Assignment

1. What is the role of the 'else' block in a try-except statement? Provide an example scenario where it would be useful.

- he 'else' block in a try-except statement is optional and is executed only if no exceptions are raised in the corresponding try block.
- try:
  # Some code that may raise an exception
  result = perform_complex_operation()
  except Exception as e:
  print("An error occurred:", str(e))
  else:
  # Code to execute when no exceptions occur
  print("Operation completed successfully")
  print("Result:", result)

2. Can a try-except block be nested inside another try-except block? Explain with an example.

Yes, a try-except block can indeed be nested inside another try-except block. This is known as nested exception handling and allows for handling specific exceptions at different levels of code execution.

```
try:
  # Outer try-except block
  try:
    # Inner try-except block
    numerator = 10
```

```
        denominator = 0
        result = numerator / denominator
        print("Result:", result)
    except ZeroDivisionError:
        print("Cannot divide by zero!")
    finally:
        print("Inner try-except block executed.")
except Exception as e:
    print("An error occurred:", str(e))
finally:
    print("Outer try-except block executed.")
```

3. How can you create a custom exception class in Python? Provide an example that demonstrates its usage.

you can create a custom exception class by subclassing the built-in Exception class or any of its subclasses. By creating a custom exception class, you can define your own exception types with specific behaviors and attributes.

```
class CustomException(Exception):
    def __init__(self, message):
        self.message = message
    def __str__(self):
        return f"CustomException: {self.message}"
def divide(a, b):
    if b == 0:
```

```
        raise CustomException("Cannot divide by zero!")
    return a / b
try:
    result = divide(10, 0)
    print("Result:", result)
except CustomException as e:
    print(e)
```

4. What are some common exceptions that are built-in to Python?

1.TypeError: Raised when an operation or function is performed on an object of inappropriate type.

2.ValueError: Raised when a function receives an argument of correct type but an inappropriate value.

3.NameError: Raised when a local or global name is not found.

4.IndexError: Raised when an index is out of range.

5.KeyError: Raised when a dictionary key is not found.

6.FileNotFoundError: Raised when an attempt to open a file fails because the file does not exist.

7.IOError: Raised when an I/O operation fails.

8.ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.

5. What is logging in Python, and why is it important in software development?

Logging is a way to store information about your script and track events that occur. When writing any complex script in Python, logging is essential for debugging software as you develop it. Without logging, finding the source of a problem in your code may be extremely time consuming.

6. Explain the purpose of log levels in Python logging and provide examples of when

each log level would be appropriate.

DEBUG: This is the lowest log level and is used for detailed debugging information. It is typically used during development and should not be enabled in production environments. Example usage: Tracking the flow of execution, variable values, or specific events within the code.

INFO: This level is used to confirm that things are working as expected. It provides general information about the application's execution. Example usage: Reporting the start and end of major processes, configuration changes, or important milestones during the application's lifecycle.

WARNING: This level indicates a potential issue or an unexpected condition that does not necessarily lead to an error but might require attention. Example usage: Reporting deprecated features, incorrect API usage, or recoverable errors.

ERROR: This level indicates a more severe issue that typically prevents the application from functioning correctly. Example usage: Reporting an unhandled exception, database connection failures, or critical errors that need immediate attention.

CRITICAL: This is the highest log level and represents a critical failure that might result in the application's termination. Example usage: Reporting a catastrophic failure, security breaches, or unrecoverable errors that require immediate intervention

7. What are log formatters in Python logging, and how can you customise the log

message format using formatters?

Loggers expose the interface that application code directly uses. Handlers send the log records (created by loggers) to the appropriate destination. Filters provide a finer grained facility for determining which log records to output. Formatters specify the layout of log records in the final output.

8. How can you set up logging to capture log messages from multiple modules or

classes in a Python application?

mport logging logging. basicConfig(level=logging. ...

DEBUG:root:This will get logged.

import logging logging. basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s') logging. ...

root - ERROR - This will get logged to a file.

ERROR:root:This is an error message.

9. What is the difference between the logging and print statements in Python? When

should you use logging over print statements in a real-world application?

To define basic logging needs, several lines of code are needed. Including additional logging information is not easy. The print() statement only displays messages on the console. Recording logging data inside a file or sending it over the internet needs additional works.

Debugging and troubleshooting: When you encounter issues or bugs in your application, logging allows you to add detailed diagnostic information.

Production environment: In a production environment, where your application is live and serving users, print statements are not suitable because they write directly to the console or standard output.

Long-term maintenance: When developing a real-world application, it's essential to consider long-term maintenance

10. Write a Python program that logs a message to a file named "app.log" with the

following requirements:

● The log message should be "Hello, World!"

● The log level should be set to "INFO."

● The log file should append new log entries without overwriting previous ones.

```
import logging


# Configure the logger
logging.basicConfig(filename='app.log', level=logging.INFO, filemode='a',
  format='%(asctime)s - %(levelname)s - %(message)s')
# Log the message
logging.info("Hello, World!")
```

11. Create a Python program that logs an error message to the console and a file named

"errors.log" if an exception occurs during the program's execution. The error

message should include the exception type and a timestamp.

```
import logging
```

```
import datetime

# Configure the logger

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

try:

   # Your code that may raise an exception

   raise ValueError("An error occurred!")

except Exception as e:

   # Log the error message to the console

   logging.error(f"{type(e).__name__} occurred at {datetime.datetime.now()}: {str(e)}")

   # Log the error message to the file

   logging.error(f"{type(e).__name__} occurred: {str(e)}")
```