Theoretical Questions

Files & Exceptional Handling Assignment

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

- Interpreted languages are executed line-by-line by an interpreter at runtime, translating source code into machine code on the fly. They are generally more portable because the same source code can be run on any platform with the appropriate interpreter. However, they tend to be slower due to the overhead of runtime translation. Errors are detected at runtime, which means the program may run until it encounters an error. Examples include Python and JavaScript.

- Compiled languages are translated into machine code by a compiler before execution, creating an executable file. This machine code is specific to the target platform, making compiled languages less portable. However, they tend to be faster because the machine code is executed directly by the hardware without the need for runtime translation. Errors are detected at compile-time, which means the program will not run until all errors are fixed. Examples include C and C++.

Q2. What is exception handling in Python?

- Exception handling in Python allows you to manage runtime errors gracefully using try, except, else, and finally blocks. This mechanism helps prevent program crashes and ensures proper resource management. The try block contains code that might raise an exception, the except block handles the exception if it occurs, the else block executes code if no exceptions occur, and the finally block executes code regardless of whether an exception occurs.

Syntax

```
try:
    risky_code()
except SomeException as e:
    handle_exception(e)
```

```
    else:
        no_exception_occurred()
    finally:
        cleanup_actions()
```

Example

```
    def divide(a, b):
        try:
            result = a / b
        except ZeroDivisionError as e:
            print(f"Error: Division by zero is not allowed. {e}")
        except TypeError as e:
            print(f"Error: Invalid input type. {e}")
        else:
            print(f"The result is {result}")
        finally:
            print("Execution of the divide function is complete.")

    divide(10, 2)
    divide(10, 0)
    divide(10, 'a')
```

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

- The finally block ensures that code runs regardless of whether an exception occurs. It is used for cleanup actions like closing files or releasing resources. This guarantees that important cleanup tasks are always performed, even if an error occurs. The finally block is particularly useful for resource management, ensuring that resources are properly released even if an error occurs.

Syntax

```
    try:
        risky_code()
    finally:
        cleanup_actions()
```

Q4. What is logging in Python?

- Logging is a way to track events that happen when software runs. The logging module provides a flexible framework for emitting log messages from Python

programs. It helps in debugging, monitoring, and understanding the flow of a program by recording significant events. Logging can be configured to output messages to different destinations, such as the console or a file, and at different levels of severity, such as DEBUG, INFO, WARNING, ERROR, and CRITICAL.

Syntax

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info('This is an info message')
```

Q5. What is the significance of the `__del__` method in Python?

- The `__del__` method is called when an object is about to be destroyed. It is used to clean up resources like closing files or network connections. This method ensures that any necessary finalization is performed before the object is removed from memory. The `__del__` method is also known as a destructor and is useful for managing resources that need to be explicitly released when an object is no longer needed.

Syntax

```
class MyClass:
    def __del__(self):
        print('Object is being deleted')
```

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

- **import:** Imports the entire module, making all its functions and classes available. You need to use the module name to access its functions and classes.
- **from import:** Imports specific attributes or functions from a module, allowing direct access without module qualification. This can make the code cleaner and more readable.

Syntax

```
import math
from math import sqrt
```

Q7. How can you handle multiple exceptions in Python?

- Use multiple except blocks or a single except block with a tuple of exceptions. This allows handling different types of exceptions separately or together, providing more control over error handling. Multiple except blocks can be used to handle different exceptions in different ways, while a single except block with a tuple can handle multiple exceptions in the same way.

Syntax

```
try:
    risky_code()
except (TypeError, ValueError) as e:
    handle_exception(e)
```

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

- The with statement ensures proper acquisition and release of resources. It is commonly used with file operations to ensure files are properly closed, even if an error occurs. This helps prevent resource leaks and ensures that files are always properly managed. The with statement simplifies resource management by automatically handling the setup and teardown of resources.

Syntax

