18th June Assignment

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

Ans- The else block in a try-except statement is used to specify a block of code to be executed if no exceptions are raised in the try block. This means that the code in the else block will only be executed if the code in the try block completes without raising any exceptions.

Example:

```
    Try:
     file = open('file.txt', 'r')
except FileNotFoundError:
     print('File not found')
else:
     data = file.read()
     file.close()
     print(data)
```

here ,open a file and read its contents. If the file doesn't exist, a FileNotFoundError exception will be raised, and the code in the except block will be executed. if the file does exist and no exceptions are raised, the code in the else block will be executed. This allows us to only attempt to read from the file and close it if it was successfully opened, without having to add additional checks or error handling.

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try-except statement.

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

Ans- try-except block can be nested inside another try-except block. This is useful when you want to handle different exceptions that may occur in different parts of the code. **Example:**

```
try:
     x = 5 / 0
except ZeroDivisionError:
     print("ZeroDivisionError occurred")
     try:
```

```
        y = int('a')
    except ValueError:
        print("ValueError occurred")
```

The first try block contains code that raises a ZeroDivisionError. The corresponding except block handles this exception and prints a message. Inside this except block, there is another try-except block that attempts to convert a string to an integer, which raises a ValueError. The inner except block handles this exception and prints another message.

Output:

```
ZeroDivisionError occurred
ValueError occurred
```

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

Ans- We can create a custom exception class in Python by defining a new class that inherits from the built-in Exception class or one of its subclasses.

Example that demonstrates how to create and use a custom exception:

```
class MyCustomError(Exception):
    def __init__(self, message):
        self.message = message


try:
    raise MyCustomError("This is a custom error message")
except MyCustomError as e:
    print(e.message)
```

We define a new class called MyCustomError that inherits from the Exception class. The constructor of this class takes a message parameter and assigns it to the message attribute of the object.

In the try block, we raise an instance of our custom exception by calling raise MyCustomError("This is a custom error message"). This causes the program to jump to the except block, where we catch the exception and print its message attribute.

Output:

This is a custom error message

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

Ans- Python has several built-in exceptions that can be generated by the interpreter or built-in functions. Some common built-in exceptions in Python include:

- ArithmeticError: Raised when an error occurs in numeric calculations.
- AssertionError: Raised when an assert statement fails.
- AttributeError: Raised when attribute reference or assignment fails.
- EOFError: Raised when the input() method hits an “end of file” condition (EOF).
- FloatingPointError: Raised when a floating-point calculation fails.
- ImportError: Raised when an imported module does not exist.
- IndentationError: Raised when indentation is not correct.
- IndexError: Raised when an index of a sequence does not exist.
- KeyError: Raised when a key does not exist in a dictionary.
- KeyboardInterrupt: Raised when the user presses Ctrl+c, Ctrl+z or Delete.
- MemoryError: Raised when a program runs out of memory.

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

Ans- Logging is an important aspect of software development because it helps developers understand what is happening within their applications, especially when something goes wrong. It helps track the performance of an application, identify and troubleshoot issues that may arise, troubleshoot bugs, monitor projects in production environments, and facilitate debugging. Proper logging and management tools are essential, especially in distributed architectures.

Logging is a very useful tool in a programmer’s toolbox. It can help you develop a better understanding of the flow of a program and discover scenarios that you might not even have thought of while developing. Logs provide developers with an extra set of eyes that are constantly looking at the flow that an application is going through. They can store information, like which user or IP accessed the application. If an error occurs, then they can provide more insights than a stack trace by telling you what the state of the program was before it arrived at the line of code where the error occurred.

6. Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.

Ans- Log levels in Python logging are used to indicate the severity of log messages. There are six log levels, each one assigned a specific integer indicating the severity of the log:

DEBUG (10): Detailed information, typically of interest only when diagnosing problems.

INFO (20): Confirmation that things are working as expected.

WARNING (30): An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.

ERROR (40): Due to a more serious problem, the software has not been able to perform some function.

CRITICAL (50): A serious error, indicating that the program itself may be unable to continue running.

