# **Python Custom Exceptions Explained**

In Python programming, exceptions are crucial for handling errors and unexpected situations gracefully. While Python provides a range of built-in exceptions, there are instances where creating custom exceptions tailored to specific scenarios is necessary. Custom exceptions allow developers to convey meaningful information about errors and provide a structured way to handle them.

## 

## Understanding Custom Exceptions

In Python, custom exceptions are defined by creating new classes that inherit from the base Exception class. This enables developers to create specialized exceptions that align with the requirements of their applications. Let's delve into how custom exceptions are defined and utilized.

### 

### Defining Custom Exceptions

To define a custom exception in Python, follow these steps:

1. Create a new class that inherits from the Exception class.
2. Optionally, define additional attributes or methods to enhance the exception's functionality.

Here's the general syntax:

class CustomError(Exception):

"Optional description of the custom error."

pass

Once the custom exception class is defined, it can be raised and caught similar to built-in exceptions within Python's try and except blocks.

### 

### Example: Python User-Defined Exception

Let's illustrate the creation and usage of a custom exception through an example:

class InvalidAgeException(Exception):

"Raised the exception error when the input value is <18"

pass

# A threshold age for voting eligibility

voting\_age = 18

try:

user\_age = int(input("Enter your age: "))

if user\_age < voting\_age:

raise InvalidAgeException

else:

print("You are eligible to vote.")

except InvalidAgeException:

print("Exception: Age is below voting threshold.")

In this example:

* We define a custom exception InvalidAgeException to handle cases where the user's age is below the voting threshold.
* Inside the try block, we prompt the user to input their age. If the age is below the threshold, we raise the InvalidAgeException.
* The except block catches the InvalidAgeException and prints an appropriate error message.

## 

## **Customising Exception Classes**

Custom exception classes can be further customized to accept additional arguments or provide more detailed information about the error. This customization is achieved by overriding methods or adding new attributes to the exception class.

Here's an example of a custom exception with additional attributes:

class SalaryNotInRangeError(Exception):

"""This Exception error raised for errors in the input salary attributes.

Attributes:

salary -- input variable salary which caused the custom exception error

message -- explanation of the error

"""

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

self.salary = salary

self.message = message

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

try:

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

if not 4000 < salary < 12000:

raise SalaryNotInRangeError(salary)

except SalaryNotInRangeError as e:

print("Exception:", e.message)

In this example:

* We define a custom exception SalaryNotInRangeError to handle cases where the provided salary is outside a specified range.
* The exception class includes additional attributes salary and message to provide context about the error.
* Within the try block, we validate the user's salary input. If it falls outside the specified range, we raise the SalaryNotInRangeError.
* The except block catches the exception and prints the associated error message.

## 

## **Here's a concise summary of the advantages of using custom exceptions:**

## **Clarity**: Custom exceptions provide clear and descriptive error messages tailored to specific situations, improving code readability and debugging.

## **Modularization**: They promote modularization by encapsulating error-handling logic, enhancing code organization and maintainability.

## **Centralized Handling**: Custom exceptions enable centralized error handling, simplifying error management and promoting consistent practices.

## **Flexibility**: Developers can customize exception classes with additional attributes and methods, providing flexibility for nuanced error handling.

## **Communication**: Well-defined exception classes facilitate clear communication between different parts of the codebase and collaborating developers.

## **Encapsulation**: They promote encapsulation and abstraction principles by isolating error-related functionality within dedicated classes.

## **Reusability**: Once defined, custom exceptions can be reused throughout the codebase, reducing duplication and streamlining development efforts.

## **Documentation**: Custom exceptions serve as self-documenting constructs, providing meaningful labels and descriptions for potential error scenarios, enhancing code comprehension.

## 

Custom exceptions in Python offer a powerful mechanism for handling errors in a structured and informative manner. By creating custom exception classes, developers can encapsulate error-specific logic and enhance the readability and maintainability of their code. Understanding how to define, raise, and handle custom exceptions is a valuable skill for building robust and reliable Python applications.