```
with open('file.txt', 'r') as file:
    content = file.read()
```

Q9. What is the difference between multithreading and multiprocessing?

- **Multithreading:** Multiple threads within the same process, sharing memory space. It is useful for I/O-bound tasks, such as reading from a file or making network requests, where the program spends a lot of time waiting for external resources.
- **Multiprocessing:** Multiple processes with separate memory space. It is useful for CPU-bound tasks, such as complex calculations, where the program spends a lot of time performing computations. Multiprocessing can take advantage of multiple CPU cores to improve performance.

Syntax

```
import threading
import multiprocessing
```

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

- Logging helps in debugging and monitoring the application by providing a record of runtime events. It allows developers to track the flow of the program and identify issues, making it easier to maintain and troubleshoot the application. Logging can also be used to monitor the application's performance and behavior in production, helping to identify and resolve issues before they become critical.

Syntax

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug('Debug message')
```

Q11. What is memory management in Python?

- Memory management in Python involves the allocation and deallocation of memory to ensure efficient use of resources and prevent memory leaks. Python uses automatic memory management, including garbage collection, to handle memory allocation and deallocation. The garbage collector automatically identifies and frees unused objects, helping to manage memory efficiently and prevent memory leaks.

Syntax

```
import gc
gc.collect()
```

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

- Identify risky code, handle exceptions, execute alternative code, and clean up using try, except, else, and finally blocks. This structured approach ensures that errors are managed gracefully and resources are properly released. The try block contains code that might raise an exception, the except block handles the exception if it occurs, the else block executes code if no exceptions occur, and the finally block executes code regardless of whether an exception occurs.

Syntax

```
try:
    risky_code()
except Exception as e:
    handle_exception(e)
else:
    no_exception_occurred()
finally:
    cleanup_actions()
```

Q13. Why is memory management important in Python?

- Memory management is crucial to prevent memory leaks and ensure efficient use of resources, which helps maintain application performance. Proper memory management ensures that the application runs smoothly without consuming excessive memory. By managing memory efficiently, developers can prevent memory-related issues, such as crashes and slow performance, and ensure that the application remains stable and responsive.

Syntax

```
import gc
gc.collect()
```

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

- **try:** Wraps code that might raise an exception.
- **except:** Handles the exception if it occurs. This structure allows for graceful error handling and ensures that the program can continue running or terminate gracefully. The try block contains code that might raise an exception, and the except block handles the exception if it occurs, allowing the program to recover from errors and continue running.

Syntax

```
try:
    risky_code()
except Exception as e:
    handle_exception(e)
```

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

- Python's garbage collection system automatically deallocates memory by identifying and freeing unused objects using reference counting and cyclic garbage collector. This helps manage memory efficiently and prevents memory leaks. The garbage collector tracks the number of references to each object and frees objects that are no longer referenced. It also detects and collects cyclic references, which are groups of objects that reference each other but are not reachable from the rest of the program.

Syntax

```
import gc
gc.collect()
```

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

- The else block executes code if no exceptions occur in the try block, allowing for clean separation of normal and exceptional code paths. This helps in organizing code and making it more readable and maintainable. The else block is useful for code that should only run if no exceptions occur, providing a clear distinction between normal and exceptional code paths.

Syntax

```
try:
    risky_code()
except Exception as e:
    handle_exception(e)
else:
    no_exception_occurred()
```

Q17. What are the common logging levels in Python?

- The common logging levels are DEBUG, INFO, WARNING, ERROR, and CRITICAL, each indicating the severity of the log messages. These levels help categorize log messages and control the output based on the importance of the events. DEBUG is used for detailed diagnostic information, INFO for general information, WARNING for potential issues, ERROR for errors that prevent the program from continuing, and CRITICAL for severe errors that may cause the program to terminate.

