Q1.Special Operators in Python

Special operators in Python are operators with special functions or usage. They include the following:

1. **Identity Operators**:
   - is: Returns True if two variables point to the same object.
   - is not: Returns True if two variables do not point to the same object.

   Example:

```
a = [1, 2, 3]
b = a
print(a is b)  # True
```

2. **Membership Operators**:
   - in: Returns True if a value is found in a sequence (such as a list, string, or tuple).
   - not in: Returns True if a value is not found in a sequence.

   Example:

```
lst = [1, 2, 3]
print(2 in lst)   # True
print(4 not in lst)  # True
```

3. **Bitwise Operators**:
   - These are used to perform bit-level operations. Common bitwise operators include & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), and >> (right shift).

Q2.Difference between Python List and Numpy Array

Feature	Python List	Numpy Array
Type	Collection of items (can be mixed types)	Homogeneous array (all elements must be of the same type)
Performance	Slower for numerical computations	Faster for numerical computations due to optimizations
Size	Can grow dynamically	Fixed size once created
Operations	Supports general operations like appending, slicing	Supports advanced mathematical operations like matrix operations, element-wise operations
Memory Usage	Takes more memory for storing items	More memory-efficient, as it uses contiguous memory blocks
Dimensionality	Typically used for 1D or simple nested lists	Supports multi-dimensional arrays (2D, 3D, etc.)

Feature	Python List	Numpy Array
Libraries	Built-in in Python	Requires importing numpy

Example:

- **Python List**:
  ```
  lst = [1, 2, 3, 4]
  lst.append(5)
  print(lst)  # Output: [1, 2, 3, 4, 5]
  ```

- **Numpy Array**:
  ```
  import numpy as np
  arr = np.array([1, 2, 3, 4])
  arr = arr + 5  # Element-wise addition
  print(arr)  # Output: [6, 7, 8, 9]
  ```

Q3. State any Four Time Modules in Python

The time module in Python provides various time-related functions. Here are four commonly used functions from the time module:

1. **time()**:
   - Returns the current time as the number of seconds since the epoch (January 1, 1970, 00:00:00 UTC).

   ```
   import time
   current_time = time.time()
   print(current_time)
   ```

2. **sleep(seconds)**:
   - Pauses the execution of the current thread for the specified number of seconds.

   ```
   time.sleep(2)  # Pauses the program for 2 seconds
   ```

3. **ctime(seconds)**:
   - Converts the given time (in seconds since the epoch) to a human-readable string.

   ```
   readable_time = time.ctime(time.time())
   print(readable_time)  # Example: 'Tue Sep 28 14:34:40 2024'
   ```

4. **strftime(format, t)**:
    - Formats a time t (or the current time if no argument is passed) into a string according to a specified format.

```
formatted_time = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
print(formatted_time)  # Output: '2024-09-28 14:34:40'
```

Q4. What is a Class Variable?

A **class variable** is a variable that is shared among all instances of a class. It is defined within the class but outside of any instance methods. Class variables are typically used to store information that is the same for all objects of the class. They are accessed using the class name or an instance.

- **Example**:

```
class Car:
    # This is a class variable
    wheels = 4

    def __init__(self, model):
        self.model = model  # Instance variable

car1 = Car("Sedan")
car2 = Car("SUV")

print(car1.wheels)  # Output: 4
print(car2.wheels)  # Output: 4

# Changing class variable
Car.wheels = 6
print(car1.wheels)  # Output: 6
print(car2.wheels)  # Output: 6
```

- **Key Points**:
    - Class variables are shared across all instances of the class.
    - Changing the class variable affects all instances.
    - They are used when a variable should have the same value across all objects of a class.

Q5. Geometry Management Methods in Python (Tkinter)

In **Tkinter**, geometry management methods are used to organize and place widgets within a window. There are three main geometry managers:

1. **pack()**:
   - Organizes widgets by packing them in order (top, bottom, left, right) within the parent widget. Widgets are placed in the next available position.
   - Example:

     python
     Copy code
     label.pack(side="top")
2. **grid(row, column)**:
   - Organizes widgets in a grid format, where each widget is placed at a specific row and column.
   - Example:

     python
     Copy code
     button.grid(row=0, column=1)
3. **place(x, y)**:
   - Organizes widgets at an absolute position by specifying the x and y coordinates relative to the parent widget.
   - Example:

     python
     Copy code
     entry.place(x=50, y=100)

These methods help arrange widgets in a window according to specific layouts or coordinate-based positions.

