# Assignment No-11

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

**Solution:** The role of the 'else' block in a try-except statement is to provide a block of code that will be executed if no exceptions are raised within the 'try' block. The 'else' block is optional and can be added after all the 'except' blocks in a try-except statement.

Here is an example scenario where the 'else' block can be helpful to:

“
```
    file = open("data.txt", "r")
    numbers = [int(line) for line in file]
    file.close()
except FileNotFoundError:
    print("File not found.")
except ValueError:
    print("Unable to convert data to integers.")
except Exception:
    print("An error occurred.")
else:
    total = sum(numbers)
    print("The sum of the numbers is:", total)
```
“

In this example, the code tries to open a file named "data.txt" and read its contents line by line, converting each line to an integer. The' else' block is executed if no exceptions are raised (e.g., the file is found, and the data can be converted). The 'numbers' list is summed up in this block, and the total is printed. The 'else' block allows us to separate the code that handles exceptions from the code that should only execute when no exceptions occur.

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

**Solution:** Yes, a try-except block can be nested inside another try-except block. This is known as exception handling hierarchy or nested exception handling.

Nested exception handling allows for more specific error handling and can handle different types of exceptions at different levels.

Here is an example:

```
    # outer try-except block
    numerator = 10
    denominator = 0
    result = numerator/denominator
except ZeroDivisionError:
    print("Error: Division by zero")
    try:
        # inner try-except block
        fallback_result = numerator / 1
        print("Fallback result:", fallback_result)
```

```
    except ZeroDivisionError:
        print("Error: Fallback division by zero")
```

In the above example, we have an outer try-except block that attempts to divide `numerator` by 'denominator'. Since the `denominator` value is 0, it raises a 'ZeroDivisionError', which is caught by the outer except block.

Inside the outer except block, we have an inner try-except block. This inner block performs a fallback division by dividing the `numerator` by 1. This division will never raise a `ZeroDivisionError`. However, if there is another error inside this inner try-except block, it can be caught by another inner except block.

Nested exception handling allows for more fine-grained handling of specific exceptions at different code levels, providing better error management and flexibility.

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

**Solution:** To create a custom exception class in Python, you can define a new class that inherits from the `Exception` class. This allows you to define your own behaviors and attributes specific to the custom exception.

Here's an example that demonstrates how to create and use a custom exception class:

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

    def __str__(self):
        return self.message
“
def divide(a, b):
    if b == 0:
        raise CustomException("Cannot divide by zero")
    else:
        return a / b
“
    result = divide(10, 0)
    print(result)
except for CustomException as e:
    print(f"Error: {e}")
“
```

This code defines a custom exception class called 'CustomException' that inherits from the base `Exception` class. It has an initializer method that takes a `message` parameter and stores it as an instance variable. The '__str__' method is overridden to provide a string representation of the exception.

We also have a ‘divide’ function that checks if the denominator ‘b’ is zero. If so, it raises a ‘CustomException’ with the message "Cannot divide by zero". Otherwise, it performs the division and returns the result.

In the ‘try-except’ block, we attempt to divide 10 by 0 using the ‘divide’ function. Since division by zero triggers the custom exception, we catch it using the ‘except’ clause. We then print the error message from the custom exception object.

The output of this code will be:
“
Error: Cannot divide by zero
“

So, in this example, we successfully created and used a custom exception class to handle division by zero.

**4. What are some common exceptions that are built-in to Python?**
**Solution:** There are several standard exceptions that are built into Python. Some of them include:

1. ‘SyntaxError’: Raised when the Python code has a syntax error.
2. `TypeError`: Raised when an operation or function is applied to an object of an inappropriate type.
3. `ValueError`: Raised when a function receives a parameter of the correct type but an invalid value.
4. `NameError`: Raised when a local or global name is not found.
5. `ZeroDivisionError`: Raised when division or modulo operation is performed with zero as the divisor.
6. `IndexError`: Raised when a sequence subscript is out of range.
7. `KeyError`: Raised when a dictionary key is not found.
8. `FileNotFoundError`: Raised when a file or directory is requested but cannot be found.
9. `ImportError`: Raised when an imported module is not found or cannot be imported.
10. `AssertionError`: Raised when an assert statement fails.

These are just a few examples of the built-in exceptions in Python. There are many more specialized exceptions available for specific purposes. Developers can create custom exceptions by creating a new class derived from the `Exception` class.

**5. What is logging in Python, and why is it important in software development?**
**Solution:** Logging in Python is the process of capturing and saving information about the execution of a program. It involves generating log messages that provide insights into the behavior of the software during runtime.

Logging is important in software development for several reasons:

1. Debugging: Logs can help developers identify and fix issues in the code by providing a detailed record of the program's execution, including error messages, exceptions, and stack traces.
2. Monitoring: Logs can be used to monitor the performance and health of a software system. By analyzing logs, developers can identify patterns, spot performance bottlenecks, and improve accordingly.
3. Auditing and compliance: Logging is crucial in ensuring accountability and compliance with various regulations and standards. Logs can be used to track user actions, record security events, and provide an audit trail.
4. Troubleshooting: When users encounter issues with a software application, logs can be valuable in troubleshooting and identifying the root cause of the problem. They can provide detailed information about the state of the application at the time of the error.
5. Analytics and business intelligence: Logs can gather statistics and insights about user behavior, application usage patterns, and performance metrics. This information can be valuable in making informed business decisions and optimizing the software.

Overall, logging helps developers gain visibility into the software's behavior, diagnose and debug issues, monitor performance, and ensure the overall quality and reliability of the software system.

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

**Solution:** In Python logging, log levels are used to categorize log messages based on their importance or severity. These levels help developers understand the significance of each log message and determine the appropriate action to take.

There are several log levels available in Python logging, including:

1. DEBUG: This level provides detailed information, primarily for debugging purposes. It shows detailed information about the flow of the program, variable values, and other internal details that can help identify issues during development. Example: Printing the values of variables at critical points in the code to understand their state during execution.
2. INFO: This level is used to provide general, informative messages that confirm that things are working as expected. It is usually used to track the progress or milestones of the program. Example: Logging the start and completion of essential processes or indicating a successful connection to a database.
3. WARNING: This level indicates that something unexpected has happened or could lead to an issue. It typically highlights non-fatal issues or potential problems that developers should be aware of. Example: Logging a warning when a deprecated function is called, suggesting using an alternative method.
4. ERROR: This level indicates an error or failure, but the program can still continue. It logs critical errors that need attention but do not entirely halt the program. Example: Logging an error when an operation fails due to missing or incorrect input.
5. CRITICAL: This level indicates a severe error that has caused the program to stop or become unusable. It is used for exceptional errors that require immediate attention.

Example: Logging a critical error when a necessary configuration file is missing or a database connection cannot be established.

By categorizing log messages into different levels, developers can configure the logger to filter out less critical messages during normal operations and only display or take action on messages of a specific level or above. This helps diagnose problems, improve system performance, and maintain log readability.

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

**Solution:** In Python's logging module, log formatters control the format of log messages before they are outputted. The formatting is done using a Formatter object.

The default log message format is usually not tailored specifically to one's needs. With formatters, you can customize the format to include specific details such as timestamp, log level, logger name, and the log message itself.

To customize the log message format, you can follow these steps:

1. Import the logging module:

“

```
import logging
```

“

2. Create a Formatter object with the desired format:

“

```
format_str = "[%(levelname)s] %(asctime)s %(filename)s:%(lineno)d - %(message)s"
formatter = logging.Formatter(format_str)
```

“

Here, the format string uses placeholders surrounded by percent signs (%). Each placeholder corresponds to a specific attribute of the log record like log level (levelname), logging time (asctime), filename, and line number (lineno).

3. Create a logging handler (e.g., a StreamHandler) and attach the formatter to it:

“

```
handler = logging.StreamHandler()
handler.setFormatter(formatter)
```

“

This example uses a StreamHandler to output the log messages to the console. You can also use other handler types depending on your requirements.

4. Add the handler to the logger:

“

```
logger = logging.getLogger()
logger.addHandler(handler)
```

“

Here, we retrieve the root logger and add our handler to it. You can also create specific loggers for different sections of your code.

You can control how the log messages are displayed by customizing the format string. The format string can include any of the available attributes, and you can also add static text between them. Here's a breakdown of some commonly used attributes:

1. %(asctime)s: When the log record was created.
2. %(levelname)s: The log level of the record (e.g., DEBUG, INFO, ERROR).
3. %(filename)s: The file name where the log statement is located.
4. %(lineno)d: The line number where the log statement is located.
5. %(message)s: The log message itself.

You can customize the log message format to meet your needs with these steps. You can include and arrange various attributes in any desired order to create informative and valuable log messages for debugging and monitoring purposes.

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

**Solution:** To set up logging to capture log messages from multiple modules or classes in a Python application, you can follow these steps:

1) Import the `logging` module: Start by importing the `logging` module in your Python script.

“

```
import logging
```

“

2) Configure logging: Configure the logging system according to your requirements. This includes setting the log level, specifying the output format, and defining the log file.

“

