Name : Arnab Mandal
Emain : mandal.arnab2004@gmail.com
Assignments: File handling, Exception Handling and Multitasking in Python

1) Write a code to read the contents of a file in python
  Ans:

```
# Read the contents of a file
file_path = 'example.txt'  # Replace with your file path

try:
    with open(file_path, 'r') as file:
        contents = file.read()
        print(contents)
except FileNotFoundError:
    print(f"The file {file_path} does not exist.")
```

2) Write a code to write to a file in python.
Ans:

```
# Write to a file
file_path = 'example.txt'  # Replace with your file path
data = "This is a sample text that will be written to the file."

with open(file_path, 'w') as file:
    file.write(data)
    print(f"Data written to {file_path}.")
```

3) Write a code to append to a file in python
Ans:

```
# Append to a file
file_path = 'example.txt'  # Replace with your file path
data = "\nThis is additional text that will be appended to the file."

with open(file_path, 'a') as file:
    file.write(data)
    print(f"Data appended to {file_path}.")
```

4)Write a code to read a binary file in python
Ans:

```
def read_binary_file(file_path):
    with open(file_path, 'rb') as file:
        data = file.read()
    return data

# Example usage
binary_data = read_binary_file('example.bin')
print(binary_data)
```

5) What happens if we don't use 'with' keyword with 'open' in python ?
Ans:
If you don't use the `with` keyword with `open`, you need to manually close the file using `file.close()`. Failing to close the file can lead to resource leaks and potential data corruption, as the file handle remains open. The `with` statement ensures that the file is properly closed after its suite finishes, even if an exception is raised.

6)Explain the concept of buffering in file handling and how it helps in improving read and write operations.
Ans:

Buffering involves temporarily storing data in memory before it is read from or written to a file. This is done to minimize the number of I/O operations, which are generally slow compared to memory operations. Buffering helps in improving performance by:

- **Reducing Disk I/O**: By accumulating data in memory and writing it to the disk in larger chunks, buffering reduces the number of I/O operations.
- **Improving Efficiency**: It allows programs to handle file operations more efficiently, especially for large files.

7) Describe the steps involved in implementing buffered file handling in programming language of your choice.
Ans:

1. **Open the File**: Open the file with buffering enabled.
2. **Perform File Operations**: Read or write data using buffered streams.
3. **Close the File**: Ensure the file is closed, which also flushes the buffer.

In Python, you typically use buffering by default, but you can specify buffering behavior explicitly.

8) Write a python function to read a text file using buffered reading and return its contents.
Ans:
```
def read_text_file_buffered(file_path):
```

```
    with open(file_path, 'r', buffering=4096) as file:
        contents = file.read()
    return contents

# Example usage
file_contents = read_text_file_buffered('example.txt')
print(file_contents)
```

9) What are the advantages of using buffered reading over direct file reading in python ?
Ans:
Performance: Buffered reading reduces the number of read operations by accumulating data in memory, which is faster than frequent disk I/O operations.
Efficiency: Handles larger files more efficiently by processing chunks of data rather than reading one byte or line at a time.
Reduced Latency: Minimizes delays associated with disk access.

10) Write a python code snipped to append content to a file using buffered writing.
Ans:

```
def append_to_file(file_path, content):
    with open(file_path, 'a', buffering=4096) as file:
        file.write(content)

# Example usage
append_to_file('example.txt', 'This is the appended content.\n')
```

11) Write a python function that demonstrates the use of close() method on a file.
Ans:

```
def demonstrate_close():
    file = open("sample.txt", "w")
    file.write("Hello, World!")
    file.close()  # Closes the file to free up resources

# Running the function
demonstrate_close()
```

12) Create a python function to showcase the detach() method on a file object.
Ans:

```
def demonstrate_detach():
    file = open("sample.txt", "w")
    buffer = file.detach()  # Detaches the underlying buffer from the file object
    print(f"Buffer: {buffer}")
    file.close()
```

```
# Running the function
demonstrate_detach()
```

13) Write a python function to demonstrate the use of the seek() method to change the file position.
Ans:

```
def demonstrate_seek():
    with open("sample.txt", "w+") as file:
        file.write("Hello, World!")
        file.seek(0)  # Move the file cursor to the beginning
        content = file.read()
        print(f"Content: {content}")

# Running the function
demonstrate_seek()
```