Q6.What is a Bind Method?

The **bind()** method in Tkinter is used to attach a specific event to a widget. When the specified event occurs (like a key press, mouse click, or other actions), the assigned function or handler is triggered. This allows for interactive behavior in the application.

- **Syntax**:
  widget.bind(event, handler)

- **Example**:
```
import tkinter as tk

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

root = tk.Tk()
button = tk.Button(root, text="Click Me")
button.bind("<Button-1>", on_click)  # Binds left mouse click to the function
button.pack()

root.mainloop()
```
- **Key Points**:
    - Events like "<Button-1>" (left-click), "<KeyPress>", "<Enter>" (mouse over), etc., can be used.
    - The **bind()** method helps make the application interactive by responding to user inputs or actions.

Q7. What is Seaborn?

Seaborn is a Python data visualization library built on top of Matplotlib. It provides a high-level interface for creating informative and attractive statistical graphics. Seaborn simplifies the creation of complex plots, such as heatmaps, violin plots, and pair plots, with fewer lines of code and improved aesthetics.

- **Key Features of Seaborn**:
    1. Built-in themes for better visual presentation.
    2. Functions to visualize univariate and bivariate distributions.
    3. Tools for visualizing linear regression models.
    4. Capability to handle Pandas DataFrames, making it easier to work with datasets.
- **Example**:
```
python
Copy code
Explain
import seaborn as sns
import matplotlib.pyplot as plt

# Load the built-in 'tips' dataset
tips = sns.load_dataset("tips")

# Create a simple scatter plot
sns.scatterplot(x="total_bill", y="tip", data=tips)
plt.show()
```

Seaborn is particularly useful for statistical plots and data analysis because it integrates well with Pandas, providing easier ways to plot and analyze data.

Q8. Two Common Exceptions in Python

1. **IndexError**:
   - Raised when trying to access an element from a list, tuple, or string using an index that is out of range.
   - **Example**:

     ```
     my_list = [1, 2, 3]
     print(my_list[3])  # This will raise an IndexError because there is no index 3
     ```
2. **KeyError**:
   - Raised when trying to access a key that does not exist in a dictionary.
   - **Example**:

     ```
     my_dict = {'name': 'Alice'}
     print(my_dict['age'])  # This will raise a KeyError because 'age' is not a key in the dictionary
     ```

These exceptions are common during runtime and can be handled using **try-except** blocks to prevent program crashes.

Q9.Advantages of Pandas

Pandas is a powerful data manipulation and analysis library in Python, especially useful for handling structured data such as tables. Here are some advantages of using Pandas:

1. **Easy Data Manipulation**:
   - Pandas provides simple functions to clean, filter, and modify large datasets with ease, such as adding/removing columns, handling missing data, and performing aggregation.
2. **Efficient Data Handling**:
   - It handles large datasets efficiently by utilizing optimized data structures, allowing operations like merging, grouping, and reshaping data to be performed quickly.
3. **Data Alignment and Indexing**:
   - Automatic alignment of data in operations makes it easier to work with datasets of different shapes and ensures that operations are performed on matching data.
4. **Integration with Other Libraries**:
   - Pandas integrates well with libraries like NumPy, Matplotlib, and Seaborn, making it easy to perform computations and visualizations in one workflow.
5. **Powerful Grouping and Aggregation**:

- Pandas provides powerful functions such as groupby() for grouping data and performing aggregate functions like sum(), mean(), and count() on grouped data.

6. **Easy Data Input/Output**:
   - Pandas makes it easy to read and write data from various file formats like CSV, Excel, JSON, SQL, and more.

