Session - 4: Exception Handling and Logging

3.1 Understanding Exception Handling

Exception handling is the process of dealing with exceptional conditions that occur during runtime so that the program/software does not crash and continues to run. Examples of exception handling are insufficient memory for running the program, database connectivity problems that can occur because of reasons such as power failure or unreachable server, invalid input entered by the user, etc.

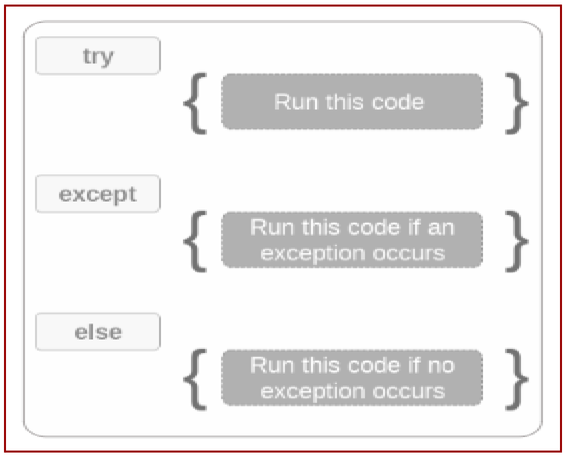
Now python is a runtime language (interpreter-based and uses no compiler); therefore, exception handling becomes a must. Here, the instructions are converted into machine code and executed one after another. On the other hand, in compile time (or compiler-based) language, the entire source code is first converted into machine code and then executed all at once. For example, C++, java, etc. In compiler-based languages, since all the code is first converted to machine language before execution, many errors get identified at the compile stage, which is not the case in interpreter-based languages like Python.

In Python, some predefined keywords are used for handling exceptions. These are as follows:

- **try**: In the **try** block, you write the code for the task to be performed, where errors can occur during runtime.

- **except**: In the **except** block, you write the tasks that need to be performed in case any exception occurs.

- **else**: The code written in the **else** block is executed only when no exception occurs. In simple terms, either the **except** or the **else** block will be executed.


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs

Note: Apart from the **try**, **except** and **else** keywords, there is one more keyword called '**finally**'. The code written in the '**finally**' block will always be executed, irrespective of whether any exception occurs. For example, if a file is opened, it must be closed in any condition, regardless of whether any exceptions occur. Such type of code can be written in the '**finally**' block.

The code that could raise errors is written inside the **try** block, and different exception handlers handle different exceptions, like ZeroDivisionError and ValueError, inside the **except** block. Apart from this, a generic exception handler can also be used, called **Exception**. This handler can handle any type of exception. So, it is recommended to place it after defining all other exception blocks. If you define this generic exception handler in between, it will not let the exception reach their corresponding blocks. The general syntax to define an exception handler is as follows:

except <Handler_type> as <object_name>

Here, **Handler_type** is the type of exception you want to handle. Python supports a lot of inbuilt exceptions, which you can find here. Then, **object_name** is the variable in which you want to store the exception object to access it later.

So, now let's summarise the advantages of exception handling:

- Helps in handling runtime errors so that the application does not go down and continues to run
- Separates the error-prone code from the main logic, making it easier to manage
- Helps in grouping errors based on different exception blocks

3.2 Logging in Python

In real-world scenarios, you need to keep track of every minor issue or bug that occurs in the application. This is where logging is extremely useful. There are many advanced logging tools available in the market like ELK Stack, Datadog, etc. Python also has inbuilt support for logging.

Note: print statement is suitable for some generic code you write on your machine. But when it comes to production-level environments, errors can occur at any time, where print statements may not be helpful as you won't be able to track them. Therefore, log files can be created to store the logs of every instance. Later, the developers can use them to debug the issue.

In python, we use the logging module to create log files and write to them. There are five different levels of logging in python, each with a different severity level. These are as follows:

- **debug**: This level is used when you are diagnosing some problems. The severity level for **debug** is 10. It is useful when you debug your software to ensure everything is running fine.

- **info**: This level is used to inform you that some specific event has occurred, like your application entered a certain state, etc. In simple words, it gives you confirmation that everything is working fine. The severity level for **info** is 20.

- **warning**: This level indicates that something unexpected has happened or a problem will arise in the future. However, the application is still working. For example, all of a sudden, the number of website users has increased, continuously affecting the response time for a specific feature. Then, it comes under the warning level. The severity level for the **warning** is 30.

- **error**: This level indicates that some of the features or functionalities are not working and need to be fixed. The severity level for **error** is 40. For example, suppose the feature recently added to the website is not working properly and also affecting other features.

- **critical**: This level indicates that some severe error has occurred that needs to be checked as soon as possible. For example, suppose a problem with some feature froze or pulled down the entire website. The severity level for **critical** is the highest, that is, 50.

There are some important functions/methods which are used to do the initial configuring settings. These are as follows:

- **basicConfig**: This method configures some initial level settings before your logger object can start logging messages. It takes various parameters, which are as follows:

    - **filename**: By default, all the log messages are printed on the terminal/ console. But if you want them saved in a file, you can specify them here.

    - **format**: This parameter is used to specify the structure of the log message. For example, in the given case, '**%(asctime)s%(message)s**', **%(asctime)s** is used for date and time, and **%(message)s** are used for the actual message that needs to be logged. You can find the complete formats available here in the LogRecord attributes section.

    - **filemode**: This parameter is used to specify the mode in which you want to open the file. For example, 'a' stands for the append mode.

- **getLogger()**: This method is used to create a logger object. For example, **logger = logging.getLogger()** will create a logger object and store the reference in the logger object. Now, whatever log messages get stored will be logged by this logger only.

  Note: If you do not create any logger object, then, by default, the root logger is used as the logger.

- **setLevel()**: This method is used to set the threshold level of the logger. For example, **logger.setLevel(logging.DEBUG)** will set the threshold level to DEBUG, which means that only the messages with an equal or greater severity level than DEBUG will be logged in the console or file. Now, suppose the level is set as WARNING(30), then only WARNING(30), ERROR(40), and

CRITICAL(50) messages will be logged. DEBUG(10) and INFO(20) will be skipped since they have a lower severity level compared to WARNING.