😊 Errors and Exception Handling

In Python, errors and exceptions are problems that occur during the execution of a program. Proper handling of these issues ensures that the program does not crash unexpectedly and can continue running smoothly.

1. Types of Errors in Python

a) Syntax Errors

- These occur when the code violates Python's syntax rules.
- Example: Missing colons, incorrect indentation, or misspelled keywords.

b) Exceptions

- Exceptions occur when a program is syntactically correct but encounters an error during execution.
- Common exceptions include:
    - **ZeroDivisionError**: Dividing by zero.
    - **TypeError**: Using incompatible data types together.
    - **NameError**: Using a variable before defining it.
    - **IndexError**: Accessing an index that does not exist in a list.
    - **KeyError**: Trying to access a non-existent key in a dictionary.
    - **FileNotFoundError**: Trying to open a file that doesn't exist.

2. Handling Exceptions with try-except

Python provides the try-except block to handle exceptions gracefully.

- **try block**: Contains the code that may raise an exception.
- **except block**: Handles the exception if one occurs.
- **finally block**: Executes code whether an exception occurs or not (useful for cleanup tasks like closing files).
- **else block**: Runs only if no exceptions occur.

3. Raising Exceptions Manually

You can raise exceptions manually using the raise keyword when you need to stop execution due to specific conditions.

4. Benefits of Exception Handling

- **Prevents program crashes** by handling unexpected issues.
- **Improves user experience** by displaying friendly error messages.

- **Ensures resource management**, like closing files or network connections properly.
- **Makes debugging easier** by logging or printing error messages.

By using proper exception handling, Python programs become more reliable and can recover from unexpected situations efficiently.

😄 To begin with the Lab

1. We start with the very basic error, which is a syntax error. As you can see below in the print statement, we wanted to print hello as a string, but we did not add the quote at the end, so we get a syntax error.

```
[1]: print('Hello)
```

```
  File "<ipython-input-1-db8c9988558c>", line 1
    print('Hello)
                 ^
SyntaxError: EOL while scanning string literal
```

2. The basic terminology and syntax used to handle errors in Python are the **try** and **except** statements. The code that can cause an exception to occur is put in the try block, and the handling of the exception is then implemented in the except block of code.
3. The code below uses a **try-except-else** structure for error handling while working with files. In the **try** block, it attempts to open a file in write mode and write some content. If an **IOError** occurs during these operations, the **except** block prints an error message. If no error happens, the **else** block confirms that the content was written successfully and then closes the file.

```
[2]: try:
        f = open('testfile','w')
        f.write('Test write this')
    except IOError:
        # This will only check for an IOError exception and then execute this print statement
        print("Error: Could not find file or read data")
    else:
        print("Content written successfully")
        f.close()
```

```
Content written successfully
```

4. This code attempts to open a file called "**testfile**" in read mode. It then tries to write to it, which is not allowed. As a result, an **IOError** occurs, and the except block prints the error message. The **else** block is not executed because an error occurred.

[3]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute this print statement
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

```
Error: Could not find file or read data
```

5. Here is a little because the code below tries to open a file named "**testfile**" in read mode and then write to it. Since writing in read mode is not allowed, an exception is raised. The except block catches any exception and prints an error message, while the else block is skipped because an error occurred.

[4]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except:
    # This will check for any exception and then execute this print statement
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

```
Error: Could not find file or read data
```

6. This time we write the code in a way that it opens a file in write mode, writes a message, and then closes the file. Regardless of whether an error occurs during these steps, the code inside the finally block always executes, printing a message.

[5]:
```
try:
    f = open("testfile", "w")
    f.write("Test write statement")
    f.close()
finally:
    print("Always execute finally code blocks")
```

```
Always execute finally code blocks
```

7. This function asks the user for an integer. It tries to convert the input into an integer and store it in a variable. If that fails, it prints an error message. The finally block always runs and prints its message. Then, it prints the integer value.
8. **Note** that if the conversion fails, printing the value will cause an error because the variable was never successfully assigned. That is why we get an error while executing in the 8th cell.

[6]:
```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!")

    finally:
        print("Finally, I executed!")
    print(val)
```

[7]:
```
askint()
```

```
Please enter an integer: 5
Finally, I executed!
5
```

[8]:
```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-8-cc291aa76c10> in <module>()
----> 1 askint()

<ipython-input-6-c97dd1c75d24> in askint()
      7     finally:
      8         print("Finally, I executed!")
----> 9     print(val)

UnboundLocalError: local variable 'val' referenced before assignment
```

9. This function, **askint**, first tries to convert the user's input into an integer. If that conversion fails, it catches the error, prints a message, and asks the user to input an integer again. Regardless of success or failure, the finally block runs, printing its message. Finally, the function prints the resulting integer value.
10. But the problem is that it only checks the first input and the second input was also indeed incorrect but it printed the message.

[9]:
```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!")
        val = int(input("Try again-Please enter an integer: "))
    finally:
        print("Finally, I executed!")
    print(val)
```

[10]:
```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Try again-Please enter an integer: four
Finally, I executed!
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-92b5f751eb01> in askint()
      2     try:
----> 3         val = int(input("Please enter an integer: "))
      4     except:

ValueError: invalid literal for int() with base 10: 'five'

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-10-cc291aa76c10> in <module>()
----> 1 askint()

<ipython-input-9-92b5f751eb01> in askint()
      4     except:
      5         print("Looks like you did not enter an integer!")
----> 6         val = int(input("Try again-Please enter an integer: "))
      7     finally:
      8         print("Finally, I executed!")

ValueError: invalid literal for int() with base 10: 'four'
```

11. So, to make things clearer we are going to keep checking until it gets us the correct answer using the **while loop**.
12. The function repeatedly asks the user for an integer. It tries to convert the input; if that fails, it prints an error and starts over. If successful, it prints a confirmation and breaks out of the loop. The finally block always runs, printing its message regardless of success or failure.

[11]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            break
        finally:
            print("Finally, I executed!")
        print(val)
```

•••

[12]:
```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: four
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 3
Yep that's an integer!
Finally, I executed!
```

13. This function again keeps asking the user for an integer until a valid number is provided. On each loop, it tries to convert the input to an integer. If it fails, it prints an error and restarts the loop. If it succeeds, it prints confirmation and the number, then stops. No matter what happens, it always prints "Finally, I executed!" before moving on.

[13]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            print(val)
            break
        finally:
            print("Finally, I executed!")
```

[14]:
```
askint()
```

```
Please enter an integer: six
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 6
Yep that's an integer!
6
Finally, I executed!
```