Question Bank on Module 4

1. Discuss the various methods available in shutil module with suitable examples.

The shutil module in Python provides various file and directory operations for copying, moving, archiving, and deleting files and directories. Here are some of the methods available in the shutil module with suitable examples:

a. **shutil.copy(src, dst, *, follow_symlinks=True):** Copies the file src to the file or directory dst. If dst is a directory, a new file will be created with the same basename as src. The optional parameter follow_symlinks determines whether to follow symbolic links or not. Here is an example:

```
import shutil

# copy file
shutil.copy('file.txt', 'newfile.txt')

# copy directory
shutil.copytree('dir', 'newdir')
```

b. shutil.move(src, dst, copy_function=copy2): Moves the file or directory src to dst. If dst is a directory, src will be moved to that directory with the same basename. The optional parameter copy_function is used to copy the file if it cannot be moved directly. Here is an example:

```
import shutil

# move file
shutil.move('file.txt', 'dir/newfile.txt')

# move directory
shutil.move('dir', 'newdir')
```

c. shutil.rmtree(path, ignore_errors=False, onerror=None): Removes the directory path and all its contents recursively. The optional parameter ignore_errors determines whether to ignore errors while deleting or not. The optional parameter onerror is a function that will be called for each error that occurs. Here is an example:

```
import shutil

# remove directory
shutil.rmtree('dir')
```

d. shutil.make_archive(base_name, format, root_dir=None, base_dir=None, verbose=False, dry_run=False, owner=None, group=None, logger=None): Creates an archive file of the directory specified by root_dir. The base_name parameter specifies the base name of the archive file. The format parameter specifies the format of the archive, such as "zip", "tar", "gztar", or "bztar". The optional parameters base_dir and verbose determine the directory to add to the archive and whether to print the name of each file added to the archive, respectively. Here is an example:

```
import shutil

# create zip archive
shutil.make_archive('archive', 'zip', root_dir='dir')
```

e. shutil.unpack_archive(filename, extract_dir=None, format=None, *, strict=False): Extracts the archive file specified by filename to the directory extract_dir. The optional parameter format specifies the format of the archive. If not specified, the format will be determined based on the file extension. Here is an example:

```
import shutil

# extract zip archive
shutil.unpack_archive('archive.zip', extract_dir='extrac
```

2. Explain os.walk() module with suitable examples

The os.walk module in Python is a very useful tool when it comes to working with file systems. It allows you to traverse directories recursively, returning all the directories, sub-directories, and files contained within them.

The os.walk method is an iterator that generates the file names in a directory tree by walking the tree either top-down or bottom-up. It yields a 3-tuple for each directory in the tree, containing the directory path, a list of all subdirectories, and a list of all the files in the directory.

Here is an example of using the os.walk module to print out all the files and directories in a specified directory and all of its subdirectories:

```
import os

# Specify the directory to traverse
root_directory = '/home/user/my_directory/'

# Traverse the directory recursively
for root, directories, files in os.walk(root_directory):
    # Print the current directory path
    print('Current directory:', root)

    # Print the list of subdirectories
    print('Subdirectories:', directories)

    # Print the list of files
    print('Files:', files)

    # Print a blank line to separate output for readability
```

```
        print()
```

In the example above, the os.walk method starts at the root directory specified and recursively traverses each directory in the tree. For each directory, it yields a 3-tuple containing the directory path, a list of all subdirectories, and a list of all the files in the directory. These values are unpacked into the variables root, directories, and files.

The print statements in the loop then print out the current directory path, the list of subdirectories, and the list of files for each directory. Finally, a blank line is printed to separate the output for readability.

Using the os.walk module can be very powerful for tasks such as searching for specific files, filtering directories based on certain criteria, or performing operations on all the files in a directory tree.

3. Write a note compressing files with zipfile module

The zipfile module in Python provides functionality for creating, reading, and extracting ZIP files. With the zipfile module, you can compress files and directories into a single ZIP file, and extract files and directories from a ZIP file.

