```
# Files Exceptional handling, logging and memory management Questoins and Answers


# Theory Questions


#Ans.1
#The primary difference between interpreted and compiled languages lies in how the source code is executed.
#Compiled languages are translated into machine code (or bytecode) before runtime,
#while interpreted languages are executed line by line by an interpreter during runtime.


#Ans.2
#Exception handling in Python is a mechanism used to manage and respond to runtime errors, known as exceptions, that occur during the ex
#It allows a program to gracefully handle unexpected situations or errors without crashing, ensuring a more robust and user-friendly app


#Ans.3
#The 'finally' block is executed regardless of whether an exception occurred or not.
#It provides a way to define cleanup actions that must be performed, such as releasing resources or closing files, irrespective of the p


#Ans.4
#Logging is a means of tracking events that happen when some software runs.
#The software's developer adds logging calls to their code to indicate that certain events have occurred.


#Ans.5
#The __del__ method in Python, also known as the destructor, holds significance primarily for resource cleanup when an object is about t
#The del keyword is used to delete objects. The del keyword can also be used to delete variables, lists, or parts of a list etc.


#Ans.6
#The ' import' keyword is used to import modules or specific functions/classes from modules, making them accessible in your code.
#The ' from' keyword is used with' import' to specify which specific functions or classes you want to import from a module.


#Ans.7
#To  catch multiple exceptions in Python in a compact form, you can use the one-line try-except syntax.
#This is especially useful for handling minor errors without breaking the program flow.
#This one-line try-except statement catches multiple exceptions and provides a concise error-handling method.


#Ans.8
#The with statement in Python simplifies resource management by automatically handling setup and cleanup, such as opening and closing fi
#For example, instead of manually opening and closing resources using a try-finally block, the with statement manages this automatically


#Ans.9
#Multithreading and multiprocessing are both techniques to achieve parallelism, but they differ in how they utilize system resources.
#Multithreading involves multiple threads within a single process, sharing the same memory space,
#while multiprocessing involves multiple processes, each with its own memory space.


#Ans.10
#It significantly aids in debugging by providing detailed records of program execution, enabling developers to pinpoint issues more effi
#Logging also helps in monitoring application behavior, understanding usage patterns, and identifying potential security vulnerabilities


#Ans.11
#Memory management is the process of allocation and deallocation of memory resources in a computer system.
#Python's internal memory manager manages memory using reference counting and garbage collection.


#Ans.12
#Exception Handling handles errors that occur during the execution of a program.
#Exception handling allows to respond to the error, instead of crashing the running program.
#It enables you to catch and manage errors, making your code more robust and user-friendly


#Ans.13
#Without proper memory management, you can face challenges while building these applications,
#as effective memory allocation is necessary in order to avoid problems that can arise when you run out of memory, such as memory leaks


#Ans.14
#If there is an exception, the try clause is skipped, and the except clause is executed.
#If the except clause within the code doesn't handle the exception, it is passed on to the outer try statement.
#If the exception is not handled, the execution will stop.
#A try statement can include more than one except clause.


#Ans.15
#Python's garbage collection automatically cleans up any unused objects based on reference counting and object allocation and deallocati
# meaning users won't have to clean these objects manually. This also helps periodically clear up memory space to help a program run mor
```

```
#Ans.16
#The 'else' block is useful when you want to perform specific actions when no exceptions occur.
#It can be used, for example, to execute additional code if the 'try' block succeeds in its operation and enhances the program flow.


#Ans.17
#Python's built-in logging module provides a set of standard logging levels to categorize the severity of messages.
#These levels help in filtering and managing log output based on the importance of the events being recorded.


#Ans.18
#os.fork() is a low-level Unix-only system call that duplicates the current process.
#multiprocessing is a high-level, cross-platform Python module that simplifies creating and managing separate processes.
#It provides tools like Process, Queue, and Pool, making parallel programming easier and portable, unlike the manual control required wi


#Ans.19
#Closing a file in Python is crucial for several reasons, primarily related to resource management, data integrity, and program stabilit


#Ans.20
#The `read()` method is used to read a specified number of characters from a file or input stream,
#while the `readline()` method is used to read a single line from a file or input stream.


#Ans.21
#Python import logging is a powerful tool for debugging and troubleshooting code. By default, Python will log all messages to the standa
#However, it is also possible to configure Python to log messages to a file, or even to a remote server.


#Ans.22
#Python has a built-in os module with methods for interacting with the operating system, like creating files and directories,
#management of files and directories, input, output, environment variables, process management, etc


#Ans.23
#Poor memory management can lead to various issues such as memory leaks, fragmentation, excessive paging, and crashes, which indirectly


#Ans.24
#To manually raise an exception in Python, you can use the raise keyword. This keyword allows you to throw an exception with a custom er


#Ans.25
#Improved Performance: Multithreading improves the performance of applications by running the task simultaneously.

#Better Responsiveness: Multithreading improves the responsiveness of your backend applications by running threads after a task is block
#It helps in handling multiple user requests in a real-time application.

#Scalability: Multithreading improves the scalability of an application by letting developers add more processors.
#It is useful when handling a large, growing number of users on a server.

#Latency: Multithreading reduces the time of user requests in applications that need immediate attention.
#The threads work in sync to process the requests and responses.

#Real-time processing: Multithreading ensures tasks or requests are executed with little delay and
#maintains a smooth performance during real-time data processing.
```

Start coding or generate with AI.

Practical Questions

#Ans.1

```
with open('example.txt', 'w') as file:
    file.write('Hello, world!')
```

#Ans.2

```
# Open the file in read mode ('r')
with open('example.txt', 'r') as file:
    # Loop through each line in the file
    for line in file:
        print(line, end='')
```

```
Hello, world!
```

#Ans.3

```
try:
    with open('example.txt', 'r') as file:
        for line in file:
            print(line, end='')
```

```
except FileNotFoundError:
    print("The file 'example.txt' does not exist.")
```

```
Hello, world!
```

```
#Ans.4
# Define source and destination file names
source_file = 'source.txt'
destination_file = 'destination.txt'

try:
    with open(source_file, 'r') as src:
        # Read all content
        content = src.read()

    with open(destination_file, 'w') as dest:
        # Write the content
        dest.write(content)

    print(f"Content copied from '{source_file}' to '{destination_file}'.")
except FileNotFoundError:
    print(f"The file '{source_file}' does not exist.")
except PermissionError:
    print("You do not have the necessary permissions to access the files.")
```

```
The file 'source.txt' does not exist.
```

```
#Ans.5
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

```
Error: Cannot divide by zero.
```

```
#Ans.6
import logging

# Configure logging
logging.basicConfig(
    filename='error.log',
    level=logging.ERROR,
    format='%(asctime)s - %(levelname)s - %(message)s'  # Log format
)

# Example function that causes division by zero
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        logging.error("Division by zero error: %s", e)
        print("An error occurred. Check the log file for details.")

# Call the function
result = divide(10, 0)
```

```
An error occurred. Check the log file for details.
```

```
#Ans.7
import logging

# Configure logging
logging.basicConfig(
    filename='app.log',
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# Log messages at different levels
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.")
```

```
#Ans.8
def read_file(filename):
    try:
        with open(filename, 'r') as file:
            content = file.read()
            print("File content:\n", content)
    except FileNotFoundError:
        print(f"Error: The file '{filename}' was not found.")
    except PermissionError:
        print(f"Error: Permission denied while trying to open '{filename}'.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

# Example usage
read_file("nonexistent_file.txt")
```

```
Error: The file 'nonexistent_file.txt' was not found.
```

```
#Ans.9
file_path = 'example.txt'

with open(file_path, 'r') as file:
    lines = file.readlines()

# Optional: remove newline characters
lines = [line.strip() for line in lines]

print(lines)
```

```
#Ans.10
with open('example.txt', 'a') as file:
    file.write("This line will be added at the end.\n")
```

```
#Ans.11
my_dict = {'name': 'Alice', 'age': 30}

try:
    # Attempt to access a key that might not exist
    value = my_dict['address']
    print("Address:", value)
except KeyError:
    print("Error: The key 'address' does not exist in the dictionary.")
```

```
Error: The key 'address' does not exist in the dictionary.
```

```
#Ans.12
def divide_numbers(a, b):
    try:
        result = a / b
        print(f"Result: {result}")

        # Trying to access an index that may not exist
        sample_list = [1, 2, 3]
        print(sample_list[a])

    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    except IndexError:
        print("Error: List index out of range.")
    except TypeError:
        print("Error: Unsupported operand type.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

# Test cases
divide_numbers(10, 0)
divide_numbers(5, 2)
divide_numbers('a', 1)
```

```
Error: Cannot divide by zero.
Result: 2.5
Error: List index out of range.
Error: Unsupported operand type.
```

