Module 9 – Debugging

Raising Exceptions

Raising Exceptions: Python's exception handling mechanism allows you to raise exceptions when encountering errors or unexpected conditions in your code.

You can use **‘raise’** statement to raise exceptions manually, providing helpful error messages to identify the issue.

Raising Exceptions

```
def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y

try:
    result = divide(10, 0)
except ValueError as e:
    print("Error:", e)
```

In this example, if **y** is 0, the **divide** function raises a **ValueError** with the message "Cannot divide by zero". The exception is caught in the **try-except** block, allowing you to handle the error gracefully.

Assertions

Assertions are used to check if a condition is true, and if it's not, it raises an **AssertionError**.
Assertions are often used to check for conditions that should never occur during the execution of the program.

```
def calculate_area(width, height):
    assert width > 0 and height > 0, "Width and height must be positive"
    return width * height

print(calculate_area(-5, 10))  # This will raise an AssertionError
```

Logging Module and File

The **‘logging’** module in Python provides a flexible framework for logging messages from your code. It supports different logging levels (**DEBUG, INFO, WARNING, ERROR, CRITICAL**) and allows you to direct log messages to different destinations such as the console, files, or network streams.

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

IDLE’s Debugger

IDLE, Python's Integrated Development and Learning Environment, comes with a built-in debugger that allows you to interactively debug your code.

You can start the debugger from the Debug menu in IDLE or by using the **‘pdb’** module programmatically.

```
import pdb

def some_function():
    x = 10
    y = 20
    pdb.set_trace()  # Start debugger at this line
    z = x + y
    print(z)

some_function()
```

Breakpoints

Python 3.7 introduced the **breakpoint()** function, which allows you to set breakpoints in your code. When executed, **breakpoint()** enters the debugger at that point in your code.

```
def some_function():
    x = 10
    y = 20
    breakpoint()  # Execution pauses here
    z = x + y
    print(z)

some_function()
```

Breakpoints

Once the debugger is activated at the **breakpoint()** call, you can step through the code and inspect variables just like with **pdb**.

These debugging techniques are essential for identifying and resolving issues in your Python code, making your development process smoother and more efficient.

Module 10 –
Working with CSV Files and JSON

CSV Modules and Delimiters

CSV Module in Python:

- The **csv** module in Python provides functionalities to read from and write to CSV files.
- It simplifies the process of handling CSV files by providing convenient functions and classes.

Delimiters:

- A delimiter is a character used to separate fields in a CSV file.
- By default, the comma (**,**) is used as the delimiter, but it can be customized based on the requirements of the data.

CSV Modules and Delimiters

Reading from a CSV File:

```
import csv

# Open the CSV file in read mode
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Iterate over each row in the CSV file
    for row in reader:
        print(row)
```

CSV Modules and Delimiters

Writing to a CSV File:

```
import csv

# Data to be written to the CSV file
data = [
    ['Name', 'Age', 'Country'],
    ['John', '30', 'USA'],
    ['Alice', '25', 'UK']
]

# Open the CSV file in write mode
with open('data.csv', 'w', newline='') as file:
    # Create a CSV writer object
    writer = csv.writer(file)

    # Write data to the CSV file
    writer.writerows(data)
```

CSV Modules and Delimiters

Customizing Delimiters:

```python
import csv

# Data to be written to the CSV file
data = [
    ['Name', 'Age', 'Country'],
    ['John', '30', 'USA'],
    ['Alice', '25', 'UK']
]

# Open the CSV file with a custom delimiter (semicolon)
with open('data_custom_delimiter.csv', 'w', newline='') as file:
    # Create a CSV writer object with a custom delimiter
    writer = csv.writer(file, delimiter=';')

    # Write data to the CSV file with the custom delimiter
    writer.writerows(data)
```

Reader and Writer Objects

Reader Object:

The **csv.reader** object is used to read data from a CSV file row by row.

It provides methods to iterate over each row and retrieve the data as a list of strings.

Reader and Writer Objects

```python
import csv

# Open the CSV file in read mode
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Iterate over each row in the CSV file
    for row in reader:
        print(row)
```

In this example, the **csv.reader** object is created to read data from the 'data.csv' file.

The **for** loop iterates over each row in the CSV file, and **row** variable contains a list of values from each row.

Reader and Writer Objects

Writer Object:

The **csv.writer** object is used to write data to a CSV file.

It provides methods to write rows of data to the file.