To compress files and directories into a ZIP file, you can use the ZipFile class in the zipfile module. Here's an example that demonstrates how to compress a directory and all of its contents into a ZIP file:

```
import zipfile
import os

def zip_directory(directory_path, zip_path):
    # Create a ZipFile object with write mode
    with zipfile.ZipFile(zip_path, 'w') as zip_file:
        # Walk the directory tree and add each file to the ZIP file
        for root, dirs, files in os.walk(directory_path):
            for file in files:
                # Create the full path of the file and add it to the ZIP file
                file_path = os.path.join(root, file)
                zip_file.write(file_path, os.path.relpath(file_path, directory_path))

    print(f'{zip_path} has been created.')
```

In this example, the zip_directory() function takes two arguments: the path to the directory to be compressed, and the path to the ZIP file to be created. The function uses the with statement to create a ZipFile object in write mode, and then walks the directory tree using os.walk(). For each file in the directory tree, the function creates the full path of the file and adds it to the ZIP file using

the write() method of the ZipFile object. The second argument to write() is the relative path of the file within the directory tree.

Once all the files have been added to the ZIP file, the function closes the ZipFile object and prints a message indicating that the ZIP file has been created.

To extract files and directories from a ZIP file, you can use the extractall() method of the ZipFile object. Here's an example that demonstrates how to extract all files and directories from a ZIP file:

```
import zipfile

def unzip_directory(zip_path, extract_path):
    # Create a ZipFile object with read mode
    with zipfile.ZipFile(zip_path, 'r') as zip_file:
        # Extract all files and directories to the specified path
        zip_file.extractall(extract_path)

    print(f'Contents of {zip_path} have been extracted to {extract_path}.')
```

In this example, the unzip_directory() function takes two arguments: the path to the ZIP file to be extracted, and the path to the directory where the contents of the ZIP file should be extracted. The function uses the with statement to create a ZipFile object in read mode, and then calls the extractall() method of the ZipFile object to extract all files and directories to the specified path.

Once the extraction is complete, the function prints a message indicating that the contents of the ZIP file have been extracted to the specified directory.

Using the zipfile module in Python, you can easily compress and extract files and directories in ZIP format, making it a useful tool for archiving and sharing files

4. Explain copying and renaming files using shutil module with suitable examples.

The shutil module in Python provides a set of high-level file operations that make it easy to work with files and directories. It includes functions for copying, moving, and deleting files and directories, among other operations.

Copying Files:

To copy a file using the shutil module, you can use the copy() method. Here's an example that demonstrates how to copy a file from one directory to another:

```
import shutil

source_file = '/path/to/source/file.txt'
destination_directory = '/path/to/destination/'
```

```
shutil.copy(source_file, destination_directory)
```

In this example, the shutil.copy() method is used to copy the file at source_file to the directory at destination_directory. The copy() method preserves the file's metadata (such as permissions and timestamps) and creates a new file at the destination path.

Renaming Files:

To rename a file using the shutil module, you can use the move() method. Here's an example that demonstrates how to rename a file:

```
import shutil

old_file_name = '/path/to/old/file.txt'
new_file_name = '/path/to/new/file.txt'

shutil.move(old_file_name, new_file_name)
```

In this example, the shutil.move() method is used to rename the file at old_file_name to new_file_name. The move() method preserves the file's metadata (such as permissions and timestamps) and creates a new file at the destination path with the new name.

Copying and Renaming Files:

To copy and rename a file at the same time using the shutil module, you can use the copy2() method. Here's an example that demonstrates how to copy and rename a file:

```
import shutil

source_file = '/path/to/source/file.txt'
destination_file = '/path/to/destination/file.txt'

shutil.copy2(source_file, destination_file)
```

In this example, the shutil.copy2() method is used to copy the file at source_file to the new path at destination_file and preserve its metadata. The resulting file has the same contents as the original file but is located at the new path with a new name.

