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 try block test a block of code for errors. The except block handle the error. The else block execute code when there is no error.**

Example:

**try:**

**print("Hello")**

**except:**

**print("Something went wrong")**

**else:**

**print("Nothing went wrong")**

Output:

**Hello**

**Nothing went wrong**

**You can use the else keyword to define a block of code to be executed if no errors were raised**

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

**A: We can have nested try-except blocks in Python. In this case, if an exception is raised in the nested try block, the nested except block is used to handle it. In case the nested except is not able to handle it, the outer except blocks are used to handle the exception.**

Example:

**x = 10**

**y = 0**

**try:**

**print("outer try block")**

**try:**

**print("nested try block")**

**print(x / y)**

**except TypeError as te:**

**print("nested except block")**

**print(te)**

**except ZeroDivisionError as ze:**

**print("outer** except block")

**print(ze)**

Output:

**outer try block**

**nested try block**

**outer except block**

**division by zero**

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

**A. In Python, we can define custom exceptions by creating a new class that is derived from the built-in Exception class.**

Here's the syntax to define custom exceptions,

**class CustomError(Exception):**

**...**

**pass**

**try:**

**...**

**except CustomError:**

**...**

Here, CustomError is a user-defined error which inherits from the Exception class.

Example:

class SalaryNotInRangeError(Exception):

"""Exception raised for errors in the input salary.

Attributes:

salary -- input salary which caused the error

message -- explanation of the error

"""

def \_\_init\_\_(self, salary, message="Salary is not in (5000, 15000) range"):

self.salary = salary

self.message = message

super().\_\_init\_\_(self.message)

salary = int(input("Enter salary amount: "))

if not 5000 < salary < 15000:

raise SalaryNotInRangeError(salary)

Output:

**Enter salary amount: 2000**

**Traceback (most recent call last):**

**File "<string>", line 17, in <module>**

**raise SalaryNotInRangeError(salary)**

**\_\_main\_\_.SalaryNotInRangeError: Salary is not in (5000, 15000) range**

**Here, we have overridden the constructor of the Exception class to accept our own custom arguments salary and message.**

**Then, the constructor of the parent Exception class is called manually with the self.message argument using super().**

**The custom self.salary attribute is defined to be used later.**

**The inherited \_\_str\_\_ method of the Exception class is then used to display the corresponding message when SalaryNotInRangeError is raised.**

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

A.

| **Exception** | **Description** |
| --- | --- |
| ArithmeticError | Raised when an error occurs in numeric calculations |
| AssertionError | Raised when an assert statement fails |
| AttributeError | Raised when attribute reference or assignment fails |
| Exception | Base class for all exceptions |
| EOFError | Raised when the input() method hits an "end of file" condition (EOF) |
| FloatingPointError | Raised when a floating point calculation fails |
| GeneratorExit | Raised when a generator is closed (with the close() method) |
| ImportError | Raised when an imported module does not exist |
| IndentationError | Raised when indentation is not correct |
| IndexError | Raised when an index of a sequence does not exist |
| KeyError | Raised when a key does not exist in a dictionary |
| KeyboardInterrupt | Raised when the user presses Ctrl+c, Ctrl+z or Delete |
| LookupError | Raised when errors raised cant be found |
| MemoryError | Raised when a program runs out of memory |
| NameError | Raised when a variable does not exist |
| NotImplementedError | Raised when an abstract method requires an inherited class to override the method |
| OSError | Raised when a system related operation causes an error |
| OverflowError | Raised when the result of a numeric calculation is too large |
| ReferenceError | Raised when a weak reference object does not exist |
| RuntimeError | Raised when an error occurs that do not belong to any specific exceptions |
| StopIteration | Raised when the next() method of an iterator has no further values |
| SyntaxError | Raised when a syntax error occurs |
| TabError | Raised when indentation consists of tabs or spaces |
| SystemError | Raised when a system error occurs |
| SystemExit | Raised when the sys.exit() function is called |
| TypeError | Raised when two different types are combined |
| UnboundLocalError | Raised when a local variable is referenced before assignment |
| UnicodeError | Raised when a unicode problem occurs |
| UnicodeEncoderror | Raised when a unicode encoding problem occurs |
| UnicodeDecoderr | Raised when a unicode decoding problem occurs |
| UnicodeTranslateError | Raised when a unicode translation problem occurs |
| ValueError | Raised when there is a wrong value in a specified data type |
| ZeroDivisionErrr | Raised when the second operator in a division is zero |

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

**A: Logging is a way to store information about your script and track events that occur. When writing any complex script in Python, logging is essential for debugging software as you develop it. Without logging, finding the source of a problem in your code may be extremely time consuming.**

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

**A: This helps to manage the granularity of information. When log levels are set using the standard logging library, only events of that level or higher will be recorded. It is important to always include a timestamp for each log entry.**

**The default contains six standard logging levels that indicate the seriousness of an event. These are**

* **Notset = 0: This is the initial default setting of a log when it is created. It is not really relevant and most developers will not even take notice of this category. In many circles, it has already become nonessential. The root log is usually created with level WARNING.**
* **Debug = 10: This level gives detailed information, useful only when a problem is being diagnosed.**
* **Info = 20: This is used to confirm that everything is working as it should.**
* **Warning = 30: This level indicates that something unexpected has happened or some problem is about to happen in the near future.**
* **Error = 40: As it implies, an error has occurred. The software was unable to perform some function.**
* **Critical = 50: A serious error has occurred. The program itself may shut down or not be able to continue running properly.**

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

A. **A formatter is created and added to the handler to transform log messages into placeholder data. In this formatter, the time of the log request, the logging level, the logger's name, the module name, and the log message will all print.**

**Description:** setFormatter() is a Handler class method. it is used to format the log message. all handlers use it for formatting.

**Signature:**  setFormatter(fmt) where fmt is a Formatter instance that obtained form logging.Formatter(). the handle can be located in the core logging package or logging.handlers module.

**Program:**

import logging

import sys

logger = logging.getLogger()

logger.setLevel(logging.DEBUG)

streamhdlr = logging.StreamHandler(sys.stdout)

filehdlr = logging.FileHandler('setFormatter.txt')

logger.addHandler(streamhdlr)

logger.addHandler(filehdlr)

streamhdlr.setLevel(logging.WARNING)

filehdlr.setLevel(logging.INFO)

formatter1 = logging.Formatter('{lineno}\*\*{message}\*\* at{asctime}|{name}',style='{')

formatter2 = logging.Formatter('{name}---> {message}',style='{')

filehdlr.setFormatter(formatter1)

streamhdlr.setFormatter(formatter2)

logger.info('we use two handlers')

logger.warning('confirm that you import sys')

logger.info('logger can have multiple handlers!')

**Output:**

root---> confirm that you import sys

setFormatter.txt:

36\*\*we use two handlers \*\* at2021-03-20 13:17:15,482|root 37\*\*confirm that you import sys\*\* at2021-03-20 13:17:15,482|root 38\*\*logger can have multiple handlers !\*\* at2021-03-20 13:17:15,482|root

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

**A: The basics of using the logging module to record the events in a file are very simple. For that, simply import the module from the library.**

1. **Create and configure the logger. It can have several parameters. But importantly, pass the name of the file in which you want to record the events.**
2. **Here the format of the logger can also be set. By default, the file works in append mode but we can change that to write mode if required.**
3. **Also, the level of the logger can be set which acts as the threshold for tracking based on the numeric values assigned to each level.   
   There are several attributes that can be passed as parameters.**
4. **The list of all those parameters is given in Python Library. The user can choose the required attribute according to the requirement.**

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:

## Difference between Logging and Print in Python

| **Logging in Python** | **Print in Python** |
| --- | --- |
| Record events and errors that occur during the execution of Python programs. | Displays the information to the console for the debugging purposes. |
| Mainly used in the production environment. | Mainly for debugging. |
| Some features are: Log levels, filtering, formatting, and more. | There are no good features. |
| It provides different log levels such as Debug, Info, Error, Warning, and Critical. | It does not have any levels, it simply prints whatever is passed to it. |
| Example:  import logging;  logging.basicConfig(level=logging.INFO);  logging.info(“Hello”)  Output:  Can be configured to log to different output destinations (e.g. console, file, network) | Example:  print(“Hello”)  Output:  Prints only on the console |

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:**

**import logging**

**logging.basicConfig(filename='app.log', filemode='w', level=logging.INFO)**

**logging.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.

**A:**

**import logging**

**logging.basicConfig(level=logging.ERROR)**

**def DivideByZero(n,d):**

**try:**

**results = n / d**

**except ZeroDivisionError:**

**logging.error("Division by Zero")**

**else:**

**print(results)**

**DivideByZero(5,0)**

Output:

**ERROR:root:Division by Zero**