
Q1. Differentiate between Compiler and Interpreter

A **compiler** and an **interpreter** are both language translators that convert high-level programming code into machine language which the computer can understand.

Although they perform the same basic task, their **working process** and **execution method** are quite different.

A **compiler** reads the entire source program as a whole, analyzes all statements together, and then translates the entire code into machine code (object code). If the program contains errors, the compiler lists all the errors after the complete scanning is done. Once the code is error-free and compiled successfully, an executable file (for example, a `.exe` file) is created. The compiled program can then be executed multiple times without recompilation, which makes it very fast at runtime.

Languages such as **C**, **C++**, and **Java** use compilers.

An **interpreter**, on the other hand, translates the source code **line by line**. It reads one line, converts it into machine code, executes it immediately, and then moves on to the next line. If an error occurs, the interpreter stops execution at that line and displays the error message right away. This makes interpreters very helpful during program development and testing because errors can be detected and corrected immediately.

However, interpreted programs run more slowly since the translation happens every time the program executes. **Python**, **JavaScript**, and **Ruby** are examples of interpreted languages.

Hence, in summary, the main difference lies in how they execute the program: the compiler translates all at once, while the interpreter works line by line.

Tabular Comparison

Basis of Difference	Compiler	Interpreter
Translation method	Translates the entire program at once into machine code.	Translates and executes code line by line.
Error detection	Displays all errors after compilation.	Displays errors one by one as soon as they occur.
Execution speed	Faster during execution because it runs the compiled file directly.	Slower since every line is re-translated each time.
Output file	Produces a separate machine-code file (e.g., <code>.exe</code>).	Does not produce any separate file.
Memory usage	Uses more memory.	Uses comparatively less memory.
Examples of languages	C, C++, Java	Python, JavaScript, PHP

Conclusion:

Both compiler and interpreter are essential tools in programming. A compiler is better suited for final product development because of its speed, while an interpreter is ideal for learning, debugging, and scripting because of its flexibility and real-time feedback.

Q2. Describe the term debugging. List some of the debugging tools.

Answer:

Debugging is the process of identifying, analyzing, and fixing errors or bugs in a computer program. When a program doesn't run as expected or produces incorrect results, debugging helps locate the cause of the problem and correct it. It is an essential step in software development that ensures the program runs smoothly and efficiently.

Debugging can involve checking logic errors, syntax errors, or runtime errors. Developers often use debugging tools to track variable values, inspect memory, and analyze the flow of control during program execution.

Common Debugging Tools:

1. **Python IDLE Debugger** – Comes built-in with Python, helps trace code step by step.
2. **PDB (Python Debugger)** – A command-line debugger that allows breakpoints, step execution, and variable inspection.
3. **PyCharm Debugger** – Part of the PyCharm IDE, supports advanced features like watches, breakpoints, and variable tracking.
4. **Visual Studio Code Debugger** – Provides an interactive debugging interface with visual feedback.

Example:

If a program gives wrong output for addition, a debugger helps track the logic and correct mistakes like using = instead of +.

Q3. Illustrate while statement with syntax and Python code.

Answer:

The `while` loop in Python is used to execute a block of code repeatedly as long as the given condition is `True`. When the condition becomes `False`, the loop terminates.

Syntax:

```
while condition:
    # statements
```

Explanation:

- The condition is checked before executing the block.
- If the condition is true, the statements inside the loop run.
- After each iteration, the condition is rechecked.

Example:

```
i = 1
while i <= 5:
    print("Number:", i)
    i += 1
```

Output:

```
Number: 1
```

```
Number: 2
Number: 3
Number: 4
Number: 5
```

This loop prints numbers from 1 to 5.

Q4. Write Python instructions to demonstrate tables and two-dimensional tables.

Answer:

A table can be created using nested loops in Python. A one-dimensional table is a single list, while a two-dimensional table is represented using lists inside lists.

Example – Multiplication Table (1D):

```
n = 5
for i in range(1, 11):
    print(n, "x", i, "=", n*i)
```

Example – Two-Dimensional Table:

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i * j, end="\t")
    print()
```

Output:

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

This creates a 5x5 multiplication table.

Q5. Develop a Python program to read 2 numbers from the keyboard and perform basic arithmetic operations based on choice.

Answer:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

print("1.Addition\n2.Subtraction\n3.Multiplication\n4.Division")
choice = int(input("Enter your choice (1-4): "))

if choice == 1:
    print("Sum =", a + b)
elif choice == 2:
    print("Difference =", a - b)
elif choice == 3:
    print("Product =", a * b)
elif choice == 4:
    print("Quotient =", a / b)
else:
```

```
print("Invalid choice")
```

Explanation:

This program takes two inputs from the user and performs an operation based on their choice using an `if-elif-else` structure.

Q6. Differentiate between syntax errors and run-time errors.

Basis	Syntax Error	Run-time Error
Definition	Occurs when code violates Python's syntax rules.	Occurs while the program is running due to invalid operations.
Detection Time	Detected before execution (during compilation).	Detected during execution.
Example	<code>print "Hello"</code> (missing parentheses)	Division by zero: <code>10/0</code>
Correction	Fixed by correcting syntax.	Fixed by handling exceptions.