Overall, the shutil module in Python provides a simple and intuitive way to perform file and directory operations such as copying and renaming, making it a useful tool for file management tasks.

5. **Write a python program to create a folder named PYTHON and under the said folder create three files with names file1, file2 and file3. Write the contents in file1 as ”NCET” and file2 as “VTU” and file3 contents should be the merge of the contents of files file1 and file2. Check out the necessary conditions before writing file3.**

```
import os
```

```
# Create the directory "PYTHON"
directory = "PYTHON"
if not os.path.exists(directory):
    os.makedirs(directory)

# Write to file1
file1_path = os.path.join(directory, "file1.txt")
with open(file1_path, "w") as file1:
    file1.write("NCET")

# Write to file2
file2_path = os.path.join(directory, "file2.txt")
with open(file2_path, "w") as file2:
    file2.write("VTU")

# Write to file3
file3_path = os.path.join(directory, "file3.txt")
if os.path.exists(file1_path) and os.path.exists(file2_path):
    with open(file1_path, "r") as file1, open(file2_path, "r") as file2, open(file3_path, "w") as file3:
        file3.write(file1.read() + file2.read())
```

6. Write Python Program to change the file extension from .txt to .csv of all the files (including from Sub Directories) for a given path.

```
import os

def change_file_extension(path):
    for root, dirs, files in os.walk(path):
        for file in files:
            if file.endswith(".txt"):
                # Get the full path of the file
                file_path = os.path.join(root, file)

                # Get the new file path with .csv extension
                new_file_path = os.path.splitext(file_path)[0] + ".csv"

                # Rename the file
                os.rename(file_path, new_file_path)

                print(f"Renamed file {file_path} to {new_file_path}")

# Example usage
change_file_extension("/path/to/directory")
```

7. Explain the syntax of the try-except-finally block with suitable examples.

The try-except-finally block is used in Python for error handling. It allows you to catch and handle exceptions that might be raised in the try block, and also to execute some cleanup code regardless of whether an exception was raised or not. The basic syntax of the try-except-finally block is as follows:

```
try:
    # code that might raise an exception
    # ...
except <exception_type>:
    # handle the exception
    # ...
finally:
    # code that will be executed regardless of whether an exception was raised or not
    # ...
```

The try block contains the code that you want to execute that might raise an exception. If an exception is raised within the try block, control is transferred to the except block, where you can handle the exception. The except block can handle one or more specific types of exceptions, or it can handle all exceptions using the generic Exception type.

The finally block contains code that will be executed regardless of whether an exception was raised or not. This block is useful for cleanup code such as closing files, releasing resources, or restoring the state of the program.

Here's an example of the try-except-finally block in action:

```
try:
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))
    result = numerator / denominator
    print(f"The result is: {result}")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("Thanks for using the program!")
```

In this example, we use the try block to get the numerator and denominator from the user and calculate the result. If either of the input values is not an integer, a ValueError will be raised, and the control will be transferred to the ValueError block to handle it. If the denominator is zero, a ZeroDivisionError will be raised, and the control will be transferred to the ZeroDivisionError block to handle it.

Regardless of whether an exception was raised or not, the finally block will be executed to print a message thanking the user for using the program.

8. **How to create user defined exception in python with an example.**

In Python, you can create user-defined exceptions by defining a new class that inherits from the Exception class. Here's an example of how to create a custom exception in Python:

```
class MyException(Exception):
    pass
```

In this example, we define a new class called MyException that inherits from the built-in Exception class. This class has no additional attributes or methods, so it will behave just like any other exception.

You can raise this exception using the raise keyword, like this:

```
def my_function(x):
    if x < 0:
        raise MyException("x cannot be negative")
    else:
        print(f"x = {x}")

try:
    my_function(-1)
except MyException as e:
    print(f"Error: {e}")
```

In this example, we define a function called my_function that takes a parameter x. If x is negative, we raise a MyException with an error message. Otherwise, we print the value of x.

