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

Ans: In a try-except statement, the 'else' block is optional and serves as a container for code that should only run if no exceptions are raised within the 'try' block. It provides a way to separate the code that may raise an exception from the code that should execute when no exceptions occur.

```
try:
   # Code that may raise an exception
except SomeException:
   # Code to handle the exception
else:
   # Code that will run if no exceptions were raised
```

```
def divide_numbers():
   try:
      num1 = int(input("Enter the first number: "))
      num2 = int(input("Enter the second number: "))
   except ValueError:
      print("Invalid input. Please enter valid integers.")
   except ZeroDivisionError:
      print("Error: Cannot divide by zero.")
   else:
      result = num1 / num2
      print(f"The result of {num1} / {num2} is {result}.")

divide_numbers()
```

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 nesting allows for more fine-grained error handling and can be useful when dealing with complex or specific scenarios.

```
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        result = None
    except TypeError:
        print("Error: Invalid data types for division.")
        result = None
    else:
        print("Division successful.")
    return result

def perform_calculation(x, y, z):
    try:
        result = divide_numbers(x, y)
        if result is not None:
            result = divide_numbers(result, z)
    except ValueError:
        print("Error: Invalid input value.")
        result = None
    else:
        print("Calculation successful.")
    return result

# Test the nested try-except blocks
a, b, c = 10, 2, 0

result = perform_calculation(a, b, c)
print("Final result:", result)
```

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

Ans:
```
# Custom exception class definition
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(message)

# Example function that raises the custom exception
def divide(a, b):
    if b == 0:
        raise CustomError("Division by zero is not allowed.")
    return a / b

# Example usage of the custom exception
try:
    result = divide(10, 2)  # This will work fine, no exception raised.
    print("Result:", result)

    result = divide(10, 0)  # This will raise the custom exception.
    print("Result:", result)  # This line will not be executed.
except CustomError as e:
    print("Custom Error occurred:", e)
except Exception as e:
    print("Other error occurred:", e)
```

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

Ans: In Python, built-in exceptions are pre-defined error classes that handle various types of errors or exceptional situations. These exceptions are a crucial part of error handling in Python and help developers identify and deal with issues that may arise during program execution. Some common built-in exceptions in Python include:

SyntaxError: Raised when there is a syntax error in the code.
IndentationError: Raised when there is an indentation-related syntax error, typically due to incorrect spacing.
NameError: Raised when a variable or name is not found in the current scope.
TypeError: Raised when an operation or function is applied to an object of an inappropriate type.
ValueError: Raised when a function receives an argument of the correct data type but an inappropriate value.
KeyError: Raised when a dictionary key is not found.
IndexError: Raised when trying to access an invalid index in a sequence (e.g., list, tuple).
AttributeError: Raised when an attribute reference or assignment fails.
FileNotFoundError: Raised when a file or directory is requested but cannot be found.
ZeroDivisionError: Raised when attempting to divide by zero.
ImportError: Raised when importing a module or package fails.
AssertionError: Raised when an **assert** statement fails.
StopIteration: Raised when there are no more items to be returned by an iterator.
KeyboardInterrupt: Raised when the user interrupts the execution of the program, usually by pressing Ctrl+C.
OverflowError: Raised when an arithmetic operation exceeds the maximum limit of the data type.

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

Ans: Logging in Python refers to the process of capturing and recording messages, events, or data during the execution of a program. It provides a systematic way to store information about the program's activities, errors, warnings, and other relevant information. Python's standard library includes a logging module that makes it easy to implement logging functionality in applications.
Logging is essential in software development for several reasons:
Debugging and Troubleshooting: When something goes wrong in a software application, logs can be invaluable in identifying the root cause of the issue. By examining the logged information, developers can understand the sequence of events leading up to the error, helping them locate and fix bugs more efficiently.

Monitoring and Performance Analysis: In production environments, logs play a crucial role in monitoring the application's behavior and performance. System administrators can analyze these logs to detect performance bottlenecks, identify usage patterns, and optimize the application's performance accordingly.
Security and Auditing: Logging can aid in tracking suspicious activities and potential security breaches. By logging relevant events, developers and security teams can detect and respond to security threats promptly.
Documentation: Logs serve as a historical record of an application's activities. They can be used as documentation to understand past behavior, which can be helpful when analyzing the system's evolution or making important business decisions.
Maintenance and Upgrades: When making updates or modifications to an application, logs can help validate the changes made and ensure that the application behaves as expected.
Python's logging module provides a flexible and customizable logging framework. It supports different log levels, such as DEBUG, INFO, WARNING, ERROR, and CRITICAL, allowing developers to control the verbosity of the logged information. This enables them to log more detailed information during development and reduce verbosity in production environments, helping to manage log sizes and prevent performance issues.

