**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 define a block of code that should run only if no exceptions were raised in the corresponding try block. In other words, the else block contains code that is executed when the try block completes successfully without encountering any exceptions.

Example Scenario:

Suppose you have a program that reads data from a file using the open() function. You want to perform some processing on the data and then close the file. If the file doesn't exist or there is an issue with processing the data, you want to catch and handle the exceptions. However, if the file is successfully read and the data is processed without errors, you want to ensure that the file is properly closed. Here's where the else block comes into play:

try:

file = open("data.txt", "r")

data = file.read()

# Process the data

except FileNotFoundError:

print("File not found.")

except Exception as e:

print("An error occurred:", e)

else:

file.close() # Close the file only if no exceptions occurred

print("File closed successfully.")

**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. This is known as nested exception handling. It allows you to handle different types of exceptions at different levels of your code. The inner try-except block can provide more specific exception handling, while the outer block can handle more general exceptions.

Example:

Suppose you have a program that reads data from a file, processes the data, and then performs a division operation. Here's how nested try-except blocks can be used:

try:

file = open("data.txt", "r")

data = file.read()

try:

result = int(data) / 0 # Division by zero

except ZeroDivisionError:

print("Cannot divide by zero.")

except FileNotFoundError:

print("File not found.")

except ValueError:

print("Invalid data format.")

except Exception as e:

print("An error occurred:", e)

finally:

file.close() # Close the file in any case

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

To create a custom exception class in Python, you need to define a new class that inherits from the built-in Exception class or any of its subclasses. By doing this, you can create a specialized exception type that provides more specific information about the error that occurred. Here's an example:

class ValueTooSmallError(Exception):

def \_\_init\_\_(self, value, min\_value):

self.value = value

self.min\_value = min\_value

self.message = f"Value {value} is too small. Minimum allowed value is {min\_value}."

super().\_\_init\_\_(self.message)

def check\_value(x):

if x < 10:

raise ValueTooSmallError(x, 10)

try:

num = int(input("Enter a number: "))

check\_value(num)

print("Value is acceptable.")

except ValueTooSmallError as vts\_err:

print(vts\_err)

except ValueError:

print("Please enter a valid number.")

In this example:

* We've created a custom exception class ValueTooSmallError that inherits from the base Exception class.
* The custom exception class takes two arguments: the value that caused the error and the min\_value that defines the minimum allowed value.
* Inside the check\_value function, we raise the custom exception if the input value is too small.
* When using the check\_value function, if an exception is raised, the specific error message from the custom exception class is displayed.
* Custom exception classes allow you to provide more descriptive and specific error messages, making it easier to understand what went wrong in your code.

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

Python provides a wide range of built-in exceptions to handle different types of errors and exceptional situations. Here are some common built-in exceptions in Python:

* ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.
* ValueError: Raised when a function receives an argument of the correct type but inappropriate value.
* TypeError: Raised when an operation or function is applied to an object of inappropriate type.
* IndexError: Raised when an index is out of range for a sequence like a list or string.
* KeyError: Raised when a dictionary key is not found.
* FileNotFoundError: Raised when an attempt to open a file fails because the file does not exist.
* IOError: Raised when an I/O operation (like reading or writing a file) fails.
* NameError: Raised when a local or global name is not found.
* ImportError: Raised when an import statement fails to import a module.
* AttributeError: Raised when an attribute reference or assignment fails.
* SyntaxError: Raised when there's a syntax error in the code.
* TypeError: Raised when an operation is performed on an object of inappropriate type.
* OverflowError: Raised when arithmetic operation exceeds the limits of a numeric type.
* MemoryError: Raised when an operation runs out of memory.

These are just a few examples of the built-in exceptions available in Python. Each exception type serves a specific purpose and helps you handle different kinds of errors that might occur during program execution.

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

Logging in Python refers to the practice of recording messages, events, and other relevant information during the execution of a software program. It involves using the built-in logging module to capture and store log messages that provide insights into the program's behavior, execution flow, errors, and more. Logging is important in software development for several reasons:

* Debugging and Troubleshooting: Logging allows developers to capture information about the program's execution, making it easier to identify and diagnose issues, errors, and unexpected behavior. By examining log messages, developers can pinpoint where and why problems occur.
* Real-time Monitoring: In production environments, logging provides a way to monitor the application's performance and behavior in real time. It helps identify performance bottlenecks, errors, and anomalies, allowing for timely intervention.
* Audit Trails: Logging helps maintain a record of significant events and user interactions, which is crucial for auditing, compliance, and tracking the flow of actions within an application.
* Documentation: Log messages serve as a form of documentation, providing a historical record of the program's execution and any changes or actions taken during runtime.
* Improving Code Quality: Logging encourages developers to write more reliable code by helping them anticipate and address potential issues. It also aids in validating assumptions about program behavior.
* Release and Deployment: Logging assists in assessing the readiness of a software application for release by providing insights into its stability, performance, and readiness for deployment.
* Communication and Collaboration: Log messages can be shared with colleagues, support teams, and other stakeholders to facilitate communication and collaboration, especially when troubleshooting issues in distributed systems.
* Security: Logging can help detect and respond to security breaches and suspicious activities by providing a record of events that might indicate unauthorized access or unusual behavior.
* By incorporating proper logging practices into software development, developers and stakeholders gain visibility into the inner workings of the application, enabling them to make informed decisions and respond effectively to various situations that arise during the software's lifecycle.

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

The purpose of log levels in Python logging is to categorize and prioritize log messages based on their severity and importance. Each log level corresponds to a different level of severity, allowing developers to control the verbosity and relevance of logged information. Different log levels are used to indicate the significance of log messages and determine how they are handled and displayed. Here are examples of when each log level would be appropriate:

DEBUG:

Use: For detailed debugging information during development and testing.

Example: Printing variable values, execution paths, and other low-level details that help diagnose issues.

Code Example:

import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("Debugging information: %s", some\_variable)

INFO:

Use: For general information about the program's progress or status.

Example: Indicating successful completion of significant program stages or displaying status updates.

Code Example:

import logging

logging.basicConfig(level=logging.INFO)

logging.info("Processing complete for file: %s", filename)

WARNING:

Use: For potential issues that might not lead to immediate errors but could cause problems in the future.

Example: Deprecation warnings, non-critical inconsistencies in data, or unusual behavior.

Code Example:

import logging

logging.basicConfig(level=logging.WARNING)

logging.warning("Data format seems unusual for user: %s", username)

ERROR:

Use: For errors that prevent certain parts of the program from functioning as intended.

Example: Failed operations, unhandled exceptions, or invalid input that leads to incorrect behavior.

Code Example:

import logging

logging.basicConfig(level=logging.ERROR)

logging.error("An error occurred while processing data: %s", error\_message)

CRITICAL:

Use: For critical errors that can lead to program failure or data loss.

Example: System crashes, unhandled exceptions in critical components, or security breaches.

Code Example:

import logging

logging.basicConfig(level=logging.CRITICAL)

logging.critical("Critical error: System is shutting down due to data corruption.")

Using appropriate log levels helps in tailoring the level of detail in logs according to the situation, making it easier to identify and resolve issues during development, testing, and production. It also aids in effective communication among developers and stakeholders by providing a clear indication of the severity of the logged events.

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

Log formatters in Python logging are used to define the structure and appearance of log messages that are generated by the logging module. They allow you to customize how log messages are formatted and presented in the output. By default, the logging module provides a basic formatter, but you can create custom formatters to display log messages with specific information, such as timestamps, log levels, and user-defined text.

To customize the log message format using formatters, you need to create an instance of the logging.Formatter class and set it as the formatter for the handlers you use. The formatter class provides various placeholders that you can use to insert different components of the log message, like timestamps, log levels, and messages.

Here's an example of how to customize log message format using a formatter:

import logging

# Create a custom formatter

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

# Create a handler and attach the formatter

handler = logging.StreamHandler()

handler.setFormatter(formatter)

# Create a logger and add the handler

logger = logging.getLogger('custom\_logger')

logger.addHandler(handler)

logger.setLevel(logging.INFO)

# Log messages with the custom format

logger.info("This is an informational message.")

logger.warning("This is a warning message.")

In this example, the custom formatter uses placeholders like %(asctime)s, %(levelname)s, and %(message)s to insert the timestamp, log level, and message text into the log output. You can modify the format string to arrange the components as desired. By using custom formatters, you can tailor the appearance of log messages to match your preferred style and include the information that is most relevant for your application's needs.

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

Setting up logging to capture log messages from multiple modules or classes in a Python application involves creating a consistent logging configuration and sharing the logger instances among the different parts of your application. Here's how you can achieve this:

Configure Logging in a Central Location:

* Create a central configuration for logging, typically at the beginning of your main script or module.
* Configure the logging format, handlers, and log levels.
* Set up loggers with unique names for different modules or classes.

Create and Use Logger Instances:

* In each module or class, create a logger instance using logging.getLogger('module\_name').
* This ensures that different parts of your application use the same logger configuration.
* The logger name should be unique to the module or class to differentiate log messages.

Use the Logger for Logging:

* Use the logger instance to log messages throughout the module or class.
* Log messages using appropriate log levels (debug, info, warning, error, critical).

Here's an example of how to set up logging to capture log messages from multiple modules:

# main.py

import logging

import module1

import module2

# Configure logging at the beginning of the main script

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

# Create logger instances for different modules

module1\_logger = logging.getLogger('module1')

module2\_logger = logging.getLogger('module2')

# Use logger instances to log messages in respective modules

module1\_logger.debug("Debug message from Module 1")

module2\_logger.info("Info message from Module 2")

# Call functions from other modules that use their respective loggers

module1.log\_function()

module2.log\_function()

# module1.py

import logging

# Create logger for this module

module1\_logger = logging.getLogger('module1')

def log\_function():

module1\_logger.warning("Warning message from Module 1")

# module2.py

import logging

# Create logger for this module

module2\_logger = logging.getLogger('module2')

def log\_function():

module2\_logger.error("Error message from Module 2")

In this example, the central configuration is set in the main.py script, and logger instances are created in each module. The log messages from different modules are captured with their respective loggers, and the messages can be controlled using the log levels specified in the configuration. This approach helps maintain consistency in logging across different parts of your application.

**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 advantages. Here's a comparison and guidance on when to use each in a real-world application:

**Logging:**

Purpose: Logging is used to record events, messages, and information about a program's execution. It's designed for debugging, monitoring, and understanding the behavior of the program during development, testing, and production.

Advantages:

* Log Levels: Logging provides different log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) to categorize messages based on their severity.
* Granular Control: You can configure logging to control which messages are displayed, saved to files, or sent to other destinations based on their severity.
* Timestamps: Logging automatically adds timestamps to log messages, aiding in understanding the sequence of events.
* Consistency: Logging provides a consistent format for log messages, making them more structured and informative.
* When to Use Logging:
* Use logging when you want to record information for analysis, troubleshooting, monitoring, and auditing.
* Logging is especially useful for production applications where you need to keep track of issues and events without interfering with the application's regular output.

**Print Statements:**

Purpose: Print statements are used to display immediate output on the console. They are primarily used for quick debugging and understanding the flow of the program during development.

Advantages:

* Simplicity: Print statements are easy to use and require minimal setup.
* Immediate Feedback: Print statements provide immediate feedback during program execution.
* Quick Debugging: You can quickly add print statements to inspect variable values and execution paths.
* When to Use Print Statements:
* Use print statements when you need a quick way to understand and troubleshoot the program's behavior during development.
* They are not recommended for production code as they might clutter the output and are harder to manage compared to structured logging.

In a Real-World Application:

* In a real-world application, you should use logging over print statements when you need to maintain a clear, organized, and structured record of your application's execution. Logging is essential for:
* Monitoring and maintaining production applications.
* Troubleshooting issues and tracking events.
* Providing detailed insights into the behavior of your application.
* Keeping a historical record of actions, errors, and interactions.
* While print statements have their place in quick debugging and experimentation during development, using structured logging with appropriate log levels provides a more professional and manageable approach for understanding and maintaining your software in the long run.

**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.**

Sure, here's a Python program that fulfills your requirements using the logging module:

import logging

# Configure logging to write to a file named "app.log"

logging.basicConfig(

filename="app.log",

level=logging.INFO,

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

)

# Log an "INFO" level message

logging.info("Hello, World!")

In this program:

* The logging.basicConfig() function sets up the logging configuration.
* The filename parameter specifies the name of the log file ("app.log").
* The level parameter sets the log level to "INFO," meaning only messages with "INFO" level or higher will be logged.
* The format parameter defines the format of the log messages, including the timestamp, log level, and message text.
* The logging.info() function is used to log an "INFO" level message with the content "Hello, World!"

When you run this program, it will create a log file named "app.log" (if it doesn't exist) and add the "Hello, World!" message to it as an "INFO" level log entry. Subsequent log entries will be appended to the same file without overwriting the previous ones.

**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.**

Python program that logs an error message to both the console and a file named "errors.log" if an exception occurs during its execution. The error message includes the exception type and a timestamp:

import logging

import sys

# Configure logging to log to console and a file named "errors.log"

logging.basicConfig(

level=logging.ERROR,

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

handlers=[

logging.StreamHandler(sys.stdout), # Log to console

logging.FileHandler("errors.log"), # Log to file

],

)

try:

# Simulate a division by zero error

result = 1 / 0

except Exception as e:

# Log the exception with timestamp and exception type

logging.error("Exception occurred: %s", e)

In this program:

* The logging.basicConfig() function configures the logging settings:
* The level parameter is set to logging.ERROR to only log error messages and more severe events.
* The format parameter defines the format of the log messages, including the timestamp, log level, and message text.
* Two handlers are specified:
* logging.StreamHandler(sys.stdout) logs to the console.
* logging.FileHandler("errors.log") logs to a file named "errors.log".
* Inside the try block, we simulate an exception by dividing by zero.
* The except block catches the exception and logs an error message that includes the exception type and timestamp. This message is sent to both the console and the "errors.log" file.
* When you run this program and an exception occurs, you will see the error message with the exception type and timestamp displayed on the console, and it will also be saved in the "errors.log" file.