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

Answer -

The else block in try except statement is executed when the exception is not raised in the try block. In the following example we see that variable *result* stores division of variable a and b. If an exception occurs in this statement then control goes to the except block. If no exception is there then the control goes over to else block.

```
def divide_numbers(a, b):
    try:
        result = a / b
    except Exception as e:
        print(e)
    else:
        print("Division result:", result)
```

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

Answer -

Yes, we can have nested try-except blocks. Whenever an exception occurs the control goes to the except block and the lines of code after the error causing statement are not executed. The nested try except blocks comes handy when we don't want to exit the outer try block.

```
try:

    num1 = int(input("Enter the first number: "))
    num2 = int(input("Enter the second number: "))


      try:

          # Inner try block

          result = num1 / num2

         print("Result:", result)

       except ZeroDivisionError:

            print("Error: Division by zero occurred.")

      except ValueError:
```

```
            print("Error: Invalid input.")
        except Exception as e:
            print("An error occurred:", str(e))

except ValueError:
    print("Invalid input. Please enter valid integers.")
except Exception as e:
    print("An error occurred:", str(e))
```

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

Answer -

We need to define a new class in order to create a custom exception. The class should inherit from the *Exception* class, which is a built in class.

```
class CustomError(Exception):
pass

def divide(a, b):
    try:
        if b == 0:
        raise CustomError("Division by zero is not allowed")
        result = a / b
        return result
    except CustomError as e:
        print("Custom Error:", e)
print(divide(10, 2))
```
Output: 5.0

```
print(divide(10, 0))
```
Output: Custom Error: Division by zero is not allowed

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

Answer -

Some of the common exceptions in python are as under.

SyntaxError: This exception is raised when the grammar of the python language is violated.

TypeError: This exception is raised when we try to perform an action on a variable which is not intended to be used on the particular data type.

NameError: This is raised when the variable/ function is not found.

ValueError: This is raised when the data type of the operand variable is correct but the value inside is not.

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

Answer -

Logging is used to record the events during the program execution. This helps us to capture valuable information while the program is getting executed. There are various advantages of logging, listed as under

Debugging - The logging helps in debugging issues in the code. The logs provide detailed information about the execution flow. It allows us to trace and analyze the sequence of events causing errors.

Error Tracking - Logging helps to trace back the errors that are happening in the code.

Auditing - Logging helps us to record events for regulatory and security purposes.

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

Answer -

Python provides different levels of logging. These levels indicate the severity of the event logged. There are following logging levels available in python

Debug - The debug level in logging provides detailed information and is used when debugging the code.

Info - This log level is used to get basic information about program execution.

Warning - This indicates potential issues in the code that may lead to an error.

Error - This indicates the error encountered in the code execution.

Critical - This indicates severe errors that lead to termination of the program. The issue requires immediate attention.

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

Answer -

The log formatters in python are used to define the format of the log. It helps us to decide how we want to display our log in the log file. It determines the log output, layout,content and styling of the message.

Formatter class allows us to customize the log message format. Below we can see we are defining a formatter where we specify the content and format of the log message.

```
import logging
# Create a logger
logger = logging.getLogger('my_logger')
logger.setLevel(logging.DEBUG)
# Create a formatter
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
```

In this example we set the format to include timestamp (asctime), severity level (levename) and actual log message (message)

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

Answer -

To setup logging from multiple modules or classes we can follow following

- Create a logger object in each module or class and use the same logger name across all modules to ensure they are part of the same logging system.
- Set the desired log level for each logger, to control the log message for each module independently
- Configure handler for each logger, which determines where the log message should be sent (console, network or a file)
- Configure formatter for each logger to customize the format of the log message.

Q 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?

Answer -

Following are the differences between logging and print statements

- Logging is used to generate log messages in order to record the events. On the other hand the print statement is used to display immediate values during the program execution.
- In case of logging, the logs messages can be stored in a file, console etc. The print statements display the message on the console.
- In case of logging we can format the log message according to our desired format and also different levels of logging can be set, which means log messages can be generated according to the level of seriousness of the event. The print statement on the other hand just displays the message on the console in the default way.

Logging is preferred in place of print statements for following reasons

- Logging allows us to control the level of the log. We can decide to raise a log message or avoid logging a message for an event based on the priority of the event.
- Logging provides mechanism to manage logs from different modules of application
- Logging helps in storing messages in log files, which makes it easy to collate and read the logs
- Additional information can be logged such as timestamp
- Logging helps in better analysis of the events.

Q 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.

Answer -

```
import logging
logging.basicConfig(filename="app.log",level=logging.INFO,format="%(asctime)s - %(levelname)s - %(message)s",filemode="a")
logging.info("Hello, World!")
```

Q 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.

Answer -

```
import logging
import datetime

logging.basicConfig(
        level=logging.ERROR,
        format="%(asctime)s - %(levelname)s - %(message)s",
        handlers=[
                logging.FileHandler("errors.log"),
                logging.StreamHandler()

         ]

)

try:
        raise ValueError("Something went wrong!")
except Exception as e:
        error_message = f" Exception: {type(e).__name__} - {str(e)}"
        logging.error(error_message)
```