We call the my_function with a negative value, which raises a MyException. We catch this exception using a try-except block and print the error message.

This is just a simple example of how to create a custom exception in Python. You can customize your exceptions by adding additional attributes and methods to the class, or by defining multiple exception classes for different types of errors.

9. **Write a python program to display the directory, its subdirectory and files using os.walk() method.**

```
import os

# Define the directory to start the search from
start_dir = '.'

# Walk through the directory tree using os.walk()
for dirpath, dirnames, filenames in os.walk(start_dir):
    # Print the current directory path
    print(f"Directory: {dirpath}")
```

```
    # Print the subdirectories
    for dirname in dirnames:
        print(f"Subdirectory: {os.path.join(dirpath, dirname)}")

    # Print the files
    for filename in filenames:
        print(f"File: {os.path.join(dirpath, filename)}")
```

10. Explain the reading and extracting zip file with suitable examples.

To read and extract the contents of a zip file in Python, you can use the zipfile module. Here's an example:

```
import zipfile

# Open the zip file for reading
with zipfile.ZipFile('my_zip_file.zip', 'r') as zip:
    # Print the list of file names in the zip file
    print(zip.namelist())

    # Extract all the files in the zip file to the current directory
    zip.extractall()
```

In this example, we first import the zipfile module. We then use the ZipFile() method to open the zip file in read mode. We pass the name of the zip file to open, as well as the mode ('r').

Once the zip file is open, we can use the namelist() method to print the list of file names in the zip file. This method returns a list of strings, where each string is the name of a file in the zip archive.

We can also extract all the files in the zip file to the current directory using the extractall() method. This method extracts all the files to the current directory, preserving the directory structure of the archive.

To extract a specific file from the zip archive, you can use the extract() method instead of extractall(). Here's an example:

```
import zipfile

# Open the zip file for reading
with zipfile.ZipFile('my_zip_file.zip', 'r') as zip:
    # Extract a specific file from the zip archive
    zip.extract('file.txt')
```

In this example, we use the extract() method to extract a single file from the zip archive. We pass the name of the file to extract as a string argument to the method.

11. Explain deleting and moving files using shutil module with suitable examples.

The shutil module in Python provides several functions for deleting and moving files. Here are examples of how to use these functions:

Deleting files with shutil

To delete a file in Python, you can use the os.remove() function or the os.unlink() function. Alternatively, you can use the shutil.rmtree() function to delete a directory and all its contents. Here's an example of using os.remove() to delete a file:

```
import os

# Define the file path to delete
file_path = 'my_file.txt'

# Delete the file
os.remove(file_path)
```

In this example, we use the os.remove() function to delete the file at the specified file_path. This function deletes the file from the file system.

Moving files with shutil

To move a file in Python, you can use the shutil.move() function. Here's an example:

```
import shutil

# Define the source and destination file paths
source_file = 'my_file.txt'
dest_dir = '/path/to/destination/directory'

# Move the file to the destination directory
shutil.move(source_file, dest_dir)
```

In this example, we use the shutil.move() function to move the file at source_file to the directory specified by dest_dir. This function moves the file from the source location to the destination location, and if a file with the same name already exists in the destination directory, it is overwritten.

Copying files with shutil

To copy a file in Python, you can use the shutil.copy() function or the shutil.copy2() function. Here's an example of using shutil.copy() to copy a file:

```
import shutil

# Define the source and destination file paths
source_file = 'my_file.txt'
dest_dir = '/path/to/destination/directory'

# Copy the file to the destination directory
shutil.copy(source_file, dest_dir)
```

In this example, we use the shutil.copy() function to copy the file at source_file to the directory specified by dest_dir. This function creates a new file with the same name as the source file in the destination directory

and copies the contents of the source file into it. If a file with the same name already exists in the destination directory, it is overwritten.

