In [ ]:

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

A) The else block in a try-except statement is optional and provides
a way to specify code that should be executed only if no exceptions occur
within the try block. It allows you to separate the code that may raise
an exception from the code that should be executed if the exception is not
raised.

The syntax of a try-except-else statement is as follows:

```
    try:
    # Code that may raise an exception
    ...
except ExceptionType:
    # Code to handle the exception
    ...
else:
    # Code to execute if no exception occurs
    ...
```

```
    Here's an example scenario where the else block would be useful:
    def divide_numbers(num1, num2):
    try:
        result = num1 / num2
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
    else:
        print("Division result:", result)
```

In this example, the divide_numbers function attempts to perform
division between two numbers. If a ZeroDivisionError occurs, it is caught
and an error message is printed. However, if no exception occurs during
the division, the else block is executed, which prints the division result.

Using the else block in this scenario allows for a clear separation between
the exception handling code and the code that should be executed when the
division is successful. It enhances code readability and ensures that the
division result is only printed when no exceptions occur, avoiding confusion
in the output.

By using the else block, you can define a specific code path to be
executed when the code in the try block completes successfully,
providing a way to handle the case when no exceptions are raised.

In [ ]:

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

A)Yes, a try-except block can be nested inside another try-except block.
This means that you can have an inner try-except block within an outer
try block or within an outer except block. This nesting allows for
handling exceptions at different levels of code execution.

Here's an example that demonstrates nested try-except blocks:

```
try:
    # Outer try block
    numerator = int(input("Enter the numerator: "))
    denominator = int(input("Enter the denominator: "))
    try:
        # Inner try block
        result = numerator / denominator
```

```
        print("Result:", result)
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter integers only.")
In this example, there are two levels of exception handling:

Outer try-except block:

The outer try block attempts to read the numerator and denominator from
the user.
If a ValueError occurs during the conversion of inputs to integers,
the outer except block is executed, which prints an error message for
invalid input.

Inner try-except block:

Inside the outer try block, there is an inner try-except block.

The inner try block attempts to perform the division operation between
the numerator and denominator.

If a ZeroDivisionError occurs during division, the inner except block
is executed, which prints an error message for division by zero.

The nested try-except blocks allow for handling exceptions at
different levels. The outer block handles the ValueError related
to incorrect input values, while the inner block handles the
ZeroDivisionError related to division by zero.

Nesting try-except blocks provides flexibility in handling exceptions
at different levels of the code execution hierarchy. It allows for
more granular control over exception handling and the ability to
handle exceptions in a hierarchical manner based on the specific
context and requirements of the program.
```

In [ ]:

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

A)In Python, We can create a custom exception class by defining a new
class that inherits from the built-in Exception class or any of its
subclasses. This allows to define your own exception types that
can be raised and caught like any other exception.

Here's an example of creating a custom exception class:

class MyCustomException(Exception):
    pass
In this example, we define a custom exception class called
MyCustomException that inherits from the base Exception class.
The pass statement indicates that the class does not have any
additional methods or attributes.

Now, let's see an example that demonstrates the usage of the
custom exception class:

class MyCustomException(Exception):
    pass

def validate_input(value):
    if not isinstance(value, int):
        raise MyCustomException("Invalid input type. Expected an integer.")
```

```
try:
    user_input = input("Enter a number: ")
    validate_input(user_input)
except MyCustomException as err:
    print("Error:", err)
```

In this example, we have a function called validate_input() that takes a value as input and raises a MyCustomException if the value is not an integer. The MyCustomException is raised using the raise statement with an instance of the exception class and an error message.

Inside the try-except block, we call the validate_input() function with user input. If the input is not an integer, the custom exception is raised, and the corresponding except block is executed, printing the error message.

By creating a custom exception class, We can define your own exception types that carry specific information or behaviors. This allows to handle exceptional situations in a more specialized and meaningful way, providing better error messages and improving the overall readability and maintainability of your code.

In [ ]:

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

A)
Python provides several built-in exceptions that cover a wide range of common error conditions. Some of the most commonly used built-in exceptions in Python include:

SyntaxError: Raised when there is a syntax error in the code.

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

NameError: Raised when a local or global name is not found.

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

IndexError: Raised when a sequence subscript is out of range.

KeyError: Raised when a dictionary key is not found.

FileNotFoundError: Raised when an attempt is made to open a file that doesn't exist.

ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.

IOError: Raised when an input/output operation fails.

AttributeError: Raised when an attribute reference or assignment fails.

OverflowError: Raised when the result of an arithmetic operation is too large to be expressed within the range of the data type.

MemoryError: Raised when the program runs out of memory resources.

ImportError: Raised when an import statement fails to find and load a module.

StopIteration: Raised to signal the end of an iterator.

