1. What is the role of the 'else' block in a try-except statement? Provide an example scenario where it would be useful.
The 'else' block in a try-except statement executes only if no exceptions were raised in the 'try' block. It's useful for code that should run only if the 'try' block was successful, avoiding accidental exception catching.

Example :-
```
try:
    f = open("myfile.txt", "r")
except FileNotFoundError:
    print("File not found.")
else:
    contents = f.read()
    print(contents)
    f.close()
```

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

Yes, a try-except block can be nested inside another. This allows for handling exceptions at different levels of code.

For example:

```
try:

    numerator = int(input("Enter numerator: "))

    try:

        denominator = int(input("Enter denominator: "))

        result = numerator / denominator

        print("Result:", result)

    except ZeroDivisionError:

        print("Cannot divide by zero.")

except ValueError:

    print("Invalid input. Please enter numbers only.")
```

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

You create a custom exception by inheriting from `Exception`.

```
class MyError(Exception):

    def __init__(self, message):

        super().__init__(message)

Try:

    raise MyError("Something went wrong!")

except MyError as e:

    print(e)
```

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

Some common built-in Python exceptions include:

- `TypeError`**: Incorrect data type used in an operation.**
- `ValueError`**: Correct data type, but inappropriate value.**
- `IndexError`**: Index out of range for a sequence.**
- `KeyError`**: Key not found in a dictionary.**
- `FileNotFoundError`**: File not found.**
- `ZeroDivisionError`**: Division by zero.**
- `IOError`**: Input/output operation failure.**
- `ImportError`**: Module import failure.**

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

Logging in Python is a way to record events that occur during the execution of a program. It's crucial for:

- Debugging: Understanding program flow and identifying errors.
- Monitoring: Tracking application performance and behavior in production.
- Auditing: Recording important events for security or compliance purposes.

Instead of just printing to the console, logging allows you to categorize messages by severity (debug, info, warning, error, critical) and output them to different destinations (files, network sockets, etc.), providing more structured and persistent records.

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

Log levels categorize log messages by severity. From least to most severe:

- DEBUG: Detailed information, useful for debugging. (e.g., variable values, function entry/exit)
- INFO: General information about program execution. (e.g., start/stop of processes)
- WARNING: Indicates potential problems or unexpected events. (e.g., low disk space, deprecated function usage)
- ERROR: Indicates errors that prevent some functionality from working. (e.g., file not found, network connection failure)
- CRITICAL: Indicates severe errors that may lead to program termination. (e.g., out of memory, data corruption)

Choosing the right level helps filter and prioritize log messages.

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

Log formatters in Python define the structure of log messages.[1] They use format strings with placeholders to include information like timestamp, log level, message, etc

You customize the format using `logging.Formatter`

```
import logging

# Create a logger

logger = logging.getLogger('my_logger')

logger.setLevel(logging.DEBUG)

# Create a handler (e.g., to write to console)

handler = logging.StreamHandler()

# Create a formatter

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# Set the formatter for the handler

handler.setFormatter(formatter)

# Add the handler to the logger

logger.addHandler(handler)

# Log a message

logger.debug('This is a debug message')
```

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

`main.py`:

```
import logging

import module_a

import module_b


logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')

logger = logging.getLogger(__name__)

logger.info("Starting")

module_a.do_something()

module_b.do_something_else()

logger.info("Finished")
```

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?

`print` displays output to the console, primarily for immediate feedback during development. `logging` is a more robust system for recording events, categorized by severity, and can output to various destinations (files, etc.).

Use `logging` over `print` in real-world applications for:

- Severity levels: Categorizing messages (debug, info, warning, error, critical).
- Output destinations: Sending logs to files, network, etc.
- Configuration: Controlling logging behavior without changing code.
- Contextual information: Including timestamps, logger names, etc.

`print` is suitable for quick debugging or simple scripts; `logging` is essential for maintainable and monitorable applications.

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

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

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

                    filemode='a') # 'a' for append

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

logging.basicConfig(level=logging.ERROR,

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

                    handlers=[

                        logging.FileHandler("errors.log"),

                        logging.StreamHandler()

                    ])


try:

    result = 10 / 0

except Exception as e:

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