Q10. How to Create a Class and Object in Python

In Python, a **class** is a blueprint for creating objects, and an **object** is an instance of a class. A class can contain attributes (variables) and methods (functions).
Creating a Class:

- **Syntax**:

```
class ClassName:
    # Constructor method to initialize object attributes
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    # A method inside the class
    def method_name(self):
        print("This is a method of the class")
```

Creating an Object:

- An object is created by calling the class as if it were a function.
- **Syntax**:
  object_name = ClassName(attribute1_value, attribute2_value)

Example:

```
# Defining a class named Car
class Car:
    # Constructor method to initialize object properties
    def __init__(self, brand, model):
        self.brand = brand  # Attribute
        self.model = model  # Attribute

    # A method to display car details
    def display_info(self):
        print(f"Car brand: {self.brand}, Model: {self.model}")
```

```
# Creating an object (instance) of the Car class
my_car = Car("Toyota", "Corolla")

# Accessing the object's method
my_car.display_info()  # Output: Car brand: Toyota, Model: Corolla
```
In this example:

- **Car** is the class.
- **my_car** is the object created from the class.
- The object uses its methods (like display_info()) to perform actions.

Q11. Math and Cmath Modules in Python

1. Math Module:

The **math** module in Python provides functions for performing mathematical operations like finding trigonometric values, logarithms, square roots, and more. It deals with real numbers.

- **Key Functions in the Math Module**:
  - **math.sqrt(x)**: Returns the square root of x.
  - **math.factorial(x)**: Returns the factorial of x.
  - **math.pow(x, y)**: Returns x raised to the power of y (equivalent to x ** y).
  - **math.log(x, base)**: Returns the logarithm of x to the specified base. If no base is provided, it returns the natural logarithm (base e).
  - **math.sin(x), math.cos(x), math.tan(x)**: Returns the sine, cosine, and tangent of x (where x is in radians).
  - **math.pi**: Returns the value of π (3.14159...).
  - **math.e**: Returns the value of Euler's number e (2.718...).
- **Example**:

```
import math
print(math.sqrt(16))       # Output: 4.0
print(math.factorial(5))   # Output: 120
print(math.log(100, 10))   # Output: 2.0
```

2. Cmath Module:

The **cmath** module is similar to the math module but is specifically used for complex numbers. In Python, complex numbers have a real part and an imaginary part, represented as a + bj (where j is the square root of -1).

- **Key Functions in the Cmath Module**:
  - **cmath.sqrt(x)**: Returns the square root of a complex number x.
  - **cmath.exp(x)**: Returns e raised to the power of a complex number x.
  - **cmath.log(x, base)**: Returns the logarithm of a complex number x to a specified base.
  - **cmath.phase(x)**: Returns the phase (or argument) of a complex number.

- **cmath.polar(x)**: Converts a complex number into its polar coordinates.
- **cmath.sin(x), cmath.cos(x), cmath.tan(x)**: Returns the sine, cosine, and tangent of a complex number x.

- **Example**:

```
import cmath
print(cmath.sqrt(-1))      # Output: 1j (imaginary unit)
print(cmath.log(1 + 1j))                    # Output:
(0.3465735902799726+0.7853981633974483j)
```

Q12. Different Data Types in Python

Python has several built-in data types that are used to define variables and store different types of data.

1. Numeric Types:

- **int (Integer)**: Represents whole numbers (both positive and negative).
  - Example: a = 10
- **float**: Represents floating-point numbers (decimal numbers).
  - Example: b = 3.14
- **complex**: Represents complex numbers, with a real part and an imaginary part.
  - Example: c = 2 + 3j

2. Sequence Types:

- **str (String)**: A sequence of Unicode characters, used for storing text.
  - Example: name = "Alice"
- **list**: An ordered and mutable collection of elements, which can hold different types of data.
  - Example: my_list = [1, 2, 3, "apple"]
- **tuple**: Similar to a list, but tuples are immutable (cannot be changed after creation).
  - Example: my_tuple = (1, 2, 3, "banana")

3. Mapping Type:

- **dict (Dictionary)**: A collection of key-value pairs. Keys are unique, and values can be any data type.
  - Example: my_dict = {"name": "Alice", "age": 25}

4. Set Types:

- **set**: An unordered collection of unique elements.
  - Example: my_set = {1, 2, 3, 4}
- **frozenset**: An immutable version of a set.
  - Example: frozen_set = frozenset([1, 2, 3])

5. Boolean Type:

- **bool**: Represents True or False values.
    - Example: is_active = True

6. Binary Types:

- **bytes**: Represents a sequence of bytes (immutable).
    - Example: my_bytes = b"Hello"
- **bytearray**: Represents a mutable sequence of bytes.
    - Example: my_bytearray = bytearray([1, 2, 3])
- **memoryview**: Provides a memory view of another object (e.g., bytes or bytearray).
    - Example: my_memoryview = memoryview(b"abc")

Q13. Inheritance in Python (Brief Explanation with Syntax)

Inheritance in Python is a feature that allows a class (child class) to inherit properties and behaviors (methods) from another class (parent class). It promotes code reuse and makes creating and maintaining complex systems easier.

Syntax:

```
class ParentClass:
    def __init__(self, attr1):
        self.attr1 = attr1

    def parent_method(self):
        print("This is a method in the parent class.")

# Child class inheriting from ParentClass
class ChildClass(ParentClass):
    def __init__(self, attr1, attr2):
        # Call the constructor of ParentClass
        super().__init__(attr1)
        self.attr2 = attr2

    def child_method(self):
        print("This is a method in the child class.")
```

Example:

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

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

dog = Dog()
dog.speak()  # Inherited method
dog.bark()   # Child class method
```

Types of Inheritance in Python:

1. **Single Inheritance**: One child class inherits from one parent class.
2. **Multiple Inheritance**: A child class inherits from more than one parent class.
3. **Multilevel Inheritance**: A child class inherits from another child class, forming a chain.
4. **Hierarchical Inheritance**: Multiple child classes inherit from the same parent class.
5. **Hybrid Inheritance**: A combination of multiple types of inheritance.

---

14. Various Types of Exception Handling in Python

Exception handling in Python is done using **try-except** blocks. It helps in managing errors gracefully without crashing the program.

```
try:
    # Code that may cause an exception
    pass
except ExceptionType:
    # Code that runs if the exception occurs
    pass
finally:
    # Code that always runs, whether an exception occurred or not (optional)
    pass
```

Common Exception Handling Types:

1. **Single try-except Block**:
   - Catches a specific exception.
   - Example:

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

2. **Multiple Exceptions**:
   - Handle multiple specific exceptions.
   - Example:

```
try:
    x = int("abc")
except (ValueError, TypeError):
    print("Invalid input")
```

3. **finally Block**:
   - The finally block is used to execute code, regardless of whether an exception occurred or not.
   - Example:

```
try:
    f = open("test.txt")
except FileNotFoundError:
    print("File not found")
finally:
    print("This will always run")
```

4. **else Block**:
   - The else block is executed if no exception occurs in the try block.
   - Example:

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print("Division successful")
```

5. **Raise Exception**:
   - Manually raising an exception using the raise keyword.
   - Example:

```
x = -1
if x < 0:
    raise ValueError("Negative value not allowed")
```

15. Principle of Keras

Keras is an open-source, high-level neural network API written in Python. It is built on top of deep learning libraries like TensorFlow and Theano. Keras is designed to enable fast experimentation and allows easy and quick prototyping of neural networks. Principles of Keras:

1. **User-Friendly and Fast Prototyping**:
   - Keras aims to reduce the cognitive load by providing a simple and intuitive API. This allows developers to quickly build and experiment with neural networks.
2. **Modularity**:
   - Neural network components such as layers, loss functions, optimizers, and metrics are all standalone, configurable objects. This modularity helps users easily create custom models and reuse components.
3. **Support for Multiple Backends**:
   - Keras supports multiple backend engines, primarily TensorFlow and Theano. The user can seamlessly switch between these backends without having to change their code.
4. **Extensibility**:
   - New components can be added and customized by writing new layers, metrics, or optimizers. This makes Keras highly flexible for research and development.
5. **Compatibility with Different Environments**:
   - Keras is compatible with both CPU and GPU, enabling users to train their models efficiently on a variety of hardware.
6. **Simple Debugging**:
   - Keras provides clear error messages and intuitive error tracing, making debugging simpler and faster.

Example of a Simple Neural Network in Keras:

```
from keras.models import Sequential
from keras.layers import Dense

# Define the model
model = Sequential()

# Add layers to the model
model.add(Dense(32, activation='relu', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam',                    loss='categorical_crossentropy',
metrics=['accuracy'])
# Train the model on a dataset
# model.fit(X_train, y_train, epochs=10)
```

16. Built-in Dictionary Functions in Python with Examples

Python dictionaries are collections of key-value pairs. Python provides several built-in functions to work with dictionaries.