```
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s', filename='app.log')
```

“

Here, `basicConfig()` method is used to set the basic configuration properties such as log level, log format, and log file. The `level` parameter specifies the log level, with `logging.DEBUG` being the lowest level and capturing all log messages. The `format` parameter defines the format of the log messages. You can customize it as per your needs. The `filename` parameter specifies the file to which the log messages will be written.

3) Create loggers: Create loggers for each module or class that you want to capture log messages from. Each logger should have a unique name.

“

```
logger1 = logging.getLogger('module1')
logger2 = logging.getLogger('module2')
```

“

4) Log messages: Use the loggers created in the previous step to log messages from different modules or classes. You can use different log levels to indicate the severity of each log message.

“

```
logger1.debug('This is a debug message from module 1')
logger2.error('This is an error message from module 2')
```

“

Here, `logger1.debug()` and `logger2.error()` are used to log debug and error messages. You can choose the appropriate log level depending on the severity of the log message.

5) Run the application: Run your Python application, and the log messages from different modules or classes will be captured according to the logging configuration.

The log messages will be written to the specified log file (`app.log` in this example) along with the specified log format. You can also choose to display the log messages on the console by including another `logging.StreamHandler()` in the logging configuration.

“

```
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s', handlers=[logging.FileHandler('app.log'), logging.StreamHandler()])
```

“

**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?**

**Solution:** In Python, the logging and print statements serve different purposes.

- The print statement is used to display information on the console during the execution of a program. It is mostly used for debugging and quick output of variables or messages. The print statement is useful for temporary code or simple debugging tasks. However, the print statement does not offer much control over the output, such as formatting or filtering the logged messages.

- On the other hand, the logging module in Python provides a more comprehensive and flexible way to record information during the execution of a program. It allows you to control and configure the logging output extensively. With the logging module, you can define the logging level (such as debug, info, warning, error, and critical), set up file logging, specify formatting for the log messages, and direct logged messages to different destinations.

In a real-world application, it is recommended to use the logging module over print statements for several reasons:

1) Versatility: The logging module provides various logging levels, allowing you to categorize and prioritize the logged messages. You can customize the output, specify where the logs should be written (console, files, etc.), and control the verbosity of certain log messages.
2) Maintainability: By using the logging module, you can easily enable or disable logging statements based on different configurations or conditions. This allows you to toggle logging on or off without modifying the code. In contrast, removing or commenting on print statements can be error-prone and time-consuming.

3) Record keeping: Logs generated through the logging module can be saved for future analysis. This is crucial for troubleshooting and tracking issues in real-world applications, as you can examine the sequence of events leading up to an error or an unexpected behavior. Print statements only show real-time output and are not persistent.
4) Debugging: The logging module has more advanced features, such as the ability to log unhandled exceptions, traceback of errors, and exceptions caught in try-except blocks. This makes it easier to identify and debug issues in production systems.

Remember, print statements have their own use cases, especially during development, but robust logging using the logging module is recommended for a real-world application.

**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.**

**Solution:** import logging

```
# Create a logger object
logger = logging.getLogger(__name__)
# Set the log level to INFO
logger.setLevel(logging.INFO)
# Create a file handler to log to the file "app.log"
file_handler = logging.FileHandler("app.log")
# Create a formatter for the log message
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
# Set the formatter for the file handler
file_handler.setFormatter(formatter)
# Add the file handler to the logger
logger.addHandler(file_handler)
# Log the 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.**

**Solution:** Sure! Here's 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:

```
“
import logging
import datetime

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

```
logger.setLevel(logging.ERROR)
# Create a file handler
file_handler = logging.FileHandler("errors.log")
file_handler.setLevel(logging.ERROR)
# Create a formatter
formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(message)s")
# Set the formatter for the file handler
file_handler.setFormatter(formatter)
# Add the file handler to the logger
logger.addHandler(file_handler)
try:
    # Your code here
    # ...
    # ...
    # Simulating an exception
    x = 10 / 0
except Exception as e:
    # Log the error message to the console
    logger.error(f"Exception occurred: {type(e).__name__} - {e}")

    # Log the error message to the file
    logger.exception(f"Exception occurred: {type(e).__name__} - {e}")
```
“

In this program, we import the `logging` module and create a logger for logging error messages. We also create a file handler to log error messages to the "errors.log" file. We set the log level to `ERROR` so that only error-level messages will be logged.

Inside the `try` block, you can replace the comment `# Your code here` with your actual program logic. In this example, I've added a line that will cause a `ZeroDivisionError` to occur for demonstration purposes.

If an exception occurs, an error message along with the exception type and a timestamp will be logged both to the console and the "errors.log" file.

Please note that you need to have write permission, and the "errors.log" file should be in the current working directory for the program to work correctly.