Python Interview Questions for Freshers

1)What is Python? List some popular applications of Python in the world of technology.

Python is a widely-used general-purpose, high-level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.
It is used for:

- Scripting
- Web Development
- Game Development
- Software Development
- Data Analysis and Data Science

2) What are the benefits of using Python language as a tool in the present scenario

The following are the benefits of using Python language:

- Object-Oriented Language
- High-Level Language
- Dynamically Typed language
- Extensive support Libraries
- Presence of third-party modules
- Open source and community development
- Portable and Interactive
- Portable across Operating systems

3) Is Python a compiled language or an interpreted language?

Actually, Python is a both compiled language and interpreted language. The compilation part is done first when we execute our code and this will generate byte code internally this byte code gets converted by the Python virtual machine(p.v.m) according to the underlying platform(machine+operating system).

4) What is the difference between a Mutable datatype and an Immutable data type?

Mutable data types can be modified i.e., they can change at runtime. Eg – List, Dictionary, etc.
Immutable data types can not be modified i.e., they can not change at runtime. Eg – String, Tuple, etc.

5)What is the difference between and List and Tuples

Lists and tuples are both data structures in Python, but they have some key differences:

1. Mutability:
   - Lists are mutable, meaning you can change their elements after they have been created. You can add, remove, or modify elements of a list.
   - Tuples, on the other hand, are immutable. Once a tuple is created, you cannot change its elements. You can't add, remove, or modify elements of a tuple.

2. Syntax:
   - Lists are created using square brackets `[]`, and elements are separated by commas.
   - Tuples are created using parentheses `()`, and elements are separated by commas.

Example:
my_list = [1, 2, 3, 4, 5]
my_tuple = (1, 2, 3, 4, 5)

3. Performance:
   - Tuples are generally faster than lists because they are immutable. Once a tuple is created, its elements cannot be changed, which allows for some optimizations by the interpreter.

4. Usage:
   - Lists are used when you have a collection of items that may need to be modified, sorted, or manipulated in some way.
   - Tuples are used when you want to store a collection of items that should not be changed, such as coordinates, database records, or configuration settings.

In summary, if you need a collection of items that will remain constant throughout the life of your program, and you want to optimize for performance, use a tuple. If you need a collection of items that may change over time, and you need the flexibility to modify, add, or remove elements, use a list.

5)What is List Comprehension? Explain why it is needed ?

List comprehension is a concise way of creating lists in Python. It allows you to generate lists based on existing iterables (like lists, tuples, strings, etc.) with a single line of code. List comprehensions are not strictly necessary, but they offer several advantages:

1. Conciseness: List comprehensions often allow you to create lists more concisely and with fewer lines of code compared to traditional methods like using loops.

2. Readability: List comprehensions can make your code more readable and expressive, especially for simple transformations or filtering operations. They convey the intent of the code more clearly.

3. Performance: In many cases, list comprehensions can be more efficient than traditional looping constructs. They are optimized by the interpreter and can sometimes run faster than equivalent loop-based code.

Here's a simple example to illustrate the syntax and usage of list comprehension:

```
# Traditional approach using a loop
squares = []
for x in range(10):
    squares.append(x ** 2)

# Using list comprehension
squares = [x ** 2 for x in range(10)]
```

Both approaches generate the same list of squares from 0 to 9. However, the list comprehension version accomplishes this in a single line, making the code more concise and readable.

In summary, list comprehension is a powerful and expressive feature of Python that can simplify your code, improve readability, and in some cases, enhance performance.

6)What is a Set and Dictionary ? Explain when to use set and Dictionary?

Sets and dictionaries are both collection data types in Python, but they serve different purposes and have different characteristics:

1. Set:
   - A set is an unordered collection of unique elements.
   - Sets are defined by enclosing elements within curly braces `{}`.
   - Sets do not allow duplicate elements. If you try to add a duplicate element, it will not be added to the set.
   - Sets are mutable, meaning you can add or remove elements after the set has been created.
   - Sets are often used for tasks such as removing duplicates from a list, performing mathematical set operations like union, intersection, difference, etc.

