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 optional and provides a block of code that is executed only if no exception is raised within the corresponding try block.

```
try:
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))
    result = numerator / denominator
except ValueError:
    print("Error: Invalid input. Please enter integers.")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
else:
    print("Division result:", result)
```

```
Division result: 5.0
```

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

Yes, it is possible to nest a try-except block inside another try-except block in Python. This is known as nested exception handling. It allows for more specific and localized handling of exceptions within different levels or sections of code. Here's an example:

```
try:
    # Outer try block
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))
    try:
        # Inner try block
        result = numerator / denominator
        print("Division result:", result)
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter integers.")
```

```
Division result: 5.0
```

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

Ans: In Python, you can create a custom exception class by defining a new class that inherits from the built-in Exception class or any of its subclasses.

```
class CustomException(Exception):
    pass

# Usage example
def divide(a, b):
    if b == 0:
        raise CustomException("Division by zero is not allowed.")
    return a / b

try:
    result = divide(10, 0)
    print("Result:", result)
```

```
except CustomException as e:
    print("Error:", str(e))
```

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

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

Exception: The base class for all exceptions.

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

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

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

IndexError: Raised when trying to access an index that is out of range in a sequence.

KeyError: Raised when trying to access a dictionary key that does not exist.

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

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

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

IOError: Raised when an input/output operation fails or is interrupted.

AttributeError: Raised when an attribute reference or assignment fails.

OverflowError: Raised when the result of an arithmetic operation exceeds the maximum representable value for a numeric type.

MemoryError: Raised when the Python interpreter runs out of memory to allocate for an object.

ImportError: Raised when an import statement fails to find and load a module.

StopIteration: Raised when the next() function is called on an iterator and there are no more items to be returned.

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

Ans: Logging in Python is a built-in module that provides a flexible and configurable way to record log messages during the execution of a program. It allows developers to track and record important information, warnings, errors, and debugging messages that occur during the program's execution.

1) Debugging and Troubleshooting

2) Monitoring and Auditing

3) Error Analysis and Maintenance

4) Documentation and Communication

5) Flexibility and Configurability

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

```
import logging

logging.debug("Entering function foo().")  #This log level is used for
detailed
#debugging information. It is typically used during development and is the
most verbose log level.



logging.info("Server started on port 8000.")  #This log level is used to
provide general information about the program's execution.
#It can be used to track important milestones or events that help
understand the program's progress.

logging.warning("Disk space is running low.") #This log level indicates a
potential issue or a situation that may lead to an error in the future.
#It is used to highlight abnormal or unexpected events that should be noted
but do not cause the program to stop

logging.error("Failed to connect to the database.") #This log level is used
to indicate errors that prevent the program from functioning as
# intended. 3 It signifies a failure in some part of the program's
execution.

logging.critical("System is running out of memory.")   #This log level
represents a critical error or a severe failure that requires immediate
# attention. It indicates a condition that might cause the program to
terminate or stop working correctly.
```

```
WARNING:root:Disk space is running low.
ERROR:root:Failed to connect to the database.
CRITICAL:root:System is running out of memory.
```

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

```
#Log levels helpdevelopers control the verbosity of the logging output and
enable
#them to focus on specific types of messages depending on the situation.
import logging

logging.basicConfig(level=logging.INFO)

# ...

logging.info("Data processing completed successfully.")
```

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

```
#import logging ----for importing the external files
#formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
Create an instance of the formatter class you want to use.
#logger = logging.getLogger()
#logger.addHandler(handler) Create a logger instance and attach the handler
to it
```

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

```
import logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
handler = logging.StreamHandler()
logger.addHandler(handler)
logger.debug("This is a debug message.")
logger.info("This is an informational message.")
logger.warning("This is a warning message.")
```

```
This is a warning message.
This is a warning message.
WARNING:__main__:This is a warning 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?

The logging and print statements in Python serve different purposes and have distinct characteristics. Here are the key differences between them:

1. Output Control
2. Configurability
3. Log Levels and Severity
4. Debugging and Development
5. Production Environment
6. Long-Term Maintenance
7. Integration and Automation

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(
    level=logging.INFO,
    filename='app.log',
    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(
    level=logging.ERROR,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler(),  # Output to console
        logging.FileHandler('errors.log')  # Output to file
    ]
)

try:
    # Code that may raise an exception
    # ...
    raise ValueError("An error occurred.")
except Exception as e:
    # Log the error message
    logging.error(f"{type(e).__name__}: {e}")
```

```
ERROR:root:ValueError: An error occurred.
```