Q7. Demonstrate the syntax of for construct in Python with example.**Answer:**

The `for` loop in Python is used to iterate over a sequence such as a list, tuple, string, or range.

Syntax:

```
for variable in sequence:
    # statements
```

Example:

```
for i in range(1, 6):
    print("Number:", i)
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

The loop iterates 5 times and prints the current value of `i` in each iteration.

Q8. Define Python programming language. Explain syntax errors and runtime errors.**Answer:**

Python is a high-level, interpreted, and general-purpose programming language developed by Guido van Rossum. It is known for its simplicity, readability, and vast library support. Python supports both object-oriented and procedural programming.

Syntax Errors:

These occur when the programmer violates the grammar or structure of the language. For example:

```
if x == 10
    print(x)
```

Here, the colon (:) is missing — it causes a syntax error.

Runtime Errors:

These occur while the program is executing. Example:

```
x = 5
y = 0
print(x / y)
```

Division by zero causes a runtime error.

Perfect 🍌 Let's continue from **Q9 to Q24**, written in **exam-style, easy to understand, detailed answers** with examples wherever needed.

Q9. What are variables? Differentiate between variables and keywords with example.

Answer:

A **variable** in Python is a name given to a memory location that stores data. It is used to hold values that can be changed or updated during program execution. Variables act as containers for data.

Example:

```
name = "Ayush"
age = 20
```

Here, name and age are variables.

Difference between Variables and Keywords:

Basis	Variables	Keywords
Definition	User-defined names that store data.	Reserved words used by Python for special purposes.
Example	age, name, total	if, while, for, def
Count	Unlimited; user can create as many as needed.	Fixed; Python has around 33 keywords.
Modifiable	Variables can change value.	Keywords cannot be used or modified.

Q10. Write a Python program to illustrate break statement.

Answer:

The `break` statement is used to exit a loop prematurely when a specific condition is met.

Example:

```
for i in range(1, 10):
    if i == 6:
        break
```

```
print(i)
```

Output:

```
1
2
3
4
5
```

Explanation:

When `i` becomes 6, the `break` statement stops the loop even though the range is up to 10.

Q11. Develop a function to print multiplication table 2 to 5 using any iteration construct.

Answer:

```
def table():
    for i in range(2, 6):
        print(f"\nMultiplication Table of {i}")
        for j in range(1, 11):
            print(i, "x", j, "=", i*j)

table()
```

Explanation:

This function prints tables for numbers 2 to 5 using nested `for` loops. The outer loop changes the number, and the inner loop prints each multiplication line.

Q12. List the operators supported by Python and give the precedence of those operators.

Answer:

Operators are special symbols that perform operations on operands.

Types of Operators in Python:

1. **Arithmetic Operators:** `+`, `-`, `*`, `/`, `%`, `**`, `//`
2. **Relational Operators:** `>`, `<`, `>=`, `<=`, `==`, `!=`
3. **Logical Operators:** `and`, `or`, `not`
4. **Assignment Operators:** `=`, `+=`, `-=`, `*=`, `/=`
5. **Bitwise Operators:** `&`, `|`, `^`, `~`, `<<`, `>>`
6. **Membership Operators:** `in`, `not in`
7. **Identity Operators:** `is`, `is not`

Operator Precedence (Highest to Lowest):

Precedence Level	Operators
1	<code>**</code> (Exponentiation)
2	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>
3	<code>+</code> , <code>-</code>
4	<code><</code> , <code>></code> , <code><=</code> , <code>>=</code>

Precedence Level	Operators
------------------	-----------

5	==, !=
6	and, or, not
7	+=, -=

Q13. Describe the term variable. List all the rules to be followed while defining a variable.

Answer:

A **variable** is a name that refers to a memory location where data is stored. It allows programs to store, retrieve, and manipulate data values.

Rules for Defining Variables:

1. Variable names must start with a letter or underscore (_).
2. They cannot start with a number.
3. Only letters, digits, and underscores are allowed.
4. Variable names are **case-sensitive** (Age and age are different).
5. Keywords cannot be used as variable names.
6. Variables should have meaningful names.

Example:

```
name = "Ayush"  
_age = 20  
roll_number = 101
```

Q14. Explain the syntax of while construct in Python with example.

Answer:

The **while** loop repeatedly executes a block of code as long as the condition is true.

Syntax:

```
while condition:  
    # statements
```

Example:

```
i = 1  
while i <= 5:  
    print("Count:", i)  
    i += 1
```

Output:

```
Count: 1  
Count: 2  
Count: 3  
Count: 4  
Count: 5
```

Here, the loop runs until the condition `i <= 5` becomes false.

Q15. With suitable examples demonstrate the working of concatenation and repetition operators on strings.

Answer:

- **Concatenation (+):** Joins two or more strings.
- **Repetition (*):** Repeats a string a given number of times.

Example:

```
s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)      # Concatenation
print(s1 * 3)             # Repetition
```

Output:

```
Hello World
HelloHelloHello
```

Q16. Differentiate between high-level and machine-level languages.

Basis	High-Level Language	Machine-Level Language
Definition	Uses human-readable syntax (like Python, C).	Uses binary code (0s and 1s).
Ease of Use	Easy to learn and understand.	Difficult for humans to read or write.
Execution	Requires compiler/interpreter.	Directly executed by the CPU.
Portability	Portable between systems.	Not portable; depends on hardware.
Example	Python, Java, C++	Assembly or binary code

Q17. Describe variables and keywords in Python programming.

Answer:

- **Variables:** Are identifiers used to store data values. For example, `x = 10`, where `x` is a variable.
- **Keywords:** Are reserved words that have predefined meanings in Python. They cannot be used as variable names.

Example of Keywords:

`if, for, while, class, def, import, return.`

There are about **33 keywords** in Python.

Q18. Explain type conversion functions with examples.

Answer:

Type conversion means changing the data type of a value from one form to another. Python supports several in-built functions for this purpose.

Common Type Conversion Functions:

1. `int()` – Converts to integer
2. `float()` – Converts to float
3. `str()` – Converts to string
4. `list()` – Converts to list
5. `tuple()` – Converts to tuple

Example:

```
x = "10"
y = int(x) + 5
print(y)
```

Output:

15

Q19. Demonstrate the working of break and continue statements with suitable example.

Answer:

- **break:** Exits the loop when a condition is met.
- **continue:** Skips the remaining code in the current iteration and moves to the next.

Example:

```
for i in range(1, 6):
    if i == 3:
        continue
    if i == 5:
        break
    print(i)
```

Output:

1
2
4

Explanation: When `i=3`, `continue` skips printing; when `i=5`, `break` ends the loop.

Q20. Describe the term function. Develop a function to print even numbers from 1 to 50 using for loop.

Answer:

A **function** is a reusable block of code that performs a specific task. It helps reduce repetition and makes programs modular.

Example:

```
def even_numbers():
    for i in range(1, 51):
        if i % 2 == 0:
            print(i)

even_numbers()
```

Output:

Prints even numbers from 1 to 50.

Q21. Describe the Collatz sequence. Implement Collatz sequence for value 78.**Answer:**

The **Collatz sequence** is defined as:

- If the number is even → divide it by 2
 - If the number is odd → multiply by 3 and add 1
- The sequence continues until it reaches 1.

Program:

```
n = 78
while n != 1:
    print(n, end=" ")
    if n % 2 == 0:
        n = n // 2
    else:
        n = 3 * n + 1
print(1)
```

Q22. Develop a program to implement Collatz sequence for number 96 and print max sequence length, number of odd and even numbers.**Answer:**

```
n = 96
count = 0
odd = 0
even = 0

while n != 1:
    print(n, end=" ")
    count += 1
    if n % 2 == 0:
        even += 1
        n = n // 2
    else:
        odd += 1
        n = 3 * n + 1
print(1)
print("\nTotal length:", count)
print("Odd numbers:", odd)
print("Even numbers:", even)
```

Q23. List different types of errors that a software program might produce.**Answer:**

Errors are problems that stop a program from running correctly. Major types include:

1. **Syntax Errors** – Occur when Python syntax rules are violated.
Example: Missing colon `if x == 5`

2. **Runtime Errors** – Occur during execution.
Example: Division by zero.
 3. **Logical Errors** – Program runs but gives wrong output due to incorrect logic.
Example: Using + instead of *.
 4. **Semantic Errors** – Occur when statements are grammatically correct but not meaningful.
-

Q24. List and explain any two data types supported by Python language.

Answer:

Python supports several built-in data types. Two common ones are:

1. **Integer (`int`)** – Represents whole numbers, positive or negative, without decimals.
Example: `x = 10`
2. **String (`str`)** – Represents a sequence of characters enclosed in quotes.
Example: `name = "Ayush"`

Other data types include `float`, `list`, `tuple`, `dict`, and `set`.

MODULE 2

Great — below are **Module-2** answers, all in one place, written in an **exam/viva style**: clear paragraphs, detailed explanations, and working Python examples for each question. I numbered each question to match yours. Where code is shown you can copy–paste it into a Python file or REPL and run it.

Module-2 — Strings, Lists, Tuples (Questions & Long Answers)

1. Describe the term strings. Apply `len()` function and indexing operators on strings.

Answer (theory + explanation):

A **string** is a sequence of characters used to represent text. In Python a string is an object of type `str`. Strings are *ordered* (each character has an index) and *immutable* (once created they cannot be changed). A string may contain letters, digits, spaces and special characters. Strings can be written with single quotes `'...'`, double quotes `"..."` or triple quotes `'''...'''` / `"""..."""` for multi-line text.

The built-in function `len(s)` returns the number of characters in string `s`. Characters in a string are accessed using square-bracket indexing: `s[i]` gives the character at index `i`. Python uses **0-based** positive indices (0 is first character) and **negative indices** (-1 is last character).

Example:

```
s = "Python"
print(len(s))      # 6
print(s[0])        # 'P'    (first char)
print(s[2])        # 't'    (third char)
print(s[-1])       # 'n'    (last char)
print(s[-2])       # 'o'    (second last)
```

Notes for exam: mention immutability and indexing rules; show both positive and negative indexing.

2. Demonstrate string comparison and use of `in` and `not in` operators on strings.

Answer (theory + examples):

String comparison uses relational operators such as `==`, `!=`, `<`, `>`, `<=`, `>=`. Comparison is **lexicographic** (dictionary order) and is case-sensitive: `'Apple' != 'apple'`. The membership operators `in` and `not in` test whether a substring exists in a string (returning `True` or `False`).

Examples:

```
a = "apple"
b = "banana"
print(a == "apple")      # True
print(a != b)            # True
print("app" in a)        # True (substring present)
print("App" in a)        # False (case-sensitive)
print("cat" not in b)    # True
# Lexicographic comparison:
print("apple" < "banana") # True because 'a' < 'b'
```

Exam tip: explain lexicographic ordering and case sensitivity when comparing strings.

3. Develop a function that computes factorial of a given number.

Answer (theory + two implementations):

The factorial of a non-negative integer n (written $n!$) is the product of all positive integers $\leq n$. By definition $0! = 1$. Factorial grows quickly; implementations often use iteration or recursion.

Iterative version:

```
def factorial_iter(n):
    if n < 0:
        raise ValueError("Factorial not defined for negative numbers")
    result = 1
    for i in range(2, n+1):
        result *= i
    return result

print(factorial_iter(5)) # 120
```

Recursive version:

```
def factorial_rec(n):
    if n < 0:
        raise ValueError("Factorial not defined for negative numbers")
    if n == 0 or n == 1:
        return 1
    return n * factorial_rec(n-1)

print(factorial_rec(5)) # 120
```

Exam note: mention base case for recursion and error handling for negative input.

4. Describe the term string. Write a function to find the length of a string.

Answer (concise definition + function):

As above, a string is a sequence of characters. The length of a string is the number of characters it contains. Use `len()` to obtain length.

Function example:

```
def string_length(s):
    return len(s)

print(string_length("Python Programming")) # 18 (includes space)
```

Remark: you may implement a manual length function (loop counting) if asked to show algorithmic understanding.

Manual counting example:

```
def manual_len(s):
    count = 0
    for _ in s:
        count += 1
    return count
```

5. Enlist the differences between tuples and lists. Give the syntax to create objects of list and tuple.

Answer (theory + table + syntax):

Both lists and tuples are built-in sequence types in Python and can hold heterogeneous items. The key difference is **mutability**:

- **List:** mutable — you can change, add or remove elements. Use square brackets `[]`.
- **Tuple:** immutable — once created you cannot change its contents. Use parentheses `()` or omit parentheses in some contexts.

Differences (table):

Feature	List	Tuple
Mutability	Mutable (can modify)	Immutable (cannot modify)
Syntax	<code>lst = [1, 2, 3]</code> <code>t = (1, 2, 3)</code> or <code>t = 1, 2, 3</code>	
Methods	Many (append, extend, remove, pop, etc.)	Few (count, index)
Use case	Collections needing change	Fixed records, keys in dict (if elements hashable)

Feature	List	Tuple
Performance	Slightly slower	Slightly faster on iteration & memory

Examples:

```
# list
my_list = [10, "apple", 3.14]

# tuple
my_tuple = (10, "apple", 3.14)
# single element tuple:
single = (5,) # comma is required
```

Exam tip: show single-element tuple rule and explain immutability significance.

6. Develop the python program to explain the common string methods.

Answer (explain methods + illustrative program):

Python strings have many useful methods. Common ones include `.lower()`, `.upper()`, `.strip()`, `.split()`, `.join()`, `.replace()`, `.find()`, `.count()`, `.startswith()`, `.endswith()`.

Sample program demonstrating several methods:

```
s = " Hello, Python Programming! "
print("Original:", repr(s))

print("lower():", s.lower())
print("upper():", s.upper())
print("strip():", s.strip()) # removes leading/trailing whitespace
print("replace():", s.replace("Python", "PY"))
print("split():", s.split()) # splits on whitespace by default
print("find('Python'):", s.find("Python")) # index or -1 if not found
print("count('o'):", s.count('o')) # number of occurrences
print("startswith(' He'):", s.startswith(" He"))
print("endswith(' '):", s.endswith(" "))
print("join example:", "-".join(["a", "b", "c"]))
```

Exam remark: explain immutability — methods return new strings; original remains unchanged.

7. Describe the term string. Demonstrate the working any two string manipulation functions with sample code.

Answer (select two & explain):

String manipulation functions alter or derive new strings. Two commonly used ones:

1. `.replace(old, new)` — returns a new string with all occurrences of `old` replaced by `new`.
2. `.split(sep)` — splits the string into a list of substrings using `sep` (default whitespace).

Examples:

```
s = "apples, bananas, cherries"
# replace
s2 = s.replace("bananas", "oranges")
print(s2) # 'apples, oranges, cherries'
```