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

Ans: DEBUG: This is the lowest log level and is used for detailed information during development and debugging. It's typically used to log variable values, function calls, and other fine-grained details that can help developers understand the flow of execution and pinpoint issues.

```
import logging

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

def some_function():
    logger.debug("This is a debug message.")
    # Some code...
```

```
some_function()
```

INFO: The INFO log level is used to convey general, high-level information about the application's operation. It is useful for tracking significant events or milestones during the program's execution.

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def some_function():
    logger.info("Application started.")
    # Some code...

some_function()
```

WARNING: This log level is used to indicate potential issues or situations that might cause problems in the future. It's higher in severity than INFO but does not necessarily represent a critical problem.

```
import logging

logging.basicConfig(level=logging.WARNING)
logger = logging.getLogger(__name__)

def some_function():
    # Some code...
    if some_condition:
        logger.warning("The condition might cause unexpected behavior.")

some_function()
```

ERROR: The ERROR level is used to indicate errors that caused the program to fail to perform a specific task or function. These errors are unexpected and can negatively impact the application's functionality.

```
import logging
```

```
logging.basicConfig(level=logging.ERROR)
logger = logging.getLogger(__name__)

def some_function():
    try:
        # Some code that might raise an exception...
        raise ValueError("Something went wrong!")
    except Exception as e:
        logger.error(f"An error occurred: {e}")

some_function()
```

CRITICAL: This is the highest log level and is reserved for catastrophic failures that prevent the application from continuing. Critical log messages typically represent severe errors or unexpected conditions that demand immediate attention.

```
import logging

logging.basicConfig(level=logging.CRITICAL)
logger = logging.getLogger(__name__)

def some_function():
    # Some code that leads to a critical failure...
    logger.critical("A critical error occurred. Shutting down the application.")
    raise SystemExit(1)

some_function()
```

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

Ans: In Python's logging module, log formatters are used to specify the format of log messages that are generated during the execution of a program. A log formatter defines the structure and content of each log entry, including information like the timestamp, log level, logger name, and the actual message being logged.

The logging module provides a set of built-in formatters, but you can also customize the log message format to suit your specific needs. To create a custom log message format using formatters, you need to follow these steps

Import the necessary module:

```
import logging
```

Create a formatter instance:

```
formatter = logging.Formatter('[%(asctime)s] [%(levelname)s] [%(name)s] %(message)s')
```

In this example, the format string uses placeholders enclosed in **%(%)**. Each placeholder corresponds to a specific piece of information in the log message:
%(asctime)s: The timestamp when the log message was created.
%(levelname)s: The log level (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL).
%(name)s: The name of the logger used to log the message.
%(message)s: The actual log message.

Configure the logger to use the custom formatter:

```
logger = logging.getLogger('my_logger')
handler = logging.StreamHandler()  # or FileHandler(filename) for logging to a file
handler.setFormatter(formatter)
logger.addHandler(handler)
```

Log messages using the configured logger:

```
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')
```

When you run your program with the above configuration, the log messages will be displayed or saved according to the format you specified in the **formatter**. For example:

[2023-07-19 12:34:56] [DEBUG] [my_logger] This is a debug message
[2023-07-19 12:34:56] [INFO] [my_logger] This is an info message
[2023-07-19 12:34:56] [WARNING] [my_logger] This is a warning message
[2023-07-19 12:34:56] [ERROR] [my_logger] This is an error message
[2023-07-19 12:34:56] [CRITICAL] [my_logger] This is a critical message

Customizing log message formats allows you to tailor the output to your specific needs, making it easier to read and analyze log data.

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

Ans: Setting up logging to capture log messages from multiple modules or classes in a Python application involves using the built-in logging module, which allows you to centralize and manage all your application's log messages in a single place. Here's a step-by-step guide on how to do it:

Import the logging module: Start by importing the logging module at the beginning of your Python script or module.

```
import logging
```

Configure the logging settings: Before you start logging messages, configure the logging settings according to your needs. This includes setting the logging level, defining a log format, and specifying a log file if necessary. The logging levels are used to control which log messages are captured based on their severity.

```
# Set the logging level (options: DEBUG, INFO, WARNING, ERROR, CRITICAL)
logging.basicConfig(level=logging.DEBUG)

# Optionally, specify a log file to store the log messages
```

logging.basicConfig(filename='app.log', level=logging.DEBUG)

Define a logger object for each module or class: In each module or class that you want to log messages from, create a logger object using the **logging.getLogger()** method. It's recommended to use the module or class name as the logger's name to easily distinguish the log origin.

logger = logging.getLogger(__name__)

Log messages using the logger object: Once you've created the logger object, you can use it to log messages with different severity levels (debug, info, warning, error, critical) and relevant information. The messages will be captured by the logging module and sent to the configured output, such as the console or a log file.