Syntax

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug('Debug message')
```

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

- **os.fork():** Creates a new process by duplicating the current process. It is available on Unix-like systems and is useful for creating child processes that run concurrently with the parent process. However, it can be complex to manage and is not available on all platforms.
- **multiprocessing:** Module to create processes with separate memory space, providing a higher-level interface for process creation and management. The multiprocessing module is platform-independent and provides a simpler and more flexible way to create and manage processes, making it easier to write cross-platform code.

Syntax

```
import os
pid = os.fork()
```

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

- Closing a file ensures that data is written and resources are released, preventing data corruption and resource leaks. It is a crucial step in file handling to maintain data integrity and system stability. When a file is closed, any buffered data is written to the file, and the file descriptor is released, making it available for other processes. Failing to close a file can lead to data loss, resource exhaustion, and other issues.

Syntax

```
file = open('file.txt', 'r')
file.close()
```

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

- **file.read():** Reads the entire file into a single string. This method is useful for reading small files that can be easily loaded into memory.
- **file.readline():** Reads one line at a time, making it suitable for processing large files line-by-line. This method is useful for reading large files that cannot be loaded into memory all at once, allowing the program to process each line individually.

Syntax

```
with open('file.txt', 'r') as file:
    content = file.read()
    line = file.readline()
```

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

- The logging module provides a flexible framework for emitting log messages from Python programs, aiding in debugging and monitoring. It allows developers to track the flow of the program and identify issues. The logging module supports different log levels, output formats, and destinations, making it a powerful tool for managing log messages in Python applications.

Syntax

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info('This is an info message')
```

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

- The os module provides functions to interact with the operating system, such as file operations, directory management, and process handling. It is essential for performing system-level tasks in Python. The os module allows developers to perform tasks like creating, deleting, and renaming files and directories, as well as managing file permissions and attributes.

Syntax

```
import os
os.remove('file.txt')
```

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

- Challenges include handling memory leaks, fragmentation, and ensuring efficient memory use to maintain application performance. Proper memory management is crucial for the stability and efficiency of applications. Memory leaks occur when objects are not properly deallocated, leading to increased memory usage over time. Fragmentation occurs when memory is allocated and deallocated in a way that leaves gaps, reducing the efficiency of memory usage.

Syntax

```
import gc
gc.collect()
```

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

- Use the `raise` statement with an exception to manually raise an error. This allows for custom error handling and can be used to enforce certain conditions in the code. Raising exceptions manually can help ensure that the program behaves as expected and can provide meaningful error messages to users and developers.

Syntax

```
raise ValueError("An error occurred")
```

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

- Multithreading improves performance by allowing concurrent execution of tasks, making applications more responsive and efficient. It is particularly useful for I/O-bound tasks where waiting for external resources can be parallelized. By using multiple threads, an application can perform multiple tasks simultaneously, improving overall performance and responsiveness.

Syntax

```
import threading
def task():
    print('Task executed')
thread = threading.Thread(target=task)
thread.start()
```

