1. **Role of 'else' block in a try-except statement**:
   The else block in a try-except statement is executed if no exceptions are raised in the try block. It's useful when you want to perform some actions only if the try block completes successfully (i.e., no exceptions occurred).

Example:

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Error: Division by zero.")
else:
    print("Success: No errors occurred.")
    print(f"Result: {result}")
```

In this case, if no exception is raised, the else block will execute, printing the result of the division.

---

2. **Nested try-except block**:
   Yes, a try-except block can be nested inside another try-except block. This is useful when you need to handle different types of exceptions in different parts of your code.

Example:

```
try:
    try:
        result = 10 / 0
    except ZeroDivisionError:
        print("Caught ZeroDivisionError.")
    num = int("abc")  # Will raise ValueError
except ValueError:
    print("Caught ValueError.")
```

In this case, the inner try-except handles the ZeroDivisionError, while the outer try-except handles the ValueError.

---

3. **Creating a custom exception class in Python**:
   You can create a custom exception by subclassing the built-in Exception class.

Example:

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message

try:
    raise CustomError("This is a custom error!")
except CustomError as e:
    print(f"Caught an exception: {e.message}")
```

This code raises and catches a custom exception with a custom message.

---

4. **Common built-in Python exceptions**:
   - IndexError: Raised when trying to access an index that is out of range.
   - KeyError: Raised when a dictionary key is not found.
   - ValueError: Raised when a function receives an argument of the correct type but inappropriate value.
   - TypeError: Raised when an operation is performed on an inappropriate data type.
   - ZeroDivisionError: Raised when division by zero occurs.
   - FileNotFoundError: Raised when a file operation fails due to the file not being found.

---

5. **What is logging and its importance**:
   Logging in Python is the process of recording events that happen during a program's execution. It's crucial for tracking issues, debugging, and understanding how the software is performing in different environments. Logging helps developers to monitor the application’s behavior and catch bugs that may not trigger exceptions

but can impact performance or correctness.

---

6. **Purpose of log levels**:
   Log levels in Python logging allow developers to categorize and filter log messages based on their severity.

Common log levels:

- DEBUG: Detailed information, useful for diagnosing problems. Example: Debugging database connections.
- INFO: Confirmation that things are working as expected. Example: Successful initialization of an application.
- WARNING: An indication that something unexpected happened, but the program is still running. Example: Deprecated features used.
- ERROR: More serious issues that prevent parts of the program from functioning. Example: A failed database connection.
- CRITICAL: Serious errors indicating the program may not continue to run. Example: Disk space running out.

---

7. **Log formatters in Python logging**:
   Log formatters define the structure of log messages. You can customize the format to include details like timestamp, log level, and the log message.

Example of customizing log format:

```
import logging

logging.basicConfig(
   format='%(asctime)s - %(levelname)s - %(message)s',
   level=logging.DEBUG
)

logging.info("This is an info message")
```

The log message would include a timestamp, the severity level (INFO), and the actual message.

---

8. **Logging from multiple modules**:
   You can configure logging in a main module and ensure that other modules use the same configuration. You typically define the logging configuration in a shared file or the main script.

Example:

```
# In the main module (main.py)
import logging
import module_a

logging.basicConfig(filename='app.log', level=logging.INFO)
logging.info("Main module started")

module_a.log_message()

# In another module (module_a.py)
import logging

def log_message():
   logging.info("Message from module_a")
```

This will capture log messages from both modules in app.log.

---

9. **Difference between logging and print**:
   - print: Displays output to the console. It's simple and temporary, typically used for debugging during development.
   - logging: Provides a more structured way to report events. It supports different log levels, message formatting, and can direct output to files, consoles, or other destinations.

When to use logging: In production, where you need to capture detailed information about

your application's behavior, especially when debugging or analyzing issues.

---

10. **Program to log a message to a file (app.log)**:

```
import logging
logging.basicConfig(filename='app.log', level=logging.INFO, filemode='a')  # append mode
logging.info("Hello, World!")
```

This program logs the message "Hello, World!" to app.log in INFO level.

---

11. **Program to log errors to the console and file**:

```
import logging
from datetime import datetime
# Set up logging to console and file
logging.basicConfig(filename='errors.log', level=logging.ERROR, format='%(asctime)s - %(levelname)s - %(message)s')
try:
    x = 10 / 0  # Will raise ZeroDivisionError
except Exception as e:
    logging.error(f"Exception occurred: {type(e).__name__} - {e}")
    print(f"Error logged: {type(e).__name__}")
```