14) Create a python function to return the file descriptor (integer number) of a file using the fileno() method.
Ans:

```
def get_file_descriptor():
    with open("sample.txt", "w") as file:
        fd = file.fileno()  # Get the file descriptor
        print(f"File Descriptor: {fd}")

# Running the function
get_file_descriptor()
```

15) Write a python function to return the current position of the file's object using the tell() method.
Ans:

```
def get_file_position():
    with open("sample.txt", "w") as file:
        file.write("Hello, World!")
        position = file.tell()  # Get the current file position
        print(f"File Position: {position}")

# Running the function
get_file_position()
```

16) Create a python program that logs a message to a file using the logging module.
Ans:

```
import logging

def log_message():
    logging.basicConfig(filename="logfile.log", level=logging.INFO)
    logging.info("This is an info message")

# Running the function
log_message()
```

17) Explain the importance of logging levels in python's logging module.
Ans:

Logging levels in Python's logging module allow you to control the granularity of log messages captured. They help in categorizing the severity or importance of events:

- **DEBUG**: Detailed information, typically of interest only when diagnosing problems.
- **INFO**: Confirmation that things are working as expected.
- **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.
- **ERROR**: Due to a more serious problem, the software has not been able to perform some function.
- **CRITICAL**: A very serious error, indicating that the program itself may be unable to continue running.

18) Create a python program that uses the debugger to find the value of a variable inside a loop.
Ans:

```
import pdb

def debug_variable():
    for i in range(5):
        pdb.set_trace()  # Set a breakpoint
        print(f"Value of i: {i}")

# Running the function
debug_variable()
```

19) Create a python program that demonstrates setting breakpoints and inspecting variables using the debugger.

Ans:

```
import pdb

def inspect_variables():
    x = 10
    y = 20
    pdb.set_trace()  # Set a breakpoint here
    result = x + y
    print(f"Result: {result}")

# Running the function
inspect_variables()
```

20) Create a python program that uses the debugger to trace a recursive function.

Ans:

```
import pdb

def factorial(n):
    pdb.set_trace()  # Set a breakpoint to trace the function
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Running the function
print(factorial(5))
```

21) Write a try-except block to handle a ZeroDividionError.

Ans:

```
 def handle_zero_division():
    try:
        result = 10 / 0
    except ZeroDivisionError:
        print("Division by zero is not allowed")

# Running the function
handle_zero_division()
```

22) How does the else block work with try-except ?

Ans:

The `else` block in a `try-except` statement executes only if no exceptions were raised in the `try` block. It's useful for code that should run only if the `try` block did not encounter any errors.

23) Implement a try-except-else block to open and read a file.
Ans:

```
def read_file():
    try:
        file = open("sample.txt", "r")
    except FileNotFoundError:
        print("File not found")
    else:
        content = file.read()
        print(f"File content: {content}")
        file.close()

# Running the function
read_file()
```

24) What is the purpose of the finally block in exception handling.
Ans:
The `finally` block is used to execute code regardless of whether an exception was raised or not. It is typically used for cleanup actions, like closing files or releasing resources.

25) Write a try-except-finally block to handle a ValueError.
Ans:

```
def handle_value_error():
    try:
        number = int("abc")
    except ValueError:
        print("Invalid input, could not convert to integer")
    finally:
        print("Execution completed")

# Running the function
handle_value_error()
```

26) How multiple except blocks work in python.
Ans:

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ValueError:
    print("ValueError: You must enter a number.")
```

```
except ZeroDivisionError:
    print("ZeroDivisionError: You can't divide by zero.")
except Exception as e:
    print(f"Unexpected error: {e}")
```

Explanation:
In this example, if a `ValueError` occurs (e.g., if the input is not an integer), the first `except` block will be executed. If a `ZeroDivisionError` occurs (e.g., if `x` is `0`), the second `except` block will handle it. If any other exception occurs, it will be caught by the general `Exception` block.

27) What is a custom exception in python
Ans:
A custom exception in Python is a user-defined exception class that extends the base `Exception` class. Custom exceptions allow you to create meaningful error types for specific situations in your code.

28) Create a custom exception class with a message.
Ans:

```
class NegativeValueError(Exception):
    def __init__(self, message="Negative value is not allowed"):
        self.message = message
        super().__init__(self.message)
```

29) Write a code to raise a custom exception in python.
Ans:

```
def check_positive(value):
    if value < 0:
        raise NegativeValueError(f"Negative value found: {value}")
    return value

try:
    check_positive(-10)
except NegativeValueError as e:
    print(e)
```

30) Write a function that raises a custom exception when a value is negative.
Ans:

```
def validate_positive(value):
    if value < 0:
        raise NegativeValueError(f"Invalid input: {value} is negative.")
```

```
    return value
```

31) What is the role of try,except,else,and finally in handling exceptions.
Ans:
`try`: The block of code where exceptions might occur.
`except`: The block that handles exceptions if they occur in the `try` block.
`else`: Executed if no exceptions occur in the `try` block.
`finally`: Executed regardless of whether an exception occurs, usually used for cleanup.

32) How can custom exceptions improve code readability and maintainability.
Ans:
Custom exceptions make it easier to understand the specific errors that can occur in your code. They can provide more meaningful error messages and help in debugging, leading to more maintainable and readable code.

33) What is multithreading ?
Ans:
Multithreading in Python is the concurrent execution of multiple threads (smaller units of a process) within a single process. It allows for parallelism in tasks that can be done simultaneously.

34) Create a thread in python.
Ans:

```
import threading

def print_hello():
    print("Hello from a thread!")

thread = threading.Thread(target=print_hello)
thread.start()
thread.join()
```

35) What is the Global Interpreter Lock(GIL) in python ?
Ans:
The Global Interpreter Lock (GIL) is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously in a multi-core processor environment. It ensures thread safety but can be a bottleneck in CPU-bound tasks.
36) Implement a simple multithreading example in python.
Ans:

```
import threading

def print_numbers():
```

```
    for i in range(1, 6):
        print(i)

def print_letters():
    for letter in 'abcde':
        print(letter)

t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

t1.start()
t2.start()

t1.join()
t2.join()
```

37) What is the purpose of the `join()` method in threading ?
Ans:
The `join()` method in threading ensures that the calling thread waits until the thread on which `join()` was called has completed its execution.

38) Describe a scenario where multithreading would be beneficial in python .
Ans:
Multithreading is beneficial when you have I/O-bound tasks, such as reading/writing to files, making network requests, or handling user input/output, where tasks spend time waiting for resources to be available.

39) What is multiprocessing in python ?
Ans:
Multiprocessing is a technique to execute multiple processes simultaneously, taking full advantage of multiple CPUs or CPU cores. Each process runs in its own memory space, avoiding the GIL problem.

40) How is multiprocessing different from multithreading in python ?
Ans:
Multithreading: Multiple threads run within the same process, sharing memory space. Subject to GIL limitations.
Multiprocessing: Multiple processes run independently, each with its own memory space. Not subject to GIL, making it better for CPU-bound tasks.

41) Create a process using the multiprocessing module in python .
Ans:

```
 import multiprocessing
```

```
def print_hello():
    print("Hello from a process!")

if __name__ == '__main__':
    process = multiprocessing.Process(target=print_hello)
    process.start()
    process.join()
```

42) Explain the concept of pool in the multiprocessing module.
Ans:
A pool in the multiprocessing module allows you to manage a pool of worker processes, and you can control how many processes can run concurrently. It is useful for executing a function in parallel across a sequence of inputs.

```
import multiprocessing

def square(x):
    return x * x

if __name__ == '__main__':
    with multiprocessing.Pool(4) as pool:
        results = pool.map(square, [1, 2, 3, 4, 5])
    print(results)
```

43) Explain inter-process communication in multiprocessing.
Ans:
Inter-process communication (IPC) in multiprocessing allows processes to communicate and share data. Python provides several ways to implement IPC, such as using `Queue`, `Pipe`, and `Manager`. These allow data to be passed between processes safely.

```
import multiprocessing

def producer(queue):
    queue.put("Data from producer")

def consumer(queue):
    data = queue.get()
    print(data)

if __name__ == '__main__':
    queue = multiprocessing.Queue()
    p1 = multiprocessing.Process(target=producer, args=(queue,))
```

```
p2 = multiprocessing.Process(target=consumer, args=(queue,))

p1.start()
p2.start()

p1.join()
p2.join()
```