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 used to specify a block of code that should be executed if no exception is raised within the associated try block.

Eg –

def divide(a, b):

try:

result = a / b

except ZeroDivisionError:

print("Error: Cannot divide by zero")

else:

print("Division result:", result)

# Example usage

divide(10, 2) # Output: Division result: 5.0

divide(10, 0) # Output: Error: Cannot divide by zero

1. 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. This is known as nested exception handling. It allows you to handle different levels of exceptions and provide more specific error handling for different parts of our code.

Eg -

def nested\_exception\_example(a, b, c):

try:

result = a / b

print("First division result:", result)

try:

another\_result = result / c

print("Second division result:", another\_result)

except ZeroDivisionError:

print("Error: Cannot divide by zero in the nested block")

except ZeroDivisionError:

print("Error: Cannot divide by zero in the outer block")

# Example

nested\_exception\_example(10, 2, 0)

1. 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 one of its subclasses. This allows you to create your own specific exception types to handle different error scenarios in your code.

Eg –

class CustomError(Exception):

"""Custom exception class."""

def \_\_init\_\_(self, message):

self.message = message

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

def process\_data(data):

if not isinstance(data, int):

raise CustomError("Data must be an integer")

if data < 0:

raise CustomError("Data cannot be negative")

return data \* 2

# Example usage

try:

value = process\_data("hello")

except CustomError as e:

print("Custom error:", e)

else:

print("Processed value:", value)

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

Ans: Some common exceptions are as follows:

1. SyntaxError: Raised when there is a syntax error in the code.
2. IndentationError: Raised when there is an issue with the indentation of the code.
3. NameError: Raised when a local or global name is not found.
4. TypeError: Raised when an operation or function is applied to an object of inappropriate type.
5. ValueError: Raised when a function receives an argument of the correct type but an inappropriate value.
6. KeyError: Raised when a dictionary key is not found.
7. IndexError: Raised when a sequence index is out of range.
8. FileNotFoundError: Raised when a file or directory is not found.
9. ZeroDivisionError: Raised when division or modulo by zero occurs.
10. IOError: Raised when an I/O operation (e.g., file operation) fails.
11. What is logging in Python, and why is it important in software development?

Ans: Logging in Python refers to the process of recording information, events, and messages generated during the execution of a program. It involves capturing various levels of details about the program's behavior and state and writing them to a log file or other output destinations.

Logging is important in software development for several reasons:

1. Debugging and Troubleshooting: Logging helps developers diagnose issues and bugs by providing a detailed history of program execution. When an error occurs, developers can examine the log to understand what happened leading up to the error.
2. Error and Exception Tracking: By logging errors and exceptions along with relevant context, developers can gain insights into the cause of unexpected behavior and quickly identify problematic code areas.
3. Monitoring and Performance Analysis: Logging can be used to monitor the performance of an application, tracking metrics like response times, request rates, and resource utilization. This information is valuable for optimizing and scaling the application.
4. Auditing and Compliance: Logging is crucial for compliance with regulations and standards in industries like finance and healthcare. Detailed logs can provide an audit trail of actions taken by the application and its users.
5. Security: Logging security-related events and access attempts can help detect unauthorized activities or potential security breaches.
6. Long-term Maintenance: As software evolves, maintaining and improving it becomes easier with a comprehensive log history. Logs can aid in understanding the evolution of the codebase and help in making informed decisions about future changes.
7. Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.

Ans: Log levels in Python logging provide a way to categorize and prioritize different types of log messages based on their severity and importance. Each log level represents a different level of detail and significance, allowing developers to control the verbosity of log output and focus on specific aspects of their application's behavior. Python's logging module defines several standard log levels, which are as follows:

* 1. **DEBUG**: The lowest log level, used for detailed debugging information. These messages are typically only relevant during development and should not be present in production code.

**Eg** - import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug message.")

* 1. **INFO**: Used to provide informational messages about the application's status or behavior. These messages can be helpful for tracking the general flow of the program.

