**Exception Handling in Python:**

#### ****What is an Exception ?****

An exception is an error that happens during the execution of a program. That is, Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called **exceptions**. Exceptions are known to non-programmers as instances that do not conform to a  general rule. The name "exception" implies that the problem (the exception) doesn't occur frequently, i.e. the exception is the "exception to the rule".

#### ****How to handle Exceptions in Python ?****

Exception handling is a construct in some programming languages to handle or deal  with errors automatically. When exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash. In Python, exceptions can be handled using a **try** statement. A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause. It is up to us, what operations we perform once we have caught the exception.

**Synatx:**

try:

# code that may cause exception

except:

# code to run when exception occurs

**Sample Example**

try:

a = 0

b = 0

result = a / b

print(result)

except ZeroDivisionError:

print("Division by zero error occurred")

except Exception as e:

print(f"An unexpected error occurred: {e}")

**Explanation:**

try **block:** This is the block where you place the code that might raise an exception. In this case, you are attempting to perform a division operation a / b, where both a and b are set to 0.

1. except ZeroDivisionError: **block:** This block is executed if a ZeroDivisionError occurs in the try block. A ZeroDivisionError occurs when you try to divide a number by zero. In this block, the program prints a message indicating that a division by zero error occurred.
2. except Exception as e: **block:** This block is a more general exception handler. It will catch any exception that is a subclass of the Exception class. If an unexpected error occurs (other than a ZeroDivisionError), this block will be executed. It prints a message that includes information about the specific error using the variable e, which holds the exception instance.

# Using finally and else with try -except block

In some situations, we might want to run a certain block of code if the code block inside try runs without any errors.

For these cases, you can use the optional else keyword with the try statement

**Example 2**

try:

num = int(input("Enter a number: "))

assert num % 2 == 0

except:

print("Not an even number!")

else:

reciprocal = 1 / num

print(reciprocal)

**Explanation:**

1. try **block:** The user is prompted to enter a number using input(), and it is converted to an integer (int). If the input is not a valid integer (for example, if the user enters a string), it may raise a ValueError. If the input is an even number, the code continues to the else block.
2. assert num % 2 == 0**:** The assert statement checks if the entered number (num) is even. If it's not even, it raises an AssertionError, and the code jumps to the except block.
3. except **block:** If an exception occurs in the try block, it is caught here. If the input is not an even number or if there's a ValueError (e.g., if the user enters a non-integer), it prints "Not an even number!"
4. else **block:** If the try block completes without any exceptions (i.e., the entered number is an even integer), the code proceeds to the else block. Here, it calculates the reciprocal of the even number and prints it.

## **Python try...finally**

In Python, the finally block is always executed no matter whether there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

**Example:3**

try:

numerator = 10

denominator = 0

result = numerator/denominator

print(result)

except:

print("Error: Denominator cannot be 0.")

finally:

print("This is finally block.")

**Explanation:**

try **block:** The code inside this block is the part where an exception might occur. In this case, it attempts to divide numerator by denominator, which is 0. This will result in a ZeroDivisionError.

except **block:** If an exception occurs in the try block, the code inside the except block is executed. In this case, it catches any exception (using a generic except), and it prints a message saying "Error: Denominator cannot be 0."

finally **block:** The finally block contains code that will be executed whether an exception occurs or not. In this case, it prints "This is the finally block."

**Here's how the program flow works:**

Since the denominator is 0, the division operation in the try block will raise a ZeroDivisionError.

The code in the except block is executed, printing the error message.

After the except block, the finally block is executed, printing "This is the finally block."

The finally block is commonly used for cleanup operations or actions that must be performed whether an exception occurred or not.

# Standard Exceptions in Python

|  |  |
| --- | --- |
| ****EXCEPTION NAME**** | ****DESCRIPTION**** |
| Exception | Base class for all exceptions |
| StopIteration | Raised when the next() method of an iterator does not point to any object. |
| SystemExit | Raised by the sys.exit() function. |
| StandardError | Base class for all built-in exceptions except StopIteration and SystemExit. |
| ArithmeticError | Base class for all errors that occur for numeric calculation. |
| OverflowError | Raised when a calculation exceeds maximum limit for a numeric type. |
| FloatingPointError | Raised when a floating point calculation fails. |
| ZeroDivisionError | Raised when division or modulo by zero takes place for all numeric types. |
| AssertionError | Raised in case of failure of the Assert statement. |
| AttributeError | Raised in case of failure of attribute reference or assignment. |
| EOFError | Raised when there is no input from either the raw\_input() or input() function and the end of file is reached. |
| ImportError | Raised when an import statement fails. |
| KeyboardInterrupt | Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| LookupError | Base class for all lookup errors. |
| IndexError  KeyError | Raised when an index is not found in a sequence.  Raised when the specified key is not found in the dictionary. |
| NameError | Raised when an identifier is not found in the local or global namespace. |
| UnboundLocalError  EnvironmentError | Raised when trying to access a local variable in a function or method but no value has been assigned to it.  Base class for all exceptions that occur outside the Python environment. |
| IOError  IOError | Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.  Raised for operating system-related errors. |
| SyntaxError  IndentationError | Raised when there is an error in Python syntax.  Raised when indentation is not specified properly. |
| SystemError | Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |
| SystemExit | Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit. |
| TypeError | Raised when an operation or function is attempted that is invalid for the specified data type. |
| ValueError | Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| RuntimeError | Raised when a generated error does not fall into any category. |
| NotImplementedError | Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |