– End Sem Question Bank and Solution –

Unit 3

Q1. What is a function? Explain code reuse.

a function is a named block of code that performs a specific task. It is a fundamental concept in most programming languages and is used to organize code into manageable and reusable chunks. Functions help make code modular, improve readability, and promote code reuse.

Code reuse refers to the practice of using existing code in multiple parts of a program or in different programs altogether. Functions enable code reuse by providing functionality that can be called whenever needed. Instead of rewriting the same code in different places, you can define a function once and reuse it in multiple locations.

Code reuse has several advantages:

Simplicity: Reusing code reduces the overall size and complexity of your program. You can focus on writing reusable functions that solve specific problems, rather than duplicating code. This simplifies the codebase and makes it easier to understand, debug, and maintain.

Efficiency: By reusing code, you avoid unnecessary duplication, resulting in smaller file sizes and improved performance.

Consistency: If you find a bug or need to make an improvement, you only need to update the function code only, and all the calling locations will benefit from the change.

Productivity: Code reuse saves time and effort. Once you have a library of reusable functions, you can build new programs more quickly by leveraging the existing functionality.

Overall, functions and code reuse are essential concepts in programming that enable efficient, modular, and maintainable code development.

Q2. Explain with example Docstring

1. A docstring, short for "documentation string," is used to describe document functions, modules, classes, or methods in Python. It provides a way to describe what the code does, its inputs, outputs, and any other relevant information

2. By including this docstring, other developers or users of the code can easily understand how to use the function, what it expects as input, and what it returns

3. The docstring is placed immediately after the function definition, enclosed within triple quotes (""").

Example:

```
def calculate_square(number):
    """
    Calculates the square of a given number.
    """
    square = number ** 2
    return square
```

Q3. Explain Lambda function in python with an example.

A lambda function, also known as an anonymous function, is a small, single-line function that does not require a def statement or a name. It is defined using the lambda keyword, followed by a list of arguments, a colon (:), and an expression. The result of the expression is the return value of the function.
Lambda functions are commonly used when you need a simple, one-time function without defining a full-fledged function using the def keyword.

Syntax:

lambda arguments: expression

Example 1: Square function using lambda

```
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

Q4. Explain the use of return statements in python.

In Python, the return statement is used to end the execution of a function and return a value (or multiple values) back to the caller. It is commonly used to provide the result of a computation or to pass data from a function back to the code that called it.

When a return statement is encountered in a function, the following actions occur:

1. The function execution is immediately terminated.
2. The value of the expression is evaluated and becomes the return value of the function.
3. The control is passed back to the line of code that called the function.

Example:

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

result = add_numbers(2, 3)
print(result)  # Output: 5
```

Q5. What are variable length arguments in Python?

In some situations, it is not known in advance how many arguments will be passed to a function. In such cases, Python allows programmers to make function calls with arbitrary (or any) number of arguments.

Positional variable length arguments are defined using an asterisk (*) before the parameter name in the function definition. This allows the function to accept any number of positional arguments, which are then collected into a tuple.

Example:

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

result = sum_numbers(1, 2, 3, 4, 5)
print(result)  # Output: 15
```

Q6. What are modules? How do you use them in your program?

A module is a file that contains Python code, including definitions, statements, and functions, that can be imported and used in other Python programs. Modules are used to organize and reuse code, making it easier to manage large programs and collaborate with other developers.

A module can contain various elements, such as variables, functions, classes, and even other modules. It provides a way to encapsulate related code into a single file, allowing you to logically group and organize your codebase.

To use a module in your program, you need to follow these steps:

Importing the module: To use a module, you first need to import it into your program. You can import an entire module or specific elements from it.

Example:

```
# Importing the entire module
import module_name

# Importing specific elements from a module
from module_name import element1, element2

# Importing a module with an alias
import module_name as alias
```

Using the module: Once a module is imported, you can use its elements in your program. You can access the elements using the module name followed by a dot (.) and the element name.

Example:

```
# Using elements from the module
module_name.element1
module_name.element2()
```

Q7. Differentiate between Local & Global variables.

Local andGlobal variables are two types of variables that differ in their scope and visibility within a program.

Local Variables:

1. Local variables are defined and used within a specific block or scope, typically within a function.
2. They are created when a function is called and destroyed when the function completes its execution or returns.
3. Local variables are not accessible outside the block or function where they are defined.
4. They have a limited lifespan and are not visible to other parts of the program.
5. Local variables can have the same name as global variables, but they are independent and separate entities.

Example:

```
def my_function():
  x = 10  # Local variable
  print(x)

my_function()
```

Global Variables:

1. Global variables are defined outside any function or block and have a global scope.
2. They can be accessed and modified by any part of the program, including functions.
3. Global variables are visible throughout the entire program, enabling them to be used in multiple functions or blocks.
4. They retain their value until explicitly changed or the program terminates.
5. If a local variable shares the same name as a global variable, the local variable takes precedence within its scope.

Q8. What do you understand by the term argument? How do we pass them to a function ?

1. An argument refers to the value or expression that is passed to a function when it is called.
2. Arguments provide a way to supply input to a function and allow the function to perform operations

or calculations based on that input.

In Python, we can pass arguments to a function by specifying them within the parentheses of the function call. The arguments can be of different types, such as numbers, strings, variables, or even other function calls. we can pass arguments in two ways: positional or required arguments, default argument, keyword arguments and variable length argument

Q9. What are types of arguments?

There are several types of arguments that can be used when defining and calling functions. These argument types allow for flexibility in how arguments are passed and used within functions. The main types of arguments are:

1. Positional Arguments:
   Positional arguments are the most common type of argument. They are passed to a function based on their position or order in the function call. The values of positional arguments are assigned to corresponding parameters in the function definition based on their order.
   Example:

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

result = add_numbers(2, 3)
```

2. Keyword Arguments:
   Keyword arguments are passed to a function using the parameter names during the function call. This allows you to specify the name of the parameter you want to assign a value to, regardless of the order of the parameters in the function definition.
   Example:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(age=25, name="Alice")
```

3. Default Arguments:
   Default arguments have predefined values assigned to the function parameters. If a value is not provided for a default argument during the function call, the default value is used. Default arguments are defined using the assignment operator (=) within the function definition.
   Example:

```
def greet(name, age=30):
    print(f"Hello, {name}! You are {age} years old.")

greet("Bob")
```

4. Variable Length Arguments:
   Variable length arguments allow a function to accept an arbitrary number of arguments. They are defined using the *args syntax for positional variable length arguments.
   Example:

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

result = add_numbers(1, 2, 3, 4, 5)
```

Unit 4

Q1.What is slice operation? Explain with examples.

or

Explain indexing and slicing operations on string with suitable examples.

1. The slice operation is a powerful feature in Python that allows you to extract subsets of sequences efficiently.
2. It is commonly used in tasks such as extracting substrings, sublists, or creating reversed copies of sequences.
3. The slice operation is denoted using the square brackets [] with start, stop, and step values specified within the brackets

   start is the index at which the slice starts.
   stop is the index at which the slice ends.
   step is an optional value that specifies the increment between elements.

Example:

```
# Slice a string
string = "Hello, World!"
slice1 = string[7:12]  # "World"

# Slice with negative indices
sequence = "Python"
slice3 = sequence[-3:-1]  # "ho"
```

Q2. Explain the following string operations with suitable examples.