Practical Questions

```
# Practical question
'''
Q1. How can you open a file for writing in Python and write a string
to it?

'''
'''
Answer:-1
'''
with open('myfile.txt', 'w') as file:
    # Write a string to the file
    file.write("HI MY NAME IS AVINESH MASIH\n")
    file.write("THIS IS MY ASSIGNMENT FOR PWSKILLS\n")

# Practical question
'''
Q2. Write a Python program to read the contents of a file and print
each line.

'''
'''
Answer:-2
'''
# i will be using `myfile.txt` for the content so that i have to not
call or write again and again
with open('myfile.txt', 'r') as file:
    for line in file:
        print(line, end='')

HI MY NAME IS AVINESH MASIH
THIS IS MY ASSIGNMENT FOR PWSKILLS

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

'''
'''
Answer:-3
'''
file_path = 'answer3.txt'

try:
    with open(file_path, 'r') as file:
```

```
        for line in file:
            print(line, end='')
except FileNotFoundError:
    print(f"Opps!!! The file '{file_path}' does not exist.")
```

```
Opps!!! The file 'answer3.txt' does not exist.
```

```
# Practical question
'''
Q4. Write a Python script that reads from one file and writes its
content to another file.

'''
'''
Answer:-4
'''

source_file_path = 'myfile.txt'
destination_file_path = 'destination.txt'

try:
    with open(source_file_path, 'r') as source_file:
        content = source_file.read()

    with open(destination_file_path, 'w') as destination_file:
        destination_file.write(content)

    print(f"Content copied from '{source_file_path}' to
'{destination_file_path}' successfully.")
except FileNotFoundError:
    print(f"Error: The file '{source_file_path}' does not exist.")
except Exception as e:
    print(f"An error occurred: {e}")
```

```
Content copied from 'myfile.txt' to 'destination.txt' successfully.
```

```
# Practical question
'''
Q5. How would you catch and handle division by zero error in Python?

'''
'''
Answer:-5
'''
try:
    numerator = 50
    denominator = 0
    result = numerator / denominator
    print(f"The result is {result}")
except ZeroDivisionError:
    print("Error: Sorry!! Division by zero is not allowed.")
```

```
Error: Sorry!! Division by zero is not allowed.
```

```
# Practical question
'''
Q6. Write a Python program that logs an error message to a log file
when a division by zero exception occurs.

'''
'''
Answer:-6
'''
import logging

# Configure logging
logging.basicConfig(filename='error.log', level=logging.ERROR)

def divide(numerator, denominator):
    try:
        result = numerator / denominator
        return result
    except ZeroDivisionError:
        logging.error("Division by zero is not allowed.")
        return None
divide(40,0)
```

```
ERROR:root:Division by zero is not allowed.
```

```
# Practical question
'''
Q7. How do you log information at different levels (INFO, ERROR,
WARNING) in Python using the logging module?

'''
'''
Answer:-7
'''
import logging

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

logging.info("This is an informational message to indicate normal
operation.")
logging.warning("This is a warning message about potential issues.")
logging.error("This is an error message for failed operations.")
```

```
WARNING:root:This is a warning message about potential issues.
ERROR:root:This is an error message for failed operations.
```

```
# Practical question
'''
Q8. Write a program to handle a file opening error using exception
handling.

'''
'''
Answer:-8
'''
def file_handling_example(filename):
    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 unexpected error occurred: {e}")

file_handling_example('myfile.txt') #in this the file is existing
file_handling_example('mfile.txt') #in this the file is not existing
```

```
HI MY NAME IS AVINESH MASIH
THIS IS MY ASSIGNMENT FOR PWSKILLS

Error: File 'mfile.txt' not found.
```

```
# Practical question
'''
Q9. How can you read a file line by line and store its content in a
list in Python?

'''
'''
Answer:-9
'''
file_path = 'myfile.txt'

lines = []
with open(file_path, 'r') as file:
    for line in file:
        lines.append(line.strip())

print(lines)
```

```
['HI MY NAME IS AVINESH MASIH', 'THIS IS MY ASSIGNMENT FOR PWSKILLS']
```

```
# Practical question
'''
Q10. How can you append data to an existing file in Python?
```

```
'''
'''
Answer:-10
'''
with open('myfile.txt', 'a') as file:
    file.write("This is appended text.\n")
    file.write("This is question number 10.\n")

# Practical question
'''
Q11. 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.

'''
'''
Answer:-11
'''
data = {"name": "AVINESH", "age": 21, "city": "LUCKNOW"}

def val(dictionary, key):
    try:
        value = dictionary[key]
        return value
    except KeyError:
        print(f"Error: The key '{key}' does not exist in the
dictionary.")
        return None

key_to_look = "country"
result = val(data, key_to_look)

#this line of code i'm writing in the where key is existing in the
dictionary
if result is not None:
    print(f"The value for key '{key_to_look}' is: {result}")
else:
    print(f"Key '{key_to_look}' was not found.")


Error: The key 'country' does not exist in the dictionary.
Key 'country' was not found.

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

'''
'''
Answer:-12
```

```
'''
def handle_exceptions():
    try:
        num = int(input("Enter a number: "))
        result = 10 / num
        print(f"The result of 10 divided by {num} is {result}")

        my_dict = {"a": 21, "b": 50}
        key = input("Enter a key to access the dictionary: ")
        value = my_dict[key]
        print(f"The value for the key '{key}' is {value}")

    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
    except ValueError:
        print("Error: Invalid input. Please enter a valid number.")
    except KeyError:
        print("Error: The specified key does not exist in the
dictionary.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

handle_exceptions()
```

```
Enter a number: 40
The result of 10 divided by 40 is 0.25
Enter a key to access the dictionary: age
Error: The specified key does not exist in the dictionary.
```

```
# Practical question
'''
Q13. How would you check if a file exists before attempting to read it
in Python?

'''
'''
Answer:-13
'''
import os

file_path = "mfile.txt"

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

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