1. dict.get(key, default=None):

- Returns the value associated with the given key. If the key does not exist, it returns the default value (optional).
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict.get("name"))         # Output: Alice
print(my_dict.get("gender", "N/A")) # Output: N/A
```

2. dict.keys():

- Returns a view object containing all the keys in the dictionary.
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict.keys())  # Output: dict_keys(['name', 'age'])
```

3. dict.values():

- Returns a view object containing all the values in the dictionary.
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict.values())  # Output: dict_values(['Alice', 25])
```

4. dict.items():

- Returns a view object containing a list of all key-value pairs as tuples.
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict.items())  # Output: dict_items([('name', 'Alice'), ('age', 25)])
```

5. dict.update(other_dict):

- Updates the dictionary with key-value pairs from other_dict. If a key exists in both dictionaries, the value is updated.
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
my_dict.update({"age": 26, "city": "New York"})
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

6. dict.pop(key, default=None):

- Removes the key-value pair with the specified key and returns the value. If the key does not exist, it returns the default value.
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
age = my_dict.pop("age")
print(age)       # Output: 25
print(my_dict)   # Output: {'name': 'Alice'}
```

7. dict.clear():

- Removes all key-value pairs from the dictionary, making it empty.
- **Example**

```
my_dict = {"name": "Alice", "age": 25}
my_dict.clear()
print(my_dict)  # Output: {}
```

8. dict.copy():

- Returns a shallow copy of the dictionary.
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
copied_dict = my_dict.copy()
print(copied_dict)  # Output: {'name': 'Alice', 'age': 25}
```

9. dict.setdefault(key, default=None):

- Returns the value associated with the key if it exists. If the key does not exist, it inserts the key with the default value and returns the default.
- **Example**:

```
my_dict = {"name": "Alice", "age": 25}
my_dict.setdefault("gender", "female")
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'gender': 'female'}
```

---

17. Features of Pandas in Python

Pandas is a powerful library in Python for data manipulation and analysis. It provides two primary data structures: Series (1-dimensional) and DataFrame (2-dimensional), which are highly efficient for handling and analyzing structured data.
Key Features of Pandas:

1. **DataFrame and Series Structures**:
   - Pandas provides two main data structures: Series (1D) and DataFrame (2D) for managing and manipulating datasets of various shapes and sizes.
   - Example:

```
import pandas as pd
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
print(df)
```

2. **Data Alignment**:
   - Pandas automatically aligns data in arithmetic operations, making sure labels and indexes are matched correctly.
   - Example:

```
s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
s2 = pd.Series([4, 5, 6], index=['b', 'c', 'd'])
result = s1 + s2  # Automatically aligns based on index
print(result)
```

3. **Handling Missing Data**:
   - Pandas provides methods like fillna(), dropna(), and isna() to handle missing data efficiently.
   - Example:

```
df = pd.DataFrame({"A": [1, 2, None], "B": [4, None, 6]})
df.fillna(0)  # Replace missing values with 0
```

4. **Label-based and Position-based Indexing**:
   - Pandas allows for powerful indexing methods, such as label-based indexing using loc[] and position-based indexing using iloc[].
   - Example:

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
print(df.loc[0, 'A'])  # Label-based
print(df.iloc[0, 1])   # Position-based
```

5. **Data Aggregation and Grouping**:
   - Pandas provides powerful grouping functions like groupby() and allows for applying aggregate functions like sum(), mean(), etc.
   - Example:

```
data = {'Name': ['Alice', 'Bob', 'Alice'], 'Age': [25, 30, 25]}
```

```
df = pd.DataFrame(data)
df_grouped = df.groupby('Name').sum()
print(df_grouped)
```

6. **Merging and Joining Data**:
   - Pandas supports functions like merge(), concat(), and join() to combine datasets.
   - Example:

```
df1 = pd.DataFrame({'key': ['A', 'B'], 'value': [1, 2]})
df2 = pd.DataFrame({'key': ['A', 'C'], 'value': [3, 4]})
result = pd.merge(df1, df2, on='key', how='outer')
print(result)
```

7. **Reading and Writing Data**:
   - Pandas supports reading and writing to various file formats, including CSV, Excel, JSON, SQL, etc.
   - Example:

```
df.to_csv('output.csv')  # Save DataFrame to CSV
df = pd.read_csv('input.csv')  # Read CSV into DataFrame
```

8. **Time Series Functionality**:
   - Pandas is well-suited for handling time series data. It supports resampling, shifting, rolling windows, and more.
   - Example:

```
date_range = pd.date_range('20230101', periods=6)
df = pd.DataFrame({"A": [1, 2, 3, 4, 5, 6]}, index=date_range)
print(df)
```

9. **Visualization Integration**:
   - Pandas integrates well with data visualization libraries like Matplotlib and Seaborn, allowing for easy plotting of DataFrames and Series.
   - Example:

```
df.plot(kind='line')
plt.show()
```

---

18. Explanation of entry.delete and entry.insert with Syntax and Example

In Python's Tkinter library, Entry widgets are used to accept text input from the user. Functions like entry.delete() and entry.insert() allow you to manipulate the content inside the Entry widget.

1. entry.delete(start, end):
   - This method deletes characters from the Entry widget.
   - **Parameters**:
     - start: The index position from where to start deleting.
     - end: The index position where to stop deleting. If omitted, it will delete one character.
     - 0 means the first character, and END means the last character.
   - **Syntax**:

     ```
     entry.delete(0, END)  # Deletes all characters from start to end
     ```
   - **Example**:

     ```
     from tkinter import *

     root = Tk()
     entry = Entry(root)
     entry.pack()
     entry.insert(0, "Enter your name")  # Insert a default message

     def clear_text():
         entry.delete(0, END)  # Clear the entire content

     button = Button(root, text="Clear", command=clear_text)
     button.pack()
     root.mainloop()
     ```

2. entry.insert(index, string):
   - This method inserts a string at the given index position in the Entry widget.
   - **Parameters**:
     - index: The position where the string will be inserted.
     - string: The text that will be inserted.
   - **Syntax**:

     ```
     entry.insert(0, "Hello")  # Inserts "Hello" at the beginning of the entry widget
     ```

- **Example**:

```
from tkinter import *

root = Tk()
entry = Entry(root)
entry.pack()

def insert_text():
    entry.insert(0, "Welcome ")  # Inserts text at the start of the Entry widget

button = Button(root, text="Insert Text", command=insert_text)
button.pack()
root.mainloop()
```

---

19. Python Program to Find Factors of a Given Number

A **factor** of a number is a number that divides it exactly without leaving a remainder. Here's a Python program to find all the factors of a given number.

```
# Function to find factors of a number
def find_factors(num):
    factors = []
    for i in range(1, num + 1):
        if num % i == 0:
            factors.append(i)
    return factors

# Input from user
number = int(input("Enter a number: "))

# Calling the function and displaying the factors
print("Factors of", number, "are:", find_factors(number))
```

Example:
If the input is 12, the output will be:
less

```
Factors of 12 are: [1, 2, 3, 4, 6, 12]
```

---

20. Python Script to Generate Fibonacci Terms Using Generator Function

A **generator function** in Python is a special type of function that yields values one at a time using the yield statement, instead of returning them all at once.
The **Fibonacci sequence** is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.
Python Program to Generate Fibonacci Terms Using Generator:

```
# Generator function to generate Fibonacci numbers
def fibonacci_gen():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Create a generator object
fib_gen = fibonacci_gen()

# Get input from user for number of terms
num_terms = int(input("Enter the number of Fibonacci terms to generate: "))

# Generate and print the Fibonacci terms
for i in range(num_terms):
    print(next(fib_gen))
```

Example:
If the input is 7, the output will be:
Copy code
Explain
0
1
1
2
3
5
8
This program will generate the Fibonacci terms up to the number of terms specified by

21. How to Define a Function in Python (Explanation with Example)

In Python, a **function** is defined using the def keyword followed by the function name and parentheses (). A function is a block of code that only runs when it is called, and it can take arguments and return results.

Explain

```
def function_name(parameters):
    """docstring (optional): describes the function"""
    # Block of code
    return result  # Optional
```

Example of a Function:

```
# Function to add two numbers
def add_numbers(num1, num2):
    """This function adds two numbers and returns the result"""
    result = num1 + num2
    return result

# Calling the function and printing the result
sum = add_numbers(5, 10)
print("Sum:", sum)  # Output: Sum: 15
```

Explanation:

- def add_numbers(num1, num2) defines the function add_numbers with two parameters num1 and num2.
- Inside the function, the two numbers are added, and the result is returned using the return statement.
- The function is called using add_numbers(5, 10), which passes the arguments 5 and 10 to the function, and the result is printed.

---

22. Explanation of the EXCEPT Clause with No Exception

In Python, the try-except block is used to handle exceptions (errors). When no exception occurs, the except block is not executed, and the program proceeds as normal. You can still have an except clause in the code, even if no exception is raised, but it will remain unused unless an error occurs.

Syntax:

```
try:
    # Block of code that might raise an exception
    pass
except:
    # Block of code to run if an exception occurs
    pass
```

Example:

```
try:
    # Code without an exception
    x = 10
    y = 20
    print("The result is:", x + y)  # This line works without raising an error
except:
    # This block is not executed because no exception occurs
    print("An error occurred.")
```

Output:

csharp

```
The result is: 30
```

Explanation:

- The try block contains code that runs without any issues (addition of two numbers).
- Since no exception occurs, the except block is not executed.
- If there were an error (e.g., division by zero), the except block would catch the error and prevent the program from crashing.

---

c) Explanation of IS-A Relationship and HAS-A Relationship with Example

1. IS-A Relationship (Inheritance):

- The **IS-A** relationship represents inheritance in object-oriented programming. It implies that a subclass **is a** type of its superclass. The subclass inherits the properties and behaviors (methods) of the superclass.
- For example, if **Dog** is a subclass of **Animal**, then **Dog IS-A Animal**. This relationship is used when we want to implement inheritance.

```
# Parent class
class Animal:
    def speak(self):
        return "Animal speaks"

# Child class (Dog IS-A Animal)
class Dog(Animal):
    def bark(self):
        return "Dog barks"

# Creating an object of Dog class
dog = Dog()
print(dog.speak())  # Inherited from Animal class
print(dog.bark())   # Dog's own method
```

Explanation:

- The Dog class inherits from the Animal class, establishing the **IS-A** relationship (Dog **is a** type of Animal).
- The Dog class can access both the speak() method (from Animal) and its own method bark().

2. HAS-A Relationship (Composition):

- The **HAS-A** relationship represents **composition**. In this case, one class contains or uses another class, implying that one object **has** another object.
- For example, if a **Car** has an **Engine**, we say **Car HAS-A Engine**. This relationship indicates that one class is composed of another class.

```
# Engine class (used as a component in Car class)
class Engine:
    def start(self):
        return "Engine starts"

# Car class (Car HAS-A Engine)
class Car:
    def __init__(self):
        self.engine = Engine()  # Composition (Car HAS-A Engine)

    def start_car(self):
        return self.engine.start()

# Creating an object of Car class
car = Car()
print(car.start_car())  # Output: Engine starts
```

Explanation:

- The Car class has an instance of the Engine class. This represents the **HAS-A** relationship (Car **has a** Engine).
- The start_car() method in the Car class calls the start() method of the Engine class, demonstrating composition.

---

d) Python Program to Check if a Given Key Already Exists in a Dictionary and Replace It

In this program, we check if a specific key exists in a dictionary. If it exists, we replace it with a new key-value pair.
Python Program:

```
# Function to replace key if it exists in the dictionary
def replace_key_in_dict(my_dict, old_key, new_key, new_value):
    if old_key in my_dict:
        del my_dict[old_key]  # Delete the old key
        my_dict[new_key] = new_value  # Insert the new key-value pair
    else:
        print(f"Key '{old_key}' not found in the dictionary.")

