Assignment 11

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 an optional block that follows all the "except" blocks. The role of the "else" block is to specify a section of code that should be executed if no exceptions are raised within the corresponding **"try"** block.
   The **"else"** block is executed only if no exceptions occur during the execution of the **"try"** block. It allows we to define code that should run when the **"try"** block completes successfully, providing an opportunity to perform additional actions or handle situations that are specific to the success case.

```python
#Example

try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
except ValueError:
    print("Error: Invalid input. Please enter valid integers.")
else:
    result = num1 + num2
    print("Sum:", result)
```

```
Enter the first number:  2
Enter the second number:  a
Error: Invalid input. Please enter valid integers.
```

```python
#Example

try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
except ValueError:
    print("Error: Invalid input. Please enter valid integers.")
else:
    result = num1 + num2
    print("Sum:", result)
```

```
Enter the first number:  4
Enter the second number:  6
Sum: 10
```

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

Ans:

Yes, a **"try-except"** block can be nested inside another **"try-except"** block in Python. This is called nested exception handling and allows for handling exceptions at different levels of code, providing more granular error handling.

```
try:
    print("Outer try block")
    try:

        print("Inner try block")
        result = 10 / 0
    except ZeroDivisionError:
        print("Caught ZeroDivisionError in inner except block")
except Exception as e:
    print("An error occurred:", str(e))
```

```
Outer try block
Inner try block
Caught ZeroDivisionError in inner except block
```

3. **How can we create a custom exception class in Python? Provide an example thatdemonstrates its usage.**

Ans:

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

    def __str__(self):
        return self.message


def process_data(data):
    if not data:
        raise CustomException("Empty data passed.")

try:
    data = []
    process_data(data)
except CustomException as e:
    print("CustomException occurred:", str(e))
```

```
CustomException occurred: Empty data passed.
```

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

Ans:

Some common exceptions that are built-in to Python are-

SyntaxError: Raised when there is a syntax error in the code.

IndentationError: Raised when there is an indentation-related error, such as incorrect or inconsistent indentation.

NameError: Raised when a variable or name is not found or not defined.

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

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

IndexError: Raised when a sequence (such as a list or string) is accessed using an invalid index.

KeyError: Raised when a dictionary is accessed with a key that does not exist.

FileNotFoundError: Raised when a file or directory is not found.

IOError: Raised when an input/output operation fails.

ZeroDivisionError: Raised when division or modulo operation is performed with a denominator of zero.

AttributeError: Raised when an attribute reference or assignment fails.

ImportError: Raised when an imported module or name cannot be found.

StopIteration: Raised to signal the end of an iterator.

KeyboardInterrupt: Raised when the user interrupts the execution of the program (typically by pressing Ctrl+C).

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

Ans:

Logging in Python is a built-in module that allows developers to record and track events, messages, warnings, and errors that occur during the execution of a program. It provides a flexible and efficient way to capture and store log information for analysis, troubleshooting, and monitoring purposes.

Few reasons why logging is important in software development:

Debugging and Troubleshooting: Logging allows developers to track the flow of execution, capture relevant information, and debug issues by providing a detailed log of events, errors, and warnings. It helps in identifying the cause of errors and facilitates the debugging process.

Error Reporting and Handling: Logging provides a mechanism to record and report errors. By logging error messages along with contextual information, developers can gain insights into the cause of errors and take appropriate action. It helps in identifying recurring issues, monitoring error rates, and improving error handling mechanisms.

Monitoring and Performance Optimization: Logging can be used to collect performance-related data, such as response times, resource utilization, and execution metrics. By analyzing these logs, developers can identify performance bottlenecks, optimize code, and improve system efficiency.

Audit Trail and Compliance: Logging plays a crucial role in maintaining an audit trail for security, compliance, and legal purposes. By logging important events and actions, developers can track system activities, user interactions, and changes made to critical data.

Understanding User Behavior: Logging can provide insights into user behavior, such as user actions, preferences, and usage patterns. By logging relevant information, developers can gain a better understanding of how users interact with the system and make data-driven decisions to improve user experience

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 categorize and prioritize log messages based on their severity or importance. They allow developers to control the verbosity of logging output and filter log messages based on their relevance.

DEBUG: The DEBUG log level is used for detailed information useful for debugging purposes. It provides the most verbose logging output and is typically used during development or troubleshooting scenarios.

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug("This is a debug message")
```

```
DEBUG:root:This is a debug message
```