```
# Practical question
'''
Q14. Write a program that uses the logging module to log both
informational and error messages.

'''
'''
Answer:-14
'''
import logging
logging.basicConfig(filename='my_app.log',
level=logging.DEBUG,format='%(asctime)s - %(levelname)s - %
(message)s',handlers=[logging.StreamHandler()])

def divide_numbers(numerator, denominator):
    try:
        logging.info("Attempting to divide %s by %s.", numerator,
denominator)
        result = numerator / denominator
        logging.info("Division successful. Result: %s", result)
        return result
    except ZeroDivisionError:
        logging.error("Error: Division by zero is not allowed.
Numerator: %s, Denominator: %s", numerator, denominator)
        return None
    except Exception as e:
        logging.error("An unexpected error occurred: %s", e)
        return None


logging.info("Program started.")
divide_numbers(50, 2)  # Successful division
divide_numbers(40, 0)  # Division by zero error
divide_numbers(11, 'a')  # Invalid input
logging.info("Program ended.")
```

```
ERROR:root:Error: Division by zero is not allowed. Numerator: 40,
Denominator: 0
ERROR:root:An unexpected error occurred: unsupported operand type(s)
for /: 'int' and 'str'
```

```
# Practical question
'''
Q15. Write a Python program that prints the content of a file and
handles the case when the file is empty.

'''
'''
Answer:-15
'''
```

```python
#creating empty file
with open('myfile.txt', 'w') as file:
  file.write("")
  file.close()
def print_file_content(filename):
    try:
        with open(filename, 'r') as file:
            content = file.read()
            if not content:
                print("The file is empty.")
            else:
                print(content)
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")

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

```
The file is empty.
```

```python
# Practical question
'''
Q16. Demonstrate how to use memory profiling to check the memory usage
of a small program.

'''
'''
Answer:-16
'''
!pip install memory-profiler
from memory_profiler import profile

@profile
def simple_function():
    a = [i for i in range(100000)]  # Create a large list
    b = sum(a)  # Calculate the sum of the list
    return b

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

```
Requirement already satisfied: memory-profiler in
/usr/local/lib/python3.10/dist-packages (0.61.0)
Requirement already satisfied: psutil in
/usr/local/lib/python3.10/dist-packages (from memory-profiler) (5.9.5)
ERROR: Could not find file <ipython-input-35-f21129a5b0f0>
NOTE: %mprun can only be used on functions defined in physical files,
and not in the IPython environment.
```

```
# Practical question
'''
Q17. Write a Python program to create and write a list of numbers to a
file, one number per line.

'''
'''
Answer:-17
'''
def write_numbers_to_file(numbers, filename):
    try:
        with open(filename, 'w') as file:
            for number in numbers:
                file.write(str(number) + '\n')
        print(f"Numbers successfully written to '{filename}'.")
    except Exception as e:
        print(f"An error occurred: {e}")

numbers = [1, 2, 3, 4, 5]
filename = 'numbers.txt'
write_numbers_to_file(numbers, filename)
```

```
Numbers successfully written to 'numbers.txt'.
```

```
# Practical question
'''
Q18. How would you implement a basic logging setup that logs to a file
with rotation after 1MB?