Example:

```
my_set = {1, 2, 3, 4, 5}
```

2. Dictionary:
   - A dictionary is an unordered collection of key-value pairs.
   - Dictionaries are defined by enclosing key-value pairs within curly braces `{}`. Each key-value pair is separated by a colon `:`.
   - Keys within a dictionary must be unique, but the values can be duplicated.
   - Dictionaries are mutable, meaning you can add, remove, or modify key-value pairs after the dictionary has been created.
   - Dictionaries are often used when you want to store data in a way that allows for fast retrieval based on some unique identifier (the key).

Example:
```
my_dict = {'name': ‘Ramesh’, 'age': 30, 'city': 'Amalapuram’}
```

When to use which:
- Use a set:
  - When you need to store a collection of unique elements and the order of elements doesn't matter.
  - When you need to perform set operations like union, intersection, difference, etc.
  - When you need to remove duplicates from a list or another iterable.

- Use a dictionary:
  - When you need to store data in key-value pairs and want to be able to retrieve the values quickly based on the keys.
  - When you need to associate each element in a collection with a unique identifier.
  - When you need to store data in an unordered but labeled format

7)What is a Function? Explain Types of Arguments in Python

In Python, a function is a block of reusable code that performs a specific task. Functions allow you to break down your program into smaller, modular pieces, making your code more organized, readable, and easier to maintain. Functions can take inputs (arguments), perform operations, and optionally return a result.

Here's a simple example of a function:
```
def greet(name):
    """This function greets the user."""
    print("Hello, " + name + "!")
```

In this example:
- `def` keyword is used to define a function.
- `greet` is the name of the function.

- `(name)` is the parameter or argument that the function accepts.
- The indented block of code under the function definition is the body of the function.
- The `print()` statement inside the function is the operation it performs.

To call this function, you simply write its name followed by parentheses and provide the required arguments:

```
greet("Alice")
```

This would output:

```
Hello, Alice!
```

Types of Arguments in Functions:

1. Positional Arguments:
   - These are the most common type of arguments in Python functions.
   - They are passed to the function in the order they are defined.
   - Positional arguments are mandatory, meaning you must provide them in the correct order when calling the function.

   Example:

```
   def add(a, b):
       return a + b

   result = add(3, 5)  # a=3, b=5
```

2. Keyword Arguments:
   - These are arguments that are preceded by the parameter name and an equals sign (`=`) when calling the function.
   - They allow you to specify arguments out of order or to provide default values for parameters.

   Example:

```
   def greet(name, greeting="Hello"):
       print(greeting + ", " + name + "!")

   greet("Ramesh")                  # Uses default greeting ("Hello")
   greet("Varma", greeting="Hi")      # Uses "Hi" as greeting
```

3. Default Arguments:
   - These are parameters that have default values specified in the function definition.

- If the caller does not provide a value for these parameters, the default value is used.

Example:

```
def power(x, n=2):
    return x ** n

result1 = power(3)    # n defaults to 2
result2 = power(3, 3)  # n=3
```

4. Variable-Length Arguments
   - These allow functions to accept a variable number of arguments.
   - There are two types: *args (non-keyword variable-length arguments) and **kwargs (keyword variable-length arguments).

Example:

```
def add(*args):
    total = 0
    for num in args:
        total += num
    return total

result = add(1, 2, 3, 4, 5)  # result is 15
```

Example with **kwargs:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(key + ": " + value)

print_info(name="Alice", age="30", city="New York")
```

8)What is Lambda Function and explain why its needed

A lambda function in Python is a small anonymous function defined using the `lambda` keyword. It's a way to create functions on the fly without needing to formally define them using the `def` keyword. Lambda functions can take any number of arguments, but they can only have one expression.

The syntax of a lambda function is:

```
lambda arguments: expression
```

Here's an example of a simple lambda function that squares its input:

```
square = lambda x: x ** 2
```

You can then use this lambda function like any other function:

```
result = square(4)  # Result will be 16
```

Lambda functions are often used in scenarios where you need a small function for a short period of time, especially when you need to pass a function as an argument to another function, like `map()`, `filter()`, or `sorted()`. They are particularly useful for writing short, concise code and are commonly used in functional programming paradigms.

For example, you could use a lambda function with `map()` to apply the same operation to every element of an iterable:

```
numbers = [1, 2, 3, 4, 5]

squared_numbers = list(map(lambda x: x ** 2, numbers))

