100 Python Interview Questions and Answers

APOORVA IYER

PYTHON BASICS AND CORE CONCEPTS

1. What is Python, and why is it commonly used in data analytics?
Python is a high-level, interpreted programming language known for its simplicity and readability. It's widely used in data analytics due to its rich ecosystem of libraries such as Pandas, NumPy, and Matplotlib, which make data manipulation, analysis, and visualization more accessible.

---

2. How do you install external libraries in Python?
External libraries in Python can be installed using package managers like pip. For example, to install the Pandas library, you can use the command:

pip install pandas

---

3. What is the difference between a list and a tuple in Python?
Lists are mutable, meaning they can be modified after creation. Tuples are immutable, meaning their content cannot be changed. Lists use square brackets [ ], while tuples use parentheses ( ). Tuples are generally faster and are used when data should remain constant.

---

4. What is the difference between == and is in Python?

- == checks for value equality — whether the values of two objects are the same.
- is checks for object identity — whether two variables refer to the same object in memory.
  Example:

a = [1, 2]

b = a

a == b # True

a is b # True

---

5. What is the use of *args and kwargs in Python functions?

- *args allows a function to accept any number of positional arguments.
- **kwargs allows it to accept any number of keyword arguments.
  This makes functions more flexible.
  Example:
- def func(*args, **kwargs):
  print(args)
  print(kwargs)

---

6. What is the purpose of the with statement in Python?
The with statement is used for resource management. It ensures that files or resources are properly closed or released after use, even if an error occurs.
Example:

```
with open("file.txt", "r") as f:
    data = f.read()
```

---

7. What is a generator in Python?
A generator is a function that yields items one at a time using the yield keyword. It doesn't store the entire sequence in memory, making it memory-efficient.
Example:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

---

8. What is the difference between shallow copy and deep copy in Python?

- Shallow Copy: Copies the outer object but not nested objects.
- Deep Copy: Copies all objects recursively.
  Use the copy module:

```
import copy
copy.copy(obj)       # Shallow
copy.deepcopy(obj)   # Deep
```

---

9. What are Python magic methods?
Magic methods (or dunder methods) are special methods with double underscores, used to implement built-in behavior. Examples:

- __init__ — constructor
- __str__ — string representation
- __len__, __getitem__, __eq__, etc.

---

10. What is the difference between str() and repr()?

- str() is used to return a readable and user-friendly version of an object.
- repr() is used to return an unambiguous string, mostly for debugging.
  Example:

```
x = "Hello"
print(str(x))   # Hello
print(repr(x))  # 'Hello'
```

DATA ANALYSIS WITH PANDAS AND NUMPY

11. What is Pandas, and how is it used in data analysis?
Pandas is a Python library designed for data manipulation and analysis. It provides two core data structures:

- DataFrame: a 2D labeled data structure
- Series: a 1D labeled array
  It simplifies data wrangling tasks like cleaning, merging, filtering, and transforming tabular data.

---

12. How do you read a CSV file into a DataFrame using Pandas?
Use the read_csv() function from Pandas to load CSV data:

```
import pandas as pd
df = pd.read_csv('filename.csv')
```

---

13. How do you handle missing values in a dataset using Pandas?
Pandas offers several methods:

- isnull() and notnull() — detect missing values
- dropna() — remove rows or columns with missing values
- fillna() — fill missing values with a specific value
- interpolate() — fill missing values using interpolation

---

14. How do you merge two DataFrames in Pandas?
Use the merge() function, which supports SQL-style joins:

```
merged_df = pd.merge(df1, df2, on='common_column', how='inner')
```

---

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

- loc: Label-based indexing. Selects rows and columns using labels.

- iloc: Integer-based indexing. Selects by row/column number.
  Example:

df.loc[0, 'name']

df.iloc[0, 1]

---

16. How do you group data in Pandas and perform aggregation?
Use groupby() to split the data and apply aggregation functions:

df.groupby('category')['sales'].sum()

---

17. How do you apply a function to each element in a Pandas Series?
Use the apply() method:

df['column'].apply(lambda x: x + 1)

---

18. What is the difference between map(), apply(), and applymap()?

- map(): Element-wise on Series
- apply(): Used on Series or DataFrame (row/column-wise)
- applymap(): Element-wise on entire DataFrame

---

19. How do you handle categorical variables in Python for data analysis?

- Label Encoding: Use LabelEncoder to convert categories into integers
- One-Hot Encoding: Use pd.get_dummies() to create binary columns
- Frequency or Target Encoding: Replace with aggregated numerical values

---

20. How do you convert a string column to datetime in Pandas?
Use pd.to_datetime():

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

---

21. What is method chaining in Pandas?
Method chaining is the technique of calling multiple Pandas operations in one expression:

```
df_cleaned = (df.dropna()
        .assign(new_col=lambda x: x['col'] * 2)
        .query('col > 100'))
```

---

22. What is the difference between pd.concat() and pd.merge()?

- pd.concat(): Combines dataframes along rows or columns
- pd.merge(): Joins two dataframes using keys like SQL joins

---

23. How do you filter rows in a DataFrame based on multiple conditions?
Use logical operators & and |:

df[(df['age'] > 30) & (df['income'] > 50000)]

---

24. What is the difference between .values, .to_numpy(), .tolist(), and .array in Pandas?

- .values: Older way to get NumPy array
- .to_numpy(): Preferred way to convert to NumPy
- .tolist(): Converts to Python list
- .array: Returns internal ExtensionArray

---

25. What is the difference between a DataFrame and a Series in Pandas?

- Series: A 1D labeled array
- DataFrame: A 2D labeled data structure made of multiple Series
  DataFrames allow for tabular data handling similar to Excel or SQL tables.

---

26. How do you read a JSON file into a DataFrame?
Use read_json():

df = pd.read_json('data.json')

---

27. How do you perform a left join in Pandas?
Use merge() with how='left':

pd.merge(df1, df2, on='key_column', how='left')

---

28. How do you create pivot tables in Pandas?
Use pivot_table():

df.pivot_table(index='Region', columns='Product', values='Sales', aggfunc='sum')

---

29. What is the use of the transform() function in Pandas?
transform() applies a function to each group and returns a DataFrame with the same shape:

df['normalized'] = df.groupby('Category')['Value'].transform(lambda x: (x - x.mean()) / x.std())

---

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

- merge(): SQL-style joins on columns
- join(): Join on index
- concat(): Stack DataFrames vertically or horizontally

NUMPY, STATISTICS & MATH OPERATIONS

31. What is NumPy, and why is it used in data analysis?
NumPy (Numerical Python) is a powerful open-source library used for numerical computing in Python. It provides a high-performance multidimensional array object called ndarray, along with a vast set of mathematical functions to perform operations efficiently on these arrays.

NumPy plays a foundational role in the data analysis and scientific computing ecosystem because of the following reasons:

- **Efficient Memory Usage**: NumPy arrays consume less memory and provide better performance compared to traditional Python lists due to their fixed type and contiguous memory allocation.
- **Vectorized Operations**: NumPy eliminates the need for explicit loops by allowing element-wise operations directly on arrays, significantly improving speed and reducing code complexity.
- **Interoperability**: NumPy integrates seamlessly with other popular Python libraries such as Pandas, scikit-learn, TensorFlow, and Matplotlib, making it a core part of any data pipeline.
- **Scientific Functions**: It provides tools for performing linear algebra, Fourier transforms, statistical analysis, and random number generation.
- **Basis for Pandas and Other Libraries**: Pandas, which is the most widely used library for tabular data analysis, is built on top of NumPy arrays, making an understanding of NumPy essential for deeper analytical work.

In summary, NumPy is the backbone of numerical and matrix computation in Python and is widely adopted by analysts and data scientists due to its performance, simplicity, and integration.

---

32. Explain the concept of broadcasting in NumPy.
Broadcasting is a set of rules used by NumPy to perform element-wise arithmetic operations on arrays of different shapes and sizes without requiring manual replication of data.

When working with arrays of different dimensions, NumPy "broadcasts" the smaller array across the larger one so that they have compatible shapes. This avoids the overhead of copying data and makes computations faster and memory efficient.

Example:

```
import numpy as np


a = np.array([[1, 2, 3],
              [4, 5, 6]])


b = np.array([10, 20, 30])


result = a + b
print(result)
```

Output:

```
[[11 22 33]
 [14 25 36]]
```

In this example, array b is automatically broadcasted to match the shape of array a. The values from b are applied row-wise to a without explicitly repeating b.

How broadcasting works:

1. Compare the shapes of both arrays from right to left.
2. Dimensions are compatible when:
    - They are equal, or
    - One of them is 1.
3. NumPy then "stretches" the smaller shape to match the larger one.

Benefits of broadcasting:

- Avoids memory duplication.
- Increases code readability and conciseness.
- Boosts computation speed for large datasets.

Common use cases:

- Adding a scalar or 1D array to a matrix.
- Normalizing rows or columns in datasets.
- Element-wise operations on mismatched array shapes.

Understanding broadcasting is critical in numerical computing because many high-performance computations rely on this concept to work efficiently.

---

33. How do you calculate the correlation between two variables in pandas?
In data analysis, correlation measures the degree to which two variables move in relation to each other. The most commonly used type is **Pearson correlation**, which captures linear relationships.

