE)        # Output: 25 30 35 (three different variables)
```

7. Types of Literals in Python

Literals are notations for representing fixed values in source code. Python supports various types of literals.

- **Numeric Literals**: Include integers, floating-point numbers, and complex numbers
- **String Literals**: Text enclosed in single, double, or triple quotes
- **Boolean Literals**: `True` and `False` values
- **Special Literals**: The `None` value represents absence of value or null

Common Literals:

```
# Numeric literals
integer_literal = 123
float_literal = 45.67
complex_literal = 3 + 4j
binary_literal = 0b1010     # Binary (10 in decimal)
octal_literal = 0o17        # Octal (15 in decimal)
hex_literal = 0xFF          # Hexadecimal (255 in decimal)
```

```
# String literals
single_quoted = 'Python'
double_quoted = "Python"
triple_quoted = '''Multiple
line string'''
raw_string = r"Raw\nString"  # Backslashes not interpreted

# Boolean literals
true_value = True
false_value = False

# None literal
empty_value = None
```

Example:

```
# Demonstrating different literals
# Numeric
print(123)               # Integer literal
print(3.14159)           # Float literal
print(0xFF)              # Hex literal (255)

# String
print("Hello")           # Basic string
print(r"Path\to\file")   # Raw string (backslashes preserved)
print("""This is a
multiline string""")     # Triple-quoted multiline string

# Boolean and None
print(True)              # Boolean literal
print(None)              # None literal

# Collection literals
print([1, 2, 3])         # List literal
print((1, 2, 3))         # Tuple literal
print({1, 2, 3})         # Set literal
print({"a": 1, "b": 2})  # Dictionary literal
```

8. Strings in Python: Characteristics and Operations

Strings are immutable sequences of characters used to store and manipulate text data.

- **Immutability**: Once created, string contents cannot be changed

- **Indexing and Slicing**: Characters can be accessed by position using indices
- **Rich Methods**: Python provides numerous built-in methods for string manipulation
- **Concatenation**: Strings can be combined using the `+` operator or join methods

Common String Operations:

```
# String creation
string_var = "Python Programming"

# Access (indexing and slicing)
char = string_var[0]       # First character 'P'
substring = string_var[0:6]  # Characters from 0 to 5 'Python'

# Common methods
upper_case = string_var.upper()  # Convert to uppercase
lower_case = string_var.lower()  # Convert to lowercase
split_words = string_var.split()  # Split into list ['Python',
'Programming']
replaced = string_var.replace('P', 'J')  # Replace 'P' with 'J'

# Checking content
contains = 'Python' in string_var  # True
starts = string_var.startswith('P')  # True
ends = string_var.endswith('ing')  # True

# Finding and counting
position = string_var.find('Pro')  # Returns 7
count = string_var.count('P')  # Returns 2
```

Example:

```
# Comprehensive string manipulation example
text = "Python is powerful"

# Basic operations
print(f"Original: {text}")
print(f"Length: {len(text)}")
print(f"First character: {text[0]}")
print(f"Last character: {text[-1]}")
print(f"Slicing (2-8): {text[2:8]}")

# String methods
print(f"Uppercase: {text.upper()}")
print(f"Title case: {text.title()}")
```

```
print(f"Replace: {text.replace('powerful', 'amazing')}")
print(f"Split: {text.split()}")

# String formatting
name = "Python"
version = 3.9
print(f"{name} version {version} is the latest release")
print("{} version {} is the latest release".format(name, version))

# String checking
print(f"Contains 'is': {'is' in text}")
print(f"Starts with 'Py': {text.startswith('Py')}")
print(f"Count of 'p': {text.count('p')}")
```

9. Categories of Operators in Python

Python provides various types of operators for different operations like arithmetic, comparison, logical, etc.

- **Arithmetic Operators**: Perform mathematical operations (+, -, *, /, //, %, **)
- **Comparison Operators**: Compare values and return boolean results (==, !=, >, <, >=, <=)
- **Logical Operators**: Combine boolean expressions (and, or, not)
- **Assignment Operators**: Assign values to variables (=, +=, -=, *=, etc.)
- **Bitwise Operators**: Perform bit-level operations (&, |, ^, ~, >>, <<)
- **Identity Operators**: Compare object identity (is, is not)
- **Membership Operators**: Test if a value exists in a sequence (in, not in)

Two Detailed Categories:

Comparison Operators

Used to compare values and return boolean results.

Syntax:

```
x == y    # Equal to
x != y    # Not equal to
x > y     # Greater than
x < y     # Less than
x >= y    # Greater than or equal to
x <= y    # Less than or equal to
```

Example:

```
# Comparison operators example
a = 10
b = 20

print(f"{a} == {b}: {a == b}")  # False
print(f"{a} != {b}: {a != b}")  # True
print(f"{a} > {b}: {a > b}")    # False
print(f"{a} < {b}: {a < b}")    # True
print(f"{a} >= {b}: {a >= b}")  # False
print(f"{a} <= {b}: {a <= b}")  # True

# Comparing different types
print(f"5 == '5': {5 == '5'}")          # False
print(f"'apple' < 'banana': {'apple' < 'banana'}")  # True (alphabetical)
print(f"[1, 2] == [1, 2]: {[1, 2] == [1, 2]}")      # True
print(f"[1, 2] is [1, 2]: {[1, 2] is [1, 2]}")      # False (different
objects)
```

Logical Operators

Used to combine conditional statements and determine the logic between variables or values.

Syntax:

```
x and y   # True if both x and y are true
x or y    # True if either x or y is true
not x     # True if x is false (inverts the boolean value)
```

Example:

```
# Logical operators example
x = True
y = False

print(f"{x} and {y}: {x and y}")  # False
print(f"{x} or {y}: {x or y}")    # True
print(f"not {x}: {not x}")        # False
print(f"not {y}: {not y}")        # True

# With conditions
age = 25
income = 50000
is_eligible = age > 18 and income >= 30000
print(f"Eligible for loan: {is_eligible}")  # True
```

```
# Short-circuit evaluation
result1 = False and print("This won't print")  # Short-circuits, doesn't
print
result2 = True or print("This won't print")    # Short-circuits, doesn't
print
```

Python Unit-5 Concise Notes

1. Structure of a Python Program

A Python program typically follows this structure:

1. **Module Documentation**: Comments at the beginning that explain what the program does
2. **Import Statements**: Lines that bring in external modules and libraries
3. **Global Variables/Constants**: Values defined at the top level of the program
4. **Function/Class Definitions**: Reusable blocks of code that perform specific tasks
5. **Main Execution Point**: Often uses `if __name__ == "__main__":` to define the entry point

Simple Example:

```
"""This program calculates the area of a circle."""

# Imports
import math

# Constants
PI = 3.14159

# Functions
def calculate_area(radius):
    return PI * radius * radius

# Main execution
if __name__ == "__main__":
    r = float(input("Enter radius: "))
    area = calculate_area(r)
    print(f"Area: {area}")
```

2. Elements of Python Programming Language

Python consists of these fundamental elements:

1. **Lexical Elements**:
   - Keywords (reserved words like `if`, `for`, `def`)
   - Identifiers (names of variables, functions)
   - Literals (fixed values like numbers, strings)
   - Operators (+, -, *, etc.)
   - Indentation (defines code blocks)
2. **Data Types**:
   - Numbers (int, float, complex)
   - Strings (text)
   - Lists, tuples (sequences)
   - Dictionaries (key-value pairs)
   - Sets, Boolean values
3. **Control Structures**:
   - Conditional statements (if-elif-else)
   - Loops (for, while)
   - Exception handling (try-except)
4. **Functions and Modules**:
   - Function definitions and calls
   - Module imports
   - Package organization

Example:

```
# Basic elements in action
age = 25                          # Variable assignment
name = "John"                     # String literal
if age >= 18:                     # Conditional with comparison operator
    print(f"{name} is an adult") # Function call with formatted string
else:
    print(f"{name} is a minor")
```

3. Python and Its Significance in Programming

Python's importance in the programming world comes from:

1. **Readability and Simplicity**: Clean syntax that resembles English makes it easy to learn and use
2. **Versatility**: Used in diverse fields including:

- Web development
- Data analysis
- Machine learning
- Scientific computing
- Automation
- Game development

3. **Rich Ecosystem**: Extensive standard library and thousands of third-party packages for almost any task
4. **Community Support**: Large, active community of developers creating resources and providing help

Popular Applications:

```
# Web development (using Flask)
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, World!"

# Data analysis (using Pandas)
import pandas as pd
data = pd.read_csv("data.csv")
average = data["Value"].mean()
```

4. Functionality of a Python Interpreter

The Python interpreter is responsible for:

1. **Code Execution**: Translates Python code into bytecode and executes it line by line
2. **Memory Management**: Automatically allocates and frees memory (garbage collection)
3. **Interactive Mode**: Provides a REPL (Read-Evaluate-Print Loop) for testing code interactively
4. **Module Management**: Handles importing and loading of external modules

Interpreter Types:

- CPython (standard)
- PyPy (faster for long-running programs)
- Jython (Java-based)

- IronPython (.NET-based)

Interactive Mode Example:

```
# In the Python shell/REPL:
>>> x = 10
>>> y = 20
>>> x + y
30
>>> for i in range(3):
...     print(i)
...
0
1
2
```