Files and Exception Handling Assignment
Course: Data Science with Gen AI (May 2025)
Anoop Verma (anoopmuz@gmail.com

Q(1) What is the difference between interpreted and compiled languages?

Ans:- Compilers and Interpreters - both are softwares which are classified as Language Translators and they both translate the computer program written in any high level language to machine-understandable language.

The main difference between these two are as follows:

** Compilers**

- The source code is converted into machine code (executable binary) in one go before execution by a compiler.
- Creates a separate compiled binary file that contains the machine understandable code in it which is directly executed on the CPU.
- Generally faster because the code is optimized before execution.
- Harder to debug because errors are checked only after compilation.
- Compiled binaries are platform-dependent (must be recompiled or linked for different OS/CPU architectures).
- Once the compiled binary file is created, no need of source code for execution.
- Examples: C, C++, COBOL, Pascal, Go, Rust, Swift.

Interpreters

- The source code is translated line-by-line and executed by an interpreter and then the next line is translated and executed and so on.
- No separate compilation step; the interpreter reads and executes code on the fly.
- Slower because each line is translated during execution.
- Easier since errors are reported immediately.
- Source code has to be present during each execution of the program, so more memory is required.
- More portable—code runs anywhere the interpreter is available.
- Examples: Python, BASIC, JavaScript (in browsers), Ruby, PHP.

Q(2) What is exception handling in Python?

Ans:- Exception handling in Python is a mechanism to manage runtime errors (exceptions), preventing the program from crashing unexpectedly. It allows us to detect, handle, and recover from errors in a controlled way.

Exception An event that disrupts the normal flow of a program (e.g., division by zero, file not found). *Try-Except Block* Catches and handles exceptions. *Else Block* Executes if no exception occurs. *Finally Block* Always executes, regardless of exceptions. *Raise* Manually trigger an exception.

Common Exceptions are:

- ZeroDivisionError = Division/modulo by zero
- FileNotFoundError = File does not exist
- IndexError = List index out of range
- KeyError = Dictionary key not found
- ValueError = Invalid value (e.g., int("abc"))
- TypeError = Operation on wrong data type
- NameError = Variable not defined

Q(3) What is the purpose of the finally block in exception handling?

Ans:- The finally block in Python is used to define code that must execute regardless of whether an exception occurs or not. It ensures cleanup or finalization tasks (like closing files, releasing resources, or logging) are always performed.

- Always Executes - Runs whether the try block succeeds, raises an exception, or even if a return statement is encountered.
- Cleanup Operations -Ideal for releasing resources (files, network connections, locks) to prevent memory leaks.
- Overrides return in try or except - If finally contains a return, it overrides any previous return statements.

Q(4) What is logging in Python?

Ans:- Logging is a built-in Python module (logging) used to track events that occur during software execution. It is essential for debugging, monitoring application behavior, and diagnosing issues in production environments. Unlike print() statements (which are temporary and messy), logging provides a structured, configurable, and persistent way to record events.

Q(5) What is the significance of the **del** method in Python?

Ans:- The **del** method is a destructor in Python, automatically called when an object is about to be destroyed (e.g., when it is garbage-collected).

- Purpose - Used for cleanup tasks (e.g., closing files, releasing network connections). Not guaranteed to run immediately when an object goes out of scope (depends on garbage collection).
- When **del** is Called - When the object's reference count drops to zero. When the garbage collector reclaims the object (if cyclic references exist). During program termination (but not always, especially if the interpreter exits abruptly).

Q(6) What is the difference between import and from ... import in Python?

Ans:- Both import and from ... import are used to include external modules or functions in your Python code, but they differ in how they load and reference the imported content.

- import Statement= Imports an entire module, and you must use the module name as a prefix to access its contents.

Syntax: import module_name

- from ... import Statement = Imports specific objects (functions, classes, variables) directly into the current namespace, so you can use them without the module prefix.

Syntax: from module_name import function_name

Q(7) How can you handle multiple exceptions in Python?

Ans:- We can handle multiple exceptions using several approaches depending on our specific needs.

- Separate except blocks (Handle each exception differently)
- Single except block with tuple (Handle multiple exceptions the same way)
- Exception hierarchy (Handle base exceptions)
- Multiple exceptions with different handling in one block
- Re-raising exceptions with context
- Using else and finally

Q(8) What is the purpose of the with statement when handling files in Python?

Ans:- The "with" statement in Python provides a clean and efficient way to handle resources for file operations. Its primary purpose is to ensure proper acquisition and release of resources, even if exceptions occur.

The following points can be noted while using "with" with files:

- Automatic Resource Management => Opens and closes files automatically. The file is closed when the block inside with ends, even if an error occurs. No need to manually call file.close()
- Exception Safety => If an error (e.g., IOError, PermissionError) occurs inside the with block, Python ensures the file is properly closed before propagating the exception. It also prevents resource leaks (e.g., too many open files).
- Cleaner and More Readable Code => Eliminates boilerplate try-finally blocks for cleanup.
- Supports Context Managers => open() returns a context manager object, which defines **enter**() (setup) and **exit**() (cleanup). The with statement leverages these methods to manage resources.

Q(9) What is the difference between multithreading and multiprocessing?

Ans:- Both multithreading and multiprocessing allow concurrent execution of tasks, but they differ in how they manage system resources, memory, and Python's limitations (like the Global Interpreter Lock (GIL)).

- Memory & Data Sharing =>
  In Multithreading, all threads share the same memory space (variables, objects). So, there is a risk of race conditions (use threading.Lock).
  But in Multiprocessing, each process has its own memory (no shared state by default). So it requires Inter-Process Communication (IPC) (e.g., multiprocessing.Queue, Pipe).
- Global Interpreter Lock (GIL)
  In Multithreading, Python's GIL allows only one thread to execute Python bytecode at a time. I/O-related tasks work well because threads release the GIL while waiting but the CPU-bound tasks (e.g., math) suffer because threads cannot run in parallel.
  In Multiprocessing, each process has its own GIL, enabling true parallelism for CPU-bound tasks.
- Performance
  Multithreading wins because of low overhead (for APIs, file handling etc.)
  Multiprocessing wins for CPU intensive works, like NumPy, ML applications etc.

Q(10) What are the advantages of using logging in a program?

Ans:- Logging is a critical practice in software development that provides structured, persistent records of program execution.

- Debugging & Troubleshooting
- Monitoring & Performance Analysis
- Crash Investigation
- Scalability & Maintenance
- Better Than print() Statements
- Security & Compliance
- Structured Logging (Modern Approach)

Q(11) What is memory management in Python?

Ans:- Memory management in Python refers to how the Python interpreter allocates, uses, and frees memory for objects in your program. Python automates memory management. It optimizes performance and avoids leaks.

- Dynamic Memory Allocation
  Python dynamically allocates memory for objects (integers, lists, classes, etc.).
- Garbage Collection (GC)
  Python uses automatic garbage collection to reclaim unused memory.
- Memory Pools & Object-Specific Allocators
  Python uses private heaps for object storage.
  Small objects (e.g., integers, tuples) use pre-allocated memory pools for efficiency.
  Large objects (e.g., NumPy arrays) use the system allocator.

- Memory Profiling & Optimization
  Tools like tracemalloc, memory_profiler, and objgraph help analyze memory usage.

Q(12) What are the basic steps involved in exception handling in Python?

Ans:- Exception handling in Python allows us to gracefully manage errors and unexpected events without crashing our program.

- Step 1
  Use try and except Blocks
- Step 2
  Catch Specific Exceptions
- Step 3
  Use else for Success Cases
- Step 4
  Clean Up with finally
- Step 5
  Raise Custom Exceptions
- Step 6
  Chain Exceptions with raise from
- Step 7
  Log Exceptions

Q(13) Why is memory management important in Python?

Ans:- Memory management is crucial in Python to ensure efficient resource usage, prevent crashes, and optimize performance.

- Prevents Memory Leaks
- Optimizes Performance
- Avoids Crashes in Resource-Limited Environments
- Handles Cyclic References
- Ensures Scalability
- Facilitates Debugging

Q(14) What is the role of try and except in exception handling?

Ans:- In Python, try and except are fundamental constructs used for exception handling, allowing us to gracefully manage errors and prevent our program from crashing when unexpected situations occur.

Role of try and except:
try Block:

Contains the code that might raise an exception (e.g., dividing by zero, accessing a non-existent file).

If an exception occurs inside try, Python stops executing the rest of the block and jumps to the matching except.

except Block:

Catches and handles the exception raised in the try block.

We can specify which exceptions to catch (e.g., ValueError, TypeError) or use a generic except to catch all exceptions.

If no exception occurs, the except block is skipped.

Thus it helps in

- Preventing Program Crashes: Instead of stopping execution, you can log errors or provide fallback behavior.
- Controlling Error Handling: Different exceptions can be handled differently.
- Cleaning up with finally: Ensures resources (like files or network connections) are properly released.

Q(15) How does Python's garbage collection system work?

Ans:- Python's garbage collection (GC) system automatically manages memory by reclaiming unused objects to free up resources. It primarily uses a combination of reference counting and a generational garbage collector to handle cyclic references.

Reference Counting (Primary Mechanism)
Every object in Python has a reference count, which tracks how many variables or data structures point to it.

When the reference count drops to zero, the object is immediately deallocated.

Generational Garbage Collector (Handles Cycles)
To handle reference cycles (e.g., circular lists or graphs), Python uses a generational GC with three generations:
Generation 0 (Young): Newly created objects.
Generation 1 (Middle): Objects that survive one GC cycle.
Generation 2 (Old): Long-lived objects.

Q(16) What is the purpose of the else block in exception handling?

Ans:- In Python's exception handling, the "else block" serves a specific and often underutilized purpose. It executes only if the try block completes successfully without raising any exceptions. This makes it distinct from finally (which always runs) and except (which runs only on errors).

-Separation of Concerns:
Code in the try block should only include statements that might raise exceptions.
The else block contains code that should run only if no exceptions occurred, keeping the logic clean.

- Avoid Accidental Exception Catching:
  If you place non-risky code in the try block, an unrelated exception might be caught unintentionally by except. The else block prevents this.
- Clarity:
  Makes it explicit which code is "protected" by exception handling and which is part of the happy path.

Q(17) What are the common logging levels in Python?

Ans:- In Python's logging module, logging levels categorize the severity of events, allowing us to filter and handle messages appropriately. Here are the standard levels (from lowest to highest severity):

Common Logging Levels

- DEBUG 10 => Detailed diagnostic info for developers (e.g., variable states).
- INFO 20 => Confirmation that things are working as expected (e.g., "Server started").
- WARNING 30 => Indicates potential issues (e.g., "Low disk space"). Non-critical.
- ERROR 40 => Serious problems that disrupt functionality (e.g., "Failed to open DB").
- CRITICAL 50 => Fatal errors causing program termination (e.g., "Out of memory").