Pandas provides a built-in method called .corr() to calculate correlation coefficients between columns of a DataFrame or between two Series.

Example:

```
import pandas as pd


df = pd.DataFrame({
    'height': [160, 165, 170, 175, 180],
    'weight': [55, 60, 65, 70, 75]
})


correlation = df['height'].corr(df['weight'])
print(correlation)
```

Output:

1.0

This output indicates a perfect positive linear correlation.

Types of correlation supported by pandas:

- method='pearson' (default): measures linear correlation.
- method='spearman': measures rank-based correlation (monotonic).
- method='kendall': measures ordinal association.

Why this is useful:

- Identifying strong or weak relationships between variables.
- Detecting multicollinearity in regression analysis.
- Selecting features based on correlation with the target variable.

Important Note: Correlation does not imply causation. Two variables can be correlated due to confounding factors, so additional analysis is often needed.

---

34. What is the difference between np.array() and np.asarray()?
Both np.array() and np.asarray() are used to convert Python sequences (like lists or tuples) into NumPy arrays, but they behave differently in how they handle existing arrays.

Feature	**np.array()**	**np.asarray()**
Always creates copy?	Yes, creates a new array by default	No, returns input if it's already an array
Efficiency	Less efficient if input is already array	More efficient for avoiding duplication
Use case	When a fresh copy is needed	When avoiding redundancy is important

Example:

```
import numpy as np


x = np.array([1, 2, 3])
y = np.asarray(x)


print(x is y)  # Output: True
```

In this example, asarray() avoids making a new copy, saving memory when working with large datasets.

---

35. How do you handle categorical variables with high cardinality?
High cardinality refers to categorical variables that have a large number of unique values (e.g., Zip Code, Product ID). Handling such variables carefully is crucial because naive one-hot encoding can result in an explosion of features, causing overfitting and slow model training.

Here are common techniques to handle high-cardinality variables:

1. **Frequency Encoding**
   Replace each category with its frequency (number of occurrences) in the dataset.
2. df['category_freq'] = df['category'].map(df['category'].value_counts())
3. **Target Encoding**
   Replace categories with the average value of the target variable (e.g., mean salary for each department).
   Be cautious of **data leakage**, especially during model evaluation. Use K-fold target encoding or perform encoding only on training data.
4. **Hash Encoding (Feature Hashing)**
   Use a hash function to map categories to a fixed number of columns. This reduces

dimensionality and works well for very large cardinalities. Used in libraries like CategoryEncoders.

5. **Embeddings (for Deep Learning)**
   Convert categories into dense vectors using neural networks. Useful for large-scale recommendation systems or NLP.

6. **Clustering-Based Grouping**
   Group rare categories into an 'Other' category or cluster similar categories using business logic or statistical metrics.

When to choose what?

- Use **frequency/target encoding** for tree-based models like XGBoost.
- Use **one-hot encoding** for linear models if cardinality is manageable.
- Use **embeddings** for deep learning or NLP problems.

📌 **Pro Tip:** Always validate encodings with cross-validation to ensure they improve model performance.

---

36. What is the purpose of the enumerate() function in Python?
The enumerate() function adds a counter to an iterable and returns it as an enumerate object, which is useful for retrieving both the **index** and **value** during iteration.

Syntax:

enumerate(iterable, start=0)

Example:

```
colors = ['red', 'green', 'blue']
for index, color in enumerate(colors, start=1):
    print(index, color)
```

Output:

1 red

2 green

3 blue

Why it's useful:

- It improves code readability.
- Avoids the need for a separate counter variable.
- Helps when working with loops that require both index and value.

In interviews, it's important to also explain that enumerate() is often used in:

- Looping through lists and maintaining position.

- Debugging and logging.
- Pairing index with row or element during iteration.

---

37. How do you detect and handle multicollinearity in a dataset?
Multicollinearity occurs when two or more independent variables in a dataset are highly correlated. This leads to **unstable coefficients** in regression models and makes it difficult to determine the effect of individual variables.

How to detect multicollinearity:

1. **Correlation Matrix**
   Use .corr() to visualize correlations between numeric variables.
2. import seaborn as sns
3. sns.heatmap(df.corr(), annot=True)
4. **Variance Inflation Factor (VIF)**
   VIF quantifies how much the variance of a coefficient is inflated due to multicollinearity.
5. from statsmodels.stats.outliers_influence import variance_inflation_factor
6. vif_data = pd.DataFrame()
7. vif_data['VIF'] = [variance_inflation_factor(df.values, i) for i in range(df.shape[1])]

How to handle it:

- **Remove one of the correlated variables**: Keep the one with higher relevance.
- **Combine features**: Use PCA or feature engineering to combine highly correlated variables.
- **Regularization**: Use Ridge or Lasso regression to penalize large coefficients and reduce multicollinearity.

📌 Multicollinearity doesn't affect the predictive power of the model drastically but makes interpretation unreliable.

---

38. What is the difference between any() and all() functions in Python?
Both any() and all() are built-in functions that work with iterables (lists, tuples, sets, etc.), typically used in conditions and filters.

Function	Description	Example	Result
any()	Returns True if **at least one** element is True	any([False, True, False])	True
all()	Returns True if **all** elements are True	all([True, True, True])	True

Use cases:

- any() is often used when checking if a list contains at least one True, non-zero, or valid value.
- all() is used for validating that all inputs or conditions meet a requirement.

Example in data filtering:

df = pd.DataFrame({'A': [1, 0, 3], 'B': [4, 5, 0]})

df['has_zero'] = df[['A', 'B']].apply(lambda x: not all(x), axis=1)

This adds a new column that flags rows with zero values.

---

39. How do you perform a left join in pandas?
A **left join** combines rows from two DataFrames, keeping all rows from the left DataFrame and matching rows from the right DataFrame. If no match is found in the right DataFrame, the result will contain NaN.

Syntax:

pd.merge(left_df, right_df, how='left', on='key_column')

Example:

df1 = pd.DataFrame({'id': [1, 2, 3], 'name': ['A', 'B', 'C']})

df2 = pd.DataFrame({'id': [2, 3], 'score': [85, 90]})

result = pd.merge(df1, df2, how='left', on='id')

Output:

	id	name	score
0	1	A	NaN
1	2	B	85.0
2	3	C	90.0

Key points:

- how='left' ensures no rows are dropped from the left DataFrame.
- on='column_name' specifies the common key for joining.
- Missing values from unmatched rows are filled with NaN.

📌 Left joins are commonly used in real-world datasets where one table has master records and another has related transactional or lookup information.

---

40. Explain the use of the zip() function in Python.
The zip() function combines multiple iterables (e.g., lists, tuples) into a single iterator of tuples. It's particularly useful for parallel iteration over multiple sequences.

Example:

```
names = ['Alice', 'Bob']
scores = [85, 92]
combined = list(zip(names, scores))
print(combined)
```

Output:

```
[('Alice', 85), ('Bob', 92)]
```

Use cases:

- Creating dictionaries: dict(zip(keys, values))
- Iterating over multiple lists simultaneously.
- Unzipping data using zip(*zipped_data)

Unzipping Example:

```
zipped = [('a', 1), ('b', 2)]
letters, numbers = zip(*zipped)
```

Output:

```
letters = ('a', 'b')
numbers = (1, 2)
```

📌 zip() is memory efficient as it returns an iterator, and it's widely used in data transformation, pairing, and looping patterns.

---

41. How do you calculate the correlation between two variables in pandas?

Correlation measures the **linear relationship** between two variables. In pandas, you can calculate it using the .corr() method, which by default uses the **Pearson correlation coefficient**.

Syntax:

```
df['column1'].corr(df['column2'])
```

Example:

```
import pandas as pd

df = pd.DataFrame({
   'height': [150, 160, 170, 180, 190],
   'weight': [50, 55, 65, 70, 80]
})
```

```
correlation = df['height'].corr(df['weight'])
print(correlation)  # Output: 0.991 (strong positive correlation)
```

Types of correlation:

- **Pearson**: Measures linear correlation (default).
- **Kendall** and **Spearman**: Use .corr(method='kendall') or .corr(method='spearman') for non-parametric data.

When to use:

- Use Pearson when data is normally distributed and linear.
- Use Spearman/Kendall for ranked or non-linear data.

📌 A value near 1 indicates strong positive correlation; near -1 indicates strong negative correlation; near 0 indicates no correlation.

---

42. What is the difference between np.array() and np.asarray()?
Both np.array() and np.asarray() are used to convert Python sequences into NumPy arrays, but there's a **key difference** in how they handle existing NumPy arrays.

