CS1280: CIE-3 Viva voce Questions and Answers

Dr. Durgarao Guttula

April 16, 2025

Basic Programming Questions

1. **Explain the purpose of built-in functions in Python by naming a few examples.**

   Built-in functions in Python are pre-defined functions that are always available for use without requiring any import. They provide essential functionality for common operations. Examples include:

   - `len()` - returns the length of an object
   - `print()` - outputs text to the console
   - `range()` - generates a sequence of numbers
   - `type()` - returns the type of an object
   - `sum()` - sums the items of an iterable

2. **Describe how dictionary comprehension works in Python with a suitable example.**

   Dictionary comprehension is a concise way to create dictionaries using an iterable. The syntax is {`key_expr: value_expr for item in iterable`}. Example:

   ```
   squares = {x: x**2 for x in range(1, 6)}
   # Result: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
   ```

3. **Describe the concept of list comprehension in Python and how it is used.**

   List comprehension provides a compact way to create lists. The syntax is `[expression for item in iterable]`. It can include conditions:

   ```
   even_squares = [x**2 for x in range(10) if x % 2 == 0]
   ```

4. **Summarize the main data structures used in Python and their typical use cases.**

   - **Lists**: Mutable, ordered sequences - for collections that need modification
   - **Tuples**: Immutable, ordered sequences - for fixed data
   - **Dictionaries**: Key-value pairs - for fast lookups by key
   - **Sets**: Unordered, unique elements - for membership testing
   - **Strings**: Immutable text sequences - for text processing

5. **Explain how to retrieve a value from a dictionary efficiently using a key.**

   Use square bracket notation or the `get()` method:

   ```
   value = my_dict['key']  # Raises KeyError if missing
   value = my_dict.get('key', default)  # Returns default if missing
   ```

6. **Identify valid data types that can be used as dictionary keys in Python and explain why.**

   Only hashable (immutable) types can be dictionary keys:

   - Numbers (int, float)
   - Strings
   - Tuples (if they contain only hashable types)

   Mutable types like lists can't be keys because their hash value could change.

7. **Describe the methods used to sort elements in a dataset in Python.**

   - `sorted()` - returns a new sorted list from any iterable
   - `list.sort()` - sorts a list in-place
   - For custom sorting, use the `key` parameter

   Example:

   ```
   sorted_list = sorted(iterable, key=lambda x: x[1])
   ```

8. **Explain the concept of negative indexing in Python and how it is applied.**

   Negative indices count from the end of a sequence:

   - -1 refers to the last element
   - -2 refers to the second last element

   Example:

```
last = my_list[-1]
```

9. **Distinguish between lists and tuples in Python by comparing their properties.**

Property	List	Tuple
Mutability	Mutable	Immutable
Syntax	Square brackets []	Parentheses ()
Performance	Slightly slower	Faster
Use case	Changing collections	Fixed data

10. **Differentiate between a Series and a DataFrame in Pandas.**

    - **Series**: 1D array with axis labels (single column)
    - **DataFrame**: 2D table with labeled axes (rows and columns)

    A DataFrame is essentially a collection of Series objects.

11. **Explain what a lambda function is in Python and when it is typically used.**

    A lambda function is an anonymous, inline function defined with `lambda`. Syntax:

    ```
    lambda arguments: expression
    ```

    Used for short operations where a full function isn't needed, often with `map()`, `filter()`, or `sorted()`.

12. **Distinguish between the / and // operators in Python.**

    - `/`: Regular division (returns float)
    - `//`: Floor division (returns integer)

    Example:

    ```
    5 / 2  # 2.5
    5 // 2 # 2
    ```

13. **Describe how you would convert an integer into a string in Python.**

    Use the `str()` function:

    ```
    num_str = str(42)
    ```

14. **Explain the difference between mutable and immutable objects in Python with examples.**

    - **Mutable**: Can be changed after creation (lists, dicts, sets)
    - **Immutable**: Cannot be changed after creation (ints, strings, tuples)

Example:

```
# Mutable
lst = [1, 2]; lst[0] = 3  # Valid

# Immutable
s = "hello"; s[0] = 'H'  # Raises TypeError
```

15. Describe the difference between shallow copy and deep copy in Python and when to use each.

- **Shallow copy**: Copies only the top level (references to nested objects)
- **Deep copy**: Recursively copies all nested objects

Use shallow copy when nested objects don't need duplication, deep copy when they do.