1. Concatenation
2. Appending
3. String repetition

1. <u>Concatenation:</u>

   - Concatenation refers to combining two or more strings or sequences into a single string or sequence.
   - In Python, the concatenation operation for strings is performed using the + operator.
   - When you concatenate two strings, the resulting string contains the characters from both strings in the specified order.
     Example:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Output: "Hello World"
```

2. <u>Appending:</u>
   - Appending refers to adding an element to the end of a sequence, such as a list.
   - In Python, the addition and assignment operation can be combined using the += operator.
     Example:

```
message = "Hello"
message += ", World!"  # Concatenate ", World!" to the existing string
```

Write short note on format operator in python

The format operator in Python is represented by the % symbol, and is used to format strings by substituting values into a string in a specified format. The format operator is used in conjunction with the % character, which is followed by a tuple or a dictionary of values to be substituted into the string.

```
name = "Alice"
age = 25
print("My name is %s and I am %d years old." % (name, age))
```

Here is a list of format operators used in Python:

%s - string
%d - decimal integer
%f - floating point decimal
%e - exponential notation
%x - hexadecimal integer
%o - octal integer
%c - character

Python strings are immutable.

In Python, strings are immutable objects, which mean that once a string is created, its contents cannot be changed.

If you try to modify a string in place by directly changing one of its characters, you will receive a TypeError. For example, the following code will raise an error:

```
my_string = "hello"
my_string[0] = "H"  # raises TypeError: 'str' object does not support item assignment
```

Because strings are immutable, it can be more memory-efficient to use them in certain contexts, since Python can optimize certain operations by reusing existing strings. However, in cases where you need to modify a string frequently, it may be more appropriate to use a mutable data structure like a list, and then convert it to a string when necessary.

Difference between method and a function

Function

1. A function is a self-contained block of code that performs a specific task.
2. It takes input arguments (if any) and returns output (if any), but does not modify any state outside of its own scope.
3. Functions in Python can be defined using the def keyword, and can be called from anywhere in the program.

Method

1. A method, on the other hand, is a function that is associated with an object.
2. It is defined within a class and can be called on instances of that class. Methods are invoked using the dot notation, in the form object.method(arguments).
3. Methods typically operate on the data contained within the object, and may modify its state.

Explain following term

1. rindex()

The rindex() method in Python is used to search for the last occurrence of a specified substring within a string. It returns the index of the first character of the last occurrence of the substring if it is found, and raises a ValueError if the substring is not found in the string.

Syntax:

```
string.rindex(substring, start, end)
```

2. strip()

strip() is a built-in string method that is used to remove leading and trailing whitespace characters (spaces, tabs, and newlines) from a string. It returns a new string with the whitespace characters removed.

Syntax:

string.strip(characters)

3. zfill()

zfill() is a built-in string method that is used to pad a string with leading zeros (0s) to a specified length. The method returns a new string with the specified number of zeros added to the beginning of the original string.

Syntax
string.zfill(width)

4. format()

format() is a built-in string method that is used to format a string. It allows you to insert values into a string in a flexible and dynamic way, rather than having to manually concatenate strings and variables.

Syntax
string.format(value1, value2, ...)

5. find()

find() is a built-in string method that is used to search for a substring within a string. It returns the index of the first occurrence of the substring in the string, or -1 if the substring is not found.

Syntax:
string.find(substring, start, end)

6. index()

index() is a built-in string method that is used to search for a substring within a string. It returns the index of the first occurrence of the substring in the string, or raises a ValueError if the substring is not found.

Syntax
string.index(substring, start, end)

7. split()

split() is a built-in string method that is used to split a string into a list of substrings, using a specified delimiter as the separator. The delimiter can be any character or sequence of characters, and if it is not specified, the default delimiter is a space.

Syntax:
string.split(delimiter, maxsplit)

8. join()

join() is a built-in method that is used to concatenate a list of strings into a single string, using a specified separator between each pair of strings. The join() method is called on the separator string, and the list of strings is passed as an argument.

Syntax:
separator.join(list)

9. enumerate()

enumerate() is a built-in function that is used to iterate over a sequence (such as a list or a tuple) and return a sequence of pairs, each containing an index (starting from zero) and the corresponding value from the original sequence. The enumerate() function is particularly useful when you need to iterate over a sequence and also track the index of each item.

Syntax:
enumerate(sequence, start=0)

10.startswith()

startswith() is a built-in string method in Python that returns a Boolean value indicating whether a string starts with a specified prefix.

Syntax:
string.startswith(prefix, start, end)

Unit 5

Define different programming paradigms.

A programming paradigm is a way of thinking about and approaching the task of programming. It is a set of principles, concepts, and techniques used to design, develop, and maintain software.

There are several programming paradigms, including:

1. Monolithic programming is a traditional software development approach in which an entire software application is designed and developed as a single, self-contained unit. This means that all the components of the application, such as the user interface, business logic, and data access layer, are tightly coupled and are part of a single executable or deployable package.

2. Procedural programming is a programming paradigm that is based on the concept of procedures, also known as functions or subroutines. In procedural programming, a program is designed as a series of procedures, each of which performs a specific task or calculation. These procedures are designed to operate on data that is stored in variables, which can be modified during the execution of the program.

3. Structured programming also emphasizes the use of subroutines or functions, which enable code reuse and modularity. By breaking the program down into smaller, reusable functions, the overall program becomes easier to maintain, as changes can be made to individual functions without affecting the entire program.

4. Object-oriented programming (OOP) is a programming paradigm that is based on the concept of objects, which are instances of classes that encapsulate data and behavior. The main idea behind OOP is to organize the code into reusable and modular structures, which makes the code easier to understand, maintain, and extend.

What is a class and object?

In object-oriented programming, a class is a blueprint or template for creating objects that have common attributes and behaviors. It defines a set of attributes and methods that objects of that class will possess.

An object is an instance of a class. It is created from the class blueprint and has its own unique set of attributes and methods. You can think of a class as a blueprint for a house, and an object as a specific house built from that blueprint.

Example:

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

person1 = Person("Alice", 25) # example of object
```

In this example, **Person** is the class and **person1** is the object created from that class. **Person** defines two attributes (**name** and **age**) and a constructor method (**__init__**) that initializes those attributes. **person1** is created with the name "Alice" and age 25, using the constructor method of the **Person** class.

Differentiate between class variable and instance variable

A class variable is a variable that is associated with the class as a whole and not with any particular

instance of the class. It is declared at the class level, outside of any method, and is shared by all instances of the class. Class variables are usually used to define values that are common to all objects of the class. They can be accessed using the class name or any object of the class.

For example:

```
class Car:
        wheels = 4  # class variable
my_car = Car()
print(Car.wheels)  # output: 4
print(my_car.wheels)  # output: 4
```

Instance variable is a variable that is associated with a particular instance of the class. It is declared inside the constructor method of the class and is unique to each instance of the class. Instance variables are usually used to define values that are specific to an object of the class. They can only be accessed using an object of the class.

For example:

```
class Car:
        def __init__(self, make, model):
        self.make = make  # instance variable
        self.model = model  # instance variable

my_car = Car("Toyota", "Corolla")
print(my_car.make)  # output: "Toyota"
print(my_car.model)  # output: "Corolla"
```

Features of Object-oriented programming (OOP)

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of "objects", which can contain data and code to manipulate that data. Some of the key features of OOP are:

1. Encapsulation: Encapsulation is the process of hiding the implementation details of an object from the outside world, while providing a simple interface to interact with it. This allows objects to be used without needing to know how they work internally, and makes it easier to maintain and modify the code.
2. Abstraction: Abstraction is the process of focusing on the essential features of an object, while ignoring the irrelevant details. This allows for a simpler and more generalized representation of the object, which makes it easier to understand and work with.
3. Inheritance: Inheritance is the process of creating new classes by deriving them from existing classes. The new class inherits all the attributes and methods of the existing class, and can add new attributes and methods or override existing ones. This allows for code reuse and makes it easier to manage large and complex codebases.
4. Polymorphism: Polymorphism is the ability of objects of different types to be used interchangeably. This allows for more flexible and modular code, since objects can be used in a wider range of situations.
5. Classes and objects: OOP is based on the concept of classes and objects. A class is a blueprint for creating objects, which defines the attributes and methods of the object. An object is an instance of a class, with its own unique set of attributes and methods.
6. Modularity: OOP encourages modular design, where code is divided into smaller, more manageable units. This makes it easier to understand, maintain, and modify the code.
7. Message passing: In OOP, objects communicate with each other by sending messages. A message is a request for an object to perform an action, which may result in a change in the state of the object or the system as a whole.

These features of OOP make it a powerful and flexible programming paradigm, particularly for large and complex projects. OOP allows for code to be organized in a more intuitive and natural way, and provides a wide range of tools and techniques for managing and manipulating data.

What is the self-argument in the class method?

Or

Explain the concept of class method and self-object with suitable examples.

In Python, the **self**-argument in a class method refers to the instance of the class that the method is being called on. It is a convention in Python to use **self** as the name of the first parameter in instance methods.

Class methods must have the first argument named as self. This is the first argument that is added to the beginning of the parameter list.

We do not pass a value for this parameter; Python provides its value automatically. The self-argument refers to the object itself. That is, the object that has called the method.

Since, the class methods uses self, they require an object or instance of the class to be used. For this reason, they are often referred to as *instance methods*.

Here is an example to illustrate this:

```
class Person:
        def __init__(self, name, age):
        self.name = name
        self.age = age
        def say_hello(self):
        print("Hello, my name is", self.name, "and I am", self.age, "years old.")
person1 = Person("Alice", 25)
person1.say_hello()
```

Explain the significance of the __init__() method.

The **__init__()** method is a special method in Python classes that is automatically called when an instance of the class is created. The purpose of the **__init__()** method is to initialize the instance with initial values for its attributes.

Here are a few key points about the significance of the **__init__()** method:

1. Initialization: The **__init__()** method is used to set initial values for the attributes of an instance. When a new instance of a class is created, the **__init__()** method is called automatically, and it sets the initial state of the object.
2. Attribute assignment: Inside the **__init__()** method, you can assign values to the attributes of the instance using the **self**-keyword. This allows you to define the state of the object when it is first created.
3. Object creation: The **__init__()** method is called during the process of creating a new object. It is responsible for creating and initializing the object, and for preparing it for use.
4. Default values: You can define default values for the attributes of the instance in the **__init__()** method. This allows you to provide default behavior for the object, while still allowing it to be customized when needed.
5. Example:

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

```
        self.name = name
        self.age = age

        def say_hello(self):
        print("Hello, my name is", self.name, "and I am", self.age, "years old.")
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)
person1.say_hello()  # Output: "Hello, my name is Alice and I am 25 years old."
person2.say_hello()  # Output: "Hello, my name is Bob and I am 30 years old."
```

Differentiate between Public and Private Variable

In Python, all instance variables are by default public, which means they can be accessed and modified from outside the class. However, you can make an instance variable private by adding a double underscore (__) prefix to its name. This will make the variable name "mangled", meaning that it will be modified by Python to include the name of the class.

Here are some key differences between public and private instance variables:

Public Instance Variables:

- Can be accessed and modified from outside the class.
- Are usually used for attributes that are part of the class's public interface.
- Are not prefixed with double underscores.

Private Instance Variables:

- Cannot be accessed or modified from outside the class.
- Are usually used for attributes that are intended to be used only by the class itself.
- Are prefixed with double underscores.

