Basic Python Questions and Answers

1. What is the difference between append() and extend() method in list ?

Answer: Append function in Python adds a single element to the end of the list, whereas the extend function adds multiple elements (from an iterable) to the list.

2. Write a program to find the second largest number in a list without using sort() or max() more than once.

Answer:

```
nums = [10, 5, 8, 20, 15]
largest = max(nums)       # find largest once
second = float('-inf')
for n in nums:
    if n != largest and n > second:
        second = n
print("Second Largest:", second)
```

3. What are the characteristics of keys in a dictionary?

Answer: - Keys must be unique and immutable (strings, numbers, or tuples are commonly used as keys).

4. What is a lambda function in Python?

Answer: - A lambda function is a small, anonymous (unnamed) function defined using the lambda keyword. It can have any number of arguments but only one expression.

5. Can sets contain mutable elements like lists. If yes then which datatypes we can store?

Answer: - No, sets cannot contain mutable elements like lists because sets require elements to be immutable for hashing. It can't store list, set, and diction?

6. What is the purpose of the elif statement?

Answer: - The elif statement allows you to check multiple conditions sequentially after an initial if statement. It executes if the previous conditions are False and its condition is True.

7. Input: a list of unique numbers from 1 to n (with one missing). Find the missing number using loops only?

Example: [1, 2, 4, 5] → Output: 3

Answer:

```
nums = [1, 2, 4, 5]
n = 5  # total numbers should be from 1 to 5
expected_sum = n * (n + 1) // 2
actual_sum = 0
for x in nums:
    actual_sum += x
print("Missing Number:", expected_sum - actual_sum)
```

8. What is a Continue statement?

Answer: - continue helps you skip one iteration (cycle) of a loop when a condition is met, and then continue with the next iteration.

9. How do you skip the current iteration of a loop?

Answer: - Use the continue statement to skip the current iteration and continue with the next one.

10. What are *args and **kwargs in Python?

Answer: - They are ways to let a function take any number of arguments. *args is for many positional (normal) arguments, and **kwargs is for many keyword (name=value) arguments.

11. Can you use *args and **kwargs together in a function?

Answer: - Yes, you can use both in the same function. *args comes first, then **kwargs.

12. What is the map() function in Python?

Answer: - map() applies a function to every item in a list (or another collection) and gives back a new list with the results.

13. Why use map() instead of a loop?

Answer: - map() can make your code shorter and easier to read when you want to apply a function to every item in a list. It can also be faster sometimes.

14. For a given string, count consecutive repeating characters and compress it.

Example: "aaabbccccd" → "a3b2c4d1"

Answer: -

```
s = "aaabbccccd"
result = ""
count = 1
for i in range(1, len(s)):
    if s[i] == s[i-1]:
        count += 1
    else:
        result += s[i-1] + str(count)
        count = 1
result += s[-1] + str(count)   # add last group
print("Compressed:", result)
```

15. What are the methods that list contains?

Answer: - Methods help you do different things with your list, such as adding new items, removing items, or changing how the list looks. Each method has a specific job it can do to your list, and you use these methods by calling their names with your list.

• append(x): Adds something new to the end of your list.

• remove(x): Takes out the first thing in your list that matches what you tell it.

• sort(): Puts all the items in your list in order from smallest to biggest, or in another way you choose. • index(x): Shows you where in the list the first item like x is.

• clear(): Removes all items from the list.

• count(x): Returns the number of times item x appears in the list.

• reverse():Reverses the elements of the list in place.

• copy():Returns a shallow copy of the list.

• pop(): removes and gives you back the last item in a list, or an item at a specific position if you tell it where.

16.What are the Functions that list contains?

Answer: -

- len(list) Returns the number of items in the list.
- max(list) Returns the largest item in the list.
- min(list) Returns the smallest item in the list.
- sum(list) Returns the sum of all items in the list.
- sorted(list, key=None, reverse=False) Returns a new sorted list from the items in the list.
- list(iterable) Converts an iterable (like a tuple) into a list.

17. You're initializing a variable max_score = 85 in your machine learning model evaluation script. Is this variable iterable or non-iterable? How can you check this programmatically?

Answer: max_score = 85 is **non-iterable** because it's an integer (single value, not a collection).

```
if hasattr(max_score, '__iter__'):
    print("Iterable")
else:
    print("Non-iterable")  # This will execute
```

18. A string is balanced if the number of vowels equals the number of consonants?

Example: "code" → Balanced (2 vowels, 2 consonants).

Answer: -

```
s = "code".lower()
vowels = "aeiou"
vowel_count = 0
consonant_count = 0

for ch in s:
    if ch.isalpha():
        if ch in vowels:
            vowel_count += 1
        else:
            consonant_count += 1
```

```
if vowel_count == consonant_count:
    print("Balanced")
else:
    print("Not Balanced")
```

19. How do you check if all characters in a string are alphanumeric?

Answer: - Use s.isalnum(). What kind of input validation might use this**.**

20. What are Python's main data types and why are they important in Data Science?

Answer:
Python provides key data types:

- int, float → numeric operations
- str → textual data
- list, tuple, set, dict → data structures for collections

In data science:

- Numbers handle mathematical operations
- Strings store categorical labels
- Lists/dicts manage datasets before using pandas/numpy

21. Explain mutable and immutable types in Python?

Answer:

- **Mutable**: can be changed after creation (list, dict, set)
- **Immutable**: cannot be changed (tuple, str, int, float)

```
a = [1,2,3]
a.append(4)  # mutable

b = (1,2,3)
# b[0] = 5  error: tuple is immutable
```

22. What is the difference between == and is operator in Python?

Answer:

- == → checks value equality
- is → checks memory location

```
a = [1,2]
b = [1,2]
print(a == b)  # True
print(a is b)  # False (different objects)
```

23. Explain list comprehensions with an example?

Answer: - A concise way to create lists using loops.

```
nums = [1,2,3,4]
squares = [n**2 for n in nums]
print(squares)  # [1,4,9,16]
```

24. Write a Python program to count words in a sentence?

Answer: -

```
text = "Infosys is a global company"
count = len(text.split())
print("Word count:", count)
```

25. How do you check for even numbers using Python conditional statements?

Answer: -

```
n = 8
if n % 2 == 0:
   print("Even")
else:
   print("Odd")
```

26. Write a Python function to calculate factorial using recursion?

Answer: -

```
def factorial(n):
```

```
    return 1 if n == 0 else n * factorial(n-1)


print(factorial(5))  # 120
```

27. How do you remove duplicates from a list?

Answer: -

```
nums = [1,2,2,3,3,4]

unique = list(set(nums))

print(unique)
```

28. Write a Python program to count frequency of elements in a list?

Answer: -

```
nums = [1,2,2,3,3,3]

freq = {}

for n in nums:

    freq[n] = freq.get(n, 0) + 1

print(freq)
```

29. What is List Comprehension? Give example?

Answer: - List comprehension provides a compact and readable way to generate lists in a single line.

```
squares = [x**2 for x in range(5)]
```

30. How to count occurrences of each element in a list?

Answer: - Counter() creates a dictionary with frequency of each element.

```
from collections import Counter

nums = [1,2,2,3,3,3]

print(Counter(nums))
```

31. What is the difference between compiler and interpreter?

Answer: - Python uses an interpreter, meaning it executes code step-by-step, which helps in debugging.

Compiler	Interpreter
Translates the whole code at once	Translates line-by-line
Faster execution	Slower (checks line by line)
Example: C, Java	Example: Python, JavaScript

32. What are Python Namespaces and Scopes? Explain with examples?

Answer: - A namespace in Python is like a container or a dictionary that maps variable names (keys) to their objects/values (values).
It helps Python keep track of all identifiers (like variables, functions, and classes) and their corresponding values to avoid name conflicts.

```
x = 10      # Global namespace

def func():

   y = 20   # Local namespace

   print(y)


func()

print(x)
```

33. What is Scope? Scope defines the region of code where a variable can be accessed or modified?

Answer: - When Python encounters a variable name, it searches for it in a specific order — known as the LEGB Rule.

Level	Meaning	Example
L – Local	Inside the current function	Variable defined inside a function
E –Enclosing	Inside enclosing (nested) functions	Variable in outer function of a nested one
G – Global	Defined at the top level of the script or module	Variable declared outside functions
B – Built-in	Predefined names in Python	e.g., len, print, sum

```
x = "Global X"

def outer():
    x = "Enclosing X"
    def inner():
        x = "Local X"
        print(x)  # searches Local → Enclosing → Global → Built-in
    inner()

outer()
```

34. What is a user-defined function in Python?

Answer: - A user-defined function is a custom function created by the programmer using the def keyword to perform a specific task.

```
def greet():
    print("Hello, welcome to Python functions!")
greet()
```

35. What is the purpose of the return statement?

Answer: - The return statement sends back a value from the function to the caller. Without it, the function returns None.

```
def square(x):
    return x * x

result = square(4)
print(result)
```

36. What are parameters and arguments in functions?

Answer: - **Parameters** are variables listed inside the function definition.

Arguments are actual values passed to the function when it is called.

```
def multiply(a, b):  # parameters
    return a * b
```

```
print(multiply(3, 5))  # arguments
```

37. What is a function with default parameters?

Answer: - Default parameters have pre-defined values. If no argument is passed, the default value is used.

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


greet()        # uses default value
greet("Dr. Siemon")  # uses given argument
```

38. What is a class in Python?

Answer: - A class is a blueprint for creating objects. It defines attributes (variables) and methods (functions).

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show_info(self):
        print("Name:", self.name, "| Age:", self.age)


# creating object
s1 = Student("Amit", 21)
s1.show_info()
```

39. What is the __init__() method?

Answer: - It is a constructor method that automatically executes when an object is created. It initializes the object's data.

```
class Employee:
    def __init__(self, name, salary):
```

```
    self.name = name
    self.salary = salary

emp1 = Employee("Riya", 50000)
print(emp1.name, emp1.salary)
```

40. What is encapsulation?

Answer: - Encapsulation means binding data (variables) and methods (functions) together into a single unit (class).
It also restricts direct access to some attributes using private variables.

```
class Account:
  def __init__(self, balance):
    self.__balance = balance   # private variable

  def show_balance(self):
    print("Balance:", self.__balance)

acc = Account(1000)
acc.show_balance()
# print(acc.__balance)  Error: private variable
```

41. What is inheritance?

Answer: - Inheritance allows one class (child) to acquire properties and methods from another class (parent).

```
class Animal:
  def speak(self):
    print("Animal speaks")

class Dog(Animal):
  def bark(self):
    print("Dog barks")
```

```
d = Dog()
d.speak()
d.bark()
```

42. What is polymorphism?

Answer: - Polymorphism means “many forms” — the same function or method can behave differently depending on the object that calls it.

```
class Bird:
    def sound(self):
        print("Some bird sound")

class Parrot(Bird):
    def sound(self):
        print("Parrot says hello")

class Crow(Bird):
    def sound(self):
        print("Crow caws")

for b in [Parrot(), Crow()]:
    b.sound()
```

43. What is abstraction?

Answer: - Abstraction hides the internal details and only exposes the necessary parts of an object to the user.
In Python, it is implemented using abstract classes and abstract methods from the abc module.

```
from abc import ABC, abstractmethod

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

```
    pass

class Circle(Shape):
  def area(self):
    return 3.14 * 5 * 5

c = Circle()
print("Area of Circle:", c.area())
```

44. What is the difference between class variable and instance variable?

Answer: - **Class variable:** Shared by all objects of the class.

Instance variable: Unique to each object.

```
class Student:
  school = "ABC School"  # class variable

  def __init__(self, name):
    self.name = name   # instance variable

s1 = Student("Ravi")
s2 = Student("Meena")

print(s1.name, s1.school)
print(s2.name, s2.school)
```

45. What is multiple inheritance in Python?

Answer: - When a child class inherits from more than one parent class, it is called **multiple inheritance**.

```
class Father:
  def skill(self):
    print("Cooking")
```

```
class Mother:
    def hobby(self):
        print("Dancing")

class Child(Father, Mother):
    pass

c = Child()
c.skill()
c.hobby()
```

46. How can OOP concepts be applied in a banking system project?

Answer: - In a banking system:

1. **Classes** can represent real entities:
    - Account, Customer, Transaction
2. **Objects** are individual accounts or customers.
3. **Encapsulation** protects sensitive data like balances and PIN.
4. **Inheritance** for account types:
    - SavingsAccount and CurrentAccount inherit from Account.
5. **Polymorphism** allows methods like calculate_interest() to behave differently for savings vs. current accounts.
6. **Abstraction** hides internal database operations from the user interface.

47. Explain how encapsulation helps in security?

Answer: - Encapsulation restricts direct access to sensitive data using private attributes.

Only authorized methods can modify data, preventing accidental or malicious changes.

Example: Bank account balance, passwords, or PINs stored in private variables.

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

```
def withdraw(self, amount):
    if amount <= self.__balance:
        self.__balance -= amount
    else:
        print("Insufficient funds")
```

48. What are the advantages and disadvantages of OOP in Python?

Answer: - Advantages:

1. Modularity: Each class is self-contained
2. Reusability: Inheritance allows reuse of code
3. Flexibility: Polymorphism and abstraction allow dynamic behavior
4. Security: Encapsulation protects sensitive data
5. Maintainability: Easier to debug and scale large applications

Disadvantages:

1. Complexity: For small projects, OOP can be overkill
2. Memory consumption: Creating multiple objects consumes more memory
3. Learning curve: Understanding OOP concepts requires effort for beginners
4. Slower execution: Compared to procedural programming for simple tasks

49. You are building an online store. Each customer can add multiple items to the cart. You need to calculate the total bill including tax.

Write a Python program to:

1. Take the prices of items as input.
2. Calculate the total.
3. Add 10% tax to the total.
4. Print the final bill amount.

Hint: Use input(), loops, arithmetic operations, and float() conversion.

Answer: -

1. input() – takes user input (converted to float for decimals).

2. for loop – runs for each item and keeps adding to total.

3. Tax calculation – 10% of total = total * 0.10.

4. Final bill – total + tax.

5. Formatted printing – .2f ensures two decimal places (currency format).

Online Store Total Bill Calculator

 # Step 1: Take input for number of items

n = int(input("Enter the number of items you purchased: "))

 # Step 2: Initialize total variable

total = 0.0

 # Step 3: Use loop to take price of each item

```
for i in range(n):
    price = float(input(f"Enter price of item {i+1}: ₹"))
    total += price   # add each price to total
```

 # Step 4: Calculate 10% tax

*tax = total * 0.10*

 # Step 5: Final bill amount

final_amount = total + tax

 # Step 6: Display results

```
print("\n--------BILL SUMMARY -----------")
print(f"Total (without tax): ₹{total:.2f}")
print(f"Tax (10%): ₹{tax:.2f}")
print(f"Final Amount to Pay: ₹{final_amount:.2f}")
print("-------------------------------------")
```

50. A school wants to calculate students' grades based on marks.

90–100 → A

80–89 → B

70–79 → C

Below 70 → D

Write a Python program to take a student's mark as input and print the grade.

Answer: -

1. A conditional statement allows a program to execute certain code only if a condition is true.

2. In Python, if-elif-else is used for multiple conditions.

3. Syntax:

Student Grade Calculator

 # Step 1: Take marks input

```
marks = float(input("Enter the student's marks (0-100): "))
```

 # Step 2: Determine grade using if-elif-else

```
if 90 <= marks <= 100:
    grade = "A"
elif 80 <= marks < 90:
    grade = "B"
elif 70 <= marks < 80:
    grade = "C"
elif 0 <= marks < 70:
    grade = "D"
else:
    grade = "Invalid marks"
```

 # Step 3: Display result

print(f"The grade for marks {marks} is: {grade}")

51. What is Python's indentation rule?

Answer: Python uses indentation instead of braces {} to define blocks of code. Incorrect indentation gives IndentationError.