# squared_numbers will be [1, 4, 9, 16, 25]
```

Lambda functions provide a way to create short, anonymous functions without having to define a formal function using `def`. While they are not always necessary, they can often make your code more concise and readable, especially in situations where you need a simple function for a specific task.

9)What is pass

Pass means performing no operation or in other words, it is a placeholder in the compound statement, where there should be a blank left and nothing has to be written there.

10) what is a string explain some string methods

A string in Python is a sequence of characters, enclosed within either single quotes (') or double quotes ("). Strings are immutable, meaning they cannot be changed after they are created. However, you can perform various operations on strings, such as concatenation, slicing, and formatting

Here are some commonly used functions

- len()
- lower() upper()
- strip()
- split()
- join()
- find()

- replace()

(Refer notes for these methods)

11)when to use while loop and when to use for loop in python

The “for” Loop is generally used to iterate through the elements of various collection types such as List, Tuple, Set, and Dictionary. Developers use a “for” loop where they have both the conditions start and the end. Whereas, the “while” loop is the actual looping feature that is used in any other programming language. When we have to deal with conditions we have to go for while loop

12)What is Indentation in Python

Indentation refers to the use of whitespace at the beginning of a line to define the scope of code blocks. Unlike many other programming languages that use curly braces {} or keywords like begin and end to delineate blocks of code, Python uses indentation to indicate where blocks of code begin and end.

The Python interpreter uses indentation to understand the structure of the code and determine which statements belong to which code blocks, such as loops, conditional statements, function definitions, and so on.

Example:

```
if True:
    print("This line is indented, so it belongs to the if statement.")
    print("So is this one.")
print("This line is not indented, so it is not part of the if statement.")
```

13)What is a break, continue, and pass in Python?

The break statement is used to terminate the loop or statement in which it is present. After that, the control will pass to the statements that are present after the break statement, if available.

Continue is also a loop control statement just like the break statement. continue statement is opposite to that of the break statement, instead of terminating the loop, it forces to execute the next iteration of the loop.

Pass means performing no operation or in other words, it is a placeholder in the compound statement, where there should be a blank left and nothing has to be written there.

14) what is map filter and reduce ?

`map()`, `filter()`, and `reduce()` are three built-in functions in Python that are commonly used for functional programming tasks. They provide concise ways to perform operations on iterables such as lists, tuples, and sets. Here's a brief explanation of each and why they are useful:

1. map():

 - The `map()` function applies a given function to each item of an iterable (like a list) and returns a new iterator with the results.

 - It takes two arguments: the function to apply and the iterable to apply it to.

 - `map()` is useful when you need to transform every item in a collection using the same function.

 Example:

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x ** 2, numbers)
# squared is now an iterator containing [1, 4, 9, 16, 25]
```

2. filter():

 - The `filter()` function filters elements from an iterable based on a given function (predicate) and returns a new iterator with the elements that satisfy the condition.

 - It takes two arguments: the function that returns `True` or `False` and the iterable to filter.

 - `filter()` is useful when you need to select elements from a collection that meet certain criteria.

 Example:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
# even_numbers is now an iterator containing [2, 4]
```

3. reduce():

 - The `reduce()` function applies a rolling computation to sequential pairs of elements in an iterable, reducing them to a single value.

 - It is part of the `functools` module in Python 3 and needs to be imported.

- `reduce()` is useful when you need to perform some operation, such as addition or multiplication, on all elements of a collection to produce a single result.

```
Example:
from functools import reduce
numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
# sum_of_numbers is now 15 (1 + 2 + 3 + 4 + 5)
```

15 What is class and object in python

In Python, a class is a blueprint for creating objects (instances). It defines the attributes (data) and methods (functions) that all objects of that class will have. Classes provide a way to encapsulate data and behavior into a single unit, promoting modularity and code reusability.

Here's a basic example of a class in Python:

```
class Dog:

    def __init__(self, name, age):

        self.name = name

        self.age = age

    def bark(self):

        print(f"{self.name} says: Woof!")

# Creating an instance of the Dog class

my_dog = Dog("Buddy", 3)
```

In this example:

- `Dog` is a class that represents a dog.

- The `__init__` method is a special method called a constructor. It is called when a new instance of the class is created and is used to initialize the object's attributes.

- `name` and `age` are attributes of the `Dog` class.

- `bark` is a method of the `Dog` class that defines the behavior of the object.

An object, also known as an instance, is a specific instance of a class. It is created using the class as a blueprint. Each object has its own set of attributes and methods defined by the class.

In the example above, `my_dog` is an object of the `Dog` class. It has attributes `name` and `age`, and a method `bark`. You can access the attributes and call the methods of an object using dot notation:

```
print(my_dog.name)  # Output: Buddy
print(my_dog.age)   # Output: 3
my_dog.bark()       # Output: Buddy says: Woof!
```

16 What is constructor in python

In Python, a constructor is a special method that is automatically called when a new instance (object) of a class is created. It is used to initialize the object's attributes or perform any other setup necessary for the object to function properly. The constructor method in Python is denoted by `__init__()`.

Here's a basic example of a class with a constructor in Python:

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Creating an instance of the Person class
person1 = Person("Alice", 30)
person1.greet()  # Output: Hello, my name is Alice and I am 30 years old.
```

In this example:
- `__init__()` is the constructor method of the `Person` class.
- When a new instance of the `Person` class is created (`person1`), the `__init__()` method is automatically called with the arguments provided during object creation (`"Alice"` and `30` in this case).
- Inside the `__init__()` method, the `name` and `age` attributes of the object (`self`) are initialized with the values passed to the constructor.
- The `self` parameter represents the current instance of the class and is used to access the object's attributes and methods within the class.

17 What is Inheritance and Types of Inheritance supported by Python

Inheritance is a key concept in object-oriented programming (OOP) that allows a class (known as a subclass or derived class) to inherit attributes and methods from another class (known as a superclass or base class). In Python, inheritance enables you to create new classes that are built upon existing classes, thus promoting code reuse and modular design.

When a class inherits from another class, it automatically inherits all attributes and methods of the superclass. The subclass can then add its own attributes and methods, or override existing ones, to tailor its behavior as needed.

Python supports several types of inheritance:

1. Single Inheritance:

   - In single inheritance, a subclass inherits from only one superclass.
   - This is the most common type of inheritance.
   - Syntax:

```
class Subclass(Superclass):
    # Subclass definition
```

2. Multiple Inheritance:

   - In multiple inheritance, a subclass inherits from multiple superclasses.
   - This allows a subclass to combine features from multiple parent classes.
   - Syntax:

```
class Subclass(Superclass1, Superclass2, ...):
    # Subclass definition
```

3. Multilevel Inheritance:

   - In multilevel inheritance, a subclass inherits from a superclass, and another subclass inherits from the first subclass, creating a chain of inheritance.
   - Syntax:

```
class Superclass:
    # Superclass definition

class Subclass(Superclass):
    # Subclass definition

class SubSubclass(Subclass):
    # SubSubclass definition
```

4. Hierarchical Inheritance:

- In hierarchical inheritance, multiple subclasses inherit from the same superclass.
- Syntax:

```
class Superclass:
    # Superclass definition

class Subclass1(Superclass):
    # Subclass1 definition

class Subclass2(Superclass):
    # Subclass2 definition
```

5. Hybrid Inheritance:

- Hybrid inheritance is a combination of multiple inheritance and any other type of inheritance.
- It allows for a mix of inheritance patterns to meet specific design requirements.

Inheritance in Python provides a powerful mechanism for code reuse and abstraction. It allows you to create hierarchies of classes with specialized behavior while maintaining a clear and organized structure. However, it's important to use inheritance judiciously and design class hierarchies carefully to avoid excessive coupling and maintainability issues.

18 What is Abstraction in python

Abstraction is a fundamental concept in object-oriented programming (OOP) that involves hiding the complex implementation details of a class and only showing the essential features or functionalities to the user. It allows you to focus on what an object does rather than how it does it. Abstraction helps in simplifying the complexity of a system by providing a clear and concise interface for interacting with objects.

The main purpose of abstraction is to reduce complexity and increase the efficiency of software development by:

Providing a high-level view of functionality, hiding unnecessary details.

Encapsulating the implementation details within the class, making it easier to understand and maintain.