```
KeyboardInterrupt: Raised when the user interrupts the execution
    of the program (e.g., by pressing Ctrl+C).
```

In [ ]:

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

A)Logging in Python refers to the process of recording messages or events
that occur during the execution of a program. It involves using the
built-in logging module to capture and store information about the program's
execution, including informational messages, warnings, errors, and
debugging information.


Logging is important in software development for several reasons:


Debugging and Troubleshooting: Logging provides a mechanism to capture
    valuable information about the program's behavior during runtime.
    It allows developers to trace the flow of execution, identify errors,
    and diagnose issues by examining the logged messages. This helps in
    debugging and troubleshooting problems in the code.

Error Tracking and Monitoring: By logging error messages and exceptions,
    developers can track and monitor the occurrence of errors in the
    application. These logs can be used for error analysis, identifying
    recurring issues, and taking corrective actions to improve the
    software's stability and reliability.

Performance Analysis: Logging can be used to measure the performance
    of an application by recording timestamps, execution times, and
    other relevant metrics. This information helps in identifying
    performance bottlenecks and optimizing critical sections of the code.

Auditing and Compliance: Logging plays a crucial role in auditing and
    compliance requirements. By recording important events, user actions,
    and system activities, logging helps in maintaining an audit trail
    for security and compliance purposes.

Maintenance and Support: Logs are invaluable for maintaining and
    supporting software applications in a production environment.
    They provide a historical record of the program's execution,
    allowing support teams to analyze issues reported by users and
    provide effective resolutions.

Application Insights: Logging can provide insights into the usage
    patterns, user behavior, and application performance. By analyzing
    the logged data, developers can gain valuable insights into the
    usage patterns of their software, identify user needs, and make
    informed decisions for future improvements.
```

In [ ]:

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

A) Log levels in Python logging provide a way to categorize and
prioritize log messages based on their severity or importance.
The logging module in Python defines several standard log levels,
each serving a specific purpose. These log levels allow developers to
control which messages should be recorded based on the desired level
of detail and the importance of the logged information.

The standard log levels in Python logging, in increasing order of
severity, are:
```

```
DEBUG: This log level is used for detailed debugging information.
    It is typically used during development or troubleshooting to
    provide granular information about the program's execution,
    variable values, and intermediate steps. Debug messages are
    usually disabled in production environments to avoid cluttering
    the logs.

Example usage:

    import logging

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

logger.debug("Debug message: Variable x = %s", x)
INFO: The INFO log level is used to record informational messages
    that indicate the normal execution flow of the program. It
    provides general information about important events or milestones
    in the application's operation. These messages are helpful for
    monitoring the program's behavior and providing a high-level overview.

Example usage:

    import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

logger.info("Application started.")
WARNING: This log level is used to indicate potential issues or
    unusual conditions that are not critical but may require attention.
    Warnings are typically used to highlight situations that might lead
    to errors or unexpected behavior in the future. They serve as a way
    to capture potential problems and provide early warnings.

Example usage:

    import logging

logging.basicConfig(level=logging.WARNING)
logger = logging.getLogger(__name__)

logger.warning("Resource usage is approaching the limit.")
ERROR: The ERROR log level is used to record errors that occurred
    during the program's execution but did not prevent it from continuing.
    It indicates important issues that should be investigated, as they
    might affect the proper functioning of the application or lead to
    incorrect results.

Example usage:

    import logging

logging.basicConfig(level=logging.ERROR)
logger = logging.getLogger(__name__)

try:
    # Some code that might raise an exception
except Exception as e:
    logger.error("An error occurred: %s", str(e))
CRITICAL: The CRITICAL log level represents the most severe level of
    logging. It is used to indicate critical errors or failures that
    cause the program to terminate or cannot be recovered from.
```

```
    Critical messages typically require immediate attention and
    often trigger alerts or notifications to the development or
    operations team.