```
if True:
    print("Hello")  # correct
```

52. What is Python's garbage collection?

Answer:

a. Garbage collection (GC) is a **memory management process** where Python automatically **reclaims memory occupied by objects that are no longer in use**.

b. This helps **prevent memory leaks** and makes Python programs more efficient.

c. Python developers usually **don't have to manually free memory** because the interpreter handles it.

Python Manages Memory

Python uses **two main strategies** for memory management:

a) Reference Counting

- Every object in Python keeps a **count of how many references point to it**.
- When an object is **no longer referenced**, its memory can be freed immediately.

```
a = [1, 2, 3]
b = a      # reference count of the list = 2
del a      # reference count = 1
del b      # reference count = 0 → memory freed
```

b) Garbage Collector for Cycles

- Reference counting alone **cannot handle circular references** (objects referencing each other).
- Python's gc module detects these **reference cycles** and cleans them up.

Garbage Collection is Important:

a) Automatic Memory Management: Developers don't need to manually free memory.
b) Prevents Memory Leaks: Unused objects are automatically removed.
c) Handles Cyclic References: Even objects referencing each other in a cycle are collected.
d) Improves Performance: Frees memory for new objects, reducing memory bloat.

53. why we need python only for data science?

Answer: Python is extremely popular for data science, but it's not the only language. Here's why Python is often preferred and why it's considered almost essential in modern data science:

1. Easy to Learn and Read

2. Large Ecosystem of Libraries

3. Integration and Compatibility

4. Community Support

5. Flexibility

6. Visualization and Reporting

7. Used in Machine Learning and AI

8. Cross-Platform and Free

54. What are control structures in Python and what are their types?

Answer: Control structures in Python direct the flow of program execution based on specific conditions, loops, and sequences.

- Sequential Execution: Code runs line by line in order
- Decision-Making Structures: if, elif, else statements for conditional execution
- Looping Structures: for and while loops for repetitive tasks
- Jump Statements: break and continue to alter loop execution

Decision-making structure

```
age = 18
if age >= 18:
    print("You can vote")
elif age >= 16:
    print("You can drive")
else:
    print("Too young for both")
```

Advanced Python Questions & Answers

1. what is python module and package how you will explain? give me a detailed answer?

Answer: A module is a file containing Python definitions, statements, functions, classes, and variables. Every Python file with a .py extension is a module. The module name is the same as the file name without the .py extension.

Key Concepts of Modules:

a. Purpose of Modules:

- Modules help organize code logically by grouping related functionality together
- They promote code reusability - write once, use multiple times across different programs
- They create separate namespaces to avoid naming conflicts between different parts of a program
- They make code more maintainable by breaking large programs into smaller, manageable units

b. Types of Modules:

- Built-in Modules: Pre-installed with Python (e.g., math, os, sys, random)
- User-defined Modules: Created by developers for specific project needs
- Third-party Modules: Installed via pip (e.g., numpy, pandas, requests)

A **package** is a hierarchical directory structure that contains modules and sub-packages. It's essentially a way of organizing related modules into a directory hierarchy. A package is a directory that contains a special file called __init__.py.

Key Concepts of Packages:

A. Structure of a Package: A package is identified by the presence of an __init__.py file in a directory. This file can be empty or contain initialization code for the package.

Directory Structure Example:

```
mypackage/
├── __init__.py        # Makes it a package
├── module1.py         # Module inside package
├── module2.py         # Another module
└── subpackage/        # Sub-package
    ├── __init__.py    # Makes subpackage a package
    ├── module3.py
```

└── *module4.py*

B. Purpose of Packages:

- Organize large projects with many modules into a structured hierarchy
- Prevent module name conflicts by creating separate namespaces
- Allow logical grouping of related functionality
- Enable dot notation for accessing nested modules
- Facilitate distribution and installation of Python libraries

2. What is advanced python?

Answer: Advanced Python refers to sophisticated Python concepts beyond basic syntax and programming. It includes:

1. **Deep OOP concepts** - metaclasses, magic methods, multiple inheritance, abstract classes
2. **Functional & async programming** - decorators, generators, closures, asyncio, multithreading
3. **Performance optimization** - memory management, profiling, efficient data structures, algorithm complexity
4. **Professional development** - package creation, testing frameworks, design patterns, context managers
5. **Specialized domains** - web frameworks (Django/Flask), data science libraries (Pandas/NumPy), database ORMs, API development

3. What are decorators in Python?

Answer: Decorators are functions that modify the behaviour of other functions or classes without changing their source code. They use the @ symbol and are applied above function/class definitions. Decorators follow the wrapper pattern - they take a function as input, add functionality, and return a modified function. Common examples include @staticmethod, @property, and custom decorators for logging or authentication.

4. What is a generator in Python? How is it different from a regular function?

Answer: A generator is a function that uses yield instead of return and produces a sequence of values lazily (one at a time) rather than computing all values at once. Generators maintain their state between calls and automatically implement the iterator protocol. They are memory-efficient for large datasets as they generate values on-demand. Unlike regular functions that execute completely and return once, generators can pause execution and resume later.

5. Explain the Global Interpreter Lock (GIL) in Python.

Answer: GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecode simultaneously in CPython. It exists because Python's memory management is not thread-safe. GIL limits multi-threading performance for CPU-bound tasks but doesn't affect I/O-bound operations. To achieve true parallelism for CPU-intensive tasks, use multiprocessing instead of threading, as each process has its own Python interpreter and GIL.

6. What is the difference between loc and iloc in Pandas?

Answer: loc and iloc are both used for indexing and selecting data, but they work differently:

- **loc**: Label-based indexing. It uses actual index labels and column names.
- **iloc**: Integer-based indexing. It uses integer positions (0-based indexing).

```
import pandas as pd


df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['x', 'y', 'z'])


# Using loc (label-based)

df.loc['x', 'A']  # Returns 1 (uses index label 'x')


# Using iloc (position-based)

df.iloc[0, 0]  # Returns 1 (uses position 0, 0)
```

Key difference: loc includes the end point in slicing (df.loc['x':'y'] includes 'y'), while iloc excludes it (df.iloc[0:2] excludes position 2).

7. Explain the concept of MultiIndex in Pandas. How do you create and work with it?

Answer: A MultiIndex (hierarchical index) allows you to have multiple levels of indexes on rows or columns, enabling you to work with higher-dimensional data in a 2D DataFrame.

Creating MultiIndex:

```
import pandas as pd


# Method 1: Using arrays

arrays = [['A', 'A', 'B', 'B'], [1, 2, 1, 2]]
```

index = pd.MultiIndex.from_arrays(arrays, names=['letter', 'number'])

df = pd.DataFrame({'value': [10, 20, 30, 40]}, index=index)

Method 2: Using tuples

index = pd.MultiIndex.from_tuples([('A', 1), ('A', 2), ('B', 1), ('B', 2)])

Method 3: Using product

index = pd.MultiIndex.from_product([['A', 'B'], [1, 2]], names=['letter', 'number'])

Working with MultiIndex:

Accessing data

df.loc['A'] # All rows with first level = 'A'

df.loc[('A', 1)] # Specific combination

Resetting index

df.reset_index()

Swapping levels

df.swaplevel()

8. What is the difference between merge(), join(), and concat() in Pandas?

Answer:

- **merge()**: SQL-style joins based on common columns or indexes. Most flexible for joining DataFrames.

pd.merge(df1, df2, on='key', how='inner') # inner, outer, left, right

- **join()**: Joins DataFrames based on their indexes. Simpler but less flexible than merge.

df1.join(df2, how='left')

- **concat()**: Concatenates DataFrames along rows (axis=0) or columns (axis=1). Used for stacking DataFrames.

pd.concat([df1, df2], axis=0) # Stack vertically

pd.concat([df1, df2], axis=1) # Stack horizontally

When to use:

- Use merge() for database-style joins with specific keys

- Use join() for quick index-based joins
- Use concat() for simple stacking of DataFrames

9. What is monkey patching in Python?

Answer: Monkey patching is dynamically modifying or extending classes/modules at runtime by adding, replacing, or modifying methods and attributes after they're defined. It's useful for testing, fixing bugs in third-party libraries, or adding functionality without modifying source code. However, it's considered bad practice in production as it makes code harder to understand, maintain, and can cause unexpected side effects. Use with caution and only when necessary.

10. What is the difference between __new__ and __init__ methods?

Answer: __new__ is a static method that creates and returns a new instance of the class; it's called before __init__. __init__ is an instance method that initializes the already-created instance with attributes. __new__ receives the class as first argument and returns an object, while __init__ receives self and returns None. __new__ is rarely overridden except for immutable types or implementing singleton pattern, while __init__ is commonly used for initialization.

11. Explain the concept of list comprehension and its advantages.

Answer: List comprehension is a concise way to create lists using a single line syntax: [expression for item in iterable if condition]. It's faster than traditional loops because it's optimized at C level and more readable/Pythonic. Comprehensions exist for lists [], dictionaries {}, sets {}, and generators (). Advantages include better performance, reduced code length, and functional programming style. Avoid complex nested comprehensions for readability.

12. Explain the concept of vectorization in Pandas. Why is it important?

Answer: Vectorization is performing operations on entire arrays/Series at once rather than using loops. Pandas is built on NumPy, which provides highly optimized vectorized operations.

Example - Bad vs Good approach:

Bad: Using loops (slow)

result = []

for val in df['column']:

```
    result.append(val * 2)
df['new_col'] = result

# Good: Vectorized operation (fast)
df['new_col'] = df['column'] * 2
```

Why it's important:

1. **Performance**: Vectorized operations are 10-100x faster than loops
2. **Readability**: Code is cleaner and more concise
3. **Memory efficiency**: Operations are performed in C/Cython under the hood
4. **Broadcasting**: Automatically handles operations across different shapes

Advanced vectorization:

```
# Using numpy functions
df['result'] = np.where(df['col'] > 0, df['col'] * 2, df['col'] / 2)

# Using apply with vectorized functions
df['result'] = df['col'].apply(np.sqrt)  # Still uses vectorization internally
```

13. What is the groupby() operation? Explain split-apply-combine concept.

Answer: groupby() is used for group-wise data analysis following the split-apply-combine pattern:

1. **Split**: Divide data into groups based on criteria
2. **Apply**: Apply a function to each group independently
3. **Combine**: Combine results into a data structure

Examples:

```
# Basic groupby
df.groupby('category')['value'].mean()

# Multiple aggregations
df.groupby('category').agg({
    'value': ['mean', 'sum', 'count'],
    'price': ['min', 'max']
```

```
})

# Custom aggregation
df.groupby('category')['value'].agg(
    total='sum',
    average='mean',
    range=lambda x: x.max() - x.min()
)

# Grouping by multiple columns
df.groupby(['category', 'region'])['sales'].sum()

# Transform (keeps original shape)
df['normalized'] = df.groupby('category')['value'].transform(
    lambda x: (x - x.mean()) / x.std()
)

# Filter groups
df.groupby('category').filter(lambda x: len(x) > 5)
```

14. Explain the difference between apply(), map(), and applymap() in Pandas.

Answer:

- **apply()**: Works on DataFrame rows/columns or Series. Can apply any function.

```
# On DataFrame (default axis=0, operates on columns)
df.apply(lambda x: x.max() - x.min())

# On rows (axis=1)
df.apply(lambda row: row['A'] + row['B'], axis=1)

# On Series
df['column'].apply(lambda x: x * 2)
```

- **map()**: Only works on Series. Maps values using a function, dict, or Series.

Using dictionary

df['category'].map({'A': 1, 'B': 2, 'C': 3})

Using function

*df['value'].map(lambda x: x ** 2)*

- **applymap()**: Works element-wise on entire DataFrame (deprecated in favor of map() in newer versions).

Applies function to every element in DataFrame

*df.applymap(lambda x: x * 2) # Old syntax*

*df.map(lambda x: x * 2) # New syntax (Pandas 2.1+)*

Key differences:

- apply(): Most flexible, works on DataFrame/Series, can reduce dimensions
- map(): Series only, element-wise, maintains shape
- applymap(): DataFrame only, element-wise, maintains shape

15. How do you handle missing data in Pandas? Explain different approaches.

Answer: Pandas provides multiple strategies for handling missing data (NaN, None, NaT):

1. Detection:

df.isnull() # or df.isna()

df.notnull() # or df.notna()

df.isnull().sum() # Count missing values per column

2. Removal:

Drop rows with any missing value

df.dropna()

Drop columns with any missing value

df.dropna(axis=1)

Drop rows where all values are missing

df.dropna(how='all')

Drop rows with less than 3 non-missing values

df.dropna(thresh=3)

3. Filling:

Fill with specific value

df.fillna(0)

Forward fill (propagate last valid observation)

df.fillna(method='ffill')

Backward fill

df.fillna(method='bfill')

Fill with mean/median/mode

df['column'].fillna(df['column'].mean())

Fill with interpolation

df.interpolate(method='linear')

4. Advanced techniques:

Fill different columns with different values

df.fillna({'col1': 0, 'col2': df['col2'].median()})

Replace specific values with NaN

df.replace([999, -999], np.nan)

16. What is the difference between multithreading and multiprocessing?

Answer: Multithreading runs multiple threads within a single process, sharing memory space, best for I/O-bound tasks. Limited by GIL for CPU-bound tasks in CPython. Multiprocessing runs multiple processes with separate memory spaces and Python interpreters, bypassing GIL, ideal for CPU-bound tasks. Threading has lower overhead and easier data sharing, while multiprocessing offers true parallelism but higher overhead and complex inter-process communication.

17. What is NumPy and why is it faster than Python lists?

Answer: NumPy (Numerical Python) is a powerful library for numerical computing that provides support for large multi-dimensional arrays and matrices. It's faster than Python lists because: (1) arrays are stored in contiguous memory blocks enabling cache optimization, (2) operations are implemented in C/Fortran for performance, (3) it uses vectorization eliminating Python loops overhead, and (4) it supports homogeneous data types avoiding type-checking overhead at runtime.

18. Explain the difference between NumPy array and Python list.

Answer: NumPy arrays are homogeneous (same data type), stored in contiguous memory, and support vectorized operations, making them faster and more memory-efficient. Python lists are heterogeneous (mixed types), stored as arrays of pointers to objects scattered in memory, and require explicit loops for operations. NumPy arrays have fixed size at creation while lists are dynamic. NumPy provides mathematical operations, broadcasting, and advanced indexing not available for lists.

19. What is vectorization in NumPy? Why is it important?

Answer: Vectorization is performing operations on entire arrays at once instead of using Python loops, leveraging low-level optimizations. Operations are pushed to highly optimized C/Fortran code, eliminating Python interpreter overhead. It makes code faster (10-100x), more readable, and concise. Example: arr * 2 is vectorized while [x*2 for x in arr] is not. Vectorization is fundamental to NumPy's performance and enables efficient scientific computing.

20. Explain the concept of array stacking in NumPy.

Answer: Stacking joins multiple arrays along a new or existing axis. np.vstack() stacks vertically (row-wise, axis=0), np.hstack() stacks horizontally (column-wise, axis=1). np.dstack() stacks depth-wise (axis=2). np.stack() provides general stacking along specified axis. np.concatenate() joins along existing axis. Arrays must have compatible shapes (same dimensions except for concatenation axis). Stacking is essential for combining datasets and building larger arrays from smaller components.

21. What is the difference between axis=0 and axis=1 in NumPy operations?

Answer: In NumPy, axis=0 refers to operations along rows (vertically, first dimension), while axis=1 refers to operations along columns (horizontally, second dimension). For 2D arrays: axis=0 operates down columns, axis=1 operates across rows. Example: arr.sum(axis=0) sums each column, arr.sum(axis=1) sums each row. The result's shape removes the specified axis. For higher dimensions, axis numbering continues (axis=2, axis=3, etc.).

22. What is the purpose of pivot_table() and how is it different from groupby()?

Answer: pivot_table() creates a spreadsheet-style pivot table that reshapes data, while groupby() performs group-wise aggregations.

pivot_table():