```
# split
parts = s.split(", ")
print(parts)  # ['apples', 'bananas', 'cherries']
```

Exam tip: mention returned types: `.replace()` returns `str`, `.split()` returns `list`.

8. Differentiate between list and tuple.

Answer (concise & exam style):

A **list** is an ordered, mutable collection of items defined using square brackets `[]`. Lists support insertion, deletion and modification of elements and provide many helper methods (`append`, `remove`, `pop`, `insert`, `sort`, etc.). For example, `arr = [1, 2, 3]`.

A **tuple** is an ordered, immutable collection defined using parentheses `()` or simply by commas. Once created, its elements cannot be changed. Tuples are typically used for heterogeneous read-only collections or to use as keys in dictionaries when elements are hashable. For example, `t = (1, 2, 3)`.

Key difference: mutability — lists can change, tuples cannot. Also tuples are slightly more memory efficient and safer when you want fixed data.

9. Given the following string "Python Programming", write a python script to print the number of ovals.

Interpretation & answer:

I assume “ovals” means **vowels** (common typo). To count vowels in "Python Programming" count letters a, e, i, o, u (case-insensitive).

Script:

```
s = "Python Programming"
vowels = set("aeiouAEIOU")
count = sum(1 for ch in s if ch in vowels)
print("Number of vowels:", count)
```

Explanation:

This code checks each character and increments the count if it is one of the vowels. For "Python Programming" the vowels are o, o, a, i → total 4.

Exam note: state you considered case and included both uppercase and lowercase vowels.

10. Demonstrate `len()` function, `for` loop, and `find()` function with an example.

Answer (combined demonstration):

`len()` returns string length; `for` can iterate over characters; `.find(sub)` returns the lowest index of substring `sub` or `-1` if not found.

Example:

```
s = "Hello World"
print("Length:", len(s))      # 11
```

```
# iterate characters with for
for i, ch in enumerate(s):
    print(i, ch)

# find substring
pos = s.find("World")
print("'World' starts at index:", pos) # 6

# find non-existing substring returns -1
print(s.find("Python")) # -1
```

Exam tip: also mention `.index()` which raises `ValueError` if substring not found (unlike `.find()`).

11. Demonstrate with suitable code various list comprehension techniques? Give an example for each.

Answer (explain & provide examples):

List comprehensions are concise, expressive constructs to create lists from iterables. They can include conditions and nested loops.

Examples:

1. Simple transformation:

```
squares = [x*x for x in range(1, 6)]
# [1, 4, 9, 16, 25]
```

2. With condition (filtering):

```
evens = [x for x in range(1, 21) if x % 2 == 0]
# [2, 4, 6, ..., 20]
```

3. Nested comprehension (flattening / pairs):

```
pairs = [(i, j) for i in range(1, 4) for j in range(1, 3)]
# [(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]
```

4. Conditional expression inside comprehension:

```
labels = ["even" if x%2==0 else "odd" for x in range(1, 6)]
# ['odd', 'even', 'odd', 'even', 'odd']
```

5. Apply function to items:

```
words = [" apple ", " banana ", " cherry "]
clean = [w.strip().upper() for w in words]
# ['APPLE', 'BANANA', 'CHERRY']
```

Exam note: show how list comprehensions replace `for` + `append` loops for brevity and readability.

12. Illustrate list operations and explain that lists are mutable with example.

Answer (theory + example showing mutation):

Lists are mutable sequences — you can change an element, append, extend, insert, remove or sort. Because they are mutable, references to the same list reflect updates.

Examples:

```
lst = [1, 2, 3]
lst.append(4)          # [1,2,3,4]
lst[1] = 20           # [1,20,3,4] # assignment to index modifies in-place
lst.insert(2, 15)      # [1,20,15,3,4]
lst.remove(3)          # [1,20,15,4]
x = lst.pop()          # removes and returns last element (4)
print(lst)             # [1,20,15]
```

Mutability demonstration (aliasing):

```
a = [1,2,3]
b = a                 # b references same list object
b.append(4)
print(a)              # [1,2,3,4] -> change via b is visible in a
```

Exam tip: explain aliasing effect and use `.copy()` to clone if separate objects needed.

13. Describe the sequential data types tuple and list. Demonstrate the creation of empty list and empty tuple with suitable examples for each.**Answer (theory + syntax):**

Sequential data types maintain an ordered sequence of elements accessible by index. Two common sequential types:

- **List** (`list`) — mutable ordered sequence.
- **Tuple** (`tuple`) — immutable ordered sequence.

Creating empty objects:

```
empty_list = []      # or list()
empty_tuple = ()     # or tuple()
```

Usage examples:

```
lst = []              # start with empty list and append
lst.append('a')       # ['a']

t = ()               # empty tuple cannot be appended (immutable)
```

Exam remark: highlight use cases: lists for collections needing modification; tuples for fixed records (like rows, coordinates).

14. With suitable example demonstrate any five methods of a list.**Answer (explain five methods with code):**

Common methods: `.append()`, `.extend()`, `.insert()`, `.pop()`, `.remove()`, `.sort()`, `.reverse()`. I'll demonstrate five.