Function	Behavior
np.array()	Always creates a **new copy** of the array
np.asarray()	Returns the original array if it's already a NumPy array (no copy)

Example:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.asarray(a)

print(a is b)  # Output: True
```

Why use asarray()?

- More efficient when you're not sure if the input is already a NumPy array.
- Prevents unnecessary memory usage.

When to use array()?

- When you explicitly need a **copy** of the input object to avoid modifying the original.

📌 Use np.asarray() in performance-sensitive operations to avoid duplication of existing arrays.

---

43. How do you read a JSON file into a pandas DataFrame?
JSON (JavaScript Object Notation) is a widely used format for data exchange. You can read it into pandas using pd.read_json().

Syntax:

```
df = pd.read_json('data.json')
```

Example JSON file:

```
[
  {"name": "Alice", "age": 25},
  {"name": "Bob", "age": 30}
]
```

Read into pandas:

```
import pandas as pd


df = pd.read_json('sample.json')
```

Output:

```
    name  age
0  Alice   25
1    Bob   30
```

Advanced Options:

- If your JSON file is nested, you can normalize it using json_normalize() from pandas.json_normalize().
- If reading from an API, use pd.read_json(url).

📌 Make sure the JSON file is properly formatted—pandas expects an array of records or a dict of dicts.

---

44. Explain the use of the groupby() function in pandas.
The groupby() function is one of pandas' most powerful features. It is used to group data based on one or more keys and perform **aggregations or transformations** on those groups.

Syntax:

```
df.groupby('column_name').agg_function()
```

Example:

```
df = pd.DataFrame({
    'department': ['HR', 'HR', 'IT', 'IT'],
    'salary': [30000, 40000, 50000, 60000]
```

})

avg_salary = df.groupby('department')['salary'].mean()

Output:

department

HR 35000.0

IT 55000.0

Functions used with groupby:

- .sum(), .mean(), .count(), .max(), .min()
- .agg({'col1': 'sum', 'col2': 'mean'}) for multiple aggregations
- .transform() to apply a function and keep the original DataFrame shape

Use cases:

- Calculating KPIs per region or category
- Customer segmentation
- Time series analysis by day/month/year

📌 Groupby follows a **split-apply-combine** pattern: it splits the data into groups, applies a function, and combines the results.

---

45. How do you handle duplicate rows in a DataFrame?
Duplicate rows can bias your analysis, especially in aggregations or machine learning tasks.

To identify duplicates:

df.duplicated()

To remove duplicates:

df = df.drop_duplicates()

Advanced options:

- Use subset=['col1', 'col2'] to check duplicates only on selected columns.
- Use keep='first' or keep='last' to control which duplicate to retain.

Example:

df = pd.DataFrame({

'id': [1, 2, 2, 3],

'name': ['Alice', 'Bob', 'Bob', 'Charlie']

})

```
df_clean = df.drop_duplicates()
```

Output:

```
   id    name
0   1   Alice
1   2     Bob
3   3 Charlie
```

📌 Always check for and handle duplicates before performing aggregations or model training.

---

46. What is the difference between isnull() and notnull() in pandas?
These are pandas functions used to identify missing data in a DataFrame or Series.

Function	Description
isnull()	Returns True for missing values (NaN)
notnull()	Returns True for non-missing values

Example:

```
df = pd.DataFrame({'A': [1, None, 3]})
df['A'].isnull()
```

Output:

```
0    False
1     True
2    False
```

Typical uses:

- Filtering missing values: df[df['A'].isnull()]
- Counting nulls: df.isnull().sum()
- Filling nulls: df.fillna(0)

📌 Proper handling of missing data is a critical part of data cleaning and preprocessing.

---

47. How do you apply a function to each row in a DataFrame?
Use df.apply() with axis=1 to apply a function **row-wise**.

Syntax:

```
df.apply(function, axis=1)
```

Example:

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

df['sum'] = df.apply(lambda row: row['A'] + row['B'], axis=1)
```

Output:

```
   A  B  sum
0  1  3    4
1  2  4    6
```

Use cases:

- Row-wise calculations (e.g., total price, net income)
- Applying complex logic that depends on multiple columns

📌 .apply() is flexible but slower for large datasets; vectorized operations are faster when available.

---

48. What are decorators in Python?

A decorator is a function that takes another function as input, **adds some functionality**, and returns it. It's commonly used to **modify behavior** without changing the original function code.

Basic structure:

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

@decorator
def greet():
    print("Hello")

greet()
```

Output:

Before function call

Hello

After function call

Use cases:

- Logging
- Timing
- Caching
- Access control and authentication

Built-in decorators:

- @staticmethod
- @classmethod
- @property
- @lru_cache (for memoization)

📌 Decorators follow the **DRY (Don't Repeat Yourself)** principle and are heavily used in Flask, Django, and FastAPI.

CORE PYTHON PROGRAMMING

49. What is the difference between .at[] and .iat[] in pandas?
.at[] and .iat[] are accessor methods provided by pandas to access scalar values from a DataFrame or Series.

- .at[] is used for **label-based** scalar access. It is optimized for retrieving or setting a **single value** in a DataFrame by specifying the **row label and column label**.
- .iat[] is used for **integer-based** scalar access. It is optimized for retrieving or setting a **single value** by using the **integer positions** of the row and column.

Why it matters in interviews:
Understanding these accessors helps optimize performance when dealing with large datasets, especially when frequent access to individual values is required.

Example:

import pandas as pd

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


print(df.at[0, 'A'])   # Output: 1
print(df.iat[1, 1])    # Output: 4
```

---

50. What is the purpose of the super() function in Python?
The super() function is used to **call methods from a parent class** in a subclass. It is especially useful in **object-oriented programming**, particularly when you want to extend or override the functionality of a parent method without rewriting the entire method.

Key Benefits:

- Promotes **code reuse**
- Supports **multiple inheritance**
- Helps maintain **clean and modular code**

Interview Insight:
Understanding super() is critical in object-oriented interviews, especially when explaining inheritance, constructor chaining, or method overriding.

Example:

```
class Parent:
    def greet(self):
        print("Hello from Parent")


class Child(Parent):
    def greet(self):
        super().greet()  # Calls the parent method
        print("Hello from Child")


child = Child()
child.greet()
# Output:
# Hello from Parent
# Hello from Child
```

---

51. What is list comprehension and how is it used?
List comprehension is a **compact and elegant** way to create lists in Python. It allows for writing **iterative logic in a single line** by combining for loops and optional if conditions.

Syntax:

[expression for item in iterable if condition]

Why interviewers ask:
It tests your ability to write concise, Pythonic code and your grasp on control structures and data handling.

Example:

```
squares = [x**2 for x in range(5)]  # [0, 1, 4, 9, 16]


# With condition
evens = [x for x in range(10) if x % 2 == 0]  # [0, 2, 4, 6, 8]
```

---

52. What is a context manager in Python?
A **context manager** is a Python construct that **manages the setup and teardown** of resources like files, network connections, or database sessions.

It ensures that resources are properly acquired and released, especially in cases where exceptions might occur. Context managers are typically used with the with statement.

Built-in example:

```
with open('file.txt', 'r') as file:
    data = file.read()
# File is automatically closed, even if an error occurs.
```

Custom context manager using class:

```
class MyContext:
    def __enter__(self):
        print("Entering context")
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting context")


with MyContext():
    print("Inside context")
```

---

53. How do you use regular expressions in Python?
Regular expressions (regex) are used for **pattern matching and text parsing**. Python provides the re module to work with regex.

Key methods:

- re.search(): Searches for the pattern anywhere in the string.
- re.match(): Checks for a match at the beginning.
- re.findall(): Returns all matching substrings.
- re.sub(): Replaces substrings matching the pattern.

Example:

```
import re


text = "There are 12 apples and 3 oranges"
numbers = re.findall(r'\d+', text)  # ['12', '3']
```

Use case in data roles:
Regex is crucial for cleaning and extracting information from unstructured text fields in datasets, logs, or web-scraped data.

---

54. What is the purpose of the yield keyword in Python?
The yield keyword is used to **create generators**. Unlike return, which terminates a function, yield **pauses the function**, saving its state for the next call.

Benefits:

- **Memory-efficient**: Values are yielded one at a time.
- Useful for **large datasets** or **infinite sequences**.

Example:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1


for num in countdown(3):
    print(num)
# Output: 3, 2, 1
```

Interview advantage:
Demonstrates your understanding of memory management, lazy evaluation, and performance-aware design.

---

55. How can you improve the performance of a Python script?
Here are several strategies:

- **Use built-in functions and libraries**: They are optimized in C.
- **Avoid unnecessary loops**: Use list comprehensions and generators.
- **Profile your code** using tools like cProfile, line_profiler, or timeit.
- **Use NumPy or pandas**: For large numeric or tabular data.
- **Leverage multiprocessing** for CPU-bound tasks.
- **Minimize global variable access**.
- **Avoid redundant operations** (e.g., don't call the same function repeatedly inside a loop).

Example profiling with cProfile:

```
python -m cProfile my_script.py
```

Pro Tip:
Showcasing performance optimization efforts in your portfolio projects impresses interviewers.

---

56. What are Python data classes and when should you use them?
Data classes, introduced in Python 3.7 via the dataclasses module, provide a simple way to define classes that are mainly used to **store data**.

With the @dataclass decorator, Python automatically generates special methods like __init__(), __repr__(), __eq__() based on the class attributes. This saves time and reduces boilerplate code.

Use Cases:

- When you need lightweight classes to hold attributes (e.g., records, configuration objects)
- When you want default values, type hints, and immutability features

Example:

```
from dataclasses import dataclass


@dataclass
class Employee:
    name: str
    age: int
    department: str = "Data"
```

```
emp = Employee("Apoorva", 29)
print(emp)
# Output: Employee(name='Apoorva', age=29, department='Data')
```

Interview Tip:
Be ready to explain the advantages of data classes over regular classes and how they improve readability and productivity in data-heavy applications.

---

57. What are magic methods in Python?
Magic methods (also called **dunder methods** because they begin and end with double underscores) are special methods that allow you to define how your objects behave with **built-in functions and operators**.

Examples include:

- __init__ – Constructor
- __str__ – String representation
- __len__, __getitem__, __setitem__ – Container behavior
- __eq__, __lt__ – Comparison behavior

Use Case: They enable operator overloading, custom object printing, object iteration, and more.

Example:

```
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return f'Book: {self.title}'

book = Book("Python Basics")
print(book)
# Output: Book: Python Basics
```

Why interviewers ask this:
To assess your knowledge of Python's object model and how well you can customize class behavior.

---

58. How do you flatten a nested list in Python?
Flattening a nested list involves converting a list of lists (or arbitrarily nested lists) into a single list.

Using recursion:

```
def flatten(lst):
    result = []
    for item in lst:
        if isinstance(item, list):
            result.extend(flatten(item))
        else:
            result.append(item)
    return result


print(flatten([1, [2, [3]], [4, 5]]))  # Output: [1, 2, 3, 4, 5]
```

Other methods include:

- Using itertools.chain for one-level nested lists
- Using numpy.ravel() for numpy arrays

Interview Relevance:
Flattening is a frequent task in data preprocessing, especially in JSON parsing or ETL operations.

---

59. What is the purpose of functools.lru_cache()?
functools.lru_cache is a **memoization decorator** used to **cache the results** of expensive function calls. It stands for **Least Recently Used Cache** and is helpful for recursive algorithms and repeated computations.

Benefits:

- Speeds up recursive functions (like Fibonacci, DFS)
- Reduces redundant computations
- Manages memory by discarding old results

Example:

```
from functools import lru_cache


@lru_cache(maxsize=128)
def fib(n):
    if n < 2:
```

```
    return n
return fib(n-1) + fib(n-2)


print(fib(50))  # Output: 12586269025
```

Pro Tip:
You can explain this in interviews when asked how you optimize recursive logic or avoid recalculating values.

---

60. What are the most important data types in Python for analysts?
Python offers several built-in and library-supported data types that are crucial for data analysts:

- int, float: For numerical calculations
- str: For handling text data
- list: Mutable ordered sequences
- tuple: Immutable ordered sequences
- set: Unordered collection of unique elements
- dict: Key-value mappings
- bool: True/False logic
- DataFrame (from pandas): 2D tabular data structure, most commonly used in data analysis
- Series (from pandas): 1D labeled array

Why this matters in interviews:
Demonstrating fluency with data types shows you're prepared to clean, transform, and model real-world data using Python.

---

61. How do you convert a DataFrame to a dictionary in pandas?
You can convert a DataFrame into a Python dictionary using the .to_dict() method in pandas. This is helpful when you want to export tabular data into a key-value format, especially for further processing, APIs, or integration into applications.

Syntax:

```
df.to_dict(orient='records')
```

Explanation of orient parameter options:

- 'dict': {column -> {index -> value}}
- 'list': {column -> [values]}
- 'records': [{column -> value}, …]
- 'series': {column -> Series}

- 'split': {‘index’ -> [...], ‘columns’ -> [...], ‘data’ -> [...]}
- 'index': {index -> {column -> value}}

Example:

```
import pandas as pd

df = pd.DataFrame({
   'Name': ['Alice', 'Bob'],
   'Age': [25, 30]
})

print(df.to_dict(orient='records'))
# Output: [{'Name': 'Alice', 'Age': 25}, {'Name': 'Bob', 'Age': 30}]
```

Why this matters:
This operation is often used when integrating pandas with web applications, databases, or APIs, where JSON or dictionary structures are required.

---

62. How do you calculate descriptive statistics in pandas?
Descriptive statistics help summarize the basic features of a dataset. In pandas, the .describe() method provides a quick overview of numeric columns, including:

- Count
- Mean
- Standard deviation
- Min and Max
- 25%, 50% (median), and 75% percentiles

Example:

```
import pandas as pd

df = pd.DataFrame({
   'Sales': [100, 150, 200, 250, 300]
})

print(df.describe())
```

Output:

```
            Sales
count    5.000000
mean   200.000000
std     79.056942
min    100.000000
25%    150.000000
50%    200.000000
75%    250.000000
max    300.000000
```

Other useful functions:

- df.mean(), df.median(), df.std(), df.min(), df.max(), df.mode()

Interview Tip:
Mention that you also use .value_counts() for categorical features and .corr() to examine relationships between variables.

---

63. How do you sort a DataFrame in pandas?
Pandas provides the .sort_values() and .sort_index() methods to sort DataFrames:

- **.sort_values(by='column_name')**: Sorts rows based on values in one or more columns
- **.sort_index()**: Sorts by the DataFrame index

Example:

```
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 22]
})


# Sort by age in descending order
df_sorted = df.sort_values(by='Age', ascending=False)
print(df_sorted)
```

Output:

```
      Name  Age
1      Bob   30
0    Alice   25
2  Charlie   22
```

Interview Insight:
Sorting is often used during reporting, ranking, or preparing data for visual dashboards. Mention how you combine sorting with filtering and aggregation in real-world scenarios.

---

64. What is a hashable object and why does it matter?
An object is **hashable** if it has a **hash value that never changes during its lifetime** and supports equality comparison.

Key points:

- **Immutable types** like str, int, float, and tuple are hashable.
- **Mutable types** like list, dict, and set are not hashable.
- Hashable objects can be used as **keys in dictionaries** or **members in sets**.

Example:

```
hash("hello")      # Valid
hash((1, 2, 3))    # Valid
hash([1, 2, 3])    # TypeError
```

Why it matters in interviews:
Understanding hashable types is essential for working with sets, dictionary keys, and understanding performance implications in lookup-based operations.

---

65. What is the difference between re.search() and re.match()?
Both functions are part of the re module in Python used for pattern matching via regular expressions, but they differ in scope:

- **re.match()** checks for a match **only at the beginning** of the string.
- **re.search()** scans through the **entire string** for a match.

Example:

```
import re

text = "Python is great"

print(re.match("Python", text))  # Match
print(re.search("great", text))  # Match
print(re.match("great", text))   # None
```

Usage tip:
In data cleaning or extraction, use re.search() to find patterns anywhere in the string and re.match() only when the pattern must start at the beginning.

66. How do you normalize data in Python?
Data normalization is a preprocessing technique used to rescale numeric values into a common range. It is especially important in machine learning when features have different scales.

Common normalization methods:

- **Min-Max Scaling**: Scales values to a fixed range [0, 1].
- **Standardization (Z-score)**: Centers the values around the mean with a standard deviation of 1.
- **Robust Scaling**: Uses median and interquartile range, good for data with outliers.

Using scikit-learn:

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler
import pandas as pd

data = pd.DataFrame({'Score': [20, 40, 60, 80, 100]})

# Min-Max
minmax = MinMaxScaler()
print(minmax.fit_transform(data))

# Standardization
standard = StandardScaler()
print(standard.fit_transform(data))
```

When to use what:

- Use **MinMaxScaler** when the data is bounded.
- Use **StandardScaler** when your model assumes data is normally distributed (e.g., logistic regression).
- Use **RobustScaler** if your data contains outliers.

---

67. What is method overloading and method overriding in Python?
These are key concepts in object-oriented programming:

Method Overloading (Not natively supported in Python):

- Same method name with different parameters.
- Simulated using default values or *args.

Example:

```
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c
```

Method Overriding:

- Subclass redefines a method from its superclass.

Example:

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


class Dog(Animal):
    def speak(self):
        print("Dog barks")


d = Dog()
d.speak()  # Output: Dog barks
```

Interview Tip:
Clarify that overloading can be mimicked using flexible parameters, but overriding is fully supported in Python and used for polymorphism.

---

68. How do you handle errors in Python?
Python uses **exception handling** to manage runtime errors, ensuring the program doesn't crash unexpectedly.

Syntax:

```
try:
    # risky code
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("Always runs")
```

Other common exceptions:

- ValueError

- TypeError
- FileNotFoundError
- KeyError

Best practices:

- Always catch specific exceptions.
- Use finally to release resources.
- Avoid catching all exceptions unless necessary.

---

69. What is a coroutine in Python?
A **coroutine** is a special function declared using async def and used for **asynchronous programming**. It allows execution to be paused and resumed using await.

Example:

```
import asyncio


async def greet():
    print("Hello")
    await asyncio.sleep(1)
    print("World")


asyncio.run(greet())
```

When to use:
Use coroutines when handling I/O-bound tasks like web scraping, APIs, or file operations to improve performance without multi-threading.

---

70. What is the use of __slots__ in Python classes?
__slots__ is a special attribute you define in a class to **limit the creation of instance attributes** to a fixed set. This saves memory and prevents the creation of a dynamic __dict__.

Example:

```
class Employee:
    __slots__ = ['name', 'age']
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Benefits:

- Reduces memory usage.
- Makes attribute access faster.
- Prevents accidental creation of new attributes.

Limitation:
You can't add attributes not listed in __slots__. Also, it doesn't work well with inheritance unless carefully managed.

---

71. How does Python handle memory management and garbage collection?
Python uses **automatic memory management** and **garbage collection** to efficiently allocate and reclaim memory during program execution. This process ensures that memory no longer in use is freed up, preventing memory leaks and optimizing performance.