```
# Basic pivot table
df.pivot_table(
    values='sales',
    index='region',
    columns='product',
    aggfunc='sum'
)

# Multiple aggregations
df.pivot_table(
    values='sales',
    index='region',
    columns='product',
    aggfunc=['sum', 'mean', 'count'],
    fill_value=0,
    margins=True  # Adds row/column totals
)

# Multiple value columns
df.pivot_table(
    values=['sales', 'profit'],
    index='region',
    columns='product',
    aggfunc='sum'
)
```

Key differences:

1. **Output shape**: pivot_table() creates a cross-tabulated table with reshaped data, groupby() returns aggregated data in original structure
2. **Use case**: pivot_table() is for creating summary tables (like Excel pivot tables), groupby() is for flexible aggregations
3. **Visualization**: pivot_table() output is better suited for heatmaps and cross-sectional analysis

Example comparison:

```
# groupby result
df.groupby(['region', 'product'])['sales'].sum()
# Returns: Series with MultiIndex


# pivot_table result
df.pivot_table(values='sales', index='region', columns='product', aggfunc='sum')
# Returns: DataFrame with regions as rows, products as columns
```

23. Explain the concept of method chaining in Pandas and provide examples.

Answer: Method chaining is writing multiple operations in a sequence where each method returns a DataFrame/Series, allowing the next method to be called. This creates clean, readable code.

Examples:

```
# Without method chaining (verbose)
df1 = df[df['age'] > 25]
df2 = df1.groupby('department')['salary'].mean()
df3 = df2.sort_values(ascending=False)
result = df3.head(5)


# With method chaining (clean)
result = (df
  .query('age > 25')
  .groupby('department')['salary']
  .mean()
  .sort_values(ascending=False)
```

.head(5)

)

Benefits:

1. **Readability**: Operations flow top-to-bottom
2. **Debugging**: Easy to comment out individual steps
3. **Memory**: No intermediate variables needed
4. **Maintainability**: Clear transformation pipeline

Useful methods for chaining:

- assign(): Add/modify columns
- query(): Filter rows using string expressions
- pipe(): Apply custom functions in the chain
- rename(): Rename columns inline

24. Explain the difference between pyplot and object-oriented interface in Matplotlib.

Answer: pyplot (plt) provides MATLAB-style state-based interface where functions modify current figure/axes implicitly - simpler for quick plots. Object-oriented (OO) interface uses explicit Figure and Axes objects giving more control and better for complex plots. OO interface is preferred for production code, subplots, and multiple figures. Pyplot is good for interactive use and simple scripts. Both can be mixed but OO is more Pythonic.

Pyplot interface (state-based)

plt.plot([1, 2, 3], [4, 5, 6])

plt.title('Pyplot Style')

Object-oriented interface (explicit)

fig, ax = plt.subplots()

ax.plot([1, 2, 3], [4, 5, 6])

ax.set_title('OO Style')

ax.set_xlabel('X')

ax.set_ylabel('Y')

plt.show()

25. What are categorical data types in Pandas? When and why should you use them?

Answer: Categorical data type is a special dtype for representing categorical variables with a limited, fixed set of possible values.

Creating categorical data:

```
# Convert to category
df['grade'] = df['grade'].astype('category')


# Create with specific categories and order
df['size'] = pd.Categorical(
    df['size'],
    categories=['S', 'M', 'L', 'XL'],
    ordered=True
)


# Define during DataFrame creation
df = pd.DataFrame({
    'color': pd.Categorical(['red', 'blue', 'red', 'green'])
})
```

Benefits:

1. **Memory efficiency**: Stores data as integer codes with a mapping, reducing memory by 50-90%

```
# String column: ~100 MB
df['country'] = ['United States'] * 1000000


# Categorical column: ~10 MB
df['country'] = df['country'].astype('category')
```

2. **Performance**: Faster operations on categorical data
3. **Ordered comparisons**: Can define logical ordering for sorting
4. **Validation**: Ensures only valid categories are used

When to use:

- Columns with repeated string values (country, state, category names)
- Ordinal data (ratings, sizes, education levels)
- When memory is a constraint
- When you need to enforce a fixed set of values

Operations on categorical data:

Get categories

df['color'].cat.categories

Add categories

df['color'].cat.add_categories(['yellow'])

Reorder categories

df['size'].cat.reorder_categories(['XL', 'L', 'M', 'S'])

Compare (if ordered)

df[df['size'] > 'M']

26. What is the difference between Figure and Axes in Matplotlib?

Answer: Figure is the entire window/canvas that contains everything - the outermost container. Axes is the actual plotting area where data is drawn (not axis!). One Figure can contain multiple Axes (subplots). Figure handles overall properties like size and background, while Axes handles plotting, labels, ticks, and legends. Understanding this hierarchy is crucial for customizing plots effectively.

fig = plt.figure(figsize=(10, 4)) # Create figure

Create two axes (subplots)

ax1 = fig.add_subplot(121) # 1 row, 2 cols, position 1

ax1.plot([1, 2, 3], [1, 4, 9])

ax1.set_title('Plot 1')

ax2 = fig.add_subplot(122) # 1 row, 2 cols, position 2

ax2.plot([1, 2, 3], [1, 2, 3])

ax2.set_title('Plot 2')

plt.tight_layout()

plt.show()

27. How do you optimize Pandas performance for large datasets?

Answer: Several strategies can significantly improve Pandas performance:

1. Use appropriate data types:

```
# Downcast numeric types
df['int_col'] = pd.to_numeric(df['int_col'], downcast='integer')
df['float_col'] = pd.to_numeric(df['float_col'], downcast='float')

# Use category for repetitive strings
df['category'] = df['category'].astype('category')

# Use datetime
df['date'] = pd.to_datetime(df['date'])
```

2. Chunking for large files:

```
# Read in chunks
chunks = []
for chunk in pd.read_csv('large_file.csv', chunksize=100000):
    processed = chunk[chunk['value'] > 0]  # Process each chunk
    chunks.append(processed)
df = pd.concat(chunks)
```

3. Vectorization over iteration:

```
# Bad: Loop
for i in range(len(df)):
    df.loc[i, 'new'] = df.loc[i, 'a'] + df.loc[i, 'b']

# Good: Vectorized
df['new'] = df['a'] + df['b']
```

Better: NumPy for complex operations

```
df['new'] = np.where(df['a'] > 0, df['a'] * df['b'], 0)
```

4. Use query() for filtering:

Slower

```
df[(df['a'] > 5) & (df['b'] < 10)]
```

Faster

```
df.query('a > 5 and b < 10')
```

5. Avoid append in loops:

Bad: O(n^2) complexity

```
df = pd.DataFrame()
for item in items:
    df = df.append(item)
```

Good: O(n) complexity

```
dfs = [process(item) for item in items]
df = pd.concat(dfs)
```

6. Use eval() for complex expressions:

Standard

```
df['result'] = df['a'] + df['b'] * df['c'] - df['d']
```

Faster with eval

```
df.eval('result = a + b * c - d', inplace=True)
```

7. Set index for frequent lookups:

```
df.set_index('id', inplace=True)
```

Now lookups are O(1) instead of O(n)

28. Explain the concept of time series functionality in Pandas.

Answer: Pandas has extensive time series capabilities for working with temporal data.

1. Creating datetime index:

Convert to datetime

```
df['date'] = pd.to_datetime(df['date'])
```

Set as index

```
df.set_index('date', inplace=True)
```

Create date range

```
dates = pd.date_range('2024-01-01', periods=365, freq='D')
```

Different frequencies

```
pd.date_range('2024-01-01', periods=12, freq='M')  # Month end
pd.date_range('2024-01-01', periods=24, freq='H')  # Hourly
pd.date_range('2024-01-01', periods=52, freq='W')  # Weekly
```

2. Time-based indexing and slicing:

Select by year