```

lst = [3, 1, 4]

# append - add single element at end
lst.append(5)          # [3,1,4,5]

# extend - add elements from another iterable
lst.extend([9, 2])     # [3,1,4,5,9,2]

# insert - insert element at given index
lst.insert(1, 7)        # [3,7,1,4,5,9,2]

# remove - remove first occurrence by value
lst.remove(4)           # [3,7,1,5,9,2]

# pop - remove and return element by index (default last)
val = lst.pop()         # val=2; lst now [3,7,1,5,9]
print(val)

```

Notes: `.sort()` sorts in place, `.reverse()` reverses in place. Use `sorted(lst)` to get new sorted list.

15. Given the following string "Python Programming", write a python script to print the number of vowels.

Answer (script + explanation):

This repeats earlier vowel count; ensure both lowercase and uppercase vowels counted.

```

s = "Python Programming"
vowels = "aeiouAEIOU"
count = sum(1 for ch in s if ch in vowels)
print("Number of vowels:", count)  # output 4

```

Explanation: the script iterates characters and increments when a vowel is found. For "Python Programming" vowels are o, o, a, i.

16. Develop a program to count how many times an element appears in a tuple.

Answer (theory + examples):

Tuples have a `.count(x)` method that returns how many times `x` appears.

Example program:

```

t = (1, 2, 3, 2, 4, 2, 5)
x = 2
print(f"{x} appears {t.count(x)} times")  # 3

```

Manual counting alternative (if method not allowed):

```

def count_in_tuple(t, x):
    c = 0
    for el in t:
        if el == x:
            c += 1
    return c

```

17. Describe the working of any three methods supported by the list class using suitable example for each.

Answer (choose three with explanation):

1. **.append(x)** — adds x to end of list.
2. `lst = [1,2]`
3. `lst.append(3)` # `[1,2,3]`
4. **.pop([i])** — removes and returns element at index i; if i omitted removes last element.
5. `lst = [1,2,3]`
6. `last = lst.pop()` # `last=3, lst=[1,2]`
7. `second = lst.pop(1)` # `second=2, lst=[1]`
8. **.sort()** — sorts list in place (numbers or strings). Use `reverse=True` for descending.
9. `nums = [3,1,4,2]`
10. `nums.sort()`
11. # `nums` now `[1,2,3,4]`
12. `nums.sort(reverse=True)`
13. # `nums` now `[4,3,2,1]`

Exam remark: distinguish between in-place methods (mutate original list) and functions that return new lists (e.g., `sorted()` returns a new list).

18. Create a tuple for the following 'Arjun', 20, "Bangalore Karnataka", 178392.89, 9976363728, arjun@gmail.com

Answer (create tuple & explain quoting):

Ensure the email is a string (must be quoted).

```
person = ('Arjun', 20, "Bangalore Karnataka", 178392.89, 9976363728, "arjun@gmail.com")
```

Explanation: tuple holds heterogeneous data types; useful for fixed records like a person's details.

19. Explain tuples with example. Illustrate with Python code tuple packing and tuple unpacking.

Answer (theory + packing/unpacking examples):

A tuple is an immutable ordered sequence. Tuple *packing* is creating a tuple by assigning multiple values to a single variable; *unpacking* extracts tuple elements into variables.

Examples:

```
# packing
t = 1, 2, 3           # t is (1,2,3)
t2 = (4, 5, 6)        # explicit

# unpacking
a, b, c = t
print(a, b, c)         # 1 2 3

# swapping values using packing/unpacking
x, y = 10, 20
x, y = y, x
# now x=20, y=10
```

Exam tip: show single-element tuple must have trailing comma: `single = (5,)`.

20. Develop Python code to demonstrate tuple assignment with tuple packing and tuple unpacking.

Answer (code + explanation):

```
# tuple packing
record = "Alice", 30, "Bengaluru"

# tuple unpacking (assignment)
name, age, city = record
print(name)    # Alice
print(age)     # 30
print(city)    # Bengaluru

# use-case: function returning multiple values
def stats(a, b):
    return a+b, a*b, a-b    # returns a tuple implicitly

sum_, prod, diff = stats(5, 3)
print(sum_, prod, diff)    # 8 15 2
```

Notes: tuple packing happens on return and multiple assignment extracts values directly.

21. Demonstrate list membership and list cloning with example.

Answer (membership `in` / cloning methods):

Membership:

```
fruits = ["apple", "banana", "mango"]
print("banana" in fruits)    # True
print("grape" not in fruits) # True
```

Cloning (creating independent copy):

- Using slice:
a = [1,2,3]
b = a[:] # shallow copy
- Using `list()`:
b = list(a)
- Using `.copy()` (Python 3.3+):
b = a.copy()

Important: these are *shallow* copies — nested mutable elements still share references. For deep copies use `copy.deepcopy()`.

22. Develop a Python program to demonstrate creation of string and lists objects.

Answer (simple demo program showing creation & basic ops):

