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 optional and provides a section of code that should be executed if no exceptions occur within the ‘try’ block. An example scenario program is as follows:

def perform\_calculation(a, b):

result = a / b

return result

try:

num1 = int(input("Enter the first number: "))

num2 = int(input("Enter the second number: "))

except ValueError:

print("Invalid input. Please enter valid integers.")

else:

try:

division\_result = perform\_calculation(num1, num2)

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

else:

print("Division successful. Result:", division\_result)

1. 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. For e.g.:

try:

# Outer try-except block

outer\_number = int(input("Enter a number: "))

try:

# Inner try-except block

result = 100 / outer\_number

print("Result:", result)

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

except ValueError:

print("Error: Invalid input!")

except Exception as e:

print("Error:", e)

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

* In Python, we can create a custom exception by inheriting from the built-in ‘Exception’ class. An example that demonstrates its usage is as follows:

class InvalidAge(Exception):

#Raised when input is less than 18

pass

try:

age = int(input(“Enter age: ”))

if age<18:

raise InvalidAge

else:

print(“Eligible to Vote”)

except InvalidAge:

print(“Invalid Age exception occured”)

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

* Some common exceptions that are built-in to Python are ‘NameError’, ‘ValueError’, ‘IndexError’, ‘KeyError’, etc.

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

* Logging in Python in a built-in module that allows developers to generate log messages and store them in various output destinations. Logging is important in software development for debugging and troubleshooting, monitoring and maintenance, performance analysis, etc.

1. 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 is to allow developers to control which log messages are displayed or recorded based on their severity, making it easier to filter and manage log output. The log levels along with their examples are as follows:
* DEBUG: The ‘DEBUG’ log level is the lowest severity level. It is used for tracking the variable values. For e.g.:

import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug(“This is a debug message.”)

* INFO: The ‘INFO’ log level provides messages that are used to check status updates. For e.g.:

import logging

logging.basicConfig(level=logging.INFO)

logging.info(“Application started successfully.”)

* WARNING: The ‘WARNING’ log level indicates a potential issue or a non-fatal that occurred during the execution of the code. For e.g.:

import logging

logging.basicConfig(level=logging.WARNING)

logging.warning(“Disk space is running out.”)

* ERROR: The ‘ERROR’ log level represents error conditions that occurred during the execution of the code. For e.g.:

import logging

logging.basicConfig(level=logging.ERROR)

logging.error(“An error occured.”)

* CRITICAL: The ‘CRITICAL’ log level represents the most severe level of logging. It is used to capture critical errors. For e.g.:

import logging

logging.basicConfig(level=logging.CRITICAL)

logging.critical("System failure detected. Shutting down.")

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

* In Python logging, log formatters are objects responsible for defining the format of log messages. We can customize the log message format using formatter as follows:

import logging

# Create a formatter

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

# Configure the logging settings

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

# Create a file handler and set its formatter

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

file\_handler.setFormatter(formatter)

# Add the file handler to the root logger

logging.getLogger().addHandler(file\_handler)

# Log messages

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

logging.info('This is an informational message')

logging.warning('This is a warning message')

logging.error('This is an error message')

logging.critical('This is a critical message')

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

* We can set up logging to capture log messages from multiple modules or classes in a Python application by following the steps below:
* Import the ‘logging’ module at the beginning of each module.
* Configure the logging settings in main script using ‘basicConfig’.
* Retrieve a logger instance using the module name as the logger name. This allows us to distinguish log messages from different modules.
* Log messages using the logger instance.

For e.g.:

module1.py:

import logging

# Retrieve the logger for this module

logger = logging.getLogger(\_\_name\_\_)

def some\_function():

logger.debug("This is a debug message from module1")

logger.info("This is an informational message from module1")

module2.py:

import logging

# Retrieve the logger for this module

logger = logging.getLogger(\_\_name\_\_)

class SomeClass:

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

logger.debug("Initializing SomeClass instance")

def some\_method(self):

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

main.py:

import logging

import module1

import module2

# Configure the logging settings

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

def main():

module1.some\_function()

obj = module2.SomeClass()

obj.some\_method()

if \_\_name\_\_ == '\_\_main\_\_':

main()

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?

* Print statements are straightforward and display output to the console whereas logging provides a better control over the output by allowing us to specify the desired log level, format and destination.

Print statements should be used for quick and temporary debugging, especially during development. Logging is more suitable for real-world applications and long-term maintenance as it provides a systematic approach to record events.

1. 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.

* The Python program is as follows:

import logging

# Set up logging configuration

logging.basicConfig(filename='app.log', level=logging.INFO, filemode='a', format='%(asctime)s - %(levelname)s - %(message)s')

# Log the message

logging.info('Hello, World!')

1. 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.

* The Python program is as follows:

import logging

import traceback

import sys

# Setting up logging configuration

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

# Creating a file handler to log errors to a file

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

file\_handler.setLevel(logging.ERROR)

file\_handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)s - %(message)s'))

# Adding the file handler to the root logger

logging.getLogger('').addHandler(file\_handler)

try:

# Program logic goes here

# Simulating an exception for demo

raise ValueError("An error occurred!")

except Exception as e:

# Logging the exception

logging.error(f"{type(e).\_\_name\_\_}: {e}")

logging.error(traceback.format\_exc())

# Printing the exception to the console

traceback.print\_exc(file=sys.stdout)