Reader and Writer Objects

```python
import csv

# Data to be written to the CSV file
data = [
    ['Name', 'Age', 'Country'],
    ['John', '30', 'USA'],
    ['Alice', '25', 'UK']
]

# Open the CSV file in write mode
with open('data.csv', 'w', newline='') as file:
    # Create a CSV writer object
    writer = csv.writer(file)

    # Write data to the CSV file
    writer.writerows(data)
```

Reader and Writer Objects

In this example, the **csv.writer** object is created to write data to the 'data.csv' file.

The **writer.writerows()** method is used to write multiple rows of data to the CSV file.

Each inner list in the **data** list represents a row in the CSV file.

These Reader and Writer objects in the CSV module provide convenient ways to read from and write to CSV files in Python, making it easier to work with tabular data.

JSON Module and I/O Functions

1. **JSON Module**:

- The **json** module in Python provides functions for encoding and decoding JSON data.

- It allows you to serialize Python objects into JSON strings and deserialize JSON strings into Python objects.

JSON Module and I/O Functions

2. **Reading from JSON**:

   - Use **json.load(file)** or **json.loads(json_string)** functions to read JSON data from a file object or a JSON-formatted string, respectively.

```
import json

# Open the JSON file in read mode
with open('data.json', 'r') as file:
    # Load JSON data from the file
    data = json.load(file)
    print(data)
```

JSON Module and I/O Functions

3. **Writing to JSON**:

Use **json.dump(data, file)** or **json.dumps(data)** functions to write Python data to a file object as JSON or to return a JSON-formatted string, respectively.

```
import json

# Python data to be converted to JSON
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Open the JSON file in write mode
with open('data.json', 'w') as file:
    # Write JSON data to the file
    json.dump(data, file)
```

JSON Module and I/O Functions

4. **I/O Functions**:

- **json.load(file)**: Loads JSON data from a file object.
- **json.loads(json_string)**: Loads JSON data from a JSON-formatted string.
- **json.dump(data, file)**: Writes JSON data to a file object.
- **json.dumps(data)**: Returns a JSON-formatted string representing the Python data.

JSON Module and I/O Functions

```
import json

# Example of using json.loads()
json_string = '{"name": "Alice", "age": 25, "city": "London"}'
data = json.loads(json_string)
print(data)

# Example of using json.dumps()
data = {'name': 'Bob', 'age': 35, 'city': 'Paris'}
json_string = json.dumps(data)
print(json_string)
```

JSON Module and I/O Functions

- **json.loads()** function parses a JSON-formatted string and returns a Python data structure.
- **json.dumps()** function serializes a Python data structure into a JSON-formatted string.

The JSON module in Python provides a convenient way to work with JSON data, which is commonly used for data interchange between web servers and clients, as well as for storing and sharing structured data.

Pytest command line arguments

1. **Run All Tests:**

```
pytest
```

   - This command runs all the tests in the current directory and its subdirectories.
   - Pytest will discover and execute all test files following its default naming conventions.

2. **Run Specific Test Files:**

```
pytest test_file.py
```

   - You can specify the name of a specific test file to run.
   - Pytest will discover and execute tests in the specified file.

File Handling

File handling in Python is a powerful and versatile tool that can be used to perform a wide range of operations. However, it is important to carefully consider the advantages and disadvantages of file handling when writing Python programs, to ensure that the code is secure, reliable, and performs well.

In this topic, we will explore

Python File Handling,

Advantages, Disadvantages and How open,

write and append functions works in python file.

Python File Handling

- Python supports file handling and allows users to handle files i.e.,

to read and write files, along with many other file handling options, to operate on files.

- The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, like other concepts of Python, this concept here is also easy and short.

Python File Handling

- Python treats files differently as text or binary and this is important.
- Each line of code includes a sequence of characters, and they form a text file.
- Each line of a file is terminated with a special character, called the **EOL** or **End of Line** characters like **comma {,}** or **newline character.**
- It ends the current line and tells the interpreter a new one has begun. Let's start with the reading and writing files.

Advantages of File Handling

Versatility:

File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.

Flexibility:

File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files, etc.), and to perform different operations on files (e.g. read, write, append, etc.).

Advantages of File Handling

User–friendly:

Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.

Cross-platform:

Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling

Error-prone:

File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).

Security risks:

File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.

Disadvantages of File Handling

Complexity:

File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.

Performance:

File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.

Python File Open

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python's inbuilt function open() but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

Let's use a training.txt file which contains some random data.

f = open(filename, mode)

Python File Open

Where the following mode is supported:

r: open an existing file for a read operation.

w: open an existing file for a write operation. If the file already contains some data, then it will be overridden but if the file is not present then it creates the file as well.

a: open an existing file for append operation. It won't override existing data.

r+: To read and write data into the file. The previous data in the file will be overridden.

w+: To write and read data. It will override existing data.

a+: To append and read data from the file. It won't override existing data.