Q(18) What is the difference between os.fork() and multiprocessing in Python?

Ans:- The followings are the differences between os.fork() and multiprocessing in Python.
os.fork()

- Directly calls the Unix/Linux system call fork(), creating a child process that is an exact copy of the parent process (including memory, file descriptors, etc.).
- The child process starts execution from the point where fork() was called.

multiprocessing Module

- Provides a cross-platform way to spawn processes using fork() (Unix) or spawn/forkserver (Windows).
- Each process has isolated memory (no shared state by default).
- Designed for parallelism, with tools like Queue, Pool, and Pipe.

Q(19) What is the importance of closing a file in Python?

Ans:- The Importance of Closing Files in Python:

Resource Management

- File Descriptor Limit:
  Every open file consumes a file descriptor (a system resource).
  Operating systems impose limits on the number of open files per process.
  Failing to close files can exhaust these limits
- Memory Leaks:
  Unclosed files may hold memory buffers, leading to unnecessary resource usage.

Data Integrity

- Python uses buffering for file I/O (data isn't written immediately to disk). If a program crashes or exits unexpectedly, buffered data may be lost.
  Closing the file (close()) flushes the buffer, ensuring all data is saved.

Preventing Corruption

- File Locking (OS-Level): Some OSs lock files while they're open (e.g., Windows).
  Other programs (or the same script) may fail to access the file if it's not closed.

Q(20) What is the difference between file.read() and file.readline() in Python?

Ans:- In Python, file.read() and file.readline() are methods used to read data from a file, but they behave differently in terms of how they retrieve and return the content.

file.read([size])

- Reads the entire file or a specified number of bytes/characters.
- If no size argument is given, it reads all content until the end of the file (EOF).
- If size is provided (e.g., read(100)), it reads up to that many bytes (binary mode) or characters (text mode).
- A single string (text mode) or bytes object (binary mode) is returned.
- Best for reading small files or when we need the entire content at once.

file.readline([size])

- Reads a single line from the file (up to the newline character \n).
- Reads until the next \n or EOF.
- If size is specified, it reads up to that many characters but stops at \n.
- Returns an empty string ("") at EOF.
- A single line as a string (including the \n at the end) is returned.

- Ideal for processing large files line by line (memory-efficient).

Q(21) What is the logging module in Python used for?

Ans:- The logging module in Python is a built-in library designed to track events that occur when software runs. It is used for recording log messages that help in debugging, monitoring, and analyzing applications.

Purpose of logging module

- Debugging: Log messages help identify issues during development and production.
- Error Tracking: Records exceptions and errors for troubleshooting.
- Monitoring Application Behavior: Tracks user activity, performance metrics, or system events.
- Audit Logs: Maintains a record of security-related events (e.g., login attempts).
- Configurable Log Levels: Different log levels allow filtering messages by severity.
- Flexible Output Destinations: Logs can be written to Console (StreamHandler), Files (FileHandler), Email (SMTPHandler), External services (via custom handlers).

Q(22) What is the os module in Python used for in file handling?

Ans:- The os module in Python provides functions for interacting with the operating system for file and directory operations. It is used in file handling to manage paths, create/delete files, and perform system-level tasks.

- File Path Manipulation
  os.path.join(): Safely joins path components (handles OS-specific separators).
  os.path.abspath(): Gets the absolute path of a file.
  os.path.exists(): Checks if a file/directory exists.
  os.path.isfile() / os.path.isdir(): Checks if a path is a file or directory.
- File and Directory Operations
  os.listdir(): Lists files/folders in a directory.
  os.mkdir() / os.makedirs(): Creates a directory (or nested directories).
  os.remove(): Deletes a file.
  os.rmdir(): Removes an empty directory.
  os.rename(): Renames/moves a file
- File Metadata & Permissions
  os.stat(): Gets file stats (size, modification time, etc.)
  os.chmod(): Changes file permissions.
- Working with File Paths Across OS
  os.sep: OS-specific path separator (\ on Windows, / on Linux)
  os.getcwd(): Gets the current working directory.
  os.chdir(): Changes the working directory.

Q(23) What are the challenges associated with memory management in Python?

Ans:- Memory management in Python is mostly handled automatically by its garbage collector and reference counting. But there are some challanges that a programmer has to face related with memory management.

Some of these challanges are listed below:

- Memory Leaks: Unintentional retention of objects (e.g., circular references, caches, global variables).
- High Memory Usage: Inefficient data structures (e.g., lists vs. generators) and Unoptimized object storage.
- Garbage Collection Overhead: Python's Garbage collector (especially generational GC) can introduce latency in real-time systems.
- Fragmentation: Frequent allocation/deallocation of objects leads to fragmented memory. This slows allocations or MemoryError even if total free memory exists.
- Large Data Structures: Native Python objects (e.g., lists, dicts) have high overhead. A list in Pythoin consumes 4 times more memory than an array in C.
- Reference Counting Limitations: Reference counting alone can't handle circular references.
- Multithreading & Memory: Python's Global Interpreter Lock(GIL) serializes memory access, limiting parallel memory operations.

Q(24) How do you raise an exception manually in Python?

Ans:- In Python, we can manually raise an exception using the "raise" keyword. This is useful for enforcing constraints, handling edge cases, or signaling errors in our code.

Q(25) Why is it important to use multithreading in certain applications?

Ans:- Multithreading is crucial in certain applications because it enables concurrent execution, improving performance, responsiveness, and efficiency—especially in tasks involving I/O-bound operations, real-time processing, or parallel processing.

- Improved Performance for I/O-Bound Tasks
- Responsiveness in GUI/Real-Time Applications
- Efficient Resource Utilization
- Parallelism in Multi-Core Systems (Despite the GIL)

```python
# Q(1) How can you open a file for writing in Python and write a string to it?

# Open the file in write mode ('w')
with open("example.txt", "w") as file:
    file.write("Hello, World!")  # Writes the string to the file


with open("example.txt", "a") as file:
    file.write("\nThis is a new line.")  # Appends text


lines = ["First line\n", "Second line\n", "Third line\n"]

with open("example.txt", "w") as file:
    file.writelines(lines)  # Writes all lines at once


# Q(2) Write a Python program to read the contents of a file and print each
# line.

# Method 1: Using readlines() (stores all lines in memory)
with open('example.txt', 'r') as file:
    lines = file.readlines()
    for line in lines:
        print(line, end='')  # end='' removes extra newlines since each line already has one

print("\n---")
```

```
First line
Second line
Third line

---
```

```python
# Method 2: Iterating directly over the file object (memory efficient)
with open('example.txt', 'r') as file:
    for line in file:
        print(line, end='')

print("\n---")
```

```
First line
Second line
Third line

---
```

```python
# Method 3: Using read() and splitting lines
with open('example.txt', 'r') as file:
    content = file.read()
    for line in content.split('\n'):
        print(line)
```

```
First line
Second line
Third line
```

```python
# Q(3) How would you handle a case where the file doesn't exist while trying to
# open it for reading?

# Using try-except is the most popuar way is to catch FileNotFoundError:

filename = "example1.txt"

try:
    with open(filename, 'r') as file:
        for line in file:
            print(line, end='')
except FileNotFoundError:
    print(f"Error: The file '{filename}' does not exist.")
except IOError as e:
    print(f"An I/O error occurred: {e}")
```

```
Error: The file 'example1.txt' does not exist.
```

```python
# Q(4) Write a Python script that reads from one file and writes its content to
# another file.

def copy_file_content(source_file, destination_file):
    """
    Copies content from source_file to destination_file.

    Args:
        source_file (str): Path to the source file to read from
        destination_file (str): Path to the destination file to write to
    """
```

```
    try:
        # Open source file for reading
        with open(source_file, 'r') as src:
            # Read all content from source file
            content = src.read()

            # Open destination file for writing
            with open(destination_file, 'w') as dest:
                # Write content to destination file
                dest.write(content)

        print(f"Successfully copied content from {source_file} to {destination_file}")

    except FileNotFoundError:
        print(f"Error: The file {source_file} does not exist")
    except PermissionError:
        print(f"Error: Permission denied when accessing files")
    except Exception as e:
        print(f"An unexpected error occurred: {str(e)}")

# Example usage
if __name__ == "__main__":
    source = "input.txt"  # Change to your source file path
    destination = "output.txt"  # Change to your destination file path
    copy_file_content(source, destination)
```

```
Error: The file input.txt does not exist
```

```
# Q(5)  How would you catch and handle division by zero error in Python?

try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

```
Error: Cannot divide by zero!
```

```
# Q(6) Write a Python program that logs an error message to a log file when a
# division by zero exception occurs.

import logging

# Configure logging to write to a file
logging.basicConfig(
    filename='division_errors.log',
    level=logging.ERROR,
    format='%(asctime)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)

def divide_numbers(a, b):
    try:
        result = a / b
        print(f"Result: {result}")
        return result
    except ZeroDivisionError:
        error_msg = f"Division by zero attempted: {a}/{b}"
        logging.error(error_msg)
        print(f"Error: {error_msg} (Logged to division_errors.log)")
        return None

# Example usage
if __name__ == "__main__":
    print("Division Calculator")
    print("-------------------")

    while True:
        try:
            num1 = float(input("Enter numerator (or 'q' to quit): "))
            if str(num1).lower() == 'q':
                break
            num2 = float(input("Enter denominator: "))

            divide_numbers(num1, num2)

        except ValueError:
            print("Error: Please enter valid numbers!")
        except KeyboardInterrupt:
            print("\nProgram terminated.")
            break

print("Goodbye!")
```

```
Division Calculator
------------------
Enter numerator (or 'q' to quit): 8
Enter denominator: 2
Result: 4.0
```

```
Enter numerator (or 'q' to quit): q
Error: Please enter valid numbers!
Enter numerator (or 'q' to quit): q
Error: Please enter valid numbers!
Enter numerator (or 'q' to quit): Q
Error: Please enter valid numbers!
Enter numerator (or 'q' to quit):
Error: Please enter valid numbers!
Enter numerator (or 'q' to quit):
Error: Please enter valid numbers!
Enter numerator (or 'q' to quit):
Error: Please enter valid numbers!

