In [ ]:
```
1. Role of the else block in a try-except statement
```

In [ ]:
```
The else block in a try-except statement runs only if no exception occurs in the try block. It's useful for code that should
execute only when the try block succeeds without errors.
```

In [ ]:
```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter a valid number.")
else:
    print("Division successful. Result:", result)
```

In [ ]:
```
2. Nested try-except Blocks
```

In [ ]:
```
Yes, a try-except block can be nested within another try-except. It helps in managing exceptions at multiple levels.
```

In [ ]:
```
try:
    print("Outer try block")
    try:
        num = int(input("Enter a number: "))
        print(10 / num)
    except ZeroDivisionError:
        print("Inner except: Cannot divide by zero.")
except ValueError:
    print("Outer except: Invalid input.")
```

In [ ]:
```
3. Custom Exception Class
```

In [ ]:
```
Custom exceptions can be created by subclassing the Exception class.
```

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

try:
    raise CustomError("This is a custom exception!")
except CustomError as e:
    print("Caught custom exception:", e.message)
```

In [ ]:
```
4. Common Built-in Exceptions
```

In [ ]:
```
Some common exceptions:

ZeroDivisionError: Division by zero.
ValueError: Invalid argument type or value.
IndexError: Out-of-range index access.
KeyError: Missing key in a dictionary.
TypeError: Incompatible operation for data types.
FileNotFoundError: File or directory not found.
```

In [ ]:
```
5. Logging in Python
```

In [ ]:
```
Logging records events during program execution. It's important for:

Debugging.
Monitoring application behavior.
Providing traceability for errors.
```

In [ ]:
```
6. Log Levels in Python Logging
```

In [ ]:
```
Python provides various log levels:

DEBUG: Detailed information for diagnosing issues.
INFO: General information about application progress.
WARNING: Indicates a potential problem.
ERROR: Logs an error.
CRITICAL: Logs a serious issue.
```

In [ ]:
```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug("Debug message")
logging.info("Info message")
logging.warning("Warning message")
logging.error("Error message")
logging.critical("Critical message")
```

In [ ]:
```
7. Log Formatters
```

In [ ]:
```
Log formatters customize how log messages are displayed
```

In [ ]:
```
import logging

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

logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.INFO)

logger.info("Custom formatted log message.")
```

In [ ]:
```
8. Logging Across Multiple Modules
```

In [ ]:
```
A centralized logging setup can handle logs from multiple modules.
```

In [ ]:
```
# main.py
import logging

logging.basicConfig(filename='app.log', level=logging.INFO, filemode='a',
                    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger('main_logger')

# module1.py
import logging

logger = logging.getLogger('main_logger')
logger.info("Message from module1")
```

In [ ]:
```
9. Logging vs Print
```

In [ ]:
```
logging: Designed for robust debugging and saving logs, with levels and formatting.
print: Simple, temporary output to console
```

In [ ]:
```
10. Log "Hello, World!" to a File
```

In [ ]:
```
Requirements:

Log message: "Hello, World!"
Log level: INFO
File: app.log (appends messages).
```

In [ ]:
```
import logging

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

In [ ]:
```
11. Log Errors to Console and File
```

In [ ]:
```
Requirements:

Log error to console and errors.log.
Include exception type and timestamp.
```

In [ ]:
```
import logging
from datetime import datetime

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

try:
    result = 10 / 0
```

```
except Exception as e:
    logging.error(f"An error occurred: {type(e).__name__} - {e}")
```

In [ ]: