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

= The try block lets you test a block of code for errors. The except block lets you handle the error. The else block lets you execute code when there is no error. The finally block lets you execute code, regardless of the result of the try- and except blocks.

allows your program to take alternative actions in case an error occurs. Python will first attempt to execute the code in the try statement . If no exception occurs, the except statement is skipped and the execution of the try statement is finished.

The code enters the else block only if the try clause does not raise an exception.

def divide(x, y):

try:

result = x // y

except ZeroDivisionError:

print("Sorry ! You are dividing by zero ")

else:

print("Your answer is :", result)

divide(3, 2)

divide(3, 0)

Your answer is : 1

2. Can a try-except block be nested inside another try-except block? Explain with an example.

"try − except − finally" blocks inside the existing try block. The except keyword for inner try now handles generic Exception, while we ask the except block of outer try to handle ZeroDivisionError.

Since exception doesn't occur in the inner try block, its corresponding generic Except

a=10

b=0

try:

print (a/b)

except Exception:

print ("General Exception")

finally:

print ("inside outer finally block")

General Exception

inside outer finally block.

2. a=10

b=0

try:

print ("This is outer try block")

try:

print (a/b)

except ZeroDivisionError:

print ("Division by 0")

finally:

print ("inside inner finally block")

except Exception:

print ("General Exception")

finally:

print ("inside outer finally block")

This is outer try block

Division by 0

inside inner finally block

inside outer finally block

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

= as defining a new class that inherits from the built-in class.

We can define our own exceptions called custom exception.

def enterAge(age):

if age<0:

raise ValueError('Only positive integers are allowed')

if age % 2 ==0:

print('Entered Age is even')

else:

print('Entered Age is odd')

try:

num = int(input('Enter your age: '))

enterAge(num)

except ValueError:

print('Only positive integers are allowed')

Enter your age: 12

Entered Age is even

Enter your age: -2

Only positive integers are allowed.

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

= SyntaxError: its occur when such a misspelled keyword, a missing colon, or an unbalanced parenthesis.

TypeError: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.

NameError: This exception is raised when a variable or function name is not found in the current scope.

IndexError: This exception is raised when an index is out of range for a list, tuple, or other sequence types.

KeyError: This exception is raised when a key is not found in a dictionary.

ValueError: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.

AttributeError: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.

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

= logging is a module that allows you to track events that occur while your program is running. You can use logging to record information about errors, warnings, and other events that occur during program execution. And logging is a useful tool for debugging, troubleshooting, and monitoring your program.

monitor a program's behavior and performance, it makes it easier for them to find and fix problems.

It create a trail of what happens throughout companies' complex applications, systems and infrastructure. Logs are digital records of what has happened anywhere within the IT infrastructure of a business.

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

= There are six levels for logging in Python; each level is associated with an integer that indicates the log severity: NOTSET=0, DEBUG=10, INFO=20, WARN=30, ERROR=40, and CRITICAL=50.

Logging levels allow users to quickly identify critical issues by categorizing logs by type and severity.

it makes it simple for programmers and system administrators to monitor a program's behavior and performance, it makes it easier for them to find and fix problems.

Debug: These are used to give Detailed information, typically of interest only when diagnosing problems.

Info: These are used to confirm that things are working as expected.

Warning: These are used as an indication that something unexpected happened, or is indicative of some problem in the near future

Error: This tells that due to a more serious problem, the software has not been able to perform some function.

Critical: This tells serious error, indicating that the program itself may be unable to continue running.

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

= Loggers expose the interface used by the application code.

Handlers are created by loggers and send log records to the appropriate destination.

Filters can determine which log records are output.

Formatters specify the layout of the final log record output.

fmt for the log message format (including placeholders for the timestamp), and datefmt for the custom time format. By using this approach, you can easily customize the time format according to your preference and requirements.

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

= Creating a logger logger = logging. getLogger(name) logging. basicConfig(filename='example’.)

import logging

import mylib

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

def main():

logging.basicConfig(filename='myapp.log', level=logging.INFO)

logger.info('Started')

mylib.do\_something()

logger.info('Finished')

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

main()

2. import logging

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

def do\_something():

logger.info('Doing something')

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?

= Logging in Python is a technique to display useful messages and warnings to users. The logging module provides a flexible way to log different messages in various output destinations such as on the console, in files, and on networks.

Logging is a best practice for production code. The logging module provides features such as log levels and filtering.

The print statment is a built-in function in Python that prints the specified value or values to the console. It is mainly used for debugging and is not recommended for logging information in production code.

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.

= import logging

# Create and configure logger to append to the log file

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

# Log message

logging.info('Hello, World!')

Filemode=a will append new log entries without overwriting previous onces.

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.

= import logging

from datetime import datetime

# Create logger

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

logger.setLevel(logging.ERROR)

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.ERROR)

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

file\_handler.setLevel(logging.ERROR)

# Create formatter

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

# Add formatter to console and file handler

console\_handler.setFormatter(formatter)

file\_handler.setFormatter(formatter)

# Add handlers to logger

logger.addHandler(console\_handler)

logger.addHandler(file\_handler)

try:

with open('non\_existent\_file.txt', 'r') as f:

pass

except Exception as e:

logger.error('Error occurred: %s', type(e).\_\_name\_\_)