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

Else block is used executed when no exception is raised in try block.

Eg:

def division(a,b):

try:

div=a/b

except ZeroDivisionError:

print("ZeroDivisionError")

else:

print("output of division is: ",div)

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

Yes, it can be nested another try-except block.

try:

print("outer try block")

print(10/0)

try:

print("Inner try block")

except ZeroDivisionError:

print("Inner except block")

finally:

print("Inner finally block")

except:

print("outer except block")

finally:

print("outer finally block")

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

# A python program to create user-defined exception

# class MyError is derived from super class Exception

class MyError(Exception):

# Constructor or Initializer

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

self.value = value

# \_\_str\_\_ is to print() the value

def \_\_str\_\_(self):

return(repr(self.value))

try:

raise(MyError(3\*2))

# Value of Exception is stored in error

except MyError as error:

print('A New Exception occurred: ', error.value)

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

Some built-in common exceptions in python are:

* TypeError
* ZeroDivisionError
* FileNotFoundError
* IndexError

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

Logging is to log events when a software runs. It is mainly used in software development to trace errors and understand flow of the code through logs.

1. Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.
2. Debug**:** These are used to give Detailed information, typically of interest only when diagnosing problems.
3. Info**:** These are used to confirm that things are working as expected
4. Warning**:** These are used as an indication that something unexpected happened, or is indicative of some problem in the near future.
5. Error**:** This tells that due to a more serious problem, the software has not been able to perform some function
6. 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?

Log Formatters specify the layout of log records in the final output.

We can customise the log message format using formatters in below format.

logFormatter = logging.Formatter(fmt=' %(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?

We can import logging module and use in different module, classes to log different information in a file to debug and trace the errors.

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 statement

The [print statement](https://www.geeksforgeeks.org/python-output-using-print-function/) 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.

Logging in Python is a technique to display useful messages and warnings to users. The [logging module](https://www.geeksforgeeks.org/logging-in-python/) 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

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 logging

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

logger.setLevel(logging.INFO)

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

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

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

logger.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.

import logging

# Create a logging instance

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

logger.setLevel(logging.ERROR)

ch = logging.StreamHandler() #StreamHandler logs to console

ch\_format = logging.Formatter('%(asctime)s - %(message)s')

ch.setFormatter(ch\_format)

logger.addHandler(ch)

# Assign a file-handler to that instance

fh = logging.FileHandler("errors.log")

# Format your logs (optional)

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

fh.setFormatter(formatter) # This will set the format to the file handler

# Add the handler to your logging instance

logger.addHandler(fh)

try:

raise ValueError()

except ValueError as e:

logger.exception(e) # Will send the errors to the file