

Exceptional Handling Logging (module 7)

Theoretical Questions

Q1. What is the difference between interpreted and compiled languages?

Ans-> The key difference between interpreted and compiled languages lies in how they process code: interpreted languages execute code line by line, while compiled languages translate the entire code into machine code before execution.

Interpreted Languages:

- Execution: The interpreter reads the source code, interprets each line, and executes the corresponding instructions immediately.
- Process: There's no separate compilation step; the source code is directly executed by the interpreter.
- Examples: Python, JavaScript, Ruby, Perl, and PHP.
- Compiled Languages:
- Execution: The compiler translates the entire source code into machine code (or bytecode) before execution, creating an executable file.
- Process: The compiled code is then executed by the CPU.
- Examples: C, C++, Go.

Q2. What is exception handling in Python?

Ans-> In Python, exception handling is the process of managing errors that occur while a program is running. It allows programs to continue running smoothly instead of crashing.

Q3. What is the purpose of the finally block in exception handling?

Ans-> The finally block in Python's exception handling ensures that a specific block of code (like closing files or releasing resources) is always executed, regardless of whether an exception occurred within the try block or not.

- Here's a more detailed explanation:
- Guaranteed Execution: The finally block is executed after the try block and any associated except blocks, whether or not an exception was raised.
- Resource Management: Its primary purpose is to perform cleanup tasks, such as closing files, releasing database connections, or releasing other resources, ensuring that these

actions are always performed, even if an exception occurs.

Q4. What is logging in Python?

Ans-> In Python, logging is a process of systematically recording events, such as errors, warnings, and informational messages, during the execution of a program, which helps developers track behavior, debug issues, and monitor application health.

Q5. What is significance of the **del** method in Python?

Ans-> In Python, the **del** method, also known as a destructor, is called by the garbage collector when an object is about to be destroyed, allowing for cleanup of resources held by the object.

- Here's a more detailed explanation:
- Purpose: The primary purpose of **del** is to perform any necessary cleanup tasks when an object is no longer in use, such as closing files, releasing database connections, or clearing up buffers.
- Automatic Invocation: Python's garbage collector automatically calls **del** when an object's reference count drops to zero, indicating that the object is no longer accessible.
- Timing: The timing of **del** execution is not deterministic, meaning it might be called at any time after the object is no longer referenced, or potentially not at all if the program terminates before garbage collection occurs.
- Not a Guarantee: It's crucial to understand that **del** is not a reliable mechanism for resource management, as its execution is not guaranteed. For reliable resource management, consider using context managers (with/as) or the atexit module.

Q6. What is the difference between import and from ...import in Python?

Ans-> 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.

Q7. How can handle multiple exceptions in Python?

Ans-> Python allows you to catch multiple exceptions in a single 'except' block by specifying them as a tuple. This feature is useful when different exceptions require similar handling logic. In this case, if either 'ExceptionType1' or 'ExceptionType2' is raised, the code within the 'except' block will be executed.

Q8. What is the purpose of the with statement when handling files in Python?

Ans-> The with statement in Python, when used with file handling, ensures that a file is automatically closed after its operations are complete, even if errors occur, simplifying resource management and making code cleaner and more robust.

- Here's a breakdown:
- Resource Management: The with statement is a context manager, meaning it handles the setup and cleanup of resources (like files) automatically.
- Automatic Closing: When you use with open(...) as file_object: the file is opened, and the file_object is available within the indented block. After the block finishes, the file is automatically closed, regardless of whether an exception occurred.
- Error Handling: Even if an error occurs during the operations within the with block, the file will still be closed, preventing resource leaks and potential issues.
- Simplified Code: Using with eliminates the need for explicit try...finally blocks to ensure file closure, making your code more concise and readable.

Q9. What is the difference between multithreading and multiprocessing?

Ans-> In Python, multithreading allows concurrent execution of tasks within a single process, while multiprocessing enables parallel execution of tasks across multiple processes.

Multithreading is useful for I/O-bound tasks, while multiprocessing is better for CPU-bound tasks.

- Here's a more detailed breakdown:
- Multithreading:
 - Concurrency: Multiple threads can execute seemingly simultaneously within the same process.
 - Shared Memory: Threads within a process share the same memory space, making data sharing easier.
 - Global Interpreter Lock (GIL): In CPython (the standard Python implementation), the GIL limits true parallelism for CPU-bound tasks, as only one thread can hold control of the Python interpreter at any given time.
 - Use Cases: I/O-bound tasks (e.g., network operations, file I/O), GUI applications. Example: Imagine a web server handling multiple client requests concurrently. Each request can be handled by a separate thread, allowing the server to process multiple requests at the same time.
- Multiprocessing:
 - Parallelism: Multiple processes can execute independently on different CPU cores, achieving true parallelism.