# Example dictionary
my_dict = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Input old key, new key and new value
old_key = 'age'
new_key = 'birth_year'
```

```
new_value = 1995

# Call the function
replace_key_in_dict(my_dict, old_key, new_key, new_value)

# Print the updated dictionary
print(my_dict)
```
Output:
python
Copy code
```
{'name': 'Alice', 'city': 'New York', 'birth_year': 1995}
```
Explanation:

- The function checks if the old_key exists in the dictionary using if old_key in my_dict.
- If it exists, the old key is deleted, and the new key-value pair is added to the dictionary.

---

e) Python Program to Swap the Value of Two Variables

There are different ways to swap the values of two variables in Python.
Method 1: Using a Temporary Variable

```
# Swapping using a temporary variable
a = 5
b = 10
print("Before swap: a =", a, "b =", b)

temp = a
a = b
b = temp

print("After swap: a =", a, "b =", b)
```
Method 2: Using Tuple Unpacking (Pythonic way)

```
# Swapping using tuple unpacking
a = 5
b = 10
print("Before swap: a =", a, "b =", b)

a, b = b, a  # Swapping values

print("After swap: a =", a, "b =", b)
```

Method 3: Using Arithmetic Operations

```
# Swapping using arithmetic operations
a = 5
b = 10
print("Before swap: a =", a, "b =", b)

a = a + b
b = a - b
a = a - b

print("After swap: a =", a, "b =", b)
```

Explanation:

- **Tuple Unpacking** is the most Pythonic and simplest way to swap values. It swaps the values of a and b without using a temporary variable.

a) Slicing Dictionaries

In Python, slicing refers to selecting a portion of a sequence, like a list or a string. However, dictionaries, being unordered collections of key-value pairs, do not support slicing directly. Unlike lists or strings, dictionaries are not indexed, so slicing them as you would a list is not possible. Still, you can manually slice dictionaries by selecting a portion of the dictionary based on keys.
Example of Slicing a Dictionary:

```
# Example dictionary
my_dict = {'name': 'Alice', 'age': 25', 'city': 'New York', 'job': 'Engineer'}

# Function to slice a dictionary based on keys
def slice_dict(my_dict, keys):
    return {key: my_dict[key] for key in keys if key in my_dict}

# Select a portion of the dictionary
sliced_dict = slice_dict(my_dict, ['name', 'job'])

print(sliced_dict)  # Output: {'name': 'Alice', 'job': 'Engineer'}
```

Explanation:

- You can't directly slice a dictionary, but you can extract a subset of key-value pairs based on specific keys.
- In the example above, slice_dict extracts key-value pairs for 'name' and 'job'.

---

b) Data Visualization

Data visualization is the graphical representation of data and information. It helps in making data easier to understand by presenting it in the form of charts, graphs, and maps. Python has several libraries for data visualization, such as **Matplotlib**, **Seaborn**, and **Plotly**.

Importance of Data Visualization:

- It helps in identifying patterns, trends, and outliers in data.
- It makes complex data more accessible and easier to understand.
- It aids in decision-making by providing insights at a glance.

Example with Matplotlib:

```
import matplotlib.pyplot as plt

# Example data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Creating a simple line plot
plt.plot(x, y)
plt.title('Simple Line Plot')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.show()
```

Explanation:

- **Matplotlib** is used to create simple line plots, bar charts, pie charts, etc.
- Other libraries like **Seaborn** provide higher-level interfaces for creating attractive and informative statistical graphics.

---

c) Custom Exception

In Python, a **custom exception** is a user-defined exception that extends the base Exception class. Custom exceptions are useful when you need to raise errors specific to your application logic, providing more meaningful error messages or handling specific conditions that built-in exceptions don't cover.

How to Create a Custom Exception:

1. Create a new class that inherits from Exception or one of its subclasses.
2. Use raise to trigger this custom exception when necessary.

Example of Custom Exception:

Explain

```
# Custom exception class
class AgeTooYoungError(Exception):
    def __init__(self, message="Age is below the minimum allowed"):
        self.message = message
        super().__init__(self.message)

# Function that raises custom exception
def check_age(age):
    if age < 18:
        raise AgeTooYoungError("You must be at least 18 years old")
    else:
        return "Access granted"

# Test the custom exception
try:
    print(check_age(16))
except AgeTooYoungError as e:
    print(e)
```

Explanation:

- The AgeTooYoungError class inherits from Exception, and a custom message is passed during the exception.
- If the age is below 18, the custom exception is raised with a specific error message.

Advantages:

- Custom exceptions provide more precise error handling for specific application conditions.
- They allow you to differentiate between built-in exceptions and errors unique to your application logic.