EXPERIMENT - 4

GROUP MEMBERS NAME:
Harshal Jain - 60004190041
Junaid Girkar - 60004190057
Khushi Chavan - 60004190061
Megh Dedhia - 60004190067

AIM: Explore the exception functionalities in python

THEORY:
The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.
Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

TRY EXCEPT :
The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped.
If the exception does occur, the program flow is transferred to the except: block. The statements in the except: block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message. Also, You can specify the type of exception after the except keyword.
The following example will throw an exception when we try to divide an integer by a string.

CODE:

```
try:
    a=int(input("Enter a number :"))
    b='0'
    print(a/b)
except:
    print('Some error occurred.')
print("Out of try except blocks.")
```

OUTPUT:

```
Enter a number :5
Some error occurred.
```

```
Out of try except blocks.
```

TRY EXCEPT ELSE:

In some situations, you might want to run a certain block of code if the code block inside tries to run without any errors. For these cases, you can use the optional else keyword with the try statement. The example below tries to read and write in the file.

CODE:

```
try:
fh = open("testfile", "r")
      fh.write("This is my test file for exception handling!!")
except IOError:
      print("Error: can\'t find file or read data")
else:
      print("Written content in the file successfully")
      fh.close()
```

OUTPUT:

```
Error: can't find file or read data
```

TRY FINALLY

In Python, keyword ‘finally’ can also be used along with the try and except clauses.
The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however, there's an exception in the try block, the appropriate except block will be processed, and then statements in the finally block will be processed before proceeding to the rest of the code.

CODE:

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print("This line will be executed always")
```

OUTPUT:

```
Error: This line will be executed always
```

TRY EXCEPT FINAL :

CODE:

```
try:
    k = 5//0 # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

OUTPUT:

```
Can't divide by zero
This is always executed
```

MULTIPLE ERRORS WITH ONE EXCEPTION:

A single try block may have multiple except clauses with different exception types in it. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

CODE:

```
def fun(a):
    if a < 4:

        # throws ZeroDivisionError for a = 3
        b = a/(a-3)

    # throws NameError if a >= 4
    print("Value of b = ", b)
```

```
try:
    fun(3)
    fun(5)

# note that braces () are necessary here for
# multiple exceptions
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

OUTPUT:

```
ZeroDivisionError Occurred and Handled
```

RAISING CUSTOM EXCEPTIONS:

Python also provides the raise keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

CODE:

```
try:
    raise NameError("Hi there") # Raise Error
except NameError:
    print ("An exception")
    raise # To determine whether the exception was raised or not
```

OUTPUT:

```
An exception

---------------------------------------------------------------------
---------
NameError                                 Traceback (most recent
call last)
~\AppData\Local\Temp/ipykernel_286296/2348914011.py in <module>
      2
      3 try:
----> 4     raise NameError("Hi there") # Raise Error
      5 except NameError:
```

```
      6     print ("An exception")

NameError: Hi there
```

ASSERTION ERROR

Assertion is a programming concept used while writing a code where the user declares a condition to be true using assert statement prior to running the module. If the condition is True, the control simply moves to the next line of code. In case if it is False the program stops running and returns AssertionError Exception.

The function of assert statements is the same irrespective of the language in which it is implemented, it is a language-independent concept, only the syntax varies with the programming language.

`CODE:`

```
try:
      x = 1
      y = 0
      assert y != 0, "Invalid Operation"
      print(x / y)
# the errror_message provided by the user gets printed
except AssertionError as msg:
      print(msg)
```

`OUTPUT:`

```
Invalid Operation
```

CONCLUSION

We learnt about the different errors in python and the various means of handling these errors using python's inbuilt functionalities. We have also tested out these functionalities by using them in a python script.