16. Explain the purpose of the with statement in Python and how it improves code reliability.

The `with` statement ensures proper resource management (like file handling) by automatically calling cleanup operations (like `close()`) even if exceptions occur.

Example:

```
with open('file.txt') as f:
    data = f.read()
# File automatically closed here
```

17. Explain the use of the else block in a try/except construct in Python.

The `else` block executes only if no exceptions were raised in the `try` block. It separates the code that might raise exceptions from code that should run only when no exceptions occur.

18. Describe the advantages of using NumPy arrays over regular Python lists.

- Faster operations (vectorized)
- Less memory usage
- Convenient mathematical operations
- Broadcasting capabilities
- Better for multidimensional data

19. Explain how to create a NumPy array and initialize it with values.

```
import numpy as np
arr = np.array([1, 2, 3])  # From list
zeros = np.zeros(5)        # Array of zeros
ones = np.ones((2,3))      # 2x3 array of ones
range_arr = np.arange(10)  # Like range(10)
```

20. Describe how to select specific rows and columns from a DataFrame using Pandas.

Use `loc[]` (label-based) or `iloc[]` (position-based):

```
# Select rows 1-3, columns 'A' and 'B'
df.loc[1:3, ['A', 'B']]

# Select first 2 rows, first 3 columns
df.iloc[:2, :3]
```

21. Distinguish between loc and iloc in Pandas with suitable examples.

- `loc`: Label-based indexing

  ```
  df.loc['row_label', 'column_label']
  ```

- `iloc`: Position-based indexing

  ```
  df.iloc[0, 1]  # First row, second column
  ```

22. Explain the use of the pd.concat() function in combining DataFrames.

`pd.concat()` combines DataFrames along an axis (default axis=0 for rows):

```
combined = pd.concat([df1, df2], axis=1)  # Side by side
```

23. Differentiate between del(), clear(), remove(), and pop() in Python.

- `del`: Deletes an item by index or the entire object
- `clear()`: Removes all items from a container
- `remove()`: Removes the first matching value
- `pop()`: Removes and returns an item by index

24. Compare and explain the behavior of pass, continue, and break in Python loops.

- `pass`: Does nothing (placeholder)
- `continue`: Skips to next iteration
- `break`: Exits the loop entirely

25. Distinguish between the append() and extend() methods used with lists in Python.

- `append()`: Adds its argument as a single element
- `extend()`: Adds elements of the iterable argument

Example:

```
lst = [1, 2]
lst.append([3,4])   # [1, 2, [3, 4]]
lst.extend([3,4])   # [1, 2, 3, 4]
```

26. Describe how exceptions are handled in Python with an example.

Python uses try-except blocks:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
except Exception as e:
    print(f"An error occurred: {e}")
else:
    print("Success!")
finally:
    print("This always executes")
```

Data Analysis Questions

1. **Explain why Python is widely used for data analysis and its advantages over other tools.**

   Python is popular for data analysis because:

   - **Rich ecosystem**: Libraries like Pandas, NumPy, SciPy, and scikit-learn
   - **Readability**: Clean syntax makes code maintainable
   - **Versatility**: Can handle all data analysis stages (cleaning, modeling, visualization)
   - **Performance**: NumPy and Pandas are optimized for speed
   - **Community**: Large user base and extensive documentation
   - **Integration**: Works well with databases, web apps, and other tools

   Advantages over tools like Excel or R:

   - Handles larger datasets more efficiently
   - More reproducible workflows
   - Better for automation and production deployment

2. **Describe what Pandas is in Python and how it supports data analysis.**

   Pandas is a Python library providing:

   - **DataFrame**: 2D labeled data structure (like spreadsheet)
   - **Series**: 1D labeled array
   - Tools for:
     - Reading/writing various file formats (CSV, Excel, SQL)
     - Data cleaning and transformation
     - Handling missing data
     - Merging and joining datasets
     - Time series functionality
     - Powerful grouping and aggregation

3. **Explain the steps to read a CSV file in Python using Pandas.**

```
import pandas as pd

# Basic reading
df = pd.read_csv('file.csv')

# With options
df = pd.read_csv('file.csv',
```

```
header=0,        # Use first row as headers
index_col=0,    # Use first column as index
na_values=['NA']) # Custom NA values
```

4. **Explain what NumPy is and how it simplifies the data analysis process.**

   NumPy is Python's fundamental package for numerical computing:

   - Provides **ndarray** - efficient multidimensional arrays
   - Enables vectorized operations (no loops needed)
   - Mathematical functions optimized for arrays
   - Memory-efficient storage
   - Basis for many other data science libraries

   Example advantage:

```
# Without NumPy
result = [x + y for x, y in zip(list1, list2)]

# With NumPy
result = array1 + array2  # Faster and cleaner
```

5. **Describe how to calculate the mean of a list or array in Python.**

   - For lists:

     ```
     mean = sum(my_list) / len(my_list)
     ```

   - With NumPy:

     ```
     import numpy as np
     mean = np.mean(my_array)
     ```

   - With Pandas:

     ```
     mean = df['column'].mean()
     ```

6. **Explain the steps involved in a typical data analysis process.**

   (a) Data collection (from files, databases, APIs)
   (b) Data cleaning (handle missing values, outliers)
   (c) Exploratory Data Analysis (EDA)
   (d) Feature engineering
   (e) Modeling (if applicable)
   (f) Visualization and communication
   (g) Deployment (if applicable)

7. **Compare Seaborn and Matplotlib and explain which you would prefer for plotting and why.**

Matplotlib	**Seaborn**
More customizable	Higher-level interface
More verbose syntax	More concise syntax
Basic statistical plots	Advanced statistical plots
Requires more styling	Attractive defaults

Preference: Seaborn for quick EDA, Matplotlib for custom publication-quality figures.

8. **Differentiate between a Series and a DataFrame in Pandas.**

Series	**DataFrame**
1-dimensional	2-dimensional
Single column	Multiple columns
One dtype per Series	Different dtypes per column

Example:

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

9. **Explain how to identify duplicate values in a dataset for a specific variable using Python.**

```
# For a specific column
duplicates = df.duplicated('column_name')

# Show duplicate rows
duplicate_rows = df[df.duplicated('column_name', keep=False)]

# Count duplicates
duplicate_count = df['column_name'].duplicated().sum()
```

10. **Describe the difference between merge, join, and concatenate operations in Pandas.**

- **concat**: Stacks DataFrames vertically or horizontally
- **merge**: Database-style joins (inner, outer, left, right)
- **join**: Convenience method for merging on indices

Example:

```
# Concatenation
pd.concat([df1, df2], axis=0)
```

```
# Merge
pd.merge(df1, df2, on='key', how='inner')

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

11. **Explain how to identify and handle missing values in a dataset.**

    Identification:

    ```
    df.isna().sum()  # Count missing per column
    df[df['col'].isna()]  # Show rows with missing values
    ```

    Handling:

    ```
    # Drop missing values
    df.dropna()

    # Fill with specific value
    df.fillna(0)

    # Fill with mean
    df.fillna(df.mean())

    # Interpolation
    df.interpolate()
    ```

12. **Describe how to normalize or standardize a dataset in Python.**

    Normalization (Min-Max scaling):

    ```
    from sklearn.preprocessing import MinMaxScaler
    scaler = MinMaxScaler()
    df_normalized = scaler.fit_transform(df)
    ```

    Standardization (Z-score):

    ```
    from sklearn.preprocessing import StandardScaler
    scaler = StandardScaler()
    df_standardized = scaler.fit_transform(df)
    ```

13. **Explain what outliers are and how you would detect them in a dataset.**

    Outliers are data points significantly different from others. Detection methods:

    - Statistical methods:

```
Q1 = df['col'].quantile(0.25)
Q3 = df['col'].quantile(0.75)
IQR = Q3 - Q1
outliers = df[(df['col'] < (Q1 - 1.5*IQR)) |
              (df['col'] > (Q3 + 1.5*IQR))]
```

- Visualization: Box plots, scatter plots
- Z-score method (for normally distributed data)

14. **Describe the steps to create a line plot using Matplotlib.**

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10,6))
plt.plot(x_values, y_values,
         label='Trend',
         color='blue',
         linewidth=2)
plt.title('Title')
plt.xlabel('X Label')
plt.ylabel('Y Label')
plt.legend()
plt.grid(True)
plt.show()
```

15. **Explain the purpose and steps involved in data cleaning during data analysis.**

    Purpose: Ensure data quality for accurate analysis. Steps:

    (a) Handle missing values
    (b) Remove or handle outliers
    (c) Fix inconsistent formatting
    (d) Correct data types
    (e) Standardize categorical values
    (f) Validate data ranges
    (g) Remove duplicates

16. **Describe common methods used to handle missing values in a DataFrame.**

    - Deletion: `dropna()`
    - Imputation:
      - Mean/median/mode: `fillna(df.mean())`
      - Forward/backward fill: `ffill()`, `bfill()`

    - Interpolation: `interpolate()`
  - Model-based imputation (using scikit-learn)
  - Mark as missing (for some algorithms)

17. **Explain how to calculate descriptive statistics for a DataFrame using Pandas.**