```
#Ans.13
import os

file_path = 'example.txt'
```

```
if os.path.exists(file_path):
    with open(file_path, 'r') as file:
        contents = file.read()
        print(contents)
else:
    print("File does not exist.")
```

```
File does not exist.
```

```
#Ans.14
import logging

# Configure logging
logging.basicConfig(
    filename='app.log',
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

def divide(a, b):
    logging.info(f"Attempting to divide {a} by {b}")
    try:
        result = a / b
        logging.info(f"Division successful: {result}")
        return result
    except ZeroDivisionError:
        logging.error("Error: Division by zero attempted!")
        return None

# Example usage
divide(10, 2)
divide(5, 0)
```

```
#Ans.15
def print_file_content(filename):
    try:
        with open(filename, 'r') as file:
            content = file.read()
            if content:
                print("File content:\n", content)
            else:
                print(f"The file '{filename}' is empty.")
    except FileNotFoundError:
        print(f"Error: The file '{filename}' does not exist.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

# Example usage
print_file_content('example.txt')
```

```
File content:
 Hello, world!This line will be added at the end.
```

```
#Ans.16
from memory_profiler import profile

@profile
def combine_lists(n):
    a = [i for i in range(n)]
    b = [i * 2 for i in range(n)]
    c = a + b
    del b
    return c

if __name__ == "__main__":
    combine_lists(n=10**6)
```

```
#Ans.17
numbers = [1, 2, 3, 4, 5, 10, 20, 100]

file_path = 'numbers.txt'

with open(file_path, 'w') as file:
    for number in numbers:
        file.write(f"{number}\n")

print(f"Successfully wrote {len(numbers)} numbers to '{file_path}'")
```

```
Successfully wrote 8 numbers to 'numbers.txt'
```

```
#Ans.18
import logging
from logging.handlers import RotatingFileHandler

# Create logger
logger = logging.getLogger("MyLogger")
logger.setLevel(logging.DEBUG)  # Set to DEBUG to log all levels

# Create a rotating file handler
log_file = "app.log"
max_bytes = 1 * 1024 * 1024  # 1 MB
backup_count = 5  # Keep up to 5 backup files

handler = RotatingFileHandler(
    log_file, maxBytes=max_bytes, backupCount=backup_count
)

# Create a logging format
formatter = logging.Formatter(
    '%(asctime)s - %(levelname)s - %(message)s'
)
handler.setFormatter(formatter)

# Add the handler to the logger
logger.addHandler(handler)

# Example log messages
logger.debug("This is a debug message")
logger.info("This is an info message")
logger.warning("This is a warning")
logger.error("This is an error message")
logger.critical("This is critical")


#Ans.19
def access_elements():
    my_list = [1, 2, 3]
    my_dict = {'a': 10, 'b': 20}

    try:
        # Access list element (may raise IndexError)
        print(my_list[5])

        # Access dictionary key (may raise KeyError)
        print(my_dict['c'])

    except IndexError:
        print("Error: List index out of range.")
    except KeyError:
        print("Error: Key not found in dictionary.")

access_elements()
```

```
Error: List index out of range.
```

```
#Ans.20
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

```
Hello, world!This line will be added at the end.
```

```
#Ans.21
def count_word_occurrences(filename, target_word):
    try:
        with open(filename, 'r') as file:
            content = file.read().lower()
            words = content.split()
            count = words.count(target_word.lower())
            print(f"The word '{target_word}' occurs {count} times in the file.")
    except FileNotFoundError:
        print(f"Error: The file '{filename}' does not exist.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Example usage
count_word_occurrences('example.txt', 'python')
```

```
The word 'python' occurs 0 times in the file.
```

```
#Ans.22
import os

file_path = 'example.txt'

if os.path.exists(file_path) and os.stat(file_path).st_size == 0:
    print("File is empty.")
else:
    print("File is not empty.")
```

```
File is not empty.
```

```
#Ans.23
import logging

logging.basicConfig(
    filename='file_errors.log',
    level=logging.ERROR,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

def read_file(filename):
    try:
        with open(filename, 'r') as file:
            content = file.read()
            print(content)
    except Exception as e:
        logging.error(f"Error occurred while handling file '{filename}': {e}")
        print(f"An error occurred. Check 'file_errors.log' for details.")

# Example usage
read_file('nonexistent_file.txt')
```

```
An error occurred. Check 'file_errors.log' for details.
```

Start coding or generate with AI.