```
df['2024']
```

Select by year and month

```
df['2024-01']
```

Range selection

```
df['2024-01-01':'2024-01-31']
```

Partial string indexing

```
df.loc['2024-Q1']
```

3. Resampling (changing frequency):

Downsample to monthly

```
df.resample('M').mean()

# Upsample to daily and forward fill
df.resample('D').ffill()

# Multiple aggregations
df.resample('W').agg({
    'sales': 'sum',
    'price': 'mean',
    'quantity': ['sum', 'max']
})

# Custom labels
df.resample('M', label='left').sum()
```

4. Time shifts and lags:

```
# Shift data by periods
df['prev_day'] = df['value'].shift(1)
df['next_day'] = df['value'].shift(-1)

# Calculate returns
df['returns'] = df['price'].pct_change()

# Shift by time
df['value'].shift(freq='2D')  # Shift by 2 days
```

5. Time zone handling:

```
# Localize timezone
df.index = df.index.tz_localize('UTC')

# Convert timezone
```

```
df.index = df.index.tz_convert('US/Eastern')
```

6. Date components:

```
df['year'] = df.index.year

df['month'] = df.index.month

df['day'] = df.index.day

df['dayofweek'] = df.index.dayofweek

df['quarter'] = df.index.quarter
```

29. What is the purpose of plt.legend() and how do you use it?

Answer: legend() displays a legend box explaining what each plot element represents using labels. Labels are set via label parameter in plot functions. Legend location can be controlled using loc parameter ('upper right', 'best', etc.) or coordinates. Supports customization of font size, frame, transparency, and columns. Essential for multi-line plots to distinguish between different datasets.

```
fig, ax = plt.subplots(figsize=(10, 6))

ax.plot([1, 2, 3, 4], [1, 4, 2, 3], label='Dataset 1')

ax.plot([1, 2, 3, 4], [2, 3, 4, 5], label='Dataset 2')

ax.plot([1, 2, 3, 4], [3, 2, 1, 4], label='Dataset 3')

# Customize legend

ax.legend(loc='best',        # Auto best position
          frameon=True,      # Show frame
          fancybox=True,     # Rounded corners
          shadow=True,       # Drop shadow
          fontsize=12,
          title='Legend Title')

ax.set_xlabel('X-axis')

ax.set_ylabel('Y-axis')
```

ax.set_title('Plot with Legend')

ax.grid(True, alpha=0.3)

plt.show()

30. What is the difference between copy() and view in Pandas? Why is it important?

Answer: Understanding copy vs view is crucial to avoid unexpected behavior when modifying DataFrames.

View:

- A view is a reference to the original data
- Changes to the view affect the original DataFrame
- No additional memory is used

Copy:

- A copy creates a completely independent duplicate
- Changes to the copy don't affect the original
- Uses additional memory

Examples:

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

This creates a view (usually)

view = df[['A']]

view.loc[0, 'A'] = 999 # May raise SettingWithCopyWarning

This creates a copy

copy = df[['A']].copy()

copy.loc[0, 'A'] = 999 # Safe, doesn't affect original

Explicit copy

df_copy = df.copy() # Deep copy

df_copy = df.copy(deep=False) # Shallow copy

The SettingWithCopyWarning:

Chained assignment (causes warning)

df[df['A'] > 1]['B'] = 100 # Ambiguous: view or copy?

Correct approaches

Method 1: loc

df.loc[df['A'] > 1, 'B'] = 100

Method 2: Explicit copy

subset = df[df['A'] > 1].copy()

subset['B'] = 100

When views are created:

- Single column selection: df['col'] (view)
- Slicing: df[1:5] (view)
- Simple indexing: df.iloc[0:5] (view)

When copies are created:

- Multiple column selection: df[['col1', 'col2']] (copy)
- Boolean indexing: df[df['A'] > 0] (copy)
- Most operations that return DataFrames

Best practices:

1. Use .loc for explicit assignment
2. Call .copy() when you want to work with independent data
3. Be aware of chained indexing
4. Pay attention to SettingWithCopyWarning

31. You need to find duplicate records in a dataset. How would you identify and handle them?

Answer: Use duplicated() to identify duplicate rows and drop_duplicates() to remove them. Check duplicates based on specific columns using subset parameter. Decide whether to keep first, last, or remove all duplicates based on business logic. Analyze why duplicates exist - data entry errors, system issues, or valid repeated records. Always keep a backup before removing duplicates.

import pandas as pd

```
# Sample data with duplicates
data = {
    'student_id': [101, 102, 103, 101, 104, 103],
    'name': ['Raj', 'Priya', 'Amit', 'Raj', 'Sonia', 'Amit'],
    'marks': [85, 90, 78, 85, 88, 78]
}
df = pd.DataFrame(data)

print("Original Data:")
print(df)

# Find duplicates
print("\nDuplicate rows:")
print(df[df.duplicated()])

# Find duplicates based on specific column
print("\nDuplicate student_ids:")
print(df[df.duplicated(subset=['student_id'])])

# Remove duplicates (keep first occurrence)
df_clean = df.drop_duplicates(keep='first')
print("\nAfter removing duplicates:")
print(df_clean)

# Remove duplicates based on specific columns
df_clean2 = df.drop_duplicates(subset=['student_id', 'name'])
print("\nRemove based on student_id and name:")
print(df_clean2)
```

32. You need to analyze sales data and find top performing products. How would you do it?

Answer: Use groupby() to aggregate sales by product, calculate total revenue using sum(), and sort using sort_values(). Use head() to get top N products. Create visualizations using bar charts or pie charts to present findings. Calculate percentage contribution of each product. Consider time period, region, and category-wise analysis for deeper insights.

```
import pandas as pd
import matplotlib.pyplot as plt

# Sample sales data
sales_data = pd.DataFrame({
    'product': ['Laptop', 'Mobile', 'Tablet', 'Laptop', 'Mobile',
                'Laptop', 'Tablet', 'Mobile', 'Laptop', 'Tablet'],
    'quantity': [2, 5, 3, 1, 4, 3, 2, 6, 2, 1],
    'price': [50000, 20000, 30000, 50000, 20000,
              50000, 30000, 20000, 50000, 30000]
})
# Calculate revenue
sales_data['revenue'] = sales_data['quantity'] * sales_data['price']

# Group by product and calculate totals
product_sales = sales_data.groupby('product').agg({
    'quantity': 'sum',
    'revenue': 'sum'
}).reset_index()

# Sort by revenue
product_sales = product_sales.sort_values('revenue', ascending=False)

print("Product Performance:")
print(product_sales)

# Calculate percentage contribution
total_revenue = product_sales['revenue'].sum()
```

```
product_sales['percentage'] = (product_sales['revenue'] / total_revenue * 100).round(2)


print("\nWith Percentage:")
print(product_sales)


# Visualize top 3 products
top_3 = product_sales.head(3)
plt.figure(figsize=(10, 6))
plt.bar(top_3['product'], top_3['revenue'], color=['gold', 'silver', 'brown'])
plt.title('Top 3 Products by Revenue')
plt.xlabel('Product')
plt.ylabel('Revenue (₹)')
plt.show()
```

33. What is exception handling and why is it important?

Answer: Exception handling is a mechanism to handle runtime errors gracefully without crashing the program. It uses try-except blocks to catch and handle errors. Important because it prevents program termination, provides meaningful error messages to users, allows recovery from errors, and helps in debugging. Without exception handling, any error would crash the entire application.

```
# Without exception handling (program crashes)
# number = int("abc")  # ValueError - program stops

# With exception handling (program continues)
try:
    number = int("abc")
    print("Conversion successful")
except ValueError:
    print("Invalid input! Please enter a valid number.")
    number = 0
```

```
print(f"Number is: {number}")
print("Program continues running...")
```

34. How do you catch multiple exceptions in Python?

Answer: Multiple exceptions can be caught in three ways: (1) Multiple except blocks for different exceptions, (2) Single except with tuple of exceptions, (3) Using base Exception class (not recommended for specific handling). Use specific exceptions first, then general ones. Catching multiple exceptions allows different handling strategies for different error types.