Working in Read Mode

There is more than one way to How to read from a file in Python. Let us see how we can read the content of a file in read mode.

Example 1: The open command will open the Python file in the read mode and the for loop will print each line present in the file.

```
1  # a file named "training", will be opened with the reading mode.
2  file = open('training.txt', 'r')
3
4  # This will print every line one by one in the file
5  for each in file:
6      print (each)
7
```

Working in Read Mode

Example 2: In this example, we will extract a string that contains all characters in the Python file then we can use **file.read()**.

```
1  # Python code to illustrate read() mode
2  file = open("training.txt", "r")
3  print (file.read())
4
```

Working in Read Mode

Example 3: In this example, we will see how we can read a file using the **with** statement in Python.

```
1  # Python code to illustrate with()
2  with open("training.txt") as file:
3      data = file.read()
4
5  print(data)
6
```

Working in Read Mode

Example 4: Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
1  # Python code to illustrate read() mode character wise
2  file = open("training.txt", "r")
3  print (file.read(5))
4
```

Working in Read Mode

Example 5: We can also split lines while reading files in Python. The split() function splits the variable when space is encountered.

You can also split using any characters as you wish.

```
# Python code to illustrate split() function
with open("training.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

Creating a File using write() function

Just like reading a file in Python, there are a number of ways to Writing to file in Python.

Let us see how we can write the content of a file using the write() function in Python.

Working in Write Mode

Let’s see how to create a file and how the write mode works.

Example 1: In this example, we will see how the write mode and the write() function is used to write in a file. The close() command terminates all the resources in use and frees the system of this particular program.

```
1  # Python code to create a file
2  file = open('training.txt','w')
3  file.write("This is the write command")
4  file.write("It allows us to write in a particular file")
5  file.close()
6
```

Working in Write Mode

Example 2: We can also use the written statement along with the with() function.

```
1  # Python code to illustrate with() alongwith write()
2  with open("file.txt", "w") as f:
3      f.write("Hello World!!!")
4
```

Working of Append Mode

Let us see how the append mode works.

Example: For this example, we will use the Python file created in the previous example.

```
1  # Python code to illustrate append() mode
2  file = open('training.txt', 'a')
3  file.write("This will add this line")
4  file.close()
```

Working in Append Mode

There are also various other commands in Python file handling that are used to handle various tasks:

rstrip(): This function strips each line of a file off spaces from the right-hand side.

lstrip(): This function strips each line of a file off spaces from the left-hand side.

It is designed to provide much cleaner syntax and exception handling when you are working with code. That explains why it’s good practice to use them with a statement where applicable. This is helpful because using this method any files opened will be closed automatically after one is done, so auto-cleanup.

Implementing all the functions in File Handling

In this example, we will cover all the concepts that we have seen above. Other than those, we will also see how we can delete a file using the remove() function from Python os module.

```
import os

def create_file(filename):
    try:
        with open(filename, 'w') as f:
            f.write('Hello, world!\n')
        print("File " + filename + " created successfully.")
    except IOError:
        print("Error: could not create file " + filename)

def read_file(filename):
    try:
        with open(filename, 'r') as f:
            contents = f.read()
            print(contents)
    except IOError:
```

Implementing all the functions in File Handling

```
19 def append_file(filename, text):
20     try:
21         with open(filename, 'a') as f:
22             f.write(text)
23         print("Text appended to file " + filename + " successfully.")
24     except IOError:
25         print("Error: could not append to file " + filename)
26
27 def rename_file(filename, new_filename):
28     try:
29         os.rename(filename, new_filename)
30         print("File " + filename + " renamed to " + new_filename + "
successfully.")
31     except IOError:
32         print("Error: could not rename file " + filename)
33
34 def delete_file(filename):
35     try:
36         os.remove(filename)
37         print("File " + filename + " deleted successfully.")
38     except IOError:
39         print("Error: could not delete file " + filename)
```

Implementing all the functions in File Handling

```
40
41
42  if __name__ == '__main__':
43      filename = "example.txt"
44      new_filename = "new_example.txt"
45
46      create_file(filename)
47      read_file(filename)
48      append_file(filename, "This is some additional text.\n")
49      read_file(filename)
50      rename_file(filename, new_filename)
51      read_file(new_filename)
52      delete_file(new_filename)
53
```

Implementing all the functions in File Handling

Output:

```
File example.txt created successfully.
Hello, world!
Text appended to file example.txt successfully.
Hello, world!
This is some additional text.
File example.txt renamed to new_example.txt successfully.
Hello, world!
This is some additional text.
File new_example.txt deleted successfully.
```