Key components of memory management:

- **Reference Counting:**
  Every object in Python has an internal counter that tracks how many references point to it. When this count reaches zero, the memory is immediately reclaimed.
  Example:
- a = [1, 2, 3] # reference count = 1
- b = a # reference count = 2
- del a # reference count = 1
- del b # reference count = 0 → memory freed
- **Garbage Collector (GC):**
  Python's gc module handles **cyclic references**, which reference counting alone cannot resolve.
  Cyclic garbage collection occurs periodically and identifies unreachable objects with reference cycles.
- import gc
- gc.collect() # Manually triggers garbage collection
- **Memory Pools (pymalloc):**
  CPython uses an internal allocator for small memory blocks to speed up performance.

Best Practices for Interview:

- Use del to explicitly remove large objects when no longer needed.
- Avoid circular references when possible.
- Use weak references (weakref) for caching or observer patterns where you don't want to increase the reference count.

📌 **Summary:** Python uses a combination of reference counting and cyclic garbage collection, backed by memory pooling, to handle memory efficiently.

---

72. What is the Global Interpreter Lock (GIL), and how does it affect multithreading in Python?
The **Global Interpreter Lock (GIL)** is a mutex (mutual exclusion lock) in **CPython**, the standard implementation of Python. It ensures that only **one thread executes Python bytecode at a time**, even on multi-core processors.

Why GIL exists:

- Simplifies memory management by preventing race conditions in object access.
- Ensures thread safety for core Python internals.

Implications:

- For **I/O-bound tasks** (file operations, API requests), multithreading works fine because threads release the GIL during I/O.
- For **CPU-bound tasks**, GIL becomes a bottleneck as threads cannot run truly in parallel.

Alternatives to overcome GIL:

- Use the **multiprocessing** module instead of threading for CPU-bound tasks.
- from multiprocessing import Pool
- Use **libraries like NumPy** or C-extensions that internally release the GIL during computation.
- Use **asyncio** for I/O-bound concurrency.

📌 **Interview Tip:** Explain that GIL limits multi-threaded parallelism for CPU-heavy tasks, and Python offers workarounds like multiprocessing for true parallelism.

---

73. Explain the difference between a Python module and a package.
Understanding Python's modular structure is essential for organizing large codebases and reusable components.

- **Module:**
  A **module** is simply a Python .py file containing functions, classes, or variables that can be imported and used in other programs.
  Example:
- # file: math_utils.py
- def add(a, b):
- return a + b

- **Package:**
  A **package** is a directory containing multiple modules and an __init__.py file (can be empty or contain initialization code). It allows for a hierarchical structure.
- analytics/
- ├── __init__.py
- ├── preprocess.py
- └── models.py
- **Importing:**
- from analytics.models import train_model

📌 **Key Difference:** A module is a single file, while a package is a folder of modules with optional sub-packages.

---

74. How can you handle circular imports in Python?
Circular imports occur when two or more modules depend on each other. This can lead to incomplete or failed imports.

Why it's a problem:
When Python starts importing module A, it tries to import module B, but B again tries to import A, which is not fully initialized yet. This leads to an ImportError or unexpected behavior.

Solutions:

1. **Reorganize the code**: Move common functionality to a separate third module that both modules import.
2. **Use local imports**: Import the module inside a function or method to delay it.
3. def func():
4. from module_b import some_function
5. some_function()
6. **Use importlib for dynamic import**:
7. import importlib
8. module = importlib.import_module('module_b')

📌 **Interview Tip:** Demonstrate your awareness of software design and how circular imports often indicate coupling that needs refactoring.

---

75. What are metaclasses in Python?
A **metaclass** in Python is a class **of a class**. Just like an object is an instance of a class, a class is an instance of a metaclass.

By default, all classes in Python are created using the built-in type metaclass.

Use cases:

- Automatically add or modify class attributes/methods
- Enforce coding standards
- Perform validations at class creation time

Basic metaclass example:

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, dct)


class MyClass(metaclass=Meta):
    pass
```

When MyClass is defined, the Meta.__new__ method runs first.

Best Practice:
Use metaclasses **only when absolutely necessary**, as they add complexity. More common alternatives include class decorators and factory functions.

📌 **Interview Edge:** Knowing metaclasses shows advanced Python expertise, but also mention when **not** to use them.

---

76. What is monkey patching in Python?
Monkey patching refers to dynamically changing or extending modules, classes, or methods **at runtime** without altering the original source code. It's a powerful technique but should be used cautiously.

Use cases:

- Fixing bugs in third-party libraries
- Adding methods to classes for testing
- Mocking behavior during unit testing

Example:

```
import math

math.sqrt = lambda x: "patched!"

print(math.sqrt(4))  # Output: patched!
```

Here, the built-in math.sqrt() is replaced at runtime with a custom lambda function.

Risks:

- Can lead to **unpredictable behavior**
- Makes **debugging difficult**
- Can **break code** silently if the patch fails

Safer alternatives:

- Use **inheritance or composition**
- Use **mocking frameworks** like unittest.mock for temporary patches

📌 **Interview Tip:** While monkey patching shows flexibility, always mention that it should be avoided in production unless absolutely necessary.

---

77. Explain Python's memory model.
Python's memory model is built around the idea that **everything is an object** and memory is managed through a combination of **reference counting**, **garbage collection**, and **memory pools**.

Key components:

- **Object Storage in Heap:**
  All objects and data structures are stored in the heap. Variables are just **references** pointing to these objects.
- **Reference Counting:**
  Each object has a count of how many references point to it. When the count drops to zero, the object is deallocated.
- **Garbage Collection:**
  For cases where reference cycles occur (A → B → A), Python uses a **cyclic garbage collector** in the gc module.
- **Memory Pools (pymalloc):**
  CPython uses pools for allocating memory blocks efficiently and reducing fragmentation.

Implications for developers:

- Use del to release memory manually if necessary.
- Be aware of reference cycles in complex data structures.
- Avoid holding on to large objects unnecessarily.

📌 **Summary:** Python's memory model is abstracted from the developer, but understanding it helps write memory-efficient and high-performance code.

---

78. What are weak references in Python?
A **weak reference** allows you to reference an object **without increasing its reference count**. This means the object can still be garbage collected when there are no strong references pointing to it.

Use cases:

- **Caching**: Avoid memory leaks by allowing unused objects to be collected
- **Observer patterns**: Avoid keeping unnecessary references

Example:

```
import weakref


class MyClass:
    pass


obj = MyClass()
weak_obj = weakref.ref(obj)
print(weak_obj())  # <__main__.MyClass object at ...>


del obj
print(weak_obj())  # None (object has been garbage collected)
```

When to use:

- Large graph-like structures (e.g., trees)
- Applications with caching requirements
- Avoiding reference cycles in design patterns

📌 **Interview Edge:** Understanding weak references shows you can manage memory in large or long-running systems efficiently.

---

79. How do you optimize recursive functions in Python?
Recursive functions are elegant but can be **inefficient** due to **redundant computations** or **stack overflows**. Optimizing them ensures better performance and avoids exceeding the recursion depth limit.

Optimization Techniques:

1. **Memoization:**
   Store intermediate results to avoid redundant computation.
   - Use functools.lru_cache:
2. from functools import lru_cache
3.
4. @lru_cache(maxsize=None)
5. def fib(n):

```
6.      if n < 2:
7.          return n
8.      return fib(n-1) + fib(n-2)
```

9. **Convert to Iterative Approach:**
   Tail recursion is not optimized in Python, so rewriting as a loop avoids deep stacks.
10. **Manual Memoization:**
    Use a dictionary to cache results.
11. **Adjust Recursion Limit:**
    You can increase the recursion limit, but it's risky:

```
12. import sys
13. sys.setrecursionlimit(2000)
```

📌 **Best Practice:** Use memoization when recursion is necessary and prefer iteration if performance is critical.

---

80. What are the different memory optimization techniques in Python?
Memory optimization is essential when working with **large datasets**, **high-frequency tasks**, or **resource-constrained environments**.

Common techniques:

- **Use Generators:**
  Replace lists with generators for lazy evaluation.
- gen = (x for x in range(10**6))
- **Avoid Unnecessary Objects:**
  Avoid building large intermediate lists inside loops.
- **Use __slots__:**
  Prevent creation of instance __dict__, saving memory in class definitions.
- class User:
- __slots__ = ['name', 'age']
- **Use Tuples instead of Lists:**
  Tuples are immutable and require less memory.
- **Release Memory Manually:**
  Use del to remove references, especially for large objects.
- **Use Efficient Libraries:**
  NumPy arrays are more memory-efficient than native Python lists for numeric data.

- **Monitor Memory:**
  Use gc, tracemalloc, and memory_profiler to detect memory leaks.

📌 **Interview Tip:** Memory management is not just about saving space—it directly affects performance and scalability.

DATA STRUCTURES AND ALGORITHMS IN PYTHON

81. How do you implement a stack in Python using lists?
A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. In Python, stacks can be easily implemented using lists, since Python lists support operations like append() and pop() which correspond to push and pop operations respectively.

Example:

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None

    def is_empty(self):
        return len(self.items) == 0
```

This implementation allows you to:

- push() to add an item
- pop() to remove and return the top item
- peek() to view the top item without removing it
- is_empty() to check if the stack is empty

This structure is commonly asked about in interviews, especially when evaluating your understanding of basic data structures.

---

82. How do you implement a queue using two stacks in Python?
A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It can be implemented using two stacks in such a way that enqueuing is done in one stack and dequeuing is managed via another. This method is useful for interview problems testing your understanding of how data structures can simulate others.

Example:

```
class QueueUsingStacks:
    def __init__(self):
        self.in_stack = []
        self.out_stack = []

    def enqueue(self, item):
        self.in_stack.append(item)

    def dequeue(self):
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())
        if self.out_stack:
            return self.out_stack.pop()
        return None
```

This implementation ensures that each element is transferred at most twice, resulting in amortized O(1) time for each operation. It's an elegant approach to build a queue when only stack operations are available.

---

83. How do you reverse a list without using slicing or built-in functions?
Reversing a list manually is a good way to test your understanding of loops and index manipulation. You can iterate from the end of the list to the start and construct a new list:

Example:

```
def reverse_list(lst):
    reversed_lst = []
    for i in range(len(lst) - 1, -1, -1):
        reversed_lst.append(lst[i])
    return reversed_lst
```

This method avoids Python's slicing ([::-1]) or the reversed() function and is often used in interviews to see if candidates understand control flow and memory allocation.

---

84. What is the difference between str() and repr() in Python?
Both str() and repr() are used to get string representations of objects, but their purposes are different:

- str() is intended for readability. It is meant to return a user-friendly or informal string representation of the object.
- repr() is intended for debugging and development. It returns an official string representation that ideally can be used to recreate the object.

Example:

```
x = "Hello"
print(str(x))   # Output: Hello
print(repr(x))  # Output: 'Hello'
```

In custom classes, you can override the __str__() and __repr__() methods to control how your objects are represented in both contexts.

---

85. What is the purpose of locals() and globals() functions in Python?

- locals() returns a dictionary of the current local symbol table. It's used to examine or modify local variables inside functions or blocks.
- globals() returns a dictionary of the current global symbol table. It's used to access or modify global variables.

Example:

```
x = 10
def test():
    y = 20
```

```
    print("Locals:", locals())
    print("Globals:", globals().keys())

test()
```

These functions are very useful in debugging, dynamic code execution (like with eval()), and in writing meta-programming logic.

---

86. Write a function to check if a string is a palindrome.

A palindrome is a string that reads the same forwards and backwards. This is a common coding interview question used to test your logic-building skills.

Here's a simple implementation using string comparison:

```
def is_palindrome(s):
    return s == s[::-1]
```

Alternative method using a loop:

```
def is_palindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True
```

This helps evaluate string handling and basic loop control knowledge.

---

87. How do you use zip() and unzip() in Python?

- zip() combines two or more iterables (like lists or tuples) into tuples.
- To unzip, we use the unpacking operator *.

Example:

```
# Zipping
a = [1, 2, 3]
b = ['x', 'y', 'z']
zipped = list(zip(a, b))  # [(1, 'x'), (2, 'y'), (3, 'z')]


# Unzipping
```

```
a_unzip, b_unzip = zip(*zipped)
print(list(a_unzip))  # [1, 2, 3]
print(list(b_unzip))  # ['x', 'y', 'z']
```

zip() is widely used in iteration, dictionary construction, and parallel processing of lists.

---

88. What is the difference between break and continue in Python loops?

- break is used to exit a loop prematurely when a condition is met.
- continue skips the current iteration and moves to the next one.

Example:

```
for i in range(5):
    if i == 3:
        break
    print(i)  # Output: 0, 1, 2


for i in range(5):
    if i == 3:
        continue
    print(i)  # Output: 0, 1, 2, 4
```

Understanding these control flow keywords is crucial in handling loop-based logic efficiently.

---

89. How do you implement a simple caching mechanism in Python?

Caching improves performance by storing the result of expensive function calls. You can use dictionaries or decorators like lru_cache.

Manual caching:

```
cache = {}
def add(a, b):
    if (a, b) in cache:
        return cache[(a, b)]
    result = a + b
    cache[(a, b)] = result
    return result
```

Using lru_cache:

```
from functools import lru_cache
```

```
@lru_cache(maxsize=None)
def add(a, b):
    return a + b
```

lru_cache automatically manages the cache and is very helpful in recursive functions or frequently repeated operations.

---

90. How do you flatten a nested list using recursion?
Flattening is often necessary when working with data that contains nested lists.

Example:

```
def flatten(lst):
    flat = []
    for item in lst:
        if isinstance(item, list):
            flat.extend(flatten(item))
        else:
            flat.append(item)
    return flat


print(flatten([1, [2, [3]], [4]]))  # Output: [1, 2, 3, 4]
```

This problem is common in interviews for checking recursion skills and handling dynamic data structures.

FUNCTIONS, DECORATORS, AND DESIGN PATTERNS

91. How do you implement decorators in Python?
Decorators are functions that modify the behavior of another function without changing its source code. They're often used for logging, authentication, timing functions, and input validation. Decorators allow you to add functionality to existing code in a clean, readable, and maintainable way.

Here's a basic decorator example:

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

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

greet()
```

Output:

Before function call

Hello!

After function call

In the above, @decorator is shorthand for greet = decorator(greet). The decorator function takes another function as input and returns a new function that adds behavior before and after the original function call.

📌 Decorators are powerful tools in Python for adding reusable logic around functions.

---

92. What are Python's built-in functions for functional programming?

Python supports functional programming concepts and provides several built-in functions to support this paradigm. The main ones are:

- **map()** – Applies a function to every item of an iterable.
- **filter()** – Filters items out of an iterable using a Boolean function.
- **reduce()** – Applies a rolling computation to sequential pairs of values (imported from functools).

Example:

```
from functools import reduce

nums = [1, 2, 3]

print(list(map(lambda x: x * 2, nums)))        # [2, 4, 6]
```

```
print(list(filter(lambda x: x > 1, nums)))      # [2, 3]
print(reduce(lambda x, y: x + y, nums))         # 6
```

These tools are efficient when working with sequences of data and allow concise expressions of common operations like transformation and reduction.

---

93. How do you create a singleton design pattern in Python?

The Singleton pattern ensures that only one instance of a class exists and provides a global point of access to it. It is useful in cases where one object is needed to coordinate actions across a system (e.g., configuration manager, logger).

Python implementation:

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

a = Singleton()
b = Singleton()
print(a is b)  # Output: True
```

Here, __new__ is overridden to ensure that only one instance is created.

📌 Singleton patterns are commonly used in systems where one global shared resource is required.

---

94. How do you sort a dictionary by its values in Python?

To sort a dictionary by its values, you can use the sorted() function with a lambda function as the key. This is useful when prioritizing elements based on associated values.

Example:

```
my_dict = {'a': 3, 'b': 1, 'c': 2}
sorted_dict = dict(sorted(my_dict.items(), key=lambda item: item[1]))
print(sorted_dict)  # {'b': 1, 'c': 2, 'a': 3}
```

Here, sorted() sorts the items (key-value pairs), and dict() reconstructs them back into a dictionary.

📌 This method is often used in data processing tasks like ranking or summarization.

---

95. How do you check if two lists are identical in Python?
There are two common ways to compare lists in Python:

- **Using ==**: Checks if the contents of both lists are the same and in the same order.
- **Using is**: Checks if both lists are the same object in memory.

Example:

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1


print(list1 == list2)  # True
print(list1 is list2)  # False
print(list1 is list3)  # True
```

Use == for value comparison and is for identity comparison.

📌 In interviews, this helps test your understanding of Python's memory model and reference handling.

---

96. How do you read and write files in Python?
Python provides built-in support for file operations using the open() function. You should always use a with block to ensure that the file is properly closed after operations.

Writing to a file:

```
with open("sample.txt", "w") as f:
    f.write("Hello World")
```

Reading from a file:

```
with open("sample.txt", "r") as f:
    content = f.read()
    print(content)
```

Modes available:

- 'r' – read
- 'w' – write (overwrite)
- 'a' – append
- 'b' – binary mode
- 'x' – exclusive creation

📌 Always use the with statement for safer and cleaner file handling.

97. How do you find the most frequent element in a list using collections.Counter?
Python's collections.Counter is an efficient way to count occurrences of items in an iterable and retrieve the most common ones.

Example:

```
from collections import Counter

lst = [1, 2, 2, 3, 3, 3]
counter = Counter(lst)
most_common = counter.most_common(1)[0][0]
print(most_common)  # Output: 3
```

Here, most_common(1) returns a list with the most frequent item and its count as a tuple.

📌 This is highly useful in NLP, frequency analysis, and classification problems.

---

98. What are Python's magic methods (dunder methods)?
Magic methods (also known as "dunder" methods because they start and end with double underscores) allow you to define how Python objects behave with built-in operators and functions.

Examples:

- __init__ – constructor
- __str__ – string representation
- __repr__ – official representation
- __len__, __getitem__, __eq__ – used with len(), indexing, and comparison

Example:

```
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return f"Book title: {self.title}"

book = Book("Python Basics")
print(book)  # Output: Book title: Python Basics
```

📌 Magic methods help you integrate your custom classes seamlessly with Python's syntax and operations.

RUNTIME MODIFICATION, MEMORY MANAGEMENT, AND CODE OPTIMIZATION

99. What is monkey patching in Python?
Monkey patching is the dynamic modification or extension of modules or classes at runtime. It is often used to change behavior of existing methods or classes without altering the original source code.

Example:

```
import math


# Original sqrt
print(math.sqrt(4))  # Output: 2.0


# Monkey patching
math.sqrt = lambda x: "Patched!"
print(math.sqrt(4))  # Output: Patched!
```

This can be helpful during:

- Testing (to mock certain behaviors)
- Adding quick fixes to third-party libraries
- Dynamic behavior insertion for prototyping

However, monkey patching can make code difficult to debug and maintain. It should be used cautiously as it can introduce hard-to-track bugs.

📌 Use monkey patching only when absolutely necessary and document it clearly.

---

100. How do you check if an element exists in a list without using the in keyword?
While in is the most Pythonic way to check for existence, an alternative method is using the any() function with a generator expression.

Example:

```
def exists(lst, element):
    return any(x == element for x in lst)
```

print(exists([1, 2, 3], 2)) # Output: True

This approach can also be used to apply custom logic during the check (like case-insensitive comparison, partial match, etc.).

📌 This technique is often tested in interviews to assess familiarity with functional constructs and logical reasoning.

---

101. How do you swap values of two variables in Python without using a temporary variable?
Python allows variable swapping using tuple unpacking, which is concise and efficient.

Example:

a, b = 5, 10

a, b = b, a

print(a, b) # Output: 10 5

This works by packing the values into a temporary tuple and then unpacking them in reversed order.

📌 Python's ability to swap values this way is often highlighted for its elegance and is a common interview question.

---

102. What is the purpose of if __name__ == "__main__": in Python?
This special condition is used to execute code only when the Python script is run directly, not when it is imported as a module into another script.

Example:

```
def main():
    print("Running directly")


if __name__ == "__main__":
    main()
```

Why it matters:

- Prevents certain code from being executed during import
- Useful in scripts that include tests or demo runs
- Encourages modular, reusable code design

📌 It is a standard Python idiom to organize executable scripts properly.

---

103. What is the purpose of the super() function in Python?
The super() function is used in object-oriented programming to call methods from a parent class. It is commonly used in inheritance to avoid code duplication and to maintain a clean hierarchy.

Example:

```
class Parent:
    def greet(self):
        print("Hello from Parent")


class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from Child")


Child().greet()
```

Output:

Hello from Parent

Hello from Child

Benefits:

- Supports cooperative multiple inheritance
- Makes code more maintainable and scalable
- Ensures that the parent class initialization is executed

📌 Always prefer super() over calling the parent class explicitly.

---

104. What is the difference between np.array() and np.asarray() in NumPy?
Both functions convert input data into a NumPy array, but there's a key distinction:

- **np.array()**: Always creates a new copy.
- **np.asarray()**: Converts the input to an array but avoids a copy if the input is already a NumPy array.

Example:

```
import numpy as np


a = np.array([1, 2, 3])
b = np.asarray(a)
```

print(a is b) # Output: True

When to use which:

- Use np.array() when you want to ensure independence from the original object.
- Use np.asarray() for performance optimization when copying is unnecessary.

📌 Choosing the right one helps in avoiding redundant memory usage.

LARGE DATASETS, AGGREGATION, AND PIVOT OPERATIONS IN PANDAS

105. How do you read large datasets efficiently in Python?
Handling large datasets in memory can lead to performance issues. Here are several techniques to read large CSV or text files efficiently using Python:

1. **Read in Chunks using chunksize:**
2. import pandas as pd
3.
4. for chunk in pd.read_csv('large_file.csv', chunksize=10000):
5. process(chunk)
6. **Load Only Required Columns using usecols:**
7. df = pd.read_csv('large_file.csv', usecols=['col1', 'col2'])
8. **Specify Data Types to Save Memory with dtype:**
9. df = pd.read_csv('large_file.csv', dtype={'id': 'int32', 'name': 'category'})
10. **Use nrows to Limit the Number of Rows Initially:**

11. df_sample = pd.read_csv('large_file.csv', nrows=1000)

12. **Use Libraries for Parallel Processing:**

    - **Dask**: Handles large datasets with parallelism.
    - **PySpark**: Ideal for distributed computing.

📌 Always profile your memory usage and tweak these parameters accordingly for optimal performance.

---

106. How do you create pivot tables in pandas?
Pivot tables allow you to reorganize and summarize data in tabular form, similar to Excel.

Use the pivot_table() function in pandas:

```
import pandas as pd


df = pd.DataFrame({
   'Region': ['North', 'South', 'North', 'South'],
   'Product': ['A', 'A', 'B', 'B'],
   'Sales': [100, 150, 200, 250]
})


pivot = df.pivot_table(index='Region', columns='Product', values='Sales', aggfunc='sum')
```

Features:

- index: Row labels
- columns: Column labels
- values: Data to aggregate
- aggfunc: Aggregation function like sum, mean, count

📌 Pivot tables are essential for data summarization and exploratory data analysis.

---

107. What is the use of the transform() function in pandas?
The transform() function is used to apply a function to a group while retaining the original index structure of the DataFrame. This is especially useful for group-wise calculations that need to be broadcasted back to the full DataFrame.

Example – Z-score normalization within groups:

```
df['normalized'] = df.groupby('Category')['Value'].transform(
   lambda x: (x - x.mean()) / x.std()
```

)

Key Benefits:

- Retains original DataFrame shape
- Useful in creating new derived features
- Works great for feature engineering in machine learning pipelines

📌 Think of transform() as the go-to function when you need per-group calculations with full DataFrame shape preservation.

---

108. What is the difference between merge(), join(), and concat() in pandas?

Method	Purpose	Key Features
merge()	SQL-style joins using keys	Highly flexible, supports left/right/outer/inner joins
join()	Joins based on the index (or a key column)	Simpler syntax for index-based joins
concat()	Concatenates DataFrames along a particular axis	Used to stack vertically or horizontally

Example of merge (on key column):

```
pd.merge(df1, df2, on='id', how='inner')
```

Example of join (on index):

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

Example of concat:

```
pd.concat([df1, df2], axis=0)  # Vertical stacking
pd.concat([df1, df2], axis=1)  # Horizontal stacking
```

📌 Use merge() when dealing with key-based relational data. Use concat() when appending or stacking similar datasets.

---

109. What is the purpose of namedtuple in Python?
namedtuple is a factory function for creating tuple subclasses with named fields. It is a memory-efficient, immutable alternative to defining custom classes.

Example:

```
from collections import namedtuple


Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
```

```
print(p.x)  # Output: 10
print(p.y)  # Output: 20
```

Benefits:

- Field names make code more readable.
- Uses less memory than classes.
- Instances are immutable and hashable (can be used as dictionary keys).

📌 Use namedtuple for lightweight, readable, and efficient record-like structures.

---

110. What is method chaining in pandas and why is it useful?
Method chaining is a practice where multiple pandas methods are called sequentially, using dots (.), often enclosed in parentheses. It enhances readability and eliminates the need for intermediate variables.

Example:

```
df_cleaned = (
   df.dropna()
     .assign(sales_tax=lambda x: x['Sales'] * 0.18)
     .query('sales_tax > 50')
     .sort_values(by='sales_tax', ascending=False)
)
```

Advantages:

- Improves code readability
- Encourages functional programming style
- Reduces clutter by avoiding temporary variables

📌 It's particularly useful in data cleaning and wrangling pipelines where many transformations are applied sequentially.

BUILT-IN FUNCTIONS AND ITERATION UTILITIES IN PYTHON

111. What is the difference between any() and all() in Python?
These are built-in Python functions used for evaluating Boolean conditions over iterable objects like lists or tuples.

- **any(iterable)**: Returns True if at least one element in the iterable is truthy.
- **all(iterable)**: Returns True only if *all* elements in the iterable are truthy.

Examples:

any([False, True, False]) # Output: True

all([True, True, True]) # Output: True

all([True, False, True]) # Output: False

Use Cases in Data Analysis:

- Checking for completeness of form inputs
- Verifying that all records meet a condition
- Filtering operations in custom functions

📌 These functions are concise and efficient alternatives to writing loops for condition checks.

---

112. What is the use of the enumerate() function in Python?
enumerate() is used to iterate through a collection while keeping track of the index.

Syntax:

enumerate(iterable, start=0)

Example:

```
for index, value in enumerate(['a', 'b', 'c']):
    print(index, value)
```

Output: 0 a, 1 b, 2 c

Use Cases:

- Replacing range(len(list)) style loops
- Tracking indices while iterating over sequences
- Labeling data for display or debugging

📌 enumerate() increases clarity and avoids the need to manage counters manually during iteration.

---

113. How do you create a virtual environment in Python?
A virtual environment is a self-contained directory where you can install project-specific dependencies without affecting global installations.

Steps to Create:

1. **Install virtualenv (if not already):**
2. pip install virtualenv
3. **Create a virtual environment:**
4. virtualenv env_name
5. **Activate the environment:**
    - On Windows:
    - env_name\Scripts\activate
    - On macOS/Linux:
    - source env_name/bin/activate
6. **Install packages locally:**
7. pip install pandas numpy
8. **Deactivate the environment:**
9. deactivate

📌 Virtual environments are essential for avoiding dependency conflicts and ensuring reproducibility.

---

114. How do you detect and handle outliers in Python?
Outliers are data points that differ significantly from other observations and can skew statistical analyses.

Techniques to Detect Outliers:

1. **Z-Score Method:**
2. from scipy import stats
3. z_scores = stats.zscore(df['column'])
4. df[z_scores < 3]
5. **IQR Method:**
6. Q1 = df['column'].quantile(0.25)
7. Q3 = df['column'].quantile(0.75)

8. IQR = Q3 - Q1
9. df = df[~((df['column'] < (Q1 - 1.5 * IQR)) | (df['column'] > (Q3 + 1.5 * IQR)))]
10. **Boxplots and Visual Inspection:**
11. import seaborn as sns
12. sns.boxplot(data=df, x='column')

Handling Methods:

- Remove them
- Cap or transform them
- Use robust algorithms that are insensitive to outliers (e.g., tree-based models)

📌 Outlier detection should align with the business logic and not be purely statistical.

---

115. What are weak references in Python and where are they used?
A **weak reference** allows one object to refer to another object without increasing its reference count. When the original object is no longer needed elsewhere, it can be garbage collected—even if a weak reference to it still exists.

Use Case:
Helpful in memory-sensitive applications like caching, where you don't want your cache to prevent the original object from being collected.

Example:

```
import weakref


class MyClass:
    pass


obj = MyClass()
weak_obj = weakref.ref(obj)


print(weak_obj())  # <__main__.MyClass object at ...>
del obj
print(weak_obj())  # None (object was garbage collected)
```

Applications:

- Observer pattern
- Caches in large-scale systems

- Managing object lifecycles in GUI or simulation applications

📌 Use weak references when you want to keep a lightweight reference to an object without preventing it from being freed.

MACHINE LEARNING, DATA SAMPLING, AND PERFORMANCE OPTIMIZATION

116. How do you handle imbalanced datasets in Python?
Imbalanced datasets occur when one class significantly outnumbers the others, leading to biased model performance.

Techniques to Handle Imbalanced Datasets:

1. **Resampling Methods:**
    - **Oversampling the minority class:**
    - from imblearn.over_sampling import SMOTE
    - smote = SMOTE()
    - X_res, y_res = smote.fit_resample(X, y)
    - **Undersampling the majority class:**
    - from imblearn.under_sampling import RandomUnderSampler
    - rus = RandomUnderSampler()
    - X_res, y_res = rus.fit_resample(X, y)
2. **Adjusting Class Weights:** Most ML algorithms in sklearn support class weights:
3. from sklearn.ensemble import RandomForestClassifier
4. clf = RandomForestClassifier(class_weight='balanced')
5. **Evaluation Metrics:** Use metrics that account for class imbalance:
    - F1 Score
    - Precision-Recall Curve
    - ROC-AUC Score
6. **Libraries:**
    - imbalanced-learn (compatible with scikit-learn)

- sklearn.utils.class_weight

📌 Always validate with **stratified sampling** to ensure fair evaluation across classes.

---

117. What are hashable objects in Python and why do they matter?
A hashable object has a hash value that remains constant throughout its lifetime and supports comparison via __eq__.

Why It Matters:

- Only hashable objects can be used as **dictionary keys** or added to **sets**
- Immutability is a key trait of hashable objects

Examples:

```
hash("hello")        # Valid
hash((1, 2, 3))      # Valid
hash([1, 2, 3])      # Error: list is unhashable
```

Hashable Types:

- int, str, float, bool
- tuple (if all its elements are hashable)

Unhashable Types:

- list, set, dict (mutable types)

📌 Understanding hashability helps avoid common runtime errors and design efficient data structures.

---

118. What is the difference between a generator expression and a list comprehension?
Both are used to create sequences from iterables, but differ in how they evaluate and store data.

Feature	Generator Expression	List Comprehension
Syntax	(x for x in iterable)	[x for x in iterable]
Evaluation	Lazy (on-the-fly)	Eager (all at once)
Memory Usage	Efficient	High for large data
Output Type	Generator object	List

Example:

```
gen = (x**2 for x in range(1000000))  # Generator
lst = [x**2 for x in range(1000000)]  # List
```

When to Use:

- Use **generators** when dealing with large data and memory optimization is a priority
- Use **list comprehensions** when you need a materialized sequence

📌 Prefer generators in loops, pipelines, or when the full result is not immediately needed.

---

119. How do you evaluate a machine learning model in Python?
Evaluation ensures the model performs well on unseen data and helps fine-tune it for production.

Classification Metrics:

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score

- **Accuracy**: Overall correctness
- **Precision**: True Positives / (True Positives + False Positives)
- **Recall**: True Positives / (True Positives + False Negatives)
- **F1 Score**: Harmonic mean of precision and recall
- **ROC AUC**: Area under ROC curve (for binary classification)

Regression Metrics:

from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

- **MAE**: Average magnitude of errors
- **MSE**: Squared error penalty
- **R^2 Score**: Explained variance

Cross-Validation:

from sklearn.model_selection import cross_val_score

cross_val_score(model, X, y, cv=5)

📌 Always evaluate with multiple metrics, and use **cross-validation** to reduce overfitting risk.

---

120. How can you profile performance bottlenecks in a Python script?
Performance profiling helps identify parts of code that consume the most time or memory.

Tools and Techniques:

1. **cProfile** – Built-in function-level profiler:
2. python -m cProfile my_script.py
3. **line_profiler** – Line-by-line profiling (install with pip install line_profiler):
4. @profile
5. def compute():

6. ...
7. **memory_profiler** – Tracks memory usage:
8. pip install memory_profiler
9. **Visualization Tools:**
    - SnakeViz: Interactive visualization for profiling results
    - Py-Spy: Lightweight sampling profiler
10. **Timeit Module** – Benchmarking small code snippets:
11. import timeit
12. timeit.timeit('sum(range(100))', number=1000)

📌 Profiling is critical for optimizing algorithms in data pipelines and reducing runtime on large datasets.

OBJECT-ORIENTED PROGRAMMING AND DESIGN PATTERNS

121. What is method overloading and method overriding in Python?
These are two key concepts in object-oriented programming that allow function customization and polymorphism.

Method Overloading

- Python does **not** support traditional method overloading.
- Instead, you can use **default arguments**, *args, or **kwargs to simulate it.

Example of Simulated Overloading:

```
class Math:
    def add(self, a, b=0):
        return a + b


m = Math()
print(m.add(5))       # 5 + 0 = 5
print(m.add(5, 10))   # 5 + 10 = 15
```

Method Overriding

- Occurs when a subclass provides its own version of a method from the parent class.
- Allows for custom behavior in derived classes.

Example of Overriding:

```
class Parent:
    def greet(self):
        print("Hello from Parent")


class Child(Parent):
    def greet(self):
        print("Hello from Child")


c = Child()
c.greet()  # Output: Hello from Child
```

📌 Overriding is widely used in polymorphism and abstract class implementations, whereas overloading is mimicked via flexible function definitions.

---

122. What is the use of the super() function in Python?
super() is a built-in function used to call methods from a parent or sibling class in the hierarchy.

Use Cases:

- Avoids explicitly referring to the base class
- Helps maintain the **method resolution order (MRO)** in multiple inheritance
- Promotes DRY (Don't Repeat Yourself) by reusing parent logic

Example:

```
class Parent:
    def greet(self):
        print("Hello from Parent")


class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from Child")


Child().greet()
```

Output:

```
Hello from Parent
```

Hello from Child

📌 super() is vital for extending or modifying base class methods in a clean and scalable manner.

---

123. What are magic methods in Python?
Magic methods (or dunder methods) are special methods with double underscores (__methodname__) that give classes special behavior.

Examples:

- __init__: Constructor
- __str__: String representation
- __repr__: Developer-facing representation
- __len__, __getitem__, __eq__, etc.

Example:

```
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return f"Book: {self.title}"

b = Book("Python Basics")
print(b)  # Output: Book: Python Basics
```

📌 Magic methods enable **operator overloading**, custom iteration, and richer object behavior.

---

124. What are Python data classes and when should you use them?
Data classes are introduced in Python 3.7 via the dataclasses module. They reduce boilerplate when defining classes meant to **store data**.

Features:

- Auto-generates __init__, __repr__, __eq__, etc.
- Supports default values and type hints

Example:

```
from dataclasses import dataclass

@dataclass
```

```
class Employee:
    name: str
    age: int
    role: str = "Analyst"
```

Benefits:

- Clean syntax
- Easier comparison and representation
- Good for simple data containers

📌 Use data classes for modeling data records or DTOs (Data Transfer Objects).

---

125. What are metaclasses in Python?
A **metaclass** is a class of a class. It defines how classes themselves behave.

Default metaclass: type

Use Cases for Custom Metaclasses:

- Enforcing coding standards
- Automatically adding methods or attributes
- Logging and validation during class creation

Example:

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass
```

Output:

```
Creating class MyClass
```

📌 Metaclasses are an advanced feature best used when building frameworks or highly dynamic systems.

---

126. What is monkey patching in Python?
Monkey patching refers to **modifying classes or modules at runtime**.

Use Cases:

- Fix bugs in third-party libraries without altering the source
- Mock external dependencies during testing

Example:

```
import math
math.sqrt = lambda x: "Patched!"
print(math.sqrt(4))  # Output: Patched!
```

Risks:

- Hard to debug
- Makes code behavior unpredictable

📌 Use monkey patching sparingly and only in controlled scenarios like testing.