1. 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. For example,

try:

# Code that may raise an exception

except ExceptionType:

# Code to handle the exception

else:

# Code to execute if no exception occurs

print("No exceptions occurred")

2. Yes, a try-except block can be nested inside another try-except block. This is known as nested exception handling.

try:

# Outer try block

try:

# Inner try block

# Code that may raise an exception

except InnerException:

# Inner except block

# Code to handle the inner exception

except OuterException:

# Outer except block

# Code to handle the outer exception

3. You can create a custom exception class in Python by defining a new class that inherits from the built-in Exception class or any of its subclasses. For example,

class CustomException(Exception):

pass

# Usage of custom exception

try:

if condition:

raise CustomException("This is a custom exception")

except CustomException as e:

print("Custom exception occurred:", str(e))

4. Some common exceptions that are built-in to Python include:

ValueError: Raised when an operation receives an argument of the correct type but an inappropriate value.

TypeError: Raised when an operation or function is applied to an object of an inappropriate type.

FileNotFoundError: Raised when trying to access a file that does not exist.

IndexError: Raised when trying to access an index that is out of range in a sequence.

KeyError: Raised when trying to access a key that does not exist in a dictionary.

ZeroDivisionError: Raised when dividing by zero.

5. Logging in Python is a module that provides a flexible and customizable way to record events or messages during the execution of a program. It allows you to track the flow of your code, record errors or exceptions, and provide insights into the behavior of your application. Logging is important in software development because it helps with debugging, troubleshooting, and monitoring the performance of applications in real-world scenarios.

6. Log levels in Python logging define the severity or importance of log messages. They provide a way to categorize and filter log messages based on their criticality. Examples of log levels and their appropriate use cases:

DEBUG: Detailed information, typically useful for debugging purposes.

INFO: Informational messages that confirm the normal operation of an application.

WARNING: Indication of potential issues or unexpected situations that do not prevent the program from running.

ERROR: Signifies an error that occurred during the execution but did not cause the program to stop.

CRITICAL: Indicates a critical error or failure that may lead to the termination of the program.

7. Formatter objects have the following attributes and methods. They are responsible for converting a LogRecord to (usually) a string which can be interpreted by either a human or an external system. The base Formatter allows a formatting string to be specified. If none is supplied, the default value of '%(message)s' is used, which just includes the message in the logging call. Log formatters use formatting codes or placeholders to represent specific information such as timestamps, log levels, module names, and message content. By customizing the log message format using formatters, you can ensure consistency, readability, and inclusion of relevant details in your log messages.

8. To capture log messages from multiple modules or classes in a Python application, you can configure a logger object to handle logging for the entire application. This involves setting up a logging configuration that specifies the desired log level, log output destination (such as a file or console), and log format. By configuring a logger at the root level, all log messages from different modules or classes in the application can be captured and managed centrally.

9. The difference between the logging and print statements in Python is that logging is a dedicated module for producing log messages, while print statements are used for general-purpose output during development or debugging. Logging offers more flexibility, control, and functionality for recording events and tracking the behavior of an application. It allows you to define log levels, redirect output to different destinations, customize log formats, and easily enable or disable logging in production environments. Print statements, on the other hand, are simpler and mainly used for temporary output or quick debugging.

10.

import logging

logging.basicConfig(filename="app.log", level=logging.INFO)

logging.info("Hello, World!")

11.

import logging

import datetime

logging.basicConfig(filename="errors.log", level=logging.ERROR)

try:

# Code that may raise an exception

except Exception as e:

error\_message = f"{type(e).\_\_name\_\_} occurred at {datetime.datetime.now()}: {str(e)}"

logging.error(error\_message)

print(error\_message)