```
# Basic statistics
df.describe()

# Specific metrics
df.mean()
df.median()
df.std()
df.quantile(0.25)

# For categorical data
df['category'].value_counts()
df['category'].mode()
```

18. **Describe what a histogram is and its role in data analysis.**

    A histogram shows the distribution of numerical data by:

    - Dividing data into bins
    - Counting observations in each bin
    - Visualizing as bars

    Roles:

    - Understand data distribution
    - Identify outliers
    - See skewness
    - Guide data transformations

19. **Explain how to generate a histogram using Matplotlib.**

```
plt.hist(df['column'], bins=20, color='blue', alpha=0.7)
plt.title('Distribution of Values')
plt.xlabel('Value Range')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

20. **Describe the role of data visualization in data analysis and its importance.**

    Roles:

- Exploratory analysis (identify patterns, outliers)
- Data quality assessment
- Feature selection guidance
- Results communication
- Storytelling with data

Importance:

- Humans process visuals faster than numbers
- Reveals insights hidden in raw data
- Facilitates decision-making
- Helps identify relationships

21. **Explain why data normalization is performed in the data analysis process.**

Reasons:

- Brings features to similar scales
- Improves algorithm performance (especially distance-based)
- Speeds up convergence in gradient descent
- Makes features comparable
- Required for some algorithms (PCA, neural networks)

22. **Describe some common techniques used for data normalization.**

- Min-Max scaling: $X' = \frac{X - X_{min}}{X_{max} - X_{min}}$
- Z-score standardization: $X' = \frac{X - \mu}{\sigma}$
- Decimal scaling
- Log transformation
- Unit vector scaling (for direction preservation)

23. **Explain how to normalize data using scikit-learn in Python.**

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler

# Min-Max normalization
scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(df[['column']])

# Standardization
scaler = StandardScaler()
standardized_data = scaler.fit_transform(df[['column']])
```

24. **Describe the concept and purpose of data aggregation in analysis.**

    Data aggregation combines multiple data points into summary statistics:

    - Purpose:
      - Reduce data complexity
      - Identify patterns
      - Compute metrics
      - Prepare for visualization
    - Common operations: sum, average, count, min, max

25. **Explain how to perform data aggregation in Pandas using group operations.**

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

# Multiple aggregations
df.groupby('category').agg({
    'value1': ['mean', 'max'],
    'value2': 'sum'
})

# Named aggregations (Pandas 0.25+)
df.groupby('category').agg(
    avg_value=('value', 'mean'),
    total=('value', 'sum')
)
```

26. **Explain the purpose of data filtering in analysis and when it is applied.**

    Purpose: Select subset of data relevant to analysis. Applied when:

    - Focusing on specific segments
    - Removing irrelevant data
    - Handling outliers
    - Preparing training/test sets
    - Analyzing time periods
    - Meeting business requirements

27. **Describe how to filter rows in a Pandas DataFrame based on conditions.**

```
# Single condition
filtered = df[df['column'] > 100]
```

```
# Multiple conditions
filtered = df[(df['col1'] > 100) & (df['col2'] == 'A')]

# Using query()
filtered = df.query("column > 100 and category == 'A'")

# Using isin()
filtered = df[df['category'].isin(['A', 'B'])]
```

28. **Explain how one-hot encoding is used in Pandas for categorical data.**

    Converts categorical variables to binary columns:

```
# Using get_dummies
encoded = pd.get_dummies(df['category'])

# For multiple columns
encoded = pd.get_dummies(df, columns=['cat1', 'cat2'])

# With drop_first to avoid multicollinearity
encoded = pd.get_dummies(df, drop_first=True)
```

29. **Describe how group-wise operations are performed on a DataFrame using Pandas.**

```
# Grouping and applying functions
df.groupby('group_column').agg({'value': 'mean'})

# Multiple operations
df.groupby('group_column')['value'].describe()

# Custom function
def normalize(x):
    return (x - x.mean()) / x.std()

df.groupby('group_column').transform(normalize)

# Iterating through groups
for name, group in df.groupby('category'):
    print(f"Processing {name}")
    # Do something with group
```