Example usage:


    import logging

logging.basicConfig(level=logging.CRITICAL)
logger = logging.getLogger(__name__)

logger.critical("System is down. Application terminated.")
```

In [ ]:

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

A) Log formatters in Python logging are objects that define the structure
and content of log messages. They allow developers to customize the format
of log records, including the timestamp, log level, log message, and other
relevant information.

The logging module in Python provides the Formatter class, which is used
to configure log formatters. This class allows to define the desired format
pattern using placeholders and special formatting codes. When a log message
is emitted, the formatter processes the log record and generates
a formatted string that represents the log message.

Here's an example of how to customize the log message format using
formatters:


import logging

# Create a formatter object
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

# Create a handler and set the formatter
handler = logging.StreamHandler()
handler.setFormatter(formatter)

# Create a logger and add the handler
logger = logging.getLogger(__name__)
logger.addHandler(handler)

# Log messages
logger.info("This is an informational message.")
logger.error("An error occurred.")
In this example, we create a Formatter object and specify the desired
format pattern using placeholders:

%(asctime)s: Placeholder for the timestamp in a human-readable format.
%(levelname)s: Placeholder for the log level (e.g., INFO, ERROR, etc.).
%(message)s: Placeholder for the log message itself.

    We then create a StreamHandler, which directs the log messages to the
console. We set the formatter for the handler using setFormatter() to
apply our custom format.

Next, we create a logger and add the handler to it. Finally, we
log some messages using the logger, and the formatter applies the
specified format pattern to generate the log messages.
```

The output will be something like:

```
2023-06-26 10:30:00,123 - INFO - This is an informational message.
2023-06-26 10:30:01,234 - ERROR - An error occurred.
By customizing the log message format using formatters, We can
control the information displayed in log records and tailor it to the
specific requirements. This allows for consistency in log formatting
across different log handlers, better readability of log messages, and
integration with log analysis tools.
```

In [ ]:

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

A) To capture log messages from multiple modules or classes in a
Python application, By following these steps:

Create a Logger: Create a logger instance using the logging.getLogger()
    method. This logger will be used to capture and handle log messages.


import logging

logger = logging.getLogger('my_logger')
The logger name can be any string that identifies the logger. It is
common to use the module or class name as the logger name to associate
the logs with specific components.

Configure Handlers: Configure one or more handlers to specify where
    the log messages should be outputted. Handlers define the destination
    for log messages, such as a file, console, or a remote server. We can
    add multiple handlers to capture log messages in different locations
    simultaneously.


import logging

logger = logging.getLogger('my_logger')

file_handler = logging.FileHandler('app.log')
console_handler = logging.StreamHandler()

logger.addHandler(file_handler)
logger.addHandler(console_handler)
In this example, we configure a FileHandler to write log messages to
a file named 'app.log' and a StreamHandler to print log messages to
the console. We can configure additional handlers as needed.

Set the Logging Level: Set the desired logging level for the logger.
    This determines the minimum severity level of log messages that
    will be captured. We can set different levels for different handlers
    if needed.


import logging

logger = logging.getLogger('my_logger')

file_handler = logging.FileHandler('app.log')
console_handler = logging.StreamHandler()

logger.addHandler(file_handler)
```

```
logger.addHandler(console_handler)

logger.setLevel(logging.DEBUG)  # Set the desired logging level
In this example, we set the logging level to DEBUG, which means all
log messages, regardless of their severity, will be captured. We can
adjust the level based on the specific needs.

Log Messages: In the  modules or classes, use the logger instance to
    log messages at appropriate points in the code.


import logging

logger = logging.getLogger('my_logger')

def my_function():
    logger.debug('Debug message')
    logger.info('Info message')
    logger.warning('Warning message')
    logger.error('Error message')
In this example, we log messages at different levels
(DEBUG, INFO, WARNING, ERROR) within a function. These
messages will be captured by the configured handlers.

By following these steps, we can set up logging to capture log messages
from multiple modules or classes in the Python application.
Each module or class can use the same logger instance, allowing us  to
centralize and manage the logging configuration effectively. The log
messages will be directed to the configured handlers, and can customize
the formatting, destinations, and levels based on the requirements.
```

In [ ]:

```
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?
A) The logging module and print statements in Python serve different
purposes and have different use cases in real-world applications.
Here are the main differences and when to use each:

Output Destination: The primary difference between logging and print
    statements is the destination of the output. Print statements
    send the output directly to the standard output (console), while
    logging allows  to configure multiple output destinations
    such as files, streams, or external logging services.

Flexibility and Configuration: The logging module provides extensive
    configuration options to control the behavior of log messages,
    including log levels, log formatting, log handlers, and log filtering.
    This flexibility allows to customize the logging behavior according
    to the specific needs for the application. On the other hand,
    print statements have limited customization options and lack the
    flexibility provided by logging.