Renaming files with shutil

To rename a file in Python, you can use the os.rename() function or the shutil.move() function. Here's an example of using os.rename() to rename a file:

```
import os

# Define the original and new file names
original_file_name = 'my_file.txt'
new_file_name = 'new_file.txt'

# Rename the file
os.rename(original_file_name, new_file_name)
```

In this example, we use the os.rename() function to rename the file with the original name original_file_name to the new name new_file_name. This function renames the file in the file system. Note that if a file with the new name already exists in the same directory, it will be overwritten.

12. Illustrate exception handling in python with suitable examples.

In Python, exception handling allows us to gracefully handle runtime errors and prevent our programs from crashing. We can use the try, except, and finally statements to handle exceptions.

Here's an example of how to handle a ZeroDivisionError exception that occurs when dividing by zero:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero")
finally:
    print("End of program")
```

In this example, we try to divide the number 10 by zero, which raises a ZeroDivisionError exception. We catch the exception using the except block and print an error message. Finally, we use the finally block to print a message indicating the end of the program. The output of the program would be:

```
Error: Cannot divide by zero
End of program
```

Here's another example that demonstrates how to handle a FileNotFoundError exception that occurs when trying to open a file that does not exist:

```
try:
    with open("nonexistent_file.txt", "r") as f:
        contents = f.read()
except FileNotFoundError:
    print("Error: File not found")
finally:
```

```
print("End of program")
```

In this example, we try to open a file called "nonexistent_file.txt" using the open() function. Since the file does not exist, a FileNotFoundError exception is raised. We catch the exception using the except block and print an error message. Finally, we use the finally block to print a message indicating the end of the program. The output of the program would be:

```
Error: File not found
End of program
```

In both of these examples, we use exception handling to gracefully handle runtime errors and ensure that our programs do not crash. By catching and handling exceptions, we can provide a better user experience and make our code more robust.

13. Develop a program to backing up a given Folder (Folder in a current working directory) into a ZIP File by using relevant modules and suitable methods.

```
import zipfile
import os

def backup_to_zip(folder):
    # Get the full path of the folder
    folder = os.path.abspath(folder)

    # Determine the filename for the ZIP archive
    base_name = os.path.basename(folder)
    zip_filename = base_name + '.zip'

    # Create the ZIP archive
    with zipfile.ZipFile(zip_filename, 'w') as zip_archive:
        # Walk the entire folder tree and compress the files in each folder
        for foldername, subfolders, filenames in os.walk(folder):
            print(f'Adding files in {foldername}...')
            # Add all the files in the current folder to the ZIP archive
            for filename in filenames:
                file_path = os.path.join(foldername, filename)
                zip_archive.write(file_path, os.path.relpath(file_path, folder))

    print('Done.')
```

14. Explain the use of assert function with suitable examples.

The assert function is a debugging aid that tests a condition and raises an error if the condition is not true. It takes two arguments: a condition to test and an optional message to display if the test fails.

Here's an example of using assert to check if a given number is even:

```
def is_even(n):
    assert n % 2 == 0, f'{n} is not even!'
    print(f'{n} is even.')
```

In this example, the is_even function takes an argument n and uses assert to check if n is even. If the condition n % 2 == 0 is true, the function prints a message indicating that n is even. If the condition is false, the assert function raises an AssertionError with the message 'n is not even!'.

Here's an example of calling the is_even function with an even number:
is_even(4)
This will output the following message:
4 is even.
And here's an example of calling the is_even function with an odd number:
is_even(3)
This will raise an AssertionError with the message '3 is not even!'.

Using assert can be a helpful way to catch bugs and ensure that your code is working correctly. However, it's important to use it appropriately and not rely on it as a substitute for proper error handling and testing.

15. Discuss absolute and relative path with suitable examples.

In Python, a file or directory can be referred to using either an absolute path or a relative path.

An absolute path is a full path that specifies the location of a file or directory in the file system. It starts with the root directory, which is denoted by a forward slash (/) on Unix-like systems or a drive letter followed by a colon (:) on Windows systems. Here's an example of an absolute path:

```
/home/user/documents/myfile.txt  # Unix-like system
C:\Users\User\Documents\myfile.txt  # Windows system
```

A relative path, on the other hand, specifies the location of a file or directory relative to the current working directory. It does not start with the root directory, but rather with a folder or file that is located in the current working directory. Here are some examples of relative paths:

```
myfile.txt  # refers to a file in the current working directory
../myfolder/myfile.txt  # refers to a file in a parent directory
./myfolder/myfile.txt  # refers to a file in a subdirectory of the current directory
```

In the above examples, ../ refers to the parent directory, ./ refers to the current directory, and myfolder/ and myfile.txt refer to a subdirectory and file within the specified directory.

To convert a relative path to an absolute path in Python, you can use the os.path.abspath() function. Here's an example:

```
import os

# Get the absolute path of a file using a relative path
relative_path = 'myfile.txt'
```

```
absolute_path = os.path.abspath(relative_path)
print(absolute_path)
```

This will output the absolute path of the file myfile.txt in the current working directory.

Understanding the difference between absolute and relative paths is important for working with files and directories in Python, especially when writing scripts that need to access files in different locations on a file system.

16. Write a function named DivExp which takes two parameters a, b and returns a value c (c=a/b). Write suitable assertion for a>0 in function DivExp and raise an exception for when b=0. Develop a suitable program which reads two values from the console and calls a function DivExp.

```
def DivExp(a, b):
    assert a > 0, "Value of a must be greater than 0"
    try:
        c = a / b
        return c
    except ZeroDivisionError:
        print("Error: division by zero")

a = int(input("Enter a value for a: "))
b = int(input("Enter a value for b: "))

result = DivExp(a, b)
print("Result: ", result)
```

17. Explain logging levels in python

In Python, the logging module provides a way to track events that occur during the execution of a program. The logging levels in Python are used to specify the severity of the events being logged. There are five standard logging levels in Python:

I. DEBUG: Detailed information, typically of interest only when diagnosing problems.
II. INFO: Confirmation that things are working as expected.
III. WARNING: An indication that something unexpected happened or indicative of some problem in the near future (e.g., 'disk space low'). The software is still working as expected.
IV. ERROR: Due to a more serious problem, the software has not been able to perform some function.
V. CRITICAL: A very serious error, indicating that the program itself may be unable to continue running.

The logging levels are ordered in increasing order of severity. So, if a particular logging level is set to DEBUG, then all log events with DEBUG, INFO, WARNING, ERROR, and CRITICAL levels will be logged. If a particular logging level is set to WARNING, then only the events with WARNING, ERROR, and CRITICAL levels will be logged.

Here's an example code that demonstrates the use of logging levels in Python:

```
import logging

logging.basicConfig(level=logging.DEBUG)

def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        logging.error("Attempted to divide by zero")
    else:
        logging.info(f"{x} divided by {y} is {result}")
    return result

divide(4, 2)
divide(4, 0)
```

In this code, we first set the logging level to DEBUG. Then, we define a function named divide that takes two parameters x and y. Inside the function, we try to divide x by y, and if y is 0, we log an error message using the logging.error method. Otherwise, we log an info message using the logging.info method. Finally, we return the result of the division.

We call the divide function twice, once with the arguments 4 and 2, and once with the arguments 4 and 0. The first call will log an info message, while the second call will log an error message. When we run this code, we should see the following output:

```
INFO:root:4 divided by 2 is 2.0
ERROR:root:Attempted to divide by zero
```

The INFO message is logged because the division was successful, while the ERROR message is logged because the second call attempted to divide by zero.