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 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.

try:

print("Hello")

except:

print("Something went wrong")

else:

print("Nothing went wrong")

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

Ans: yes we can try except block be nested inside antother try except block.

x = 10

y = 5

try:

print("outer try block")

try:

print("nested try block")

print(x / y)

except TypeError as te:

print("nested except block")

print(te)

except ZeroDivisionError as ze:

print("outer except block")

print(ze)

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

class InvalidAgeException(Exception):

"Raised when the Age value is less than 18"

pass

Age = 18

try:

Age = int(input("Enter the Age: "))

if Age < number:

,

else:

print("Eligibe to vote")

except InvalidAgeException:

print("Exception occured :Invalid Age")

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

Ans: some common error are ZeroDivisionError, TypeError, ValueError, IndexError, KeyError, FileNotFoundError, IOError, ImportError, MemoryError, OverflowError, AttributeError, SyntaxError, IndentationError, NameError**.**

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

Ans: Logging is a way to store information about your script and track events that occur. When writing any complex script in Python, logging is essential for debugging software as you develop it. Without logging, finding the source of a problem in your code may be extremely time consuming.

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

Log levels relate to the “importance” of the log. For example, an “error” log is a top priority and should be considered more urgent than a “warn” log. A “debug” log is usually only useful when the application is being debugged

* DEBUG: Detailed information, typically of interest only when diagnosing problems.
* INFO: Confirmation that things are working as expected.
* WARNING: An indication that something unexpected happened, or may happen in the future (e.g. ‘disk space low’). The software is still working as expected.
* ERROR: More serious problem that prevented the software from performing a function.
* CRITICAL: A very serious error, indicating that the program itself may be unable to continue running.

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

Ans: Log formatters are those which are used to customize the log messages

We can customize format handler using following code

handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s%(message)s'))

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

Ans:

def main():

import logging.config

logging.config.fileConfig('/path/to/logging.conf')

# your program code

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?

Ans: To define basic logging needs, several lines of code are needed. Including additional logging information is not easy. The print() statement only displays messages on the console. Recording logging data inside a file or sending it over the internet needs additional works   
One of the biggest advantages of proper logging is that you can categorize messages and turn them on or off depending on what you need. For example, it might be useful to turn on debugging level messages for a certain part of the project, but tone it down for other parts, so as not to be taken over by information overload and to easily concentrate on the task for which you need logging.

Also, logs are configurable. You can easily filter them, send them to files, format them, add timestamps, and any other things you might need on a global basis. Print statements are not easily managed

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.

import os

import logging

dir\_path =r'C:\Users\Home'

log\_file = 'app1.log'

full\_path = os.path.join(dir\_path,log\_file)

os.makedirs(dir\_path,exist\_ok=True)

logger.setLevel(logging.INFO)

handler = logging.FileHandler(full\_path)

handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(message)s'))

logger.addHandler(handler)

def LetSeeInfo(info):

logging.info('user %s welcomes you',info)

LetSeeInfo("Hello,World!")

handler.close()

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.

Ans:

import logging

class LoggerDemoConsole:

def testLog(self):

logger = logging.getLogger('demologger')

logger.setLevel(logging.INFO)

consoleHandler = logging.StreamHandler()

consoleHandler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s%(levelname)s: %(message)s', datefmt='%m/%d/%Y %I:%M:%S%p')

consoleHandler.setFormatter(formatter)

logger.addHandler(consoleHandler)

logger.debug('debug message')

logger.info('info message')

logger.warn('warn message')

logger.error('error message')

logger.critical('critical message')

demo = LoggerDemoConsole()

demo.testLog()