Program terminated.
Goodbye!
```

```
# Q(7) How do you log information at different levels (INFO, ERROR, WARNING) in
# Python using the logging module?

import logging

# Basic configuration (writes to console)
logging.basicConfig(
    level=logging.DEBUG,  # Minimum level to capture
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# Example log messages
logging.debug("Debugging details")      # Level 10 (verbose)
logging.info("Program started")         # Level 20 (normal operation)
logging.warning("Low disk space")       # Level 30 (potential issue)
logging.error("Failed to open file")    # Level 40 (serious problem)
logging.critical("Server crashed")      # Level 50 (fatal error)
```

```
WARNING:root:Low disk space
ERROR:root:Failed to open file
CRITICAL:root:Server crashed
```

```
# Q(8) Write a program to handle a file opening error using exception handling.

import logging
import sys

def read_file_safely(filename):
    """
    Attempts to open and read a file, handling common errors gracefully.

    Args:
        filename (str): Path to the file to be read

    Returns:
        str: Content of the file if successful, None otherwise
    """
    try:
        with open(filename, 'r') as file:
            content = file.read()
            logging.info(f"Successfully read file: {filename}")
            return content

    except FileNotFoundError:
        logging.error(f"File not found: {filename}")
        print(f"Error: The file '{filename}' does not exist.", file=sys.stderr)
    except PermissionError:
        logging.error(f"Permission denied for file: {filename}")
        print(f"Error: You don't have permission to read '{filename}'.", file=sys.stderr)
    except IsADirectoryError:
        logging.error(f"Attempted to read a directory: {filename}")
        print(f"Error: '{filename}' is a directory, not a file.", file=sys.stderr)
    except UnicodeDecodeError:
        logging.error(f"Encoding error in file: {filename}")
        print(f"Error: Could not decode the contents of '{filename}'.", file=sys.stderr)
    except OSError as e:
        logging.error(f"OS error occurred with file {filename}: {str(e)}")
        print(f"System error: {str(e)}", file=sys.stderr)
    except Exception as e:
        logging.critical(f"Unexpected error with file {filename}: {str(e)}")
        print(f"An unexpected error occurred: {str(e)}", file=sys.stderr)

    return None

def main():
    # Configure logging
    logging.basicConfig(
        filename='file_operations.log',
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s'
    )
```

```
    # Example usage
    filename = input("Enter the file path to read: ")
    content = read_file_safely(filename)

    if content is not None:
        print("\nFile contents:")
        print("-" * 40)
        print(content)
        print("-" * 40)
    else:
        print("Failed to read the file.")

if __name__ == "__main__":
    main()
```

```
Enter the file path to read: c:\Python
ERROR:root:File not found: c:\Python
Failed to read the file.
Error: The file 'c:\Python' does not exist.
```

```
# Q(9) How can you read a file line by line and store its content in a list in
# Python.

# Method 1

with open('example.txt', 'r') as file:
    lines = file.readlines()  # Reads all lines into a list

# Remove newline characters from each line
lines = [line.strip() for line in lines]

print(lines)
```

```
['First line', 'Second line', 'Third line']
```

```
# Method 2

with open('example.txt', 'r') as file:
    lines = list(file)  # Converts file lines into a list

# Strip newlines (optional)
lines = [line.strip() for line in lines]

print(lines)
```

```
['First line', 'Second line', 'Third line']
```

```
# Q(10) How can you append data to an existing file in Python?

# Appends single line

with open('data.txt', 'a') as file:
    file.write("This will be appended.\n")  # \n adds a new line


# Appends multiple lines

lines_to_add = ["Line 1\n", "Line 2\n", "Line 3\n"]

with open('data.txt', 'a') as file:
    file.writelines(lines_to_add)  # Appends each string in the list


# Q(11) Write a Python program that uses a try-except block to handle an error
# when attempting to access a dictionary key that doesn't exist.

my_dict = {"name": "Anoop", "age": 49, "city": "Muzaffarpur"}

try:
    print("Occupation:", my_dict["occupation"])  # Key doesn't exist
except KeyError:
    print("Error: The key 'occupation' does not exist in the dictionary.")
```

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

```
# Q(12) Write a program that demonstrates using multiple except blocks to handle
# different types of exceptions.

def handle_operations():
    try:
        # Example 1: Handle division (ZeroDivisionError)
        num = int(input("Enter a numerator: "))
        den = int(input("Enter a denominator: "))
        result = num / den
        print(f"Division result: {result}")
```

```
        # Example 2: Handle file operations (FileNotFoundError)
        filename = input("Enter a filename to read: ")
        with open(filename, 'r') as file:
            content = file.read()
            print(f"File content:\n{content}")

        # Example 3: Handle list index (IndexError)
        my_list = [10, 20, 30]
        index = int(input(f"Enter an index (0-2) to access from {my_list}: "))
        print(f"Value at index {index}: {my_list[index]}")

    except ValueError:
        print("Error: Invalid input (expected an integer).")
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
    except IndexError:
        print(f"Error: Index {index} is out of range.")
    except Exception as e:
        print(f"An unexpected error occurred: {type(e).__name__} - {e}")
    else:
        print("All operations completed successfully!")
    finally:
        print("Program execution complete.\n")

# Run the program
if __name__ == "__main__":
    handle_operations()
```

```
Enter a numerator: 8
Enter a denominator: 2
Division result: 4.0
Enter a filename to read: exapmle
Error: File 'exapmle' not found.
Program execution complete.
```

```
# Q(13) How would you check if a file exists before attempting to read it in
# Python.

import os

file_path = "check.txt"

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

```
Error: File 'check.txt' does not exist.
```

```
# Q(14) Write a program that uses the logging module to log both informational
# and error messages.

import logging
import os
import math

def setup_logging():
    """Configure logging to write to both console and file with different levels"""
    # Create logs directory if it doesn't exist
    os.makedirs("logs", exist_ok=True)

    # Basic configuration
    logging.basicConfig(
        level=logging.DEBUG,  # Lowest level to capture
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler("logs/app.log"),  # All logs to file
            logging.StreamHandler()               # Only warnings+ to console
        ]
    )

    # Set console handler to only show WARNING+ messages
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.WARNING)
    logging.getLogger().addHandler(console_handler)

def calculate_square_root(number):
    """Demonstrate logging at different levels"""
    try:
        logging.info(f"Calculating square root of {number}")

        if number < 0:
```

```
            logging.warning("Negative number provided - using absolute value")
            number = abs(number)

        result = math.sqrt(number)
        logging.info(f"Result: {result}")
        return result

    except Exception as e:
        logging.error(f"Error in calculation: {str(e)}", exc_info=True)
        return None

def main():
    setup_logging()

    logging.info("Program started")

    # Example calculations
    calculate_square_root(16)     # Normal operation
    calculate_square_root(-4)     # Warning case
    calculate_square_root("text") # Error case

    logging.info("Program completed")

if __name__ == "__main__":
    main()
```

```
WARNING:root:Negative number provided - using absolute value
Negative number provided - using absolute value
ERROR:root:Error in calculation: '<' not supported between instances of 'str' and 'int'
Traceback (most recent call last):
  File "/tmp/ipython-input-29-813391973.py", line 33, in calculate_square_root
    if number < 0:
       ^^^^^^^^^^
TypeError: '<' not supported between instances of 'str' and 'int'
Error in calculation: '<' not supported between instances of 'str' and 'int'
Traceback (most recent call last):
  File "/tmp/ipython-input-29-813391973.py", line 33, in calculate_square_root
    if number < 0:
       ^^^^^^^^^^
TypeError: '<' not supported between instances of 'str' and 'int'
```

```
# Q(15) Write a Python program that prints the content of a file and handles the
# case when the file is empty.

import os

def print_file_contents(filename):
    """Prints the contents of a file, handling empty files gracefully."""
    try:
        # Check if file exists first
        if not os.path.exists(filename):
            print(f"Error: File '{filename}' does not exist.")
            return

        # Check if file is empty
        if os.path.getsize(filename) == 0:
            print(f"Note: The file '{filename}' is empty.")
            return

        # Read and print file contents
        with open(filename, 'r') as file:
            content = file.read()
            print(f"Contents of '{filename}':")
            print("-" * 40)
            print(content)
            print("-" * 40)

    except PermissionError:
        print(f"Error: No permission to read '{filename}'.")
    except Exception as e:
        print(f"An unexpected error occurred: {str(e)}")

# Example usage
if __name__ == "__main__":
    filename = input("Enter the path of the file to read: ")
    print_file_contents(filename)
```

```
Enter the path of the file to read: example.txt
Contents of 'example.txt':
----------------------------------------
First line
Second line
Third line

----------------------------------------
```

```
!pip install memory_profiler
```

```
Requirement already satisfied: memory_profiler in /usr/local/lib/python3.11/dist-packages (0.61.0)
Requirement already satisfied: psutil in /usr/local/lib/python3.11/dist-packages (from memory_profiler) (5.9.5)
```

```python
# Q(16) Demonstrate how to use memory profiling to check the memory usage of
# a small program.

# Memory profiling is a technique used to measure how much memory your program
# uses during execution. Here's how to perform memory profiling on a small
# Python program using the memory-profiler package.

# memory_demo_profiled.py
import numpy as np
from memory_profiler import profile

@profile
def create_large_array(size):
    """Create and return a large numpy array"""
    arr = np.random.rand(size, size)
    return arr

@profile
def process_data():
    """Perform memory-intensive operations"""
    data1 = create_large_array(1000)
    data2 = create_large_array(1000)
    result = np.dot(data1, data2)
    return result

if __name__ == "__main__":
    process_data()
```

```
ERROR: Could not find file /tmp/ipython-input-10-2201665239.py
ERROR: Could not find file /tmp/ipython-input-10-2201665239.py
ERROR: Could not find file /tmp/ipython-input-10-2201665239.py
```

```python
# Q(17) Write a Python program to create and write a list of numbers to a file,
# one number per line

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

    Args:
        numbers (list): List of numbers to write
        filename (str): Name of the file to create/write to
    """
    try:
        with open(filename, 'w') as file:
            for number in numbers:
                file.write(f"{number}\n")
        print(f"Successfully wrote {len(numbers)} numbers to {filename}")
    except IOError as e:
        print(f"Error writing to file: {e}")

# Example usage
if __name__ == "__main__":
    # Create a list of numbers (could be integers or floats)
    numbers = [1, 2, 3, 4, 5, 10, 20, 30, 3.14, 2.718]

    # Specify the output filename