Promoting code reusability by allowing objects to be used in different contexts without exposing their internal workings.

19 What is Encapsulation in Python

Encapsulation is one of the fundamental principles of object-oriented programming (OOP) that involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit, known as a class. Encapsulation allows you to hide the internal state of objects from the outside world and only expose the necessary functionality through well-defined interfaces.

In Python, encapsulation is achieved through the use of classes and access control mechanisms, such as public, protected, and private attributes and methods.

```
class Person:
    def __init__(self, name, age):
        self._name = name  # Protected attribute
        self.__age = age   # Private attribute

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age

# Usage
person = Person("Alice", 30)
print(person._name)  # Accessing protected attribute
print(person.get_age())  # Accessing private attribute using getter method
person.set_age(35)  # Modifying private attribute using setter method
print(person.get_age())  # Output: 35
```

20 What is Polymorphism and explain what is method overriding

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying forms or types.

There are two main types of polymorphism:

1. Compile-time Polymorphism (Static Binding)**:

   - Also known as method overloading.

- It occurs when multiple methods in the same class have the same name but different parameters.

- The appropriate method to call is determined by the number and types of arguments passed to it at compile-time.

- This type of polymorphism is resolved at compile-time, hence the name static binding.

2. Runtime Polymorphism (Dynamic Binding):

- Also known as method overriding.
- It occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
- The method to call is determined by the actual object type at runtime.
- This type of polymorphism is resolved at runtime, hence the name dynamic binding.

Method overriding is a specific form of polymorphism that occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The signature (name and parameters) of the method in the subclass must match the signature of the method in the superclass.

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

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

class Cat(Animal):
    def speak(self):
        print("Cat meows")

# Usage
animal = Animal()
animal.speak()  # Output: Animal speaks

dog = Dog()
dog.speak()  # Output: Dog barks

cat = Cat()
cat.speak()  # Output: Cat meows
```

21 what is exception handling in python

Exception handling in Python is a mechanism that allows you to gracefully handle errors or exceptional situations that occur during the execution of a program. By handling exceptions, you can prevent your program from crashing and provide alternative paths or error messages to deal with unexpected conditions.

In Python, exception handling is achieved using `try`, `except`, `else`, and `finally` blocks:

1. try block:

   - The `try` block is used to enclose the code that may raise an exception.
   - If an exception occurs within the `try` block, the execution of the block is interrupted, and control is transferred to the nearest `except` block.
   - If no exception occurs, the `except` block is skipped, and the execution continues with the next block.

2. except block:

   - The `except` block is used to handle specific exceptions that occur within the `try` block.
   - You can specify the type of exception you want to catch, or you can use a generic `except` block to catch any exception.
   - Multiple `except` blocks can be used to handle different types of exceptions.

3. else` block:

   - The `else` block is optional and is executed if no exception occurs in the `try` block.

   - It is typically used to contain code that should only run if the `try` block completes successfully.

4. finally` block:

   - The `finally` block is optional and is always executed, regardless of whether an exception occurs in the `try` block or not.

   - It is typically used to contain cleanup code that should be executed regardless of whether an exception occurs.

Here's a basic example of exception handling in Python:

```
try:

    num1 = int(input("Enter a number: "))

    num2 = int(input("Enter another number: "))

    result = num1 / num2

except ZeroDivisionError:
    print("Error: Division by zero")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
else:
    print("Result:", result)
finally:
    print("This block is always executed, regardless of whether an exception occurs.")
```

In this example:

- The `try` block attempts to divide two numbers input by the user.
- If a `ZeroDivisionError` occurs (division by zero), it is caught by the first `except` block, and an error message is printed.
- If a `ValueError` occurs (invalid input), it is caught by the second `except` block, and an error message is printed.
- If no exception occurs, the `else` block is executed, and the result is printed.
- Regardless of whether an exception occurs, the `finally` block is executed, printing a message indicating that it is always executed.

Exception handling in Python allows you to write robust and reliable code by handling errors gracefully and providing appropriate responses to unexpected situations.

Interview Programs List (Refer this document)

https://docs.google.com/document/d/1hP4lPqygQS2iqX-fFuqyPSFntfvDOU7FgVjqOX5petM/edit?usp=sharing