```
# string creation and operations
s = "Hello, Python!"
```

```

print("String:", s)
print("Upper:", s.upper())
print("Slice:", s[7:13])    # 'Python'

# list creation and operations
lst = [10, 20, "apple", 3.14]
print("List:", lst)
lst.append("new")
print("After append:", lst)
print("First item:", lst[0])

```

Explanation: show how to create, index, slice, and perform basic methods.

23. Illustrate any 4 string slicing operations.

Answer (explain slicing `s[start:stop:step]` and show examples):

Slicing allows extracting a substring with `s[start:stop]` (start inclusive, stop exclusive) and optional step `s[start:stop:step]`.

Given `s = "PythonProgramming"`:

1. **From start to index (exclusive):**
2. `s[:6]` # 'Python' (start omitted => 0)
3. **From index to end:**
4. `s[6:]` # 'Programming'
5. **Middle slice:**
6. `s[6:11]` # 'Progr' (characters at indices 6..10)
7. **With step (every 2nd char):**
8. `s[::2]` # characters at indices 0,2,4,... e.g. 'Pto rgamn'
9. **Reverse string using step -1:**
10. `s[::-1]` # reversed string

Exam note: show examples and mention default values and negative steps.

24. What will be the output of the following program? Provide detailed explanation.

```

print("i\ti**2\ti**3\ti**5\ti**10\ti**20")
for i in range(1, 11):
    print(i, "\t", i**2, "\t", i**3, "\t", i**5, "\t", i**10, "\t", i**20)

```

Answer (step-by-step explanation and sample output):

Explanation:

- The first `print` prints a header showing column titles separated by tab characters `\t`. So the header line will be:
 - `i i**2 i**3 i**5 i**10 i**20`
- (tabs control spacing — actual alignment depends on console).
- The `for` loop iterates `i` from 1 to 10 inclusive (`range(1, 11)`).
 - For each `i` the `print` prints:
 - `i` (the number)
 - `i**2` (`i` squared)

- `i**3` (i cubed)
- `i**5` (i to the 5th power)
- `i**10` (i to the 10th power)
- `i**20` (i to the 20th power)
- Each value is separated by a tab `\t` (except the first which is printed with default spacing because the `print` call includes commas and explicit `"\t"` strings). The printed results will be a table of powers for numbers 1 through 10.

Sample partial output (values shown; huge numbers for high powers — I'll show first 3 rows and last row as representative):

i	<code>i**2</code>	<code>i**3</code>	<code>i**5</code>	<code>i**10</code>	<code>i**20</code>
1	1	1	1	1	1
2	4	8	32	1024	1048576
3	9	27	243	59049	3486784401
...					
10	100	1000	100000		100000000000 100000000000000000000

Notes on large numbers:

- `i**20` grows extremely large even for moderate `i`. Python supports big integers so it will print full integer values (no overflow).
- The alignment in console may not be perfectly column-aligned because numbers have different widths. Using formatted printing (like `f"{i:2d}"` or `print(f"{i}\t{i**2}\t...")`) and fixed-width formatting helps alignment.

Exam tip: explain that Python computes integer powers exactly, resulting in very large integers for `i**20`, and mention how many digits for e.g. `10**20` is 21 digits (1 followed by 20 zeros).

MODULE 3

Q1. Describe the term dictionary. Discuss the working of all the methods supported by the dictionary using an example for each.

Answer:

A **dictionary** in Python is a built-in data structure used to store data in **key–value pairs**. Each key in a dictionary is unique and is used to access its corresponding value. Unlike lists or tuples, dictionaries are **unordered and mutable**, meaning the order of elements is not fixed and values can be changed after creation.

Dictionaries are enclosed in **curly braces** `{}`, and each element is written as **key: value**.

Example:

```
student = {"name": "Arjun", "age": 20, "city": "Bangalore"}
```

Here, "name", "age", and "city" are keys, while "Arjun", 20, and "Bangalore" are their values.

Common Dictionary Methods:

1. **get(key, default)** – Returns the value of the specified key. If the key is not found, returns the default value.
2. `print(student.get("name"))` # Output: Arjun
3. `print(student.get("email", "Not Found"))` # Output: Not Found
4. **keys()** – Returns all keys of the dictionary.
5. `print(student.keys())` # Output: dict_keys(['name', 'age', 'city'])
6. **values()** – Returns all values in the dictionary.
7. `print(student.values())` # Output: dict_values(['Arjun', 20, 'Bangalore'])
8. **items()** – Returns all key–value pairs as tuples.
9. `print(student.items())` # Output:
dict_items([('name', 'Arjun'), ('age', 20), ('city', 'Bangalore')])
10. **update()** – Updates the dictionary with new key–value pairs.
11. `student.update({"email": "arjun@gmail.com"})`
12. **pop(key)** – Removes the item with the given key.
13. `student.pop("age")`
14. **clear()** – Removes all items from the dictionary.
15. `student.clear()`
16. **copy()** – Returns a shallow copy of the dictionary.
17. `new_dict = student.copy()`

Thus, dictionaries provide a flexible and efficient way to store and manage data in Python.

Q2. List and demonstrate any two ways of creating a dictionary with suitable example.

Answer:

There are two main ways to create a dictionary in Python:

1. **Using Curly Braces {}**
2. `student = {"name": "Arjun", "age": 21, "city": "Bangalore"}`
3. `print(student)`

Output:

```
{'name': 'Arjun', 'age': 21, 'city': 'Bangalore'}
```

4. **Using the dict() Constructor**
5. `student = dict(name="Arjun", age=21, city="Bangalore")`
6. `print(student)`

Output:

```
{'name': 'Arjun', 'age': 21, 'city': 'Bangalore'}
```

Both methods create the same result — a dictionary with key–value pairs.

Q3. Develop a program to check if a key exists in a dictionary.

Answer:

```
student = {"name": "Arjun", "age": 20, "course": "Python"}

key = input("Enter key to check: ")

if key in student:
    print("Key exists. Value is:", student[key])
else:
    print("Key does not exist.")
```

Explanation:

This program checks if the given key is present using the `in` operator. If found, it prints the corresponding value.

Q4. Discuss about Dictionaries and Dictionary methods with example.

Answer:

A **dictionary** is an unordered collection of items in which each element consists of a key and its associated value. It allows **fast lookups, updates, and deletions**. Dictionaries are written in `{ }` brackets, separating key–value pairs with colons.

Example:

```
fruits = {"apple": 50, "banana": 30, "grapes": 40}
```

Common Methods:

- `fruits.keys()` → returns all keys
- `fruits.values()` → returns all values
- `fruits.items()` → returns key–value pairs
- `fruits.pop("banana")` → removes “banana” key
- `fruits.update({"orange": 20})` → adds a new key

Dictionaries are very useful for representing data like records or mappings.

Q5. Demonstrate counting letters in dictionaries.

Answer:

```
text = "banana"
count = {}

for ch in text:
    count[ch] = count.get(ch, 0) + 1

print(count)
```

Output:

```
{'b': 1, 'a': 3, 'n': 2}
```

Explanation:

Each letter becomes a key, and its frequency (number of occurrences) becomes the value. The `get()` method ensures missing letters start from 0.

Q6. Illustrate aliasing and copying in dictionaries.

Answer:

Aliasing means assigning the same dictionary to another variable. Both names refer to the same object.

```
dict1 = {"a": 10, "b": 20}
dict2 = dict1 # aliasing
dict2["a"] = 100
print(dict1) # Output: {'a': 100, 'b': 20}
```

Changing one affects the other.

Copying creates a new independent copy of the dictionary.

```
dict3 = dict1.copy()
dict3["b"] = 50
print(dict1) # Output: {'a': 100, 'b': 20}
```

Here, changes in `dict3` don't affect `dict1`.

Q7. Develop a program to create an empty dictionary and add some key–value pairs to it.

Answer:

```
data = {} # empty dictionary

data["name"] = "Arjun"
data["age"] = 21
data["city"] = "Bangalore"

print("Dictionary contents:", data)
```

Output:

```
Dictionary contents: {'name': 'Arjun', 'age': 21, 'city': 'Bangalore'}
```

Q8. Define Dictionaries in Python programming. Discuss dictionary operators with example.

Answer:

A **dictionary** in Python stores data in the form of **key–value pairs** and allows quick retrieval using keys.

Dictionary Operators:

1. **Membership Operator** (`in`, `not in`)

```

2. student = {"name": "Arjun", "age": 21}
3. print("name" in student)      # True
4. print("email" not in student) # True
5. Deletion Operator (del)
6. del student["age"]
7. print(student)    # {'name': 'Arjun'}

```

These operators help to search and manage items efficiently.

Q9. Give the Python interpreter's response to each of the following:

```
dictionary = {"apples": 15, "bananas": 35, "grapes": 12}
```

Statement	Explanation	Output
dictionary["bananas"]	Returns the value of key "bananas"	35
dictionary["oranges"] = 20	Adds a new key-value pair	{'apples': 15, 'bananas': 35, 'grapes': 12, 'oranges': 20}
len(dictionary)	Returns number of items	4
"grapes" in dictionary	Checks if key exists	True
dictionary["pears"]	Key not found, raises KeyError	KeyError: 'pears'
dictionary.get("pears", 0)	Returns default value when key not found	0

Q10. Develop a program to find the length of a dictionary.

Answer:

```

fruits = {"apple": 5, "banana": 10, "grapes": 7}
length = len(fruits)
print("Length of dictionary:", length)

```

Output:

```
Length of dictionary: 3
```

Explanation:

The `len()` function returns the number of key-value pairs present in the dictionary. In this example, the dictionary contains 3 pairs.

✅ Summary of Module 3 Concepts:

- Dictionaries store **key-value pairs** for fast lookup.
 - They support methods like `get()`, `keys()`, `values()`, `update()`, `copy()`, etc.
 - The `in` operator checks for key existence.
 - They are **mutable**, meaning values can be changed after creation.
 - Useful in real-world applications like databases, word frequency counters, and record keeping.
-