Examples of when each log level would be appropriate:

DEBUG: You might use this level when you're trying to diagnose a problem and need detailed information about what's happening within your code. For example, you might log the values of variables at certain points in your code to see how they're changing over time.

INFO: This level is used for informational messages that confirm that things are working as expected. For example, you might log a message when a user successfully logs in to your application.

WARNING: This level is used to indicate potential problems that aren't necessarily causing your application to fail but could cause issues in the future. For example, you might log a warning if your application is running low on disk space.

ERROR: This level is used when something has gone wrong and your application is unable to perform some function as a result. For example, you might log an error if your application is unable to connect to a database.

CRITICAL: This level is used for very serious errors that may cause your application to stop running altogether. For example, you might log a critical error if your application runs out of memory.

7. **What are log formatters in Python logging, and how can you customise the log message format using formatters**?

   Ans- Log formatters in Python logging are used to configure the final structure and content of the logs. Using a log formatter, you can include log name, time, date, severity, and other information along with the log message.

   There are different types of formatters, such as JsonFormatter that transforms the log output into a JSON object, or Formatter() that uses the % operator to include log name, time, date, severity, and other information.

   To define the format of a log, you can use the Formatter() method or subclass the logging.Formatter class. To use a formatter, you set it on a handler that is attached to a logger. The default format of a handler without a formatter is %(message)s, and the default **format of a handler with basicConfig is %(levelname)s:%(name)s:%(message).**

Example:

```
import logging
```

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

# Create a console handler
console_handler = logging.StreamHandler()
console_handler.setFormatter(formatter)

# Create a logger and add the console handler to it
logger = logging.getLogger(__name__)
logger.addHandler(console_handler)

# Log some messages
logger.warning('This is a warning message')
logger.error('This is an error message')
```

We create a custom formatter that includes the timestamp, logger name, log level, and message in the log output. We then create a console handler and set its formatter to our custom formatter. Finally, we create a logger and add the console handler to it. When we log messages using this logger, they will be formatted using our custom formatter.

8. **How can you set up logging to capture log messages from multiple modules or classes in a Python application?**

Ans- To capture log messages from multiple modules or classes in a Python application, you can create a logger in each module or class using the logging.getLogger() method and passing in the __name__ global variable as an argument. This will create a logger with the name of your module or class and ensures no name collisions. You can also configure handlers and formatters for the root logger to control how log messages from all modules are handled and formatted. For example, you can add a file handler to the root logger to write log messages from all modules to a file.

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?**

Ans- The logging module in Python provides a flexible framework for emitting log messages from applications. It is more powerful and versatile than using print statements for debugging and provides several advantages, such as the ability to output log messages to different locations (e.g., console, file, email), filter log messages based on severity, and configure the format of log messages.

In a real-world application, it is generally recommended to use logging over print statements. This is because logging provides more control over the output of log messages and makes it easier to manage and analyse log data. Additionally, using logging makes it easier to maintain and update your code, as you can easily enable or disable logging or change the logging level without having to modify your code.

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

Ans-

Example:

```
import logging

# Set up logging

logging.basicConfig(filename='app.log',  level=logging.INFO, filemode='a')

# Log the message

logging.info('Hello, World!')
```

This program sets logging to write log messages to a file named “app.log” with the log level set to “INFO.” The filemode parameter is set to 'a' to append new log entries to the file without overwriting previous ones. The program logs the message “Hello, World!” using the logging.info() method**.**

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.

Ans-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:

```
import logging

from datetime import datetime


# Set up logging to the console and a file
```

```
logging.basicConfig(handlers=[logging.StreamHandler(),
logging.FileHandler('errors.log')], level=logging.ERROR)


try:
# Code that might raise an exception goes here
    x = 1 / 0
except Exception as e:
  # Log the error message with the exception type and
timestamp
    logging.error(f'{datetime.now()} - {type(e).__name__}:
{e}')
```

This program sets up logging to write log messages to both the console and a file named “errors.log” with the log level set to “ERROR.” The program then includes some code that might raise an exception within a try block. If an exception occurs, the except block is executed, and the program logs an error message that includes the exception type, the exception message, and a timestamp using the logging.error() method.