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

   The else block in try-except statement is optional block and it gets executed only when there is no exception in corresponding try block, and code in else block runs when try blocks execution completed without raising exception.

   Eg:

```
try:
    num1 = int(input("Enter a dividend: "))
    num2 = int(input("Enter a divisor: "))
    result = num1 / num2
except ValueError:
    print("Please enter valid integer values.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("The division result is:", result)
```

```
Output1➔ Enter a dividend: 12
Enter a divisor: 15.0
Please enter valid integer values.

Output2➔ Enter a dividend: 14
         Enter a divisor: 5
         The division result is: 2.8
```

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

   Yes, try-except block can be nested inside another try-except block.

   Eg:

```
try:
    a=int(input("Enter the numerator:"))
    try:
        b=int(input("Enter the denominator:"))
        print(f" The result is:{a/b}")
    except ValueError:
        print("Entered denominator value not valid")
    except ZeroDivisionError:
        print("You cannot input Zero as denominator")
except ValueError:
    print("Please enter valid numerator value")
```

```
Output1➔ Enter the numerator:12
Enter the denominator:0
You cannot input Zero as denominator

Output2➔ Enter the numerator:12.1
Please enter valid numerator value
```

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

We create custom exception class, as per the need of the project or situation

Eg:

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

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

def Filter_num(lst1):
    for n in lst1:
        if n==2:
            raise CustomException("List contains value 2")  #defining message

try:
    lst1 = [1,2,6,7,8]
    Filter_num(lst1)
    print(lst1)
except CustomException as e:
    lst1.remove(2)
    print(f"The Newlist is:{lst1}")
    print(e)
```

```
Output1➔ The Newlist is:[1, 6, 7, 8]
Custom Exception: List contains value 2
```

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

Python provides a set of common exceptions that are built-in to the language. These exceptions are raised when specific errors or exceptional situations occur during the execution of a program. Here are some commonly used built-in exceptions in Python:

1. **SyntaxError**: Raised when there is a syntax error in the code.
2. **IndentationError**: Raised when there is an incorrect indentation in the code.
3. **NameError**: Raised when a local or global name is not found.
4. **TypeError**: Raised when an operation or function is performed on an object of inappropriate type.
5. **ValueError**: Raised when a function receives an argument of the correct type but an inappropriate value.
6. **IndexError**: Raised when a sequence (list, tuple, string, etc.) index is out of range.
7. **KeyError**: Raised when a dictionary key is not found.
8. **FileNotFoundError**: Raised when a file or directory is not found.
9. **IOError**: Raised when an input/output operation fails.
10. **ZeroDivisionError**: Raised when division or modulo operation is performed with zero as the divisor.
11. **AssertionError:** Raised when an assertion fails.
12. **AttributeError:** Raised when an attribute reference or assignment fails.
13. **ImportError**: Raised when an import statement fails to find the specified module.
14. **OverflowError**: Raised when the result of an arithmetic operation is too large to be expressed within the limits of the integer type.

These are just a few examples of the many built-in exceptions available in Python. Each exception serves a specific purpose and allows for appropriate error handling and exception propagation in the code.

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

   Logging in python is a module and technique used for recording the log messages during the program execution. These log messages can be stored in various outputs such as console, files and can send them remotely.

   Logging is important during product development for several reasons.

   1. **Debugging and Troubleshooting:**
   2. **Error and Exception tracking**
   3. **Monitoring and performance analysis**
   4. **Auditing and Compliances**
   5. **Documentation and historical analysis.**

   By incorporating logging into software development, developers can enhance the maintainability, reliability and troubleshooting capabilities of their applications.

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

   Log levels in python logging are used to categorize log messages based on their level of severity or importance. This allow programmer to understand the verbosity and granularity of the logs generated by their application.

   The standard log levels in increasing order of their severity are:

   1. **DEBUG:** Lowest log level used for detailed debugging information. This is typically used during development and provide most verbose logs including variable values, function calls and other fine grained details.

      Eg: Tracking the flow of program execution, inspecting variable values, and diagnosing the issues during development.

   2. **INFO:** It is used to indicate the progress or significant events in the application. It provides little higher level of program execution details without excessive details.

      Eg: Logging startup and shutdown details, major configuration changes. Important milestones in the application's execution.

   3. **WARNING:** This log level is used to indicate the potential issues or unexpected conditions that are not critical but may require attention.

      Eg: Deprecation warnings, resource usage near limits, or potential errors that may occur if specific conditions are met.

   4. **ERROR:** It is used to indicate errors and exceptional conditions which affects the normal execution of the program. Errors are used to log critical issues that prevent the application from functioning correctly or produce incorrect results.

      Eg: Exception stack traces, failed operations or unexpected errors that requires immediate attention.

5. **CRITICAL:** It is the highest level of log levels used to indicate the failure of application or severe errors that may lead to termination of the program.
   Critical logs are used for catastrophic failures that require immediate action to prevent further damage.
   Eg: System crashes, unhandled exceptions, or other critical errors that result in the inability to continue program execution.

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

Log formatters in python are objects that determine the structure and format of the log messages. Formatters are used to customize the appearance of log messages by specifying the desired format including, timestamp, log level, module name, message and other relevant information.

```
import logging

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

# Create a logger
logger = logging.getLogger('my_logger')
logger.setLevel(logging.DEBUG)

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

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

# Log some messages
logger.debug('Debug message')
logger.info('Info message')
logger.warning('Warning message')
logger.error('Error message')
logger.critical('Critical message')
```

```
Output➔ 2023-07-10 16:05:40,616 - DEBUG - Debug message
2023-07-10 16:05:40,619 - INFO - Info message
2023-07-10 16:05:40,621 - WARNING - Warning message
2023-07-10 16:05:40,622 - ERROR - Error message
2023-07-10 16:05:40,623 - CRITICAL - Critical message
```

Here , we have used formatter class and constructed desired message using formatter codes. Few, log codes are:

%(asctime)s: The timestamp of the log record.
%(levelname)s: The log level.
%(message)s: The log message.
%(name)s: The name of the logger.
%(filename)s: The name of the file where the logging call was made.
%(lineno)d: The line number in the file where the logging call was made.

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

We can setup logging in followings steps to capture log messages from multiple modules or classes in python application.

1. Import the logging module
   Import logging

2. Configure the logging settings
   logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

3. Add loggers to module or classes
   In each module or class where we want to capture log messages, we add the following line to create a logger.
   logger = logging.getLogger(__name__)

4. Log messages using the created logger
   Within each module or class, we can use the logger to emit log messages of various severity level. Eg. DEBUG, INFO, WARNING, ERROR,CRITICAL.

We can choose appropriate log messages depending on the level of severity

```
logger.debug('Debug message')
logger.info('Info message')
logger.warning('Warning message')
logger.error('Error message')
logger.critical('Critical message')
```

Eg:

```
import logging
logging.basicConfig(level=logging.WARNING) #after this severity level bigins

def Balance(amount):
    if amount < 40000:
        logging.warning('%s is  your remaining balance, please recharge', amount)

    Balance(10000)
```

Output➔ `WARNING:root:10000 is  your remaining balance, please recharge`

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?

Both Logging and print statements produce output. But their intended use , and purpose is quite different.

- For print, output destination is console, while logging can store its output in different specified locations like console, file, network or system log.
- Logging allow to categorize the output messages based on its severity or importance, while print statement output messages are treated equally. We need to manually do conditional checks, in order to understand its importance.
- Print statements are primarily intended to quick debugging of simple output,

as they have limited configurability, whereas logging is highly configurable and flexible. It allows us to customize log formats, severity levels ,storage,etc. We can enable or disable logging based on its severity.

- Logging is designed for long-term maintenance and debugging of an application. It allows to keep a record of applications lifecyle, hence aiding to troubleshooting and debugging issues. Whereas print statements are used for temporary debugging during development.

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.

```
import logging

# Configure logging to write to a file

logging.basicConfig(
    level=logging.INFO,
    filename='app.log',
    filemode='a',          # a means append
    format='%(asctime)s - %(levelname)s - %(message)s'
)

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

Output➔ 2023-07-10 17:23:27,564 - 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.

```
import logging
import datetime

# Configure logging to write to a file
logging.basicConfig(
    level=logging.ERROR,
    filename='Error1.log',
    filemode='a', # a means append
    format='%(asctime)s - %(levelname)s - %(message)s'
)
try:
      x = 1 / 0
except Exception as e:
    timestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    logging.error(f'Exception occurred: {type(e).__name__} at {timestamp}')
```

output➔ 2023-07-10 18:06:46,365 - ERROR - Exception occurred: ZeroDivisionError at 2023-07-10 18:06:46