```
def process_data(data, index):
    # Method 1: Multiple except blocks
    try:
        value = int(data[index])
        result = 100 / value
        return result
    except IndexError:
        print("Error: Index out of range")
        return None
    except ValueError:
        print("Error: Cannot convert to integer")
        return None
    except ZeroDivisionError:
        print("Error: Cannot divide by zero")
        return None

# Method 2: Tuple of exceptions (same handling)
def process_data2(data, index):
    try:
        value = int(data[index])
        result = 100 / value
        return result
```

```
    except (IndexError, ValueError, ZeroDivisionError) as e:
        print(f"Error occurred: {type(e).__name__} - {e}")
        return None

# Test cases
print("--- Method 1 ---")
process_data(['10', '20'], 5)     # IndexError
process_data(['abc', '20'], 0)    # ValueError
process_data(['0', '20'], 0)      # ZeroDivisionError

print("\n--- Method 2 ---")
process_data2(['10', '20'], 5)    # All handled together
```

35. What is logging and why is it better than using print statements?

Answer: Logging is a systematic way to record events, errors, and information during program execution using Python's logging module. Better than print because: (1) logs can be saved to files for later analysis, (2) different severity levels (DEBUG, INFO, WARNING, ERROR, CRITICAL), (3) can be easily enabled/disabled, (4) supports multiple outputs simultaneously, (5) includes timestamps and context automatically. Production applications should always use logging instead of print.

```
import logging

# Using print (not recommended for production)
print("User logged in")  # Goes to console, lost after restart
print("Error occurred")  # No severity level, timestamp

# Using logging (recommended)
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
```

```
logging.info("User logged in")        # Informational message
logging.warning("Low disk space")      # Warning message
logging.error("Database connection failed")  # Error message
logging.critical("System crash imminent")    # Critical message

# Logs can be saved to file
logging.basicConfig(
   filename='app.log',
   level=logging.INFO,
   format='%(asctime)s - %(levelname)s - %(message)s'
)

logging.info("This will be saved to app.log file")
```

36. if you have data in mongodb how to load data from it explain in step by step?

Answer: 1. Get connection info — MongoDB URI (or host:port), DB name, collection name, credentials.

2. Install client — pip install pymongo pandas (for Python).

3. Connect

```
from pymongo import MongoClient
client = MongoClient("mongodb://user:pass@host:27017")
db = client["my_db"]
coll = db["my_coll"]
```

4. Query — use find() with filter/projection to limit fields

5. Load into DataFrame

6. Close: client.close()

37. what is duck typing, give a example of it?

Answer: Duck typing means Python doesn't check the *type* of an object, it only checks if the object behaves like what's expected — i.e., has the required methods or attributes.

```
class Duck:
   def quack(self):
```

```
    print("Quack Quack!")

class Person:
  def quack(self):
    print("I can also quack like a duck!")

def make_it_quack(thing):
  thing.quack()  # no type checking

d = Duck()
p = Person()

make_it_quack(d)
make_it_quack(p)
```

38. what is public, protective and private attributes what is its important in class?

Answer: In Object-Oriented Programming (OOP), we often want to control how data (variables or methods) inside a class can be accessed or modified.
This is called Encapsulation — one of the core principles of OOP.

Python supports three levels of data access control for class members (attributes or methods):

1. Public Attributes

- These can be accessed from anywhere — inside or outside the class.
- There are no restrictions on their access or modification.
- Defined without any underscore prefix.

Example name: self.name

2. Protected Attributes

- These are meant to be accessed only within the class and its subclasses.
- They start with a single underscore (_) as a convention (not enforced by Python).
- It's just a *warning* to other programmers: "Use it carefully, this is internal data."

Example name: self._roll

3. Private Attributes

- These are completely hidden from outside the class.
- Defined using double underscore (__) before the variable name.
- Python performs name mangling, which changes the variable name internally (e.g. __marks becomes _ClassName__marks).
- This helps in data hiding and security.

Importance

1. **Encapsulation:** Keeps class data safe and organized.
2. **Data Hiding:** Prevents direct access to sensitive information.
3. **Controlled Access:** Encourages using getter/setter methods instead of direct modification.
4. **Code Maintenance:** Makes the class easier to update and debug later.

39. Explane encapsulation in OOPs ?

Answer: Encapsulation is one of the main principles of Object-Oriented Programming (OOP). It means binding data (variables) and methods (functions) that operate on that data into a single unit — a class.

It also helps to hide internal details of an object and protect data from direct access by the user. This is also known as Data Hiding.

Key Points:

1. Encapsulation = Data + Methods wrapped together in one class.
2. Protects the data by making variables **private or protected**.
3. Users interact with the data only through **public methods (getters and setters)**.
4. Helps maintain **security, reusability, and clarity** in code.

Example:

```
class Student:
  def __init__(self, name, marks):
    self.__name = name     # private variable
    self.__marks = marks   # private variable

  # getter methods
```

```
    def get_name(self):
        return self.__name

    def get_marks(self):
        return self.__marks

    # setter method
    def set_marks(self, marks):
        if 0 <= marks <= 100:
            self.__marks = marks
        else:
            print("Invalid marks!")

# object creation
s = Student("Amit", 85)

print(s.get_name())    # Access using getter
print(s.get_marks())   # Access using getter
s.set_marks(90)        # Modify using setter
print(s.get_marks())
```

40. What is a Regular Expression in Python?

Answer: A Regular Expression (regex) is a pattern used to match, search, or replace text. It is supported by Python's re module.

```
import re
text = "My age is 25"
if re.search(r"\d+", text):
    print("Digits found")
```

41. What is the difference between re.match() and re.search()?

Answer:

1. re.match() → checks only at the beginning of the string.
2. re.search() → checks anywhere in the string.

```
import re
txt = "Hello Python"
print(re.match("Hello", txt))   #  Match at start
print(re.search("Python", txt)) # Found later
```

42. How to check if a string starts with a specific word using regex?

Answer: Use ^ symbol → represents start of string.

Example:

```
import re
text = "Python is powerful"
if re.match(r"^Python", text):
   print("Starts with Python")
```

43. Write a Python program using regular expressions to extract all valid phone numbers from a text.
A valid phone number has 10 digits, and may include spaces, hyphens (-), or parentheses.

Answer:

```
 import re

text = "Call me at 98765-43210 or (91234 56789), office: 9999988888"

# Regex pattern for 10-digit numbers with optional symbols
pattern = r'\(?\d{5}[-\s]?\d{5}\)?'

phones = re.findall(pattern, text)
print(phones)
```

44. What is the use of the math module in Python?

Answer: The math module provides mathematical functions like square root, power, trigonometry, log, etc.
You must import math before using it.

```
import math

print(math.sqrt(16))  # 4.0

print(math.pow(2, 3))  # 8.0
```

45. What is the purpose of the statistics module?

Answer: The statistics module provides functions to calculate statistical values like mean, median, mode, stdev, etc.

```
import statistics as stats

data = [10, 20, 30, 40]

print(stats.mean(data))  # 25

print(stats.median(data)) # 25.0
```

46. How to calculate correlation or variance using the statistics module?

Answer:

1. variance() → measures how data points differ from the mean

2. correlation() → measures linear relation between two datasets

```
import statistics as stats

x = [10, 20, 30, 40]

y = [15, 25, 35, 45]


print(stats.variance(x))     # 166.666...

print(stats.correlation(x, y)) # 1.0 → perfect positive relation
```

47. How to get only the current date or time?

Answer:

1. date.today() returns only the current date.

2. datetime.now().time() returns only the current time.

```
from datetime import date, datetime
```

```
today = date.today()

current_time = datetime.now().time()

print("Date:", today)

print("Time:", current_time)
```

48. How to format date and time using strftime()?

Answer: strftime() converts a date or datetime object into a formatted string using special format codes.