- **Separate Memory Spaces:** Each process has its own memory space, requiring explicit mechanisms for data sharing between processes.
- **No GIL Limitation:** Processes are not subject to the GIL, allowing for true parallel execution of CPU-bound tasks.
- **Use Cases:** CPU-bound tasks (e.g., image processing, scientific simulations), tasks that require high computational power.

Q10. What are the advantages of using logging in a program?

Ans-> Python logging is a module that allows you to track events that occur while your program is running. You can use logging to record information about errors, warnings, and other events that occur during program execution. And logging is a useful tool for debugging, troubleshooting, and monitoring your program.

Q11. What is memory management in Python?

Ans-> In Python, memory management is handled automatically by the interpreter using a private heap, reference counting, and a garbage collector, allowing developers to focus on code logic rather than manual memory allocation and deallocation.

Q12. What are the basic steps involved in exception handling in Python?

Ans-> In Python, exception handling involves using try, except, else, and finally blocks to gracefully manage errors and prevent program crashes. The try block contains code that might raise an exception, the except block handles specific exceptions, the else block executes if no exception occurs, and the finally block always executes for cleanup.

Q13. Why is memory management important in Python?

Ans-> Memory management is crucial in Python because it directly impacts program performance, resource usage, and stability, especially for large-scale applications, and is handled automatically by the Python interpreter, allowing developers to focus on code logic rather than memory allocation.

Q14. What is the role of try and except in exception handling?

Ans-> The try block lets you test a block of code for errors. The except block lets you handle the error. The finally block lets you execute code, regardless of the result of the try- and except blocks.

Q15. How does Python's garbage collection system work?

Ans-> Python's garbage collection system automatically manages memory by reclaiming memory occupied by objects that are no longer referenced, primarily using reference counting and a cyclic garbage collector to handle circular references.

Q16. What is the purpose of the else block in exception handling?

Ans-> 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.

Q17. What are the common logging levels in Python?

Ans-> The specific log levels available to you may differ depending on the programming language, logging framework, or service in use. However, in most cases, you can expect to encounter levels such as FATAL , ERROR , WARN , INFO , DEBUG , and TRACE .

Q18. What is the difference between os.fork() and multiprocessing in Python?

Ans-> The only real difference between the os.fork and multiprocessing.Process is portability and library overhead, since os.fork is not supported in windows, and the multiprocessing framework is included to make multiprocessing.Process work. This is because os.fork is called by multiprocessing.Process

Q19. What is the importance of closing a file in Python?

Ans-> Closing files in Python is an essential practice that helps maintain data integrity, prevent resource leaks, and ensure the reliability of your applications. By mastering file handling techniques, you can write more robust and efficient Python code that effectively manages file resources.

Q20. What is the difference between file.read() and file.readline() in Python

Ans-> In Python, read() reads the entire file content as a single string, while readline() reads a single line from the file at a time, returning it as a string.

- Here's a more detailed explanation:
- read(): Reads the entire file content into a single string. If no argument is provided, it reads the entire file. If a size argument is provided, it reads that many bytes from the current file position. Can be less efficient for large files as it loads the entire file into memory.
- readline(): Reads a single line from the file, up to the newline character (\n). Returns the line as a string. If the end of the file is reached, it returns an empty string. Suitable for

reading files line by line.

Q21. What is the logging module in Python used for?

Ans-> The Python logging module is used to track events, debug issues, and monitor the health of Python applications by capturing and storing information about program execution. It's a powerful tool for developers to understand application behavior and troubleshoot problems.

Q22. What is the os module in Python used for in file handling?

Ans-> 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.

Q23. What are the challenges associated with memory management in Python?

Ans-> Python's automatic memory management, while simplifying development, presents challenges like potential memory leaks due to circular references, performance overhead from garbage collection, and difficulty in optimizing memory usage for specific applications.

Q24. How do you raise an exception manually in Python?

Ans-> As a Python developer you can choose to throw an exception if a condition occurs. To throw (or raise) an exception, use the raise keyword. In Python, you can raise either built-in or custom exceptions. When you raise an exception, the result is the same as when Python does it. You get an exception traceback, and your program crashes unless you handle the exception on time.