Here is an example to illustrate the difference between public and private instance variables:

```
class Car:

        def __init__(self, make, model, year):

        self.make = make  # public instance variable

        self.model = model  # public instance variable

        self.year = year  # public instance variable

        self.__mileage = 0  # private instance variable


        def drive(self, distance):

         self.__mileage += distance


        def get_mileage(self):
```

```
        return self.__mileage


car1 = Car("Toyota", "Camry", 2022)

print(car1.make, car1.model, car1.year)  # Output: Toyota Camry 2022

car1.drive(100)

print(car1.get_mileage())  # Output: 100


# Try to access the private variable directly:

# print(car1.__mileage)  # Raises an Attribute Error
```

Explain the concept of classmethod and staticmethod

A class method is a method that operates on the class itself rather than on instances of the class. It can access and modify the class-level data and methods, but it cannot access the instance-level data and methods. Class methods are defined using the **@classmethod** decorator and have a **cls** parameter as their first argument, which refers to the class itself.

Example:

```
class MyClass:
        class_variable = 0

        @classmethod
        def class_method(cls):
        cls.class_variable += 1
```

A static method, on the other hand, is a method that belongs to the class and does not depend on either the instance or the class. It cannot access either the class-level or the instance-level data and methods. Static methods are defined using the **@staticmethod** decorator and do not take any special parameters like **cls** or **self**.

Here's an example of a static method:

```
class MyClass:

        @staticmethod

        def static_method(x, y):

        return x + y
```

What is inheritance and how does inheritance allow users to reuse code?

Inheritance is a fundamental concept in object-oriented programming that allows you to define a new class based on an existing class. The new class, called the "subclass" or "derived class", inherits all the properties and methods of the existing class, called the "superclass" or "base class", and can also add its own properties and methods.

Inheritance allows users to reuse code by creating a new class that is a modified version of an existing

class. Instead of starting from scratch and re-implementing all the features of the base class, the user can simply extend the base class and add or modify the functionality as needed.

Here's an example to illustrate how inheritance allows users to reuse code:

```
class Animal:
        def __init__(self, name):
        self.name = name
        def make_sound(self):
        print("The animal makes a sound")
class Dog(Animal):
        def __init__(self, name):
        super().__init__(name)
        def make_sound(self):
        print("The dog barks")
class Cat(Animal):
   def __init__(self, name):
        super().__init__(name)
        def make_sound(self):
        print("The cat meows")
```

Explain the significance of super () function

super() is a built-in function that provides a way to access the methods and attributes of a superclass from a subclass. The **super()** function is used to call a method or attribute from the parent class that has been overridden in the child class.

The **super()** function takes two arguments: the first argument is the subclass, and the second argument is the instance of the subclass. By calling **super()** with these arguments, you can access the superclass's methods and attributes.

Here's an example to illustrate the use of **super()**:

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

        def speak(self):
        print("An animal is speaking")

class Dog(Animal):
        def __init__(self, name):
        super().__init__(name)

        def speak(self):
        super().speak()
        print("A dog is barking")
```

What are types of inheritance

In object-oriented programming, there are generally four types of inheritance:

1. Single inheritance: A class inherits from a single parent class.
2. Multiple inheritances: A class inherits from multiple parent classes.
3. Multilevel inheritance: A class inherits from a parent class, which in turn inherits from another parent class.
4. Hierarchical inheritance: Multiple classes inherit from a single parent class.

What is polymorphism in python?

Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as if they are of the same class. In Python, polymorphism is achieved through method overriding and method overloading.

Method overriding is when a subclass provides a different implementation of a method that is already defined in its superclass. This allows objects of the subclass to be treated as if they are objects of the superclass, but with their own specific behavior.

Method overloading, on the other hand, is when multiple methods with the same name but different parameters are defined in a class. This allows the same method name to be used for different purposes, and Python will choose the appropriate method to call based on the arguments passed.

Polymorphism can be useful when you have a collection of objects of different classes but want to perform the same operation on all of them. By treating them as if they are of the same class, you can write code that is more flexible and easier to maintain.

Define the term operator overloading

Operator overloading is a feature in object-oriented programming that allows operators, such as +, -, *, /, and %, to be redefined or extended beyond their original use to work with user-defined types. This means that operators can be used to perform operations on objects of a class in a way that is intuitive and consistent with the behavior of the operator for built-in types.

In Python, operator overloading is achieved by defining special methods that correspond to the operators. By overloading operators, you can define the behavior of operators for your own custom types, allowing you to write code that is more concise and readable. For example, if you define a class representing complex numbers, you can overload the + and - operators to perform complex addition and subtraction, respectively, in a natural and intuitive way.

Unit 6

What is a file? Why do we need them?

A file is a collection of data or information that is stored on a computer or other electronic device. Files can contain various types of information, such as text, images, audio, video, programs, or other types of data.

We need files for several reasons:

1. Data storage: Files allow us to store and organize information on our computers or other electronic devices.
2. Data sharing: Files can be shared with others, either through email, cloud storage services, or other means.
3. Backup and recovery: Files can be backed up to protect against data loss and can be recovered in the event of a system failure or other disaster.
4. Program execution: Files containing programs or software are required for running applications on computers or other devices.
5. Documentation: Files can be used to document various types of information, such as project plans, reports, and other types of documents.

Overall, files are essential for the efficient and effective management of data and information in our digital world.

Differentiate between absolute and relative file path

A file path is a reference to a location in a file system that contains a file or directory. There are two types of file paths: absolute and relative.

An absolute file path provides the complete path from the root directory to the file or directory.

A relative file path provides the path relative to the current working directory.

Here are the main differences between absolute and relative file paths:

1. Syntax: An absolute file path always starts from the root directory and includes the complete path to the file or directory. It is represented with a leading forward slash (/) on Unix-like systems, or with a drive letter (e.g., C:) on Windows systems. In contrast, a relative file path is expressed relative to the current working directory and does not include the root directory.
2. Portability: Absolute file paths may not be portable across different systems or operating systems, as the root directory may be different. On the other hand, relative file paths are generally more portable and can be used on different systems.
3. Flexibility: Relative file paths are more flexible because they can be used to refer to files or directories at different locations, without having to know the complete path from the root directory. Absolute file paths are less flexible because they are tied to a specific location in the file system.
4. Convenience: Relative file paths are generally shorter and easier to read and write, especially for files and directories located within the current working directory. Absolute file paths can be more cumbersome to type out and maintain, especially for deep file structures.

Differentiate between text and binary file

Text and binary files are two types of computer files that differ in the way they store and represent data.

A text file is a type of file that stores textual data as a sequence of characters encoded using a character encoding such as ASCII or Unicode. Text files are human-readable and editable using a text editor or word processing software. They are commonly used to store source code, configuration files, and plain text documents such as notes and letters.

A binary file, on the other hand, stores data in a non-textual format using binary code, which consists of a

sequence of 0s and 1s. Binary files are typically not human-readable and cannot be edited using a text editor or word processor. They are used to store executable code, images, audio, video, and other non-textual data.

The key difference between text and binary files is the way they store and represent data. Text files store data in a human-readable form, while binary files store data in a machine-readable form. Text files are typically smaller in size than binary files, but they may require more storage space if they contain a lot of formatting or special characters. Binary files, on the other hand, can be much larger in size than text files, but they are typically more efficient in terms of storage space and processing time.

Explain different access modes for opening files.

- In Python open a file, we can specify the access mode, which determines how the file will be opened and how you can interact with it.
- The access modes are represented by different characters that are passed as the second argument to the open() function.

Here are the different access modes:

1. Read Mode ('r'):
   Syntax: open('filename', 'r')

   Read mode is used to open a file for reading. With this mode, you can read the content of the file but cannot modify or write to it. If the file does not exist, it will raise a FileNotFoundError.

2. Write Mode ('w'):
   Syntax: open('filename', 'w')

   Write mode is used to open a file for writing. If the file already exists, it will be truncated (emptied), and you can write new content to it. If the file doesn't exist, a new empty file with the given name will be created. Be cautious when using this mode, as it will overwrite the existing content of the file.

3. Append Mode ('a'):
   Syntax: open('filename', 'a')

   Append mode is used to open a file for writing, but it doesn't truncate the existing content. Instead, any new data you write will be appended to the end of the file. If the file doesn't exist, a new empty file with the given name will be created.

4. Binary Mode ('b'):
   Syntax: open('filename', 'rb') (for reading) or open('filename', 'wb') (for writing)

   Binary mode is used to read or write binary data, such as images, audio, video, etc. This mode is often used in combination with 'r' or 'w' for reading or writing binary files.

5. Read and Write Mode ('r+'):
   Syntax: open('filename', 'r+')

   Read and write mode is used to open a file for both reading and writing. You can read and write to the file using the same file object.

6. Write and Read Mode ('w+'):
   Syntax: open('filename', 'w+')

   Write and read mode is used to open a file for both writing and reading. It truncates the file if it exists, and you can both read from and write to the file.

7. Append and Read Mode ('a+'):
   Syntax: open('filename', 'a+')

   Append and read mode is used to open a file for both reading and appending. You can read from the file, and any data you write will be appended to the end of the file.

Give the significance of "with" keyword

In Python, the "with" statement is used to ensure that a block of code is executed in a context controlled by a context manager. It provides a way to ensure that resources are properly and automatically managed, regardless of whether the code block completes successfully or raises an exception.

When used with file I/O operations, the "with" statement provides a convenient way to open and close files, without having to worry about explicitly calling the "close()" method on the file object. When the "with" statement is used to open a file, the file is automatically closed when the block of code inside the "with" statement is exited, regardless of whether the block was exited normally or due to an exception.

Here's an example that demonstrates the use of the "with" statement for file I/O:

```
# Open a file using with statement
with open('example.txt', 'r') as f:
        data = f.read()
# File is automatically closed after with block
```

In this code, we use the "with" statement to open a file named "example.txt" in read mode. The file object is assigned to the variable "f". Inside the "with" block, we read the contents of the file using the "read()" method and assign the result to the variable "data". We then perform some operation on the data.

When the block of code inside the "with" statement is completed, the file is automatically closed, even if an exception is raised while the block is executing. This ensures that the file is always closed, and resources are properly managed.

Using the "with" statement is considered a best practice for file I/O operations in Python, as it ensures that resources are properly managed and reduces the risk of resource leaks and other issues.

What are methods for reading and writing files?

Methods for reading files in Python:

1. read(): This method is used to read the entire content of the file as a single string.
2. readline(): This method reads a single line from the file at a time.
3. readlines(): This method reads all the lines of the file and returns them as a list of strings.

Methods for writing files in Python:

1. write(): This method is used to write a string to the file. It overwrites the existing content of the file if it's opened in write mode ('w').
2. writelines(): This method is used to write a list of strings to the file. It does not add line breaks between the strings, so you need to include them explicitly if required.
3. append(): This method is used to open the file in append mode ('a') and add new content at the end of the file without overwriting the existing content..

Write note on Directory method in python

The **os** module in Python provides a way to interact with the underlying operating system. It provides a wide variety of functions for working with files and directories, executing commands, and managing processes.

Here are some commonly used functions in the **os** module:

1. **os.getcwd()**: Returns the current working directory.
2. **os.listdir(path)**: Returns a list of all files and directories in the specified directory.
3. **os.mkdir(path)**: Creates a new directory with the specified path.
4. **os.remove(path)**: Deletes the file at the specified path.
5. **os.chdir()** is a function in the Python **os** module used to change the current working directory to the specified path. The name **chdir** stands for "change directory".

There are many other functions in the **os** module that provide additional functionality for working with the operating system, including functions for working with environment variables, changing permissions on files and directories, and more.