Log Levels and Filtering: Logging allows to set different log levels
    for different types of messages (e.g., debug, info, warning, error,etc.)
    This makes it easier to filter and control the verbosity of
    log messages based on their importance and severity. Print statements,
    on the other hand, don't have built-in levels, so all statements
    will be printed unless manually removed from the code.

Production Readiness: Logging is more suitable for production-ready
    applications. It provides a structured approach to capturing and
    managing log messages, making it easier to monitor, analyze, and
    troubleshoot issues in real-time or later. Logging supports long-term
```

logging strategies, such as rotating log files and integration with log aggregation tools. Print statements are more suitable for quick debugging or ad hoc output during development but are not designed for long-term logging or maintenance.

Given these differences, it is recommended to use logging over print statements in a real-world application for the following reasons:

Debugging and Troubleshooting: Logging provides a more systematic and controlled approach to capture information during debugging and troubleshooting. It allows you to selectively enable or disable log messages based on their importance, making it easier to focus on specific areas of interest and avoid excessive output.

Centralized and Configurable Logging: Logging allows you to configure logging behavior centrally, making it easier to manage and control the logging settings across the entire application. We can control log levels, formatting, and output destinations without modifying the code. This is particularly useful in complex applications with multiple modules or classes.

Integration and Scalability: Logging is designed to be integrated into larger systems and can scale well with growing applications. It provides the flexibility to log messages to different output destinations, making it easier to integrate with existing monitoring tools, log analysis platforms, or external logging services.

Information Security: In some cases, printing sensitive information to the standard output (console) may pose a security risk. Logging allows to define custom handlers that write log messages to secure locations or encrypt sensitive data, providing better control over the security of the application's logs.

Overall, logging is more robust, configurable, and suitable for production-ready applications, offering better control, scalability, and maintainability compared to print statements, which are more suited for quick debugging or ad hoc output during development.

In [ ]:

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.

A) To fulfill the requirements and log a message to a file named "app.log" with the specified log message, log level, and appending behavior, we can use the logging module in Python.
Here's an example program that accomplishes this:

```
import logging

# Configure logging to write to the file
logging.basicConfig(filename='app.log', level=logging.INFO, filemode='a')

# Log the message
logging.info('Hello, World!')
```

In this program:

We import the logging module.

We configure the logging behavior using basicConfig().

We specify the log file name as "app.log" using the filename parameter. The level parameter is set to logging.INFO to ensure that only log messages with the level "INFO" and above will be recorded. The filemode parameter is set to 'a' to append new log entries without overwriting the existing ones.

We log the message using logging.info(), passing the log message "Hello, World!" as the argument.

When we run this program, it will create a file named "app.log" if it doesn't exist and append the log message "Hello, World!" to the file. Subsequent executions of the program will continue appending new log entries to the same file without overwriting the previous ones.

In [ ]:

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.

A)To log an error message to the console and a file named "errors.log" when an exception occurs during the program's execution, we can utilize the logging module in Python. Here's an example program that fulfills these requirements:

```
import logging
import datetime

# Configure logging to write to the console and file
logging.basicConfig(level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(message)s',
                    handlers=[logging.StreamHandler(),
                              logging.FileHandler('errors.log')])

try:
    # Your program code here
    # ...

    # Simulating an exception
    raise ValueError("Something went wrong!")

except Exception as e:
    # Log the error message with exception type and timestamp
    logging.error(f"Exception: {type(e).__name__} - {e} - Timestamp:
                  {datetime.datetime.now()}")

                  In this program:

We import the necessary modules: logging for logging functionality
                  and datetime for timestamp generation.

We configure the logging behavior using basicConfig(). The level parameter
                  is set to logging.ERROR to capture only log messages
                  with the level "ERROR" and above. The format parameter
                  specifies the format of the log message, including the
                  timestamp, log level, and message itself. We provide
                  two handlers: logging.StreamHandler() to write log
                  messages to the console and logging.FileHandler
                  ('errors.log') to write log messages to the "errors.log"
                  file.

Inside the try block, you can place your program code that may
                  raise an exception. In this example, we simulate
```

```
                    an exception by raising a ValueError.

The except block captures any exception that occurs and logs an error
                    message using logging.error(). The error message includes
                    the exception type (type(e).__name__), the exception
                    message (e), and the timestamp generated by datetime.
                    datetime.now().

When an exception occurs during the program's execution, the error message
                    will be logged to the console and appended to the
                    "errors.log" file. The log message will include
                    the exception type, the exception message, and the
                    timestamp of when the error occurred.
```