    output_file = "numbers.txt"

    # Write the numbers to file
    write_numbers_to_file(numbers, output_file)
```

```
Successfully wrote 10 numbers to numbers.txt
```

```python
# Q(18) How would you implement a basic logging setup that logs to a file with
# rotation after 1MB.

import logging
from logging.handlers import RotatingFileHandler

def setup_logging(log_file='app.log', max_size=1_000_000, backup_count=5):
    """
    Set up logging with file rotation.

    Args:
        log_file (str): Path to the log file
        max_size (int): Maximum file size in bytes before rotation (default: 1MB)
        backup_count (int): Number of backup files to keep (default: 5)
    """
    # Create a logger
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)  # Set to lowest level
```

```
    # Create formatter
    formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )

    # Create rotating file handler
    file_handler = RotatingFileHandler(
        filename=log_file,
        maxBytes=max_size,
        backupCount=backup_count
    )
    file_handler.setFormatter(formatter)

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

    # Optionally add console handler
    console_handler = logging.StreamHandler()
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    return logger

# Example usage
if __name__ == "__main__":
    logger = setup_logging('my_app.log')

    # Log some messages
    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")

    # Generate enough logs to trigger rotation
    for i in range(1000):
        logger.info(f"Log message {i}")
```

```
DEBUG:root:This is a debug message
2025-07-10 06:43:19,375 - root - DEBUG - This is a debug message
INFO:root:This is an info message
2025-07-10 06:43:19,377 - root - INFO - This is an info message
WARNING:root:This is a warning message
2025-07-10 06:43:19,380 - root - WARNING - This is a warning message
ERROR:root:This is an error message
2025-07-10 06:43:19,382 - root - ERROR - This is an error message
CRITICAL:root:This is a critical message
2025-07-10 06:43:19,384 - root - CRITICAL - This is a critical message
INFO:root:Log message 0
2025-07-10 06:43:19,385 - root - INFO - Log message 0
INFO:root:Log message 1
2025-07-10 06:43:19,387 - root - INFO - Log message 1
INFO:root:Log message 2
2025-07-10 06:43:19,388 - root - INFO - Log message 2
INFO:root:Log message 3
2025-07-10 06:43:19,389 - root - INFO - Log message 3
INFO:root:Log message 4
2025-07-10 06:43:19,390 - root - INFO - Log message 4
INFO:root:Log message 5
2025-07-10 06:43:19,392 - root - INFO - Log message 5
INFO:root:Log message 6
2025-07-10 06:43:19,393 - root - INFO - Log message 6
INFO:root:Log message 7
2025-07-10 06:43:19,394 - root - INFO - Log message 7
INFO:root:Log message 8
2025-07-10 06:43:19,395 - root - INFO - Log message 8
INFO:root:Log message 9
2025-07-10 06:43:19,396 - root - INFO - Log message 9
INFO:root:Log message 10
2025-07-10 06:43:19,404 - root - INFO - Log message 10
INFO:root:Log message 11
2025-07-10 06:43:19,406 - root - INFO - Log message 11
INFO:root:Log message 12
2025-07-10 06:43:19,408 - root - INFO - Log message 12
INFO:root:Log message 13
2025-07-10 06:43:19,410 - root - INFO - Log message 13
INFO:root:Log message 14
2025-07-10 06:43:19,412 - root - INFO - Log message 14
INFO:root:Log message 15
2025-07-10 06:43:19,414 - root - INFO - Log message 15
INFO:root:Log message 16
2025-07-10 06:43:19,416 - root - INFO - Log message 16
INFO:root:Log message 17
2025-07-10 06:43:19,417 - root - INFO - Log message 17
INFO:root:Log message 18
2025-07-10 06:43:19,421 - root - INFO - Log message 18
INFO:root:Log message 19
2025-07-10 06:43:19,424 - root - INFO - Log message 19
INFO:root:Log message 20
2025-07-10 06:43:19,426 - root - INFO - Log message 20
INFO:root:Log message 21
2025-07-10 06:43:19,427 - root - INFO - Log message 21
INFO:root:Log message 22
```

```
2025-07-10 06:43:19,428 - root - INFO - Log message 22
INFO:root:Log message 23
2025-07-10 06:43:19,430 - root - INFO - Log message 23
```

```
# Q(19) Write a program that handles both IndexError and KeyError using a
# try-except block.

def handle_errors_example():
    # Example data structures
    my_list = [10, 20, 30]
    my_dict = {'a': 1, 'b': 2, 'c': 3}

    # Example 1: Handle IndexError for list access
    try:
        print("\nAttempting to access list elements:")
        print("First element:", my_list[0])
        print("Fourth element:", my_list[3])  # This will raise IndexError
    except IndexError as e:
        print(f"IndexError occurred: {e}")
        print("The list only has", len(my_list), "elements")

    # Example 2: Handle KeyError for dictionary access
    try:
        print("\nAttempting to access dictionary elements:")
        print("Value for 'a':", my_dict['a'])
        print("Value for 'd':", my_dict['d'])  # This will raise KeyError
    except KeyError as e:
        print(f"KeyError occurred: {e}")
        print("Available keys are:", list(my_dict.keys()))

    # Example 3: Handle both in one block
    try:
        print("\nCombined access attempts:")
        print("Fifth element:", my_list[4])
        print("Value for 'e':", my_dict['e'])
    except IndexError:
        print("List index out of range!")
    except KeyError:
        print("Dictionary key not found!")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
    else:
        print("All accesses were successful!")
    finally:
        print("This cleanup code always runs")

if __name__ == "__main__":
    handle_errors_example()
```

```
Attempting to access list elements:
First element: 10
IndexError occurred: list index out of range
The list only has 3 elements

Attempting to access dictionary elements:
Value for 'a': 1
KeyError occurred: 'd'
Available keys are: ['a', 'b', 'c']

Combined access attempts:
List index out of range!
