Assignment by- Pranjal Sharma

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

Ans- The `else` block in a try-except statement is used to execute code only if no exception occurs in the `try` block. It is typically used to perform cleanup or post-processing operations.

```
def open_file(filename):
try:
file = open(filename, "r")
except FileNotFoundError:
print("File not found:", filename)
return None
else: # File was opened successfully, so read and process the contents
contents = file.read()
file.close() return contents # Example usage: contents =
#open_file("my_file.txt") if contents is not None: print(contents)
```

Other useful scenarios for the `else` block in a try-except statement include:

- Performing cleanup operations: For example, if you are opening a file or database connection, you can use the `else` block to close the file or connection even if no exception occurs.
- Performing post-processing operations: For example, if you are downloading a file, you can use the `else` block to verify the integrity of the file after it has been downloaded.
- Handling unexpected exceptions: If you are not sure what type of exception may be raised in the `try` block, you can use the `else` block to catch any unexpected exceptions and handle them gracefully.

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

Yes, we can have the nested try-except block.We can apply the try block which contains the another try -except blocks as given in below example

```
def read_file(filename):
  try:
    # Try to open the file.
    file = open(filename, "r")
  except FileNotFoundError:
```

```
    # File not found, so handle the exception.
    print("File not found:", filename)
  else:
    # File was opened successfully, so read and process the contents.
    try:
      contents = file.read()
    except ValueError:
      # Invalid file contents, so handle the exception.
      print("Invalid file contents:", filename)
    else:
      # File was read successfully, so return the contents.
      return contents

# Example usage:
contents = read_file("my_file.txt")
if contents is not None:
  print(contents)

```

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

Ans- To create a custom exception class in Python, you need to inherit from the built-in `Exception` class. You can then override the constructor and other methods of the `Exception` class to customize the behavior of your custom exception.

```
class MyCustomException(Exception):
  def __init__(self, message):
    super().__init__(message)

  def __str__(self):
    return f"MyCustomException: {self.message}"

```

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

Ans- Here are some common exceptions that are built-in to Python:

- AssertionError: Raised when an `assert` statement fails.
- AttributeError: Raised when an attribute reference or assignment fails.
- EOFError: Raised when a built-in function like `input()` hits an end-of-file condition (EOF) without reading any data.
- FloatingPointError: Raised when a floating point operation fails.
- GeneratorExit: Raised when a generator is closed (with the `close()` method).

- ImportError: Raised when an imported module does not exist.
- IndexError: Raised when an index is out of range.
- KeyError: Raised when a key is not found in a dictionary or other mapping object.
- LookupError: Raised when a lookup operation fails.
- MemoryError: Raised when there is not enough memory to allocate a new object.
- NameError: Raised when a name is not found in the current scope.
- OverflowError: Raised when an arithmetic operation results in a value that is too large to be represented.
- SyntaxError: Raised when there is a syntax error in the Python code.
- TypeError: Raised when an operation is attempted on a value of the wrong type.
- ValueError: Raised when an operation is attempted with an invalid value.
- ZeroDivisionError: Raised when an attempt is made to divide by zero.

5. What is logging in Python, and why is it important in software development?
Ans- Logging in Python is the process of recording events that occur while a Python program is running. This can be useful for a variety of purposes, such as:

- Debugging: Logs can be used to track down the source of errors in a program.
- Monitoring: Logs can be used to monitor the performance and health of a program.
- Auditing: Logs can be used to audit the activity of a program, for security or compliance purposes.

Python has a built-in logging module that makes it easy to add logging to your programs. The logging module provides a variety of features, such as:

- Different logging levels: You can specify the severity of each log message, so that you can filter and prioritize your logs.
- Different handlers: You can specify where to send your log messages, such as to a file, the console, or a remote server.
- Formatters: You can specify how your log messages should be formatted.

6. Explain the purpose of log levels in Python logging and provide examples of when each log level would be appropriate.
Ans- Log levels in Python logging are used to indicate the severity of a log message. This allows you to filter and prioritize your logs, so that you can focus on the most important information.

There are six log levels in Python:

- DEBUG: This is the lowest log level and is used for very detailed information that is only useful for debugging.
- INFO: This is the default log level and is used for general information about the program's execution.
- WARNING: This is used for potential problems that may not stop the program from running.
- ERROR: This is used for errors that prevent the program from continuing.
- CRITICAL: This is used for serious errors that require immediate attention.
- NOTSET: This is the highest log level and means that all log messages will be logged.

Here are some examples of when each log level would be appropriate:

- DEBUG:
    - When you are developing a new feature and want to track down a bug.
    - To log performance data, such as the time it takes to load a page or process a request.
- INFO:
    - To log the start and end of a program.
    - To log the successful completion of a task, such as sending an email or writing a file to disk.
- WARNING:
    - To log a potential problem that may not stop the program from running, such as a low disk space warning or a failed network connection.
- ERROR:
    - To log an error that prevents the program from continuing, such as a database connection failure or a syntax error in the code.
- CRITICAL:
    - To log a serious error that requires immediate attention, such as a security breach or a system outage.