**Eg** - import logging

logging.basicConfig(level=logging.INFO)

logging.info("Application started.")

* 1. **WARNING**: Used to indicate potential issues or situations that might cause problems in the future, but the program can continue executing.

**Eg** - import logging

logging.basicConfig(level=logging.WARNING)

logging.warning("Resource usage is nearing critical levels.")

* 1. **ERROR**: Used to indicate errors that caused a specific operation to fail or unexpected behavior that may impact the application's functionality.

**Eg** - import logging

logging.basicConfig(level=logging.ERROR)

logging.error("An error occurred while processing user input.")

* 1. **CRITICAL**: The highest log level, used to indicate severe errors or critical failures that could lead to application termination.

**Eg** - import logging

logging.basicConfig(level=logging.CRITICAL)

logging.critical("Server is down. Application cannot continue.")

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

Ans: Log formatters in Python logging are responsible for defining the structure and content of log messages that are outputted to various destinations, such as log files, console outputs, or external logging services.

Here's an example of how to customize the log message format using formatters:

import logging

# Configure logging

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

# Create a custom formatter

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

# Create a file handler and attach the formatter

file\_handler = logging.FileHandler('custom.log')

file\_handler.setFormatter(formatter)

# Create a logger and attach the file handler

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

logger.addHandler(file\_handler)

# Example usage

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

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

1. 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 configuring the **logging** module and creating loggers for each module or class. Each logger can be configured to use a specific log level, formatter, and output destination. This allows you to have fine-grained control over the logging behavior for different parts of your application.

Steps to setup logging to capture log messages from multiple modules or classes:

* 1. **Import the logging module**: Import the **logging** module at the beginning of your Python script or module. Code - import logging
  2. **Configure the logging settings**: Configure the logging settings for the root logger using **basicConfig**. You can set the default log level, format, and output destination for log messages.

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

* 1. **Create loggers for modules or classes**: In each module or class that you want to log messages from, create a logger using the **logging.getLogger()** method. Use a unique name for each logger to distinguish between different parts of your application.

Code- logger = logging.getLogger('module\_name')

* 1. **Configure the logger settings**: Configure each logger with its desired log level, formatter, and output destination. You can create and attach handlers to the logger to specify where the log messages will be written.

Code- # Create a file handler for the logger

file\_handler = logging.FileHandler('module\_name.log')

# Set the desired log level

file\_handler.setLevel(logging.DEBUG)

# Create a formatter for the log messages

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

# Attach the formatter to the handler

file\_handler.setFormatter(formatter)

# Add the handler to the logger

logger.addHandler(file\_handler)

* 1. **Use the logger to log messages**: In your module or class, use the logger to log messages at various log levels.

Code- logger.debug("This is a debug message.")

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

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

1. 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: Both logging and print statements are used for displaying information during the execution of a Python program.

Logging vs. Print Statements:

* Logging is for structured, configurable output with different levels (debug, info, warning, error) and contextual information (timestamps, module names).
* Print statements are simple, static output to the console, lacking configurability and different message levels.

Use Logging Over Print Statements:

* For debugging, troubleshooting, and maintaining applications.
* When you need different levels of detail and context.
* To control output with configurability, format, and destinations.
* In larger projects with collaboration and long-term maintenance.

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: Program is as follows:

import logging

# Configure logging

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

Ans: Program is as follows:

import logging

import traceback

# Configure logging to log errors to both console and file

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

logging.FileHandler('errors.log', mode='a'),

logging.StreamHandler()

])

def divide(a, b):

try:

result = a / b

return result

except ZeroDivisionError as e:

# Log the error with exception type and timestamp

logging.error("Error: %s", e)

traceback.print\_exc()

# Example usage

try:

result = divide(10, 0)

except Exception as e:

# Log the exception type and timestamp

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

traceback.print\_exc()