This cleanup code always runs
```

```
# Q(20) How would you open a file and read its contents using a context manager
# in Python?

# Using a context manager to read a file
with open('example.txt', 'r') as file:
    contents = file.read()
    print(contents)
```

```
First line
Second line
Third line
```

```
# Q(21) 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):
    """
    Counts occurrences of a specific word in a file.

    Args:
        filename (str): Path to the file to read
        target_word (str): Word to count occurrences of

    Returns:
```

```
        int: Number of occurrences
    """
    try:
        with open(filename, 'r', encoding='utf-8') as file:
            content = file.read()
            words = content.split()
            return words.count(target_word)
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return 0
    except Exception as e:
        print(f"An error occurred: {e}")
        return 0

if __name__ == "__main__":
    # Get user input
    filename = input("Enter the file path: ")
    word_to_count = input("Enter the word to count: ").strip()

    # Count occurrences
    count = count_word_occurrences(filename, word_to_count)

    # Print result
    print(f"The word '{word_to_count}' appears {count} times in the file.")
```

```
Enter the file path: example.txt
Enter the word to count: line
The word 'line' appears 3 times in the file.
```

```
# Q(22) How can you check if a file is empty before attempting to read its
# contents?

# Method 1: Check file size using os.path.getsize()
import os

def is_file_empty(file_path):
    """Check if file is empty by confirming its size is 0 bytes"""
    return os.path.getsize(file_path) == 0

# Usage
if not is_file_empty('example.txt'):
    with open('example.txt', 'r') as file:
        content = file.read()
        print(content)
else:
    print("File is empty")
```

```
First line
Second line
Third line
```

```
# Method 2: Read first character (most efficient for large files)

def is_file_empty(file_path):
    """Check if file is empty by reading first character"""
    with open(file_path, 'r') as file:
        first_char = file.read(1)
        return first_char == ''  # True means empty

# Usage
if not is_file_empty('example.txt'):
    with open('example.txt', 'r') as file:
        print(file.read())
else:
    print("File is empty")
```

```
First line
Second line
Third line
```

```
# Method 3: Using file.tell() after seeking to end

def is_file_empty(file_path):
    """Check if file is empty by seeking to end"""
    with open(file_path, 'r') as file:
        file.seek(0, 2)  # Seek to end
        return file.tell() == 0  # True if position is 0 (empty)

# Usage
if not is_file_empty('example.txt'):
    with open('example.txt', 'r') as file:
        print(file.read())
```

```
First line
Second line
```

```
     Third line


# Method 4: Using pathlib (Python 3.4+)

from pathlib import Path

def is_file_empty(file_path):
    """Check if file is empty using pathlib"""
    return Path(file_path).stat().st_size == 0

# Usage
if not is_file_empty('example.txt'):
    print(Path('example.txt').read_text())
```

```
First line
Second line
Third line
```

```
# Q(23) Write a Python program that writes to a log file when an error occurs
# during file handling.

import logging
from datetime import datetime

def configure_logging(log_file='file_errors.log'):
    """Set up logging configuration"""
    logging.basicConfig(
        filename=log_file,
        level=logging.ERROR,
        format='%(asctime)s - %(levelname)s - %(message)s',
        datefmt='%Y-%m-%d %H:%M:%S'
    )

def safe_file_operation(file_path, operation='read', content=None):
    """
    Perform file operations with error handling and logging

    Args:
        file_path (str): Path to the file
        operation (str): 'read' or 'write'
        content (str): Content to write (for write operations)
    """
    try:
        if operation == 'read':
            with open(file_path, 'r') as file:
                return file.read()
        elif operation == 'write':
            with open(file_path, 'w') as file:
                file.write(content)
            return True
        else:
            raise ValueError(f"Invalid operation: {operation}")

    except FileNotFoundError as e:
        logging.error(f"File not found: {file_path} - {str(e)}")
        print(f"Error: File {file_path} does not exist")
    except PermissionError as e:
        logging.error(f"Permission denied: {file_path} - {str(e)}")
        print(f"Error: Permission denied for {file_path}")
    except IOError as e:
        logging.error(f"I/O error: {file_path} - {str(e)}")
        print(f"Error: Could not complete operation on {file_path}")
    except Exception as e:
        logging.error(f"Unexpected error: {file_path} - {str(e)}", exc_info=True)
        print(f"An unexpected error occurred: {str(e)}")
    return False

if __name__ == "__main__":
    # Configure logging
    configure_logging()

    # Example usage
    print("=== File Reading Example ===")
    result = safe_file_operation('nonexistent.txt', 'read')
    if result:
        print("File content:", result)

    print("\n=== File Writing Example ===")
    if safe_file_operation('test.txt', 'write', "Hello, World!"):
        print("File written successfully")

    print("\nCheck 'file_errors.log' for any error details")
```

```
ERROR:root:File not found: nonexistent.txt - [Errno 2] No such file or directory: 'nonexistent.txt'
2025-07-10 06:55:13,809 - root - ERROR - File not found: nonexistent.txt - [Errno 2] No such file or directory: 'nonexistent.txt'
=== File Reading Example ===
```

```
Error: File nonexistent.txt does not exist

=== File Writing Example ===
File written successfully

Check 'file_errors.log' for any error details
```