Q25. Why is it important to use multithreading in certain applications?

Ans-> Multithreading is crucial for certain applications as it enables concurrent execution of tasks, leading to improved performance, resource utilization, and responsiveness, especially in applications involving I/O or waiting for external events.

- Here's a more detailed explanation of why multithreading is important:
- Improved Performance: By allowing multiple tasks to run concurrently, multithreading can significantly improve the overall performance of an application, especially on multi-core or multi-processor systems.
- Enhanced Responsiveness: In applications with a user interface (UI), multithreading allows the UI to remain responsive even when long-running tasks are being performed in the background, preventing the UI from freezing or becoming unresponsive.

- **Better Resource Utilization:** Multithreading can help optimize the utilization of CPU resources by allowing different threads to execute concurrently, potentially leading to higher throughput and efficiency.
- **Concurrency:** Multithreading enables applications to handle multiple tasks or requests simultaneously, which is essential for applications that need to serve multiple users or perform multiple operations at the same time.
- **Scalability:** Multithreading plays a vital role in building scalable applications that can handle increasing workloads or a large number of users without sacrificing performance.
- **Utilizing Microprocessor Architecture:** Multithreading can leverage the parallelism inherent in modern microprocessor architectures, allowing applications to take full advantage of the available processing power.
- **Improved System Reliability:** By isolating tasks into separate threads, multithreading can help improve system reliability and fault tolerance, as a failure in one thread is less likely to cause the entire application to crash.
- **Web Servers:** Web servers often use multithreading to handle multiple client requests concurrently, ensuring that the server can respond to a large number of users simultaneously.
- **Image Analysis:** Image analysis algorithms can use multithreading to process different parts of an image concurrently, speeding up the overall processing time.
- **Background Tasks:** Multithreading is used for performing background tasks, such as downloading data, processing user input, or performing long-running calculations, without blocking the main application thread

Practical Questions

✓ Q1. How can you open a file for writing in Python and write a string to it?

prompt: How can you open a file for writing in Python and write a string to it

```
def write_string_to_file(filename, text):  
    """Opens a file for writing and writes a string to it.  
  
    Args:  
    filename: The name of the file to write to.  
    text: The string to write to the file.  
    """  
    try:  
        with open(filename, 'w') as file:  
            file.write(text)  
    except Exception as e:  
        print(f"An error occurred: {e}")
```

```
# Example usage
write_string_to_file('output.txt', 'This is the text to write to the file.')
```

✓ Q2. Write a Python program to read the contents of a file and print each line

prompt: Write a Python program to read the contents of a file and print each line

```
def read_and_print_lines(filename):
    """Reads a file and prints each line.

    Args:
        filename: The name of the file to read.
    """
    try:
        with open(filename, 'r') as file:
            for line in file:
                print(line, end='') # end='' prevents extra newline
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Example usage (assuming the file 'output.txt' exists from the previous example):
read_and_print_lines('output.txt')
```

➡ This is the text to write to the file.

✓ Q3. How would you handle a case where the file doesn't exist while trying to open it for reading


```
def read_file_safely(filename):
    """Reads a file and handles the case where it doesn't exist.

    Args:
        filename: The name of the file to read.
    """
    try:
        with open(filename, 'r') as file:
            contents = file.read()
            print(contents)
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")
```

✓ Q4. Write a Python script that reads from one file and writes its content to another file

prompt: Write a Python script that reads from one file and writes its content to another

```
def copy_file_contents(source_filename, destination_filename):
    """Reads from one file and writes its content to another file.

    Args:
        source_filename: The name of the file to read from.
        destination_filename: The name of the file to write to.
    """
    try:
        with open(source_filename, 'r') as source_file:
            with open(destination_filename, 'w') as destination_file:
                for line in source_file:
                    destination_file.write(line)
    except FileNotFoundError:
        print(f"Error: Source file '{source_filename}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Example usage
copy_file_contents('source.txt', 'destination.txt')
```

➞ Error: Source file 'source.txt' not found.