```
from datetime import datetime

now = datetime.now()

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

print("Formatted datetime:", formatted)
```

49. How to get day, month, year, or weekday from a date?

Answer: Date objects have attributes .year, .month, .day and .weekday() to get date parts. Weekday returns 0 (Monday) to 6 (Sunday).

```
from datetime import date

d = date(2025, 10, 25)

print("Year:", d.year)

print("Month:", d.month)

print("Day:", d.day)

print("Weekday:", d.weekday())  # 0=Monday, 6=Sunday
```

50. Calculate the number of days left until a future event?

Answer: Can calculate how many days are left until a future date using timedelta by subtracting today's date from the event date.

```
from datetime import date


deadline = date(2025, 12, 31)

today = date.today()

days_left = (deadline - today).days
```

print("Days left until deadline:", days_left)

51. Convert between string and datetime?

Answer: Often, dates come as strings (from user input or files). strptime() converts strings to datetime objects, and strftime() formats datetime back to strings.

from datetime import datetime

date_str = "25-10-2025 14:30"

dt = datetime.strptime(date_str, "%d-%m-%Y %H:%M")

formatted = dt.strftime("%B %d, %Y at %I:%M %p")

print("Formatted date:", formatted)

52. Tell us how many lybraries we required for a datascinec project and why explain briefly?

Answer: brief explanation of the key libraries usually required for a data science project and why each is needed. I'll keep it concise but practical.

1. NumPy

- **Purpose:** Numerical computing, arrays, linear algebra, math operations.
- **Why needed:** Handles large datasets efficiently and provides fast vectorized operations.

2. Pandas

- **Purpose:** Data manipulation, cleaning, and analysis using **DataFrames**.
- **Why needed:** Makes it easy to read/write CSV, Excel, SQL; filter, group, and transform data.

3. Matplotlib

- **Purpose:** Basic data visualization (plots, graphs).
- **Why needed:** Helps understand data distributions and trends visually.

4. Seaborn

- **Purpose:** Advanced statistical plots built on top of Matplotlib.

- **Why needed:** Makes **correlation, heatmaps, pairplots** easy and visually appealing.

5. Scikit-learn

- **Purpose:** Machine learning library (classification, regression, clustering).
- **Why needed:** Provides ready-to-use algorithms, preprocessing, model evaluation, and pipelines.

6. SciPy

- **Purpose:** Scientific computing (optimization, statistics, linear algebra).
- **Why needed:** Useful for **statistical tests** and advanced mathematical operations.

7. Statsmodels

- **Purpose:** Statistical modeling, regression, hypothesis testing.
- **Why needed:** For **detailed statistical analysis** beyond scikit-learn.

8. TensorFlow / PyTorch

- **Purpose:** Deep learning frameworks for neural networks.
- **Why needed:** Needed when working with **image, text, or complex prediction models**.

9. NLTK / spaCy

- **Purpose:** Natural Language Processing (text preprocessing, tokenization).
- **Why needed:** Required for **text analytics, sentiment analysis, or NLP projects**.

10. OpenCV / PIL

- **Purpose:** Image processing.
- **Why needed:** For **computer vision projects**, reading and manipulating images.

53. If you are doing a ML project what will be you approch in python to make complete structure?

Answer: I'll focus on the project structure, stages, and best practices

1. Understand the Problem

Theory:

- Clearly define the **business or research problem**.
- Identify whether it's **supervised (regression/classification)** or **unsupervised (clustering, dimensionality reduction)**.
- Understand the **success criteria** (accuracy, RMSE, F1-score, business KPI).

Example:
Predicting customer churn in a telecom company → classification problem.

2. Collect Data

Theory:

- Gather data from **CSV/Excel, databases (SQL/MongoDB), APIs, or web scraping**.
- Ensure data is **relevant, sufficient, and representative**.

Python Tools:

- pandas for CSV/Excel
- SQLAlchemy or pymongo for database access
- requests or BeautifulSoup for web data

3. Explore and Understand Data (EDA)

Theory:

- Perform **Exploratory Data Analysis (EDA)** to understand patterns, relationships, and anomalies.
- Check for **missing values, duplicates, outliers**, and **data types**.
- Use **summary statistics and visualizations** to get insights.

Python Tools:

- pandas for statistics (.describe(), .info())
- matplotlib & seaborn for plots (histograms, scatterplots, heatmaps)

4. Data Cleaning and Preprocessing

Theory:

- Handle **missing values** (drop, impute, or predict).
- Encode categorical variables (Label Encoding, One-Hot Encoding).
- Normalize or scale numerical features for ML algorithms.
- Split **features (X)** and **target (y)**.

Python Tools:

- scikit-learn → SimpleImputer, StandardScaler, OneHotEncoder

5. Feature Engineering

Theory:

- Create new features that **improve model performance**.
- Reduce dimensionality if needed (PCA, feature selection).
- Handle feature correlations to avoid multicollinearity.

Python Tools:

- pandas for transformations
- scikit-learn for PCA, SelectKBest

6. Split Data

Theory:

- Split data into **training, validation, and test sets** to evaluate model generalization.
- Typical split: 70-80% training, 20-30% testing (or use cross-validation).

Python Tools:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

7. Choose and Train ML Model

Theory:

- Select algorithm based on problem type:
    - Regression → Linear Regression, Random Forest, XGBoost
    - Classification → Logistic Regression, Decision Tree, Random Forest, SVM
    - Clustering → KMeans, DBSCAN
- Train the model using the **training set**.

Python Tools:

- scikit-learn for classical ML
- XGBoost, LightGBM, CatBoost for advanced models
- TensorFlow or PyTorch for deep learning

8. Model Evaluation

Theory:

- Evaluate model performance using **appropriate metrics**:
    - Classification → Accuracy, F1-score, ROC-AUC
    - Regression → MSE, RMSE, R^2
- Use **cross-validation** to check for overfitting.

Python Tools:

from sklearn.metrics import accuracy_score, confusion_matrix, mean_squared_error

9. Hyperparameter Tuning

Theory:

- Optimize model parameters for better performance using **GridSearchCV** or **RandomizedSearchCV**.
- Helps improve accuracy/generalization without overfitting.

Python Tools:

from sklearn.model_selection import GridSearchCV

10. Model Deployment (Optional for projects)

Theory:

- Once the model is finalized, deploy it for real-world use.
- Options include:
    - Save model with pickle or joblib
    - Deploy as **API using Flask/FastAPI**
    - Integrate with dashboards or web apps

Python Tools:

- pickle, joblib for saving models
- Flask, FastAPI for serving models

11. Project Structure Recommendation

Theory:
Organize project folders for clarity and reproducibility:

ML_Project/

│

├── data/ # Raw and processed datasets

├── notebooks/ # Jupyter notebooks for exploration

├── scripts/ # Python scripts for preprocessing, training

├── models/ # Saved models (pickle/joblib)

├── reports/ # EDA reports, plots, metrics

├── requirements.txt # Project dependencies

└── README.md # Project description

54. What are some common debugging tools or techniques you use to find and fix errors in your code?

Answer:
Common debugging tools and techniques include:

- Print statements: Adding print() to check variable values during execution.
- Debugger tools: Using built-in debuggers (e.g., Python's pdb, VS Code Debugger) to step through code line by line.
- Error messages: Reading the error traceback to locate the issue.
- Rubber Duck Debugging: Explaining your code logic to someone (or even a "rubber duck") to spot mistakes.
- Logging: Using logs to track what happens in the program over time.

55. How would you approach debugging an issue that only occurs intermittently (e.g., once every few hours)?

Answer: Approach step-by-step:

1. Add logging: Record key steps and variable values in a log file.
2. Reproduce the issue: Try to find the exact conditions when it happens.
3. Check timing or data issues: Often caused by race conditions, network delays, or memory leaks.
4. Use debugging tools: Use profilers or monitoring tools to watch program behavior over time.

5. Document findings: Note what triggers the issue for future fixes.

Example:
If an app crashes randomly, add log statements to track which function runs last before the crash.