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 is used to specify a block of code that should execute if no exceptions are raised within the associated "try" block. Eg:

In [2]:
```
try:
    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("You cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter valid numbers.")
else:
    print("The result of division is:", result) # This block prints the re
```

```
Enter the first number: 30
Enter the second number: 5
The result of division is: 6.0
```

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 try-except block.

In [4]:
```
try:
    # Outer try block
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))

    try:
        # Inner try block
        result = numerator / denominator
    except ZeroDivisionError:
        print("Inner exception: You cannot divide by zero.")
    except ValueError:
        print("Inner exception: Invalid input. Please enter valid numbers.

except ZeroDivisionError:
    print("Outer exception: You cannot divide by zero.")
except ValueError:
    print("Outer exception: Invalid input. Please enter valid numbers.")
else:
    print("Result of division:", result)
```

```
Enter the numerator: 4
Enter the denominator: g
Outer exception: Invalid input. Please enter valid numbers.
```

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

In [7]:
```python
# Custom exception for handling negative values.
class NegativeValueError(Exception):

    def __init__(self, value):
        super().__init__(f"Negative value not allowed: {value}")

def process_positive_number(number):
    if number < 0:
        raise NegativeValueError(number)
    return number * 2

try:
    user_input = int(input("Enter a positive number: "))
    result = process_positive_number(user_input)
    print("Result:", result)
except NegativeValueError as e:
    print("Custom exception caught:", e)
except ValueError:
    print("Invalid input. Please enter a positive number.")
```

```
Enter a positive number: -7
Custom exception caught: Negative value not allowed: -7
```

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

Some common built-in python exceptions are given below:

1. SyntaxError: Raised when there is a syntax error in the code, typically due to a violation of the Python language rules.
2. IndentationError: Raised when there is an issue with the indentation of code, such as mixing tabs and spaces or incorrect indentation levels.
3. NameError: Raised when a local or global name is not found. This can happen if you try to access a variable or function that has not been defined.
4. TypeError: Raised when an operation is performed on an object of an inappropriate type. For example, attempting to add a string and an integer.
5. ValueError: Raised when a function receives an argument of the correct type but with an inappropriate value. For instance, trying to convert a non-integer string to an integer.
6. ZeroDivisionError: Raised when an attempt is made to divide a number by zero.
7. IndexError: Raised when you try to access an index in a sequence (e.g., list or tuple) that is outside the valid range.

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

Logging in Python refers to the process of recording or capturing information about the execution of a program, such as messages, errors, warnings, and other relevant data, and writing this information to various output destinations like files, the console, or external services.

Logging is a critical component of software development that helps with debugging, monitoring, maintenance, security, and overall software quality.

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

Log levels in Python logging are used to categorize log messages based on their severity or importance. They allow developers to control which messages get recorded and reported, making it easier to filter and prioritize log information.

In [13]:
```python
# DEBUG (10):
import logging
x=4
logging.debug("Debug message: Variable x = %s", x)
```

In [11]:
```python
# INFO (20):
import logging
logging.info("Application started.")
```

In [12]:
```python
# WARNING (30):
import logging
logging.warning("Resource usage is approaching the limit.")
```

```
WARNING:root:Resource usage is approaching the limit.
```

In [ ]:
```python
# ERROR (40):
import logging
try:
    # Code that may raise an error
except Exception as e:
    logging.error("An error occurred: %s", e)
```

In [15]:
```python
# CRITICAL (50):
import logging
logging.critical("Critical system failure. Shutting down.")
```

```
CRITICAL:root:Critical system failure. Shutting down.
```

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

Log Formatters allow developers to customize the appearance and information included in log messages, making them more readable and suitable for the specific needs of an application.

In [18]:
```
import logging

# Create a logger
logger = logging.getLogger("my_logger")
# Create a formatter and set the desired log message format
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(

# Log messages
logger.debug("This is a debug message.")
logger.info("This is an info message.")
logger.warning("This is a warning message.")
logger.error("This is an error message.")
logger.critical("This is a critical message.")

```

```
2023-09-05 13:51:26,619 - my_logger - WARNING - This is a warning message.
WARNING:my_logger:This is a warning message.
2023-09-05 13:51:26,622 - my_logger - ERROR - This is an error message.
ERROR:my_logger:This is an error message.
2023-09-05 13:51:26,625 - my_logger - CRITICAL - This is a critical message.
CRITICAL:my_logger:This is a critical message.
```

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

To set up logging to capture log messages from multiple modules or classes using the built-in logging module. Here's a step-by-step guide on how to set up logging for multiple modules or classes:

1. Import the logging module in each module or class that needs logging.
2. Use logging.getLogger() to create a logger object with a name for each module or class.
3. Use logging.basicConfig() to configure the filename, level, and format of the log messages in the main module or class.
4. Use logger.info(), logger.debug(), logger.error(), etc. to write log messages with different levels of severity.

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?

Difference between logging and print statements:

- Logging:
  - Configurable and flexible for controlling log levels and outputs.
  - Provides structured information (timestamps, log levels, source).
  - Suitable for production use and long-running applications.
  - Supports log file persistence.
  - Used in production environments and for monitoring.
- Print Statements:
  - Simple and immediate output to the console.
  - Lacks configurability for filtering or structured data.
  - Convenient for quick debugging during development.

When to use logging over print statements:

- Use logging in production code, long-running applications, and collaborative projects.
- Use print statements for quick debugging during development but replace them with logging for production and monitoring.

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.

In [4]:

```
import logging

# Configure logging to write to the "app.log" file in append mode
logging.basicConfig(filename='app.log', level=logging.INFO, format='%(asct

# Log the message "Hello, World!" with INFO level
logging.info("Hello, World!")

# Closing the log file
logging.shutdown()
```

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.

In [4]:

```python
import logging

# Configure logging to write errors to both console and "errors.log" file
logging.basicConfig( level=logging.ERROR, format='%(asctime)s - %(levelnam
        handlers=[ logging.StreamHandler(),  # Log to console
        logging.FileHandler('errors.log', mode='a')])  # Log to "errors.lo

try:
    # Your code that may raise an exception
    result = 10 / 0  # This will raise a ZeroDivisionError
except Exception as e:
    # Log the exception with timestamp
    logging.error(f"Exception: {e}")
```

```
2023-09-07 13:31:14,108 - ERROR - Exception: division by zero
```