✓ Q5. How would you catch and handle division by zero error in Python

prompt: How would you catch and handle division by zero error in Python

```
def safe_division(numerator, denominator):
    """Performs division and handles ZeroDivisionError.

    Args:
        numerator: The numerator.
        denominator: The denominator.

    Returns:
        The result of the division, or an error message if division by zero occurs.
    """
    try:
        result = numerator / denominator
        return result
    except ZeroDivisionError:
        return "Error: Division by zero."
    except TypeError:
        return "Error: Invalid input types." # Added for more robust error handling
    except Exception as e:
        return f"An unexpected error occurred: {e}"

# Example usage
print(safe_division(10, 2)) # Output: 5.0
print(safe_division(10, 0)) # Output: Error: Division by zero.
print(safe_division(10, "a")) #Output: Error: Invalid input types.
```



```
5.0
Error: Division by zero.
Error: Invalid input types.
```

✓ Q6. Write a Python program that logs an error message to a log file when a division by zero exception occurs

```
import logging

def division_with_logging(numerator, denominator):
    """Performs division and logs a division by zero error to a log file."""

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

    try:
        result = numerator / denominator
        return result
    except ZeroDivisionError:
        logging.error("Division by zero occurred.") # Log the error
        return "Error: Division by zero."
    except Exception as e:
        logging.exception(f"An unexpected error occurred: {e}") #Log other exceptions
```

```
return f"An unexpected error occurred: {e}"
```

```
# Example Usage
```

```
print(division_with_logging(10, 2)) # Output: 5.0
print(division_with_logging(10, 0)) # Output: Error: Division by zero. (and logs error)
print(division_with_logging(10, "a")) # Output: An unexpected error occurred (and logs er
```

```

ERROR:root:Division by zero occurred.
ERROR:root:An unexpected error occurred: unsupported operand type(s) for /: 'int' and
Traceback (most recent call last):
  File "<ipython-input-8-47617724f1d5>", line 11, in division_with_logging
    result = numerator / denominator
              ~~~~~^~~~~~
TypeError: unsupported operand type(s) for /: 'int' and 'str'
5.0
Error: Division by zero.
An unexpected error occurred: unsupported operand type(s) for /: 'int' and 'str'

```

Q7. How do you log information at different levels (Info, ERROR, WARNING) in Python using the logging module

```
import logging
```

```

def log_different_levels():
    # Configure logging
    logging.basicConfig(filename='app.log', level=logging.DEBUG, # Set the root logger l
                        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.')

```

```
log_different_levels()
```

```


WARNING:root:This is a warning message.
ERROR:root:This is an error message.
CRITICAL:root:This is a critical message.

```

Q8. Write a program to handle a file opening error using exception handling

```
def handle_file_opening_error(filename):
    """Handles file opening errors using exception handling."""
    try:
        with open(filename, 'r') as file:
            contents = file.read()
            print(contents)
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
    except PermissionError:
        print(f"Error: You don't have permission to open '{filename}'.")
    except Exception as e:
        print(f"An unexpected error occurred while opening '{filename}': {e}")

# Example usage
handle_file_opening_error("my_file.txt") # Replace with an actual filename
```

 Error: File 'my_file.txt' not found.

✓ Q9. How can you read a file line and store its content in a list in Python


```
def read_file_into_list(filename):
    """Reads a file line by line and stores its content in a list.

    Args:
        filename: The name of the file to read.

    Returns:
        A list of strings, where each string is a line from the file,
        or None if an error occurs.
    """
    try:
        with open(filename, 'r') as file:
            lines = file.readlines()
            return lines
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None

# Example usage
file_content = read_file_into_list("my_file.txt") # Replace with your filename

if file_content:
    for line in file_content:
        print(line.strip()) # .strip() removes leading/trailing whitespace
```

 Error: File 'my_file.txt' not found.

✓ Q10. How can you append data to an existing file in Python

prompt: How can you append data to an existing file in Python

```
def append_to_file(filename, text):
    """Appends text to an existing file. Creates the file if it doesn't exist.

    Args:
        filename: The name of the file.
        text: The text to append.
    """
    try:
        with open(filename, 'a') as file: # Open in append mode ('a')
            file.write(text)
    except Exception as e:
        print(f"An error occurred: {e}")

# Example usage
append_to_file("my_file.txt", "\nThis is appended text.")
```

Q11. Write a Python program that use a try-except block to

✓ handle an error when attempting to access a dictionary key that doesn't exist

prompt: Write a Python program that use a try-except block to handle an error when atte

```
def access_dictionary_safely(my_dict, key):
    """Accesses a dictionary key safely using a try-except block.

    Args:
        my_dict: The dictionary to access.
        key: The key to look up.

    Returns:
        The value associated with the key if it exists, otherwise None.
    """
    try:
        value = my_dict[key]
        return value
    except KeyError:
        print(f"Error: Key '{key}' not found in the dictionary.")
        return None
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        return None
```

```
# Example usage
my_dict = {"a": 1, "b": 2, "c": 3}

print(access_dictionary_safely(my_dict, "b")) # Output: 2
print(access_dictionary_safely(my_dict, "d")) # Output: Error: Key 'd' not found... and
```

```
2
Error: Key 'd' not found in the dictionary.
None
```

Q12. Write a program that demonstrates using multiple except blocks to handle different types of exceptions

prompt: Write a program that demonstrates using multiple except blocks to handle differ

```
def handle_multiple_exceptions():
    try:
        # Code that might raise different exceptions
        result = 10 / 0 # ZeroDivisionError
        # my_list = [1, 2, 3]
        # print(my_list[5]) # IndexError
        # my_dict = {"a": 1, "b": 2}
        # print(my_dict["c"]) # KeyError
        # int("abc") # ValueError
    except ZeroDivisionError:
        print("Error: Division by zero.")
    except IndexError:
        print("Error: List index out of range.")
    except KeyError:
        print("Error: Dictionary key not found.")
    except ValueError:
        print("Error: Invalid value.")
    except Exception as e: # Catch-all for other exceptions
        print(f"An unexpected error occurred: {e}")

# Example usage
handle_multiple_exceptions()
```

```
Error: Division by zero.
```


Q13. How would you check if a file exists before attempting to read it in Python

prompt: How would you check if a file exists before attempting to read it in Python

```
import os

def read_file_if_exists(filename):
    """Reads a file only if it exists."""
    if os.path.exists(filename):
        try:
            with open(filename, 'r') as file:
                contents = file.read()
                print(contents)
        except Exception as e:
            print(f"An error occurred while reading the file: {e}")
    else:
        print(f"Error: File '{filename}' not found.")

# Example usage
read_file_if_exists("my_file.txt")
```

 Error: File 'my_file.txt' not found.

✓ Q14. Write a program that uses the logging module to log both informational and error messages

prompt: Write a program that uses the logging module to log both informational and error messages
import logging # Import the logging module at the beginning


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

    try:
        # Some code that might produce informational or error messages
        result = 10 / 2 # Successful operation
        logging.info("Division successful. Result: %s", result) # Log informational message

        result = 10 / 0 # Error
        logging.info("Division successful. Result: %s", result) # This won't be printed

    except ZeroDivisionError:
        logging.error("Division by zero error occurred!") # Log an error message
    except Exception as e: # catch all other exceptions
        logging.exception(f"An unexpected error occurred: {e}")

log_info_and_error()
```

 ERROR:root:Division by zero error occurred!

✓ Q15. Write a Python program that prints the content of a file and handles the case when the file is empty

```
def print_file_content(filename):
    """Prints the content of a file and handles empty files."""
    try:
        with open(filename, 'r') as file:
            contents = file.read()
            if contents: # Check if the file has content
                print(contents)
            else:
                print(f"The file '{filename}' is empty.")
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Example usage
print_file_content("my_file.txt") # Replace "my_file.txt" with the actual filename
```

➞ Error: File 'my_file.txt' not found.

✓ Q16. Demonstrate how to use memory profiling to check the memory usage of a small program

```
# prompt: Demonstrate how to use memory profiling to check the memory usage of a small pr

!pip install memory_profiler

%load_ext memory_profiler

# Example function to profile (replace with your function)
def my_function():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

%memit my_function()
```

➞ Collecting memory_profiler
 Downloading memory_profiler-0.61.0-py3-none-any.whl.metadata (20 kB)
 Requirement already satisfied: psutil in /usr/local/lib/python3.11/dist-packages (from memory_profiler) (5.9.0)
 Downloading memory_profiler-0.61.0-py3-none-any.whl (31 kB)
 Installing collected packages: memory_profiler
 Successfully installed memory_profiler-0.61.0
 peak memory: 260.88 MiB, increment: 158.16 MiB

✓ Q17. Write a Python program to create and write a list of numbers to a file, one number per line

prompt: Write a Python program to create and write a list of numbers to a file, one num

```
def write_numbers_to_file(filename, numbers):
    """Writes a list of numbers to a file, one number per line.

    Args:
        filename: The name of the file to write to.
        numbers: A list of numbers.
    """
    try:
        with open(filename, 'w') as file:
            for number in numbers:
                file.write(str(number) + '\n')
    except Exception as e:
        print(f"An error occurred: {e}")

# Example usage
numbers = [1, 2, 3, 4, 5]
write_numbers_to_file("numbers.txt", numbers)
```

✓ Q18. How would implement a basic logging setup that logs to a file, with rotation after 1MB

```
import logging
from logging.handlers import RotatingFileHandler # Import RotatingFileHandler

def setup_rotating_log(log_filename, max_bytes=1024*1024): # 1MB
    """Sets up a rotating log handler."""

    # Create a logger
    logger = logging.getLogger("my_rotating_logger")
    logger.setLevel(logging.DEBUG) # Set the logging level

    # Create a rotating file handler
    handler = RotatingFileHandler(log_filename, maxBytes=max_bytes, backupCount=5) # Keep
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
    handler.setFormatter(formatter)
    logger.addHandler(handler)

    return logger

# Example usage:
```

```
logger = setup_rotating_log('my_app.log')

# Now use the logger
logger.debug('This is a debug message')
logger.info('This is an info message')
logger.warning('This is a warning message')
logger.error('This is an error message')
logger.critical('This is a critical message')

➡ DEBUG:my_rotating_logger:This is a debug message
   INFO:my_rotating_logger:This is an info message
   WARNING:my_rotating_logger:This is a warning message
   ERROR:my_rotating_logger:This is an error message
   CRITICAL:my_rotating_logger:This is a critical message
```

✓ Q19. Write a program that handles both `IndexError` and `keyerror` using a try-except block

prompt: Write a program that handles both Index Error and key error using a try-except

```
def handle_errors(data_structure, index_or_key):
    """Handles IndexError and KeyError exceptions.

    Args:
        data_structure: The data structure (list or dictionary) to access.
        index_or_key: The index or key to use for access.
    """
    try:
        if isinstance(data_structure, list):
            value = data_structure[index_or_key]
            print("Value at index", index_or_key, ":", value)
        elif isinstance(data_structure, dict):
            value = data_structure[index_or_key]
            print("Value at key", index_or_key, ":", value)
        else:
            print("Unsupported data structure type.")
    except IndexError:
        print("Error: Index out of range.")
    except KeyError:
        print("Error: Key not found in the dictionary.")
```

Example usage:

```
my_list = [1, 2, 3]
```

```
my_dict = {"a": 1, "b": 2}
```

```
handle_errors(my_list, 1) # Valid index
handle_errors(my_list, 5) # Invalid index - IndexError
handle_errors(my_dict, "a") # Valid key
handle_errors(my_dict, "c") # Invalid key - KeyError
handle_errors("hello", 0) # Unsupported data structure type
```

```
➡ Value at index 1 : 2
Error: Index out of range.
Value at key a : 1
Error: Key not found in the dictionary.
Unsupported data structure type.
```

✓ Q20. How would you open a file and read its contents using a context manager in Python

```
def read_file_with_context_manager(filename):
    """Reads a file using a context manager."""
    try:
        with open(filename, 'r') as file:
            contents = file.read()
            return contents
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
```

```

    return None
except Exception as e:
    print(f"An error occurred: {e}")
    return None

```

✓ Q21. Write a Python program that reads a file and prints the number of occurrences of a specific word

```

def count_word_occurrences(filename, target_word):
    """Reads a file and counts the occurrences of a specific word.

    Args:
        filename: The name of the file to read.
        target_word: The word to count occurrences of.

    Returns:
        The number of times the target word appears in the file, or None if an error occurs
    """
    try:
        with open(filename, 'r') as file:
            contents = file.read()
            words = contents.split() # Split the content into words
            count = words.count(target_word)
            return count
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None
    except Exception as e:
        print(f"An error occurred: {e}")
        return None

# Example usage
filename = "your_file.txt" # Replace with the actual filename
target_word = "logging" # Replace with the word you want to count
occurrences = count_word_occurrences(filename, target_word)

if occurrences is not None:
    print(f"The word '{target_word}' appears {occurrences} times in the file.")

```

➡ Error: File 'your_file.txt' not found.

✓ Q22. How can you check if a file is empty before attempting to read its contents

```

def read_file_if_not_empty(filename):
    """Reads a file only if it's not empty."""

```

```
try:
    with open(filename, 'r') as file:
        # Efficiently check for emptiness without reading the whole file
        if os.stat(filename).st_size == 0:
            print(f"The file '{filename}' is empty.")
            return # Exit early if empty

        contents = file.read()
        print(contents)
except FileNotFoundError:
    print(f"Error: File '{filename}' not found.")
except Exception as e:
    print(f"An error occurred while reading the file: {e}")
```