INFO: The INFO log level is used to convey general information about the program's execution. It provides less verbose output compared to DEBUG. It is often used to track the flow of execution and provide high-level details about the system's operation.

```
import logging

logging.basicConfig(level=logging.INFO)
logging.info("Processing data...")
```

```
INFO:root:Processing data...
```

WARNING: The WARNING log level is used to highlight potentially harmful or unexpected events that do not necessarily cause the program to terminate. It indicates situations that may lead to issues if not addressed but do not necessarily indicate errors.

```
import logging

logging.basicConfig(level=logging.WARNING)
logging.warning("Disk space is running low.")
```

```
WARNING:root:Disk space is running low.
```

ERROR: The ERROR log level is used to indicate error conditions that affect the normal operation of the program. It highlights unexpected errors or exceptions that occurred but did not cause the program to terminate.

```
import logging

logging.basicConfig(level=logging.ERROR)
logging.error("An error occurred while processing the request.")
```

```
ERROR:root:An error occurred while processing the request.
```

CRITICAL: The CRITICAL log level is used to indicate critical errors that may result in the program's termination or severe consequences. It represents the highest severity level and should be used sparingly for exceptional situations.

```
import logging

logging.basicConfig(level=logging.CRITICAL)
logging.critical("System failure! Shutting down.")
```

```
CRITICAL:root:System failure! Shutting down.
```

7. **What are log formatters in Python logging, and how can we customize the logmessage format using formatters?**

Ans:

Log formatters in Python logging are used to define the structure and content of log messages. They allow developers to customize the format in which log records are displayed or written to various output destinations, such as console, files, or external logging services. Log formatters provide flexibility in defining the lawet of log messages, including timestamps, log levels, module names, and additional contextual information.

```
import logging

formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s') #Creating a "formatter"

handler = logging.StreamHandler()
handler.setFormatter(formatter)

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

logger.debug("This is a debug message")
logger.info("This is an info message")
```

```
DEBUG:root:This is a debug message
2023-07-08 09:01:17,266 - DEBUG - This is a debug message
2023-07-08 09:01:17,266 - DEBUG - This is a debug message
2023-07-08 09:01:17,266 - DEBUG - This is a debug message
```

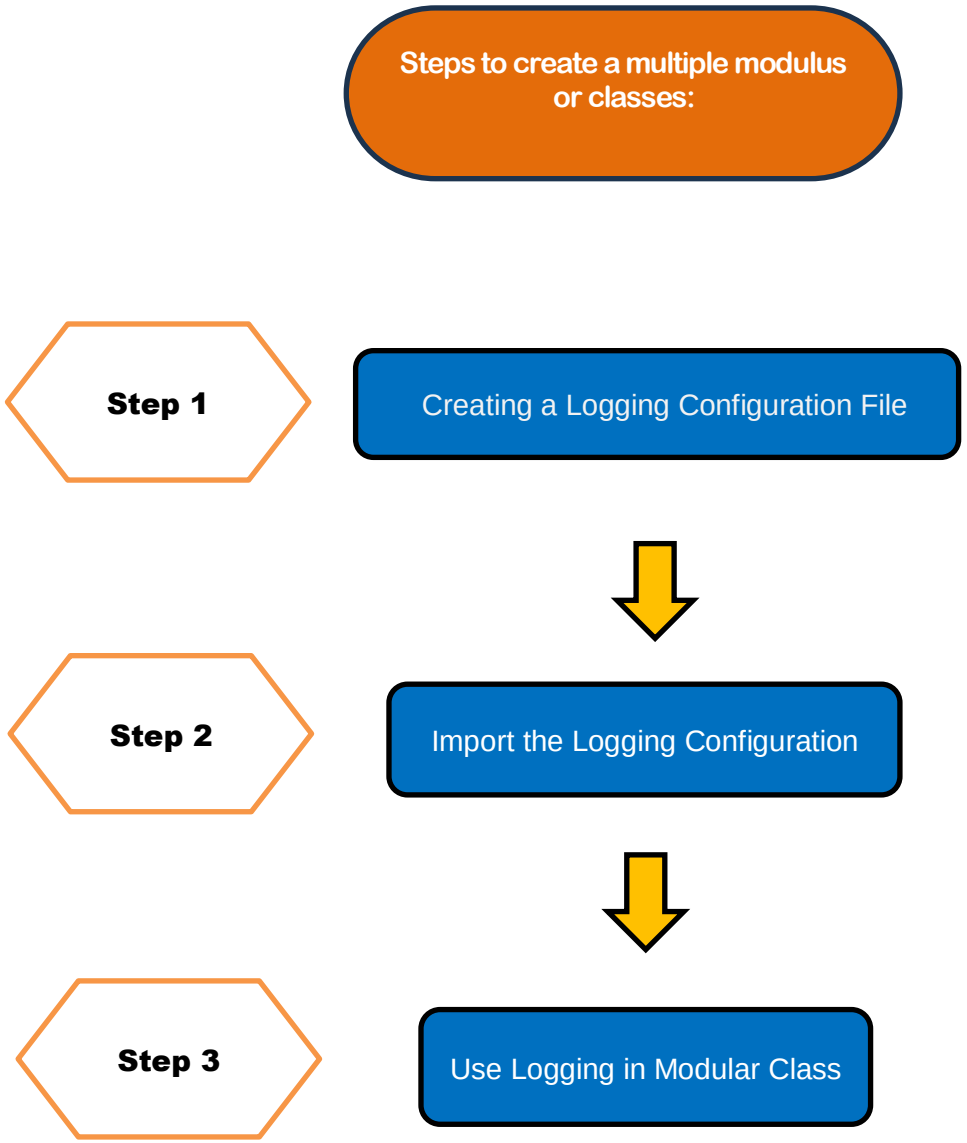
In this format pattern:

%(asctime)s is replaced with the timestamp of the log record.
%(levelname)s is replaced with the log level (e.g., DEBUG, INFO, etc.).
%(message)s is replaced with the log message itself.

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

Ans:

To capture log messages from multiple modules or classes in a Python application, we can set up a logging configuration that spans across the entire application. This allows you to centralize the logging configuration and have consistent log handling across various modules or classes.



9. **What is the difference between the logging and print statements in Python? Whenshould we use logging over print statements in a real-world application?**

Ans:

The difference between the logging and print statements are-

Output Destination: "print" statements typically write the output to the standard output (console), whereas the logging module provides more flexibility to route log messages to various output destinations, such as files, external services, or custom handlers.

Granularity and Control: Logging allows for different log levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL) to be used, allowing you to control the level of detail and importance of log messages. This enables selective logging based on severity and provides the ability to filter and manage log messages effectively. On the other hand, print statements provide a single level of output and cannot be easily controlled or filtered based on severity.

Flexibility and Configuration: The logging module offers extensive configuration options to control log message formatting, handlers, levels, and other parameters. This allows for centralized and customizable log management across modules or classes. In contrast, print statements are less configurable and require manual modifications if formatting or behavior changes are needed.

Error Handling and Debugging: Logging is particularly useful for error handling and debugging purposes. Unlike print statements, logging provides more detailed information, including timestamps, log levels, and other contextual data. It allows for a structured approach to log errors, warnings, and other important events, making it easier to identify and diagnose issues.

The following are the reasons why we use logging over print statements in a real-world application:

Production-Ready Logging: Logging provides a standardized and professional approach to handle log messages in real-world applications.

Selective Logging and Error Reporting: Logging allows you to log messages with different levels of severity, enabling you to prioritize and filter log output based on importance.

Centralized Log Management: The logging module allows you to configure and manage logging behavior centrally, across the entire application.

Efficient Debugging: Logging is especially useful during the debugging phase. It allows you to log detailed information, including stack traces, variable values, and context-specific data.

Flexibility for Different Outputs: The logging module supports various output destinations, such as log files, syslog, databases, or external logging services. This flexibility allows for integration with existing log management systems and facilitates log analysis and monitoring.

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

```python
import logging

# Configure logging
logging.basicConfig(
    filename='app.log',
    level=logging.INFO,
    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.**

Ans:

```python
import logging
import datetime

logging.basicConfig(
    level=logging.ERROR,
    format="%(asctime)s - %(levelname)s - %(message)s",
    handlers=[
        logging.FileHandler("errors.log"),
        logging.StreamHandler()
    ]
)

try:

    raise ValueError("An error occurred!")
except Exception as e:

    error_msg = f"Exception: {type(e).__name__} | Timestamp: {datetime.datetime.now()}"
    logging.error(error_msg)
```