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

Ans- Log formatters in Python logging are used to specify the format of log messages. This allows you to control how your log messages are displayed, so that you can easily find the information you need.

The default log formatter in Python is the `logging.Formatter` class. This formatter formats log messages with the following information:

- The date and time of the log message
- The logging level
- The name of the logger
- The log message

You can customize the log message format by passing a format string to the `logging.Formatter()` constructor. The format string is a Python string that uses %-formatting to specify the order and format of the log message fields.

For example, the following format string will format log messages with the following information:

```
"%(asctime)s %(levelname)s %(name)s %(message)s"
```

- The date and time of the log message, in ISO 8601 format
- The logging level, in all caps
- The name of the logger
- The log message

You can also use the `logging.Formatter()` class to format log messages in other ways, such as JSON or XML.

To use a custom log formatter, you need to assign it to a log handler. You can do this using the `setFormatter()` method.

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

Ans- To set up logging to capture log messages from multiple modules or classes in a Python application, you can use the following steps:

1. Create a logger object for each module or class. You can do this using the following code:

```
import logging

logger = logging.getLogger(__name__)
```

The `__name__` argument is the name of your module or class. This is used to identify the source of your log messages.

2. Add the logger objects to a root logger object. You can do this using the following code:

```

root_logger = logging.getLogger()

root_logger.addHandler(logger)




```

```

import logging

# Create a logger object for each module

logger_module1 = logging.getLogger("module1")

logger_module2 = logging.getLogger("module2")

# Add the logger objects to the root logger

root_logger = logging.getLogger()

root_logger.addHandler(logger_module1)

root_logger.addHandler(logger_module2)

# Configure the root logger

root_logger.setLevel(logging.INFO)
```

```
handler = logging.FileHandler("my_log.txt")

handler.setFormatter(logging.Formatter("%(asctime)s %(levelname)s %(name)s %(message)s"))

root_logger.addHandler(handler)
```

```

This code will create a new log file called `my_log.txt`. The log file will contain all of the log messages from the `module1` and `module2` modules.

You can also use log levels to control which log messages are captured. For example, you can set the log level for the `module2` logger to `logging.WARNING`. This will ensure that only warning and error messages from the `module2` module are captured.

By setting up logging in this way, you can easily capture log messages from all of the modules and classes in your Python application. This can be helpful for debugging and troubleshooting problems.

Here are some additional tips for setting up logging in a Python application:

- Use a consistent log format for all of your log messages. This will make it easier to read and understand your logs.
- Use log levels to control which log messages are captured. This can help to reduce the amount of noise in your logs and make it easier to find the information you need.
- Use log handlers to send your log messages to different destinations, such as a file, the console, or a remote server. This can help you to distribute your logs and make them more accessible to different team members.
- Regularly review your logs to identify any potential problems. This can help you to catch bugs and resolve issues early on.

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?
Ans- The main difference between logging and print statements in Python is that logging is designed to be used for recording events and errors that occur while a program is
```

running, while print statements are designed to be used for displaying information to the console.

Logging statements are typically used to generate logs that can be used to debug, monitor, and audit a program. Print statements are typically used to display information to the user or to debug a program.

Feature	Logging	Print
Purpose	To record events and errors that occur while a program is running	To display information to the console
Audience	Developers, system administrators, and other technical users	Users of the program
Output	Typically written to a file or sent to a remote server	Typically written to the console
Level of detail	Can be configured to generate different levels of detail, from debug messages to error messages	Typically only generates a single level of detail

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.
Ans- To 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.

```
import logging
```

```
# Create a logger object
logger = logging.getLogger(__name__)

# Set the log level
logger.setLevel(logging.INFO)

# Create a file handler
handler = logging.FileHandler("app.log", mode="a")

# Add the file handler to the logger
logger.addHandler(handler)

# Log a message
logger.info("Hello, World!")


```
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.
Ans-
```
import logging
import os

dir_path='D:\\iNeuron\\Untitled'
log_file='logging.txt'
full_dir=os.path.join(dir_path,log_file)
full_dir
os.makedirs(full_dir,exist_ok=True)

  # Corrected file path
logger = logging.getLogger()
logger.setLevel(level=logging.INFO)

handler = logging.FileHandler(full_dir)

handler.setFormatter('%(asctime)s:%(levelname)s:%(message)s')
logger.addHandler(handler)

def LetUsCheckSystem(sys):
    if sys != 'OK':
        logging.critical('System failure: %s', sys)
```

```
LetUsCheckSystem('You need to handle the issue now')
handler.close()


```