'''
'''
Answer:-18
'''
import logging
from logging.handlers import RotatingFileHandler

log_file = 'app.log'
log_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %
(message)s')

handler = RotatingFileHandler(log_file, maxBytes=1*1024*1024,
backupCount=5)
handler.setFormatter(log_formatter)

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
logger.addHandler(handler)

def main():
    logger.info("This is an informational message.")
```

```
    logger.debug("This is a debug message.")
    logger.warning("This is a warning message.")
    logger.error("This is an error message.")
    logger.critical("This is a critical message.")

if __name__ == "__main__":
    main()
    print("Logging setup complete. Check the log file for messages.")
```

```
INFO:root:This is an informational message.
DEBUG:root:This is a debug message.
WARNING:root:This is a warning message.
ERROR:root:This is an error message.
CRITICAL:root:This is a critical message.

Logging setup complete. Check the log file for messages.
```

```
# Practical question
'''
Q19. Write a program that handles both IndexError and KeyError using a
try-except block.

'''
'''
Answer:-19
'''
my_list = [1, 2, 3]
my_dict = {"a": 1, "b": 2}

try:
    print(my_list[5])
except IndexError:
    print("Error: List index out of range.")

try:
    value = my_dict["c"]
    print(value)
except KeyError:
    print("Error: The specified key does not exist in the
dictionary.")
```

```
Error: List index out of range.
Error: The specified key does not exist in the dictionary.
```

```
# Practical question
'''
Q20. How would you open a file and read its contents using a context
manager in Python?

'''
'''
```

```
Answer:-20
'''
file_path = "destination.txt"

with open(file_path, 'r') as file:
    content = file.read()

print(content)


HI MY NAME IS AVINESH MASIH
THIS IS MY ASSIGNMENT FOR PWSKILLS


# Practical question
'''
Q21. Write a Python program that reads a file and prints the number of
occurrences of a specific word.

'''
'''
Answer:-21
'''
def count_word_occurrences(filename, word):
    """Counts the occurrences of a specific word in a file.

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

    Returns:
        The number of times the word appears in the file, or -1 if the
file
        is not found.
    """
    try:
        with open(filename, 'r') as file:
            content = file.read()
            occurrences = content.lower().count(word.lower())  # Case-
insensitive count
            return occurrences
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return -1

# Example usage:
filename = "destination.txt"
word_to_count = "AVINESH"
count = count_word_occurrences(filename, word_to_count)

if count != -1:
```

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

```
The word 'AVINESH' appears 1 times in the file.
```

```
# Practical question
'''
Q22. How can you check if a file is empty before attempting to read
its contents?

'''
'''
Answer:-22
'''
import os

def print_file_content(filename):
    if not os.path.exists(filename):
        print(f"Error: File '{filename}' not found.")
        return

    try:
        with open(filename, 'r') as file:
            content = file.read()
            if not content:
                print("The file is empty.")
            else:
                print(content)
    except Exception as e:
        print(f"An error occurred while reading the file: {e}")

print_file_content('numbers.txt')
```

```
1
2
3
4
5
```

```
# Practical question
'''
Q23. Write a Python program that writes to a log file when an error
occurs during file handling.

'''
'''
Answer:-23
'''
import logging
logging.basicConfig(filename='file_handling.log', level=logging.ERROR,
```

```
                    format='%(asctime)s - %(levelname)s - %
(message)s')

def process_file(filename):
    try:
        with open(filename, 'r') as file:
            contents = file.read()
            print(contents)
    except FileNotFoundError:
        logging.error(f"Error: File '{filename}' not found.")
    except Exception as e:
        logging.error(f"An unexpected error occurred while processing
'{filename}': {e}")

process_file('destinatio.txt')
```

```
ERROR:root:Error: File 'destinatio.txt' not found.
```