```
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.")
```

Run your application and check the logs: After setting up the logging in your application, run the script or application, and the log messages will be displayed on the console if you haven't specified a log file. If you have specified a log file, the log messages will be written to that file.
With this setup, you can easily capture log messages from multiple modules or classes, control the log level, and have a centralized way of managing your application's logs. It helps in debugging and monitoring the application's behavior during development and production.

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: In Python, both logging and print statements are used for displaying information during program execution, but they serve different purposes and have distinct features. Here are the main differences between the two:
Purpose and Functionality:

print statements: They are primarily used for debugging and simple information display. When you use print, the output is sent to the standard output (usually the console or terminal). It is commonly used during development to quickly check the values of variables, flow of execution, or to get insights into the program's behavior.
logging: The logging module in Python is designed for more robust and flexible logging capabilities. It provides a range of log levels (debug, info, warning, error, critical), allowing you to control the granularity of the logged information. The logs can be configured to be stored in different destinations (e.g., file, console, network, etc.) and can include additional details such as timestamps, log levels, etc. Logging is intended for providing information about the program's execution, errors, and events, even in production environments.
Control and Flexibility:
print statements: With print, you have limited control over the format of the output. You can display values of variables or strings, but if you want to add more details or control the output's appearance, it requires extra manipulation of the printed text. Additionally, you need to manually remove or comment out print statements after debugging to avoid cluttering the final code.
logging: Logging provides more flexibility and control. You can easily customize the log format, specify different handlers for different log levels (e.g., displaying info messages on the console and writing error messages to a file), and adjust the logging behavior without modifying the source code. This makes it easier to maintain and manage logging in a complex application.
Granularity:
print statements: Usually, you use print statements for temporary and quick debugging purposes. They are not suitable for large-scale or production-level logging due to their limited capabilities and the extra effort required to manage them.
logging: Logging is well-suited for real-world applications, especially in production environments. You can choose the appropriate log level, and based on that, control the amount of information being logged. This ensures that only relevant and necessary information is recorded, preventing log file bloat and allowing you to analyze and troubleshoot issues effectively.
When to use logging over print statements in a real-world application:
Debugging: During the development phase, using print statements can be helpful for simple debugging and understanding the program's flow. However, when it comes to identifying and fixing issues in a production environment, logging is more appropriate. It allows you to keep a record of program execution, identify errors, and trace issues without modifying the source code.

Production Logging: In a real-world application running in production, you should avoid using print statements altogether. Instead, use the logging module to handle logs effectively. Logging provides the necessary controls to manage log levels, store logs in files, integrate with other monitoring systems, and ensure the application's stability and performance.
Informational Messages: For providing informative messages about the application's behavior or significant events, logging is the way to go. It allows you to categorize and manage different types of information systematically.
Security and Privacy: In production applications, you might want to log certain sensitive information for troubleshooting, but you don't want it to be exposed to the standard output. The logging module provides secure ways to handle such data, like setting up appropriate log handlers to store sensitive logs securely.
Overall, while print statements serve their purpose during development and quick debugging, the logging module is a more powerful and reliable tool for handling logging in real-world applications, offering better control, flexibility, and management of logs.

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: import logging

```
def setup_logger(log_file):
    # Create a logger
    logger = logging.getLogger('my_logger')
    logger.setLevel(logging.INFO)

    # Create a file handler that appends logs to the file
    file_handler = logging.FileHandler(log_file, mode='a')

    # Create a formatter for the log messages
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s',
datefmt='%Y-%m-%d %H:%M:%S')
```

```
    # Set the formatter for the file handler
    file_handler.setFormatter(formatter)

    # Add the file handler to the logger
    logger.addHandler(file_handler)

    return logger

def main():
    log_file = "app.log"
    logger = setup_logger(log_file)

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

if __name__ == "__main__":
    main()
```

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:

```
import logging
import datetime

def main():
    # Set up the logger
    logging.basicConfig(
        level=logging.ERROR,
        format='%(asctime)s [%(levelname)s]: %(message)s',
        datefmt='%Y-%m-%d %H:%M:%S',
        handlers=[
            logging.StreamHandler(),  # Log to console
            logging.FileHandler('errors.log')  # Log to file
        ]
    )
```

```
    try:
        # Your main program logic goes here
        # For demonstration purposes, let's raise an exception
        x = 10 / 0
    except Exception as e:
        # Log the exception with the error level
        logging.error(f'Exception occurred: {e}')

if __name__ == "__main__":
    main()
```