

VIDHI ROHIRA
S.Y B.TECH
SEM III
COMPUTER ENGINEERING

PD LAB 11

231071052

BATCH – C

LABORATORY 11

AIM:- To study exception handling in python.

THEORY:-

Q] WHAT IS EXCEPTION HANDLING?

Exception handling in Python is a mechanism that allows you to handle runtime errors, or exceptions, in a structured way. Instead of your program crashing when an error occurs, you can use exception handling to catch and manage errors, allowing your program to continue running or provide a meaningful error message to the user.

Python uses **try**, **except**, **else**, and **finally** blocks to handle exceptions.

Key Concepts:

1. **try** block:

- You write the code that may potentially raise an exception inside the try block. If an error occurs in this block, the rest of the code in the block is skipped.

2. **except** block:

- If an exception occurs in the try block, it is caught by the except block. You can specify the type of exception to catch, or use a generic exception handler to catch all exceptions.

3. **else** block (optional):

- If no exception occurs in the try block, the code inside the else block will execute. It's useful when you want to specify code that runs only when the try block is successful.

4. **finally** block (optional):

- This block will always execute, regardless of whether an exception occurred or not. It's often used for cleanup actions (e.g., closing files or network connections).

Q] WHAT ARE LOGICAL ERRORS IN PYTHON?

A **logical error** in Python occurs when the code runs without any syntax or runtime errors but does not produce the expected results. These errors are usually the result of incorrect logic in the program, which means the program runs successfully, but the output or behavior is incorrect.

Unlike **syntax errors** (which are caught by Python's interpreter) or **runtime errors** (which are caught during execution), **logical errors** do not stop the program from running. However, they can lead to wrong outputs, incorrect program behavior, or undesired results.

Q] HOW TO FIX THESE ERRORS?

Fixing Exception Handling Errors:

Ensure that you catch specific exceptions, not all exceptions, so that you can handle them appropriately. Also, avoid using bare `except:` clauses unless absolutely necessary.

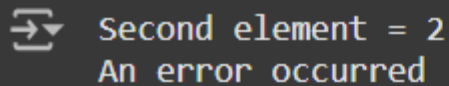
Fixing Logical Errors:

Carefully review the logic of your code to ensure that conditions and operations are correct. Use debugging tools (like `print()` statements or a debugger) to trace variable values and execution flow.

SOME CODES AND EXAMPLES

1] Index Error:

```
a = [1, 2, 3]
try:
    print ("Second element = %d" %(a[1]))
    print ("Fourth element = %d" %(a[3]))
except:
    print ("An error occurred")
```

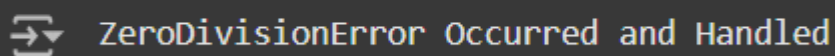


```
⇒ Second element = 2
   An error occurred
```

Reason: The error is caught by the `except` block. Since you did not specify the type of exception (e.g., `except IndexError:`), it catches **any** exception, including the `IndexError` that occurs when trying to access `a[3]`. Therefore, "An error occurred" is printed.

2] Zero Division Error and Name Error:

```
def fun(a):
    if a < 4:
        b = a/(a-3)
        print("Value of b = ", b)
try:
    fun(3)
    fun(5)
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```



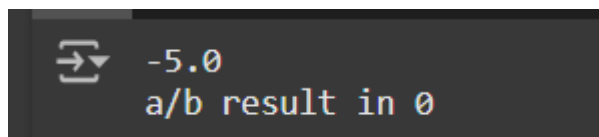
```
⇒ ZeroDivisionError Occurred and Handled
```

Reason:

- In the function `fun(a)`, when $a = 3$, the expression $a / (a - 3)$ will attempt to divide by zero ($3 / (3 - 3)$), which raises a **ZeroDivisionError**. However, this doesn't immediately raise the error because the variable `b` is defined within the `if` block.
- After the `if` block, the program tries to print `b`. But if the condition $a < 4$ is not met (like when $a = 5$), `b` is not assigned any value. This results in a **NameError** because you're trying to print an undefined variable `b` when $a = 5$.

3] Zero Division Error:

```
def AbyB(a , b):  
    try:  
        c = ((a+b) / (a-b))  
    except ZeroDivisionError:  
        print ("a/b result in 0")  
    else:  
        print (c)  
AbyB(2.0, 3.0)  
AbyB(3.0, 3.0)
```



```
-5.0  
a/b result in 0
```

Reason:

For `AbyB(2.0, 3.0)`:

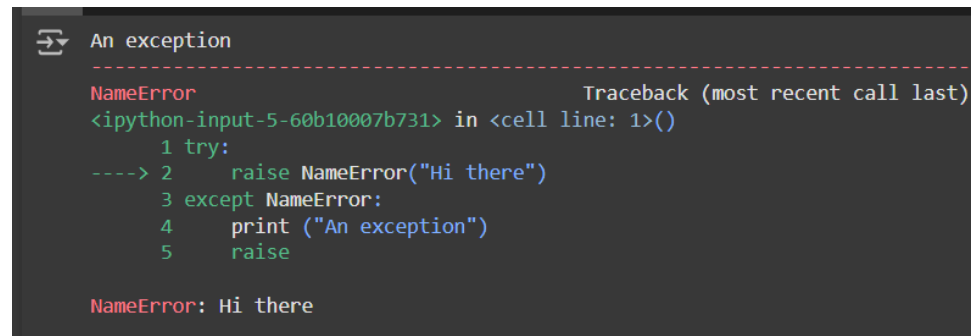
- The denominator $(a - b)$ is $2.0 - 3.0 = -1.0$, which is non-zero. Therefore, the division happens successfully, and the result is printed.

For `AbyB(3.0, 3.0)`:

- The denominator $(a - b)$ is $3.0 - 3.0 = 0.0$. Since division by zero is not allowed in Python, a **ZeroDivisionError** occurs, and the `except` block is triggered, printing "a/b result in 0".

4] Name Error:

```
try:
    raise NameError("Hi there")
except NameError:
    print ("An exception")
    raise
```



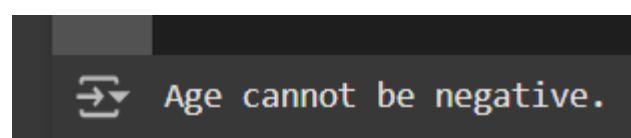
The screenshot shows a Jupyter Notebook cell with a NameError exception. The output of the cell is "An exception". Below this, a traceback is displayed, showing the error occurred in a cell at line 1, column 1. The traceback includes the code from the cell: a try block that raises a NameError("Hi there"), which is caught by an except NameError: block that prints "An exception" and then re-raises the exception. The final line of the traceback shows the NameError: Hi there.

Reason:

- The raise NameError("Hi there") statement explicitly raises a NameError with the message "Hi there".
- The except NameError: block catches this NameError and prints "An exception".
- The raise statement inside the except block re-raises the **same** NameError that was caught.

5] Negative Age Error:

```
class NegativeAgeError(Exception):
    pass
def check_age(age):
    if age < 0:
        raise NegativeAgeError("Age cannot be negative.")
    return f"Age is {age}"
try:
    print(check_age(-5))
except NegativeAgeError as e:
    print(e)
```

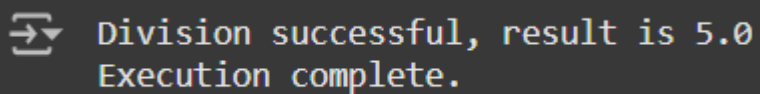


The screenshot shows a Jupyter Notebook cell with a NegativeAgeError exception. The output of the cell is "Age cannot be negative.". The error message is displayed in a red font, indicating an exception.

Reason: Negative age entered

6] Exception Handling case:

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print(f"Division successful, result is {result}")
finally:
    print("Execution complete.")
```



```
⇒ Division successful, result is 5.0
   Execution complete.
```

Execution flow:

1. The **try block** successfully divides 10 / 2 and assigns the result (5.0) to the variable result.
2. Since no **ZeroDivisionError** occurs, the **except block** is skipped.
3. The **else block** executes, printing "Division successful, result is 5.0".
4. The **finally block** always executes, so it prints "Execution complete.".

7] File not found Error:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    open("non_existent_file.txt")
```

Reason: Put a non existent file so it has nothing to read or execute or open.

8] Exception Handling:

```
class InsufficientBalanceError(Exception):
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        super().__init__(f"Insufficient balance: {balance}, attempted to
withdraw: {amount}")

try:
    balance = 100
    amount = 150
    if amount > balance:
        raise InsufficientBalanceError(balance, amount)
except InsufficientBalanceError as e:
    print(e)
```

```
⇒ Insufficient balance: 100, attempted to withdraw: 150
```

Execution Flow:

- The exception **InsufficientBalanceError**(balance, amount) is raised with the message "Insufficient balance: 100, attempted to withdraw: 150".
- This message is captured by the except block and printed.

9] Syntax Error:

```
x = 5
if x = 5: # Error: should be `==`
    print("x is 5")
```

```
⇒ File "<ipython-input-14-40263eb0454b>", line 2
    if x = 5: # Error: should be `==`
        ^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Reason: Invalid syntax

It must be “==” and not “=”

10] Index Error:

```
items = [1, 2, 3]
for i in range(len(items) + 1): # Off-by-one error, extra iteration
    print(items[i]) # Will raise IndexError on the last iteration
```

```
1
2
3
-----
IndexError                                Traceback (most recent call last)
<ipython-input-15-934523f53acf> in <cell line: 2>()
      1 items = [1, 2, 3]
      2 for i in range(len(items) + 1): # Off-by-one error, extra iteration
----> 3     print(items[i]) # Will raise IndexError on the last iteration

IndexError: list index out of range
```

Reason:

The error occurs because of an **off-by-one error** in the `range(len(items) + 1)`.

- The list `items` has 3 elements: `[1, 2, 3]`, which means the valid indices are 0, 1, and 2.
- The `range(len(items) + 1)` generates a range from 0 to 4 (because `len(items)` is 3, and `3 + 1 = 4`), which results in the following iteration sequence: 0, 1, 2, 3.
- When `i = 3`, the expression `items[i]` tries to access `items[3]`, which is out of range because the list only has indices 0, 1, and 2. This raises an **IndexError**.

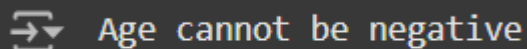
11] Logical error :

```
x = 7
if x > 10 and x < 5: # Error: this condition can never be true
    print("Invalid logic.")
```

Reason: The above makes no sense.

12] Negative Age Error:

```
class InvalidAgeError(Exception):  
    pass  
  
def check_age(age):  
    if age < 0:  
        raise InvalidAgeError("Age cannot be negative")  
    elif age < 18:  
        print("Minor")  
    else:  
        print("Adult")  
  
try:  
    check_age(-5)  
except InvalidAgeError as e:  
    print(e)
```

A screenshot of a Python error message. It features a small icon of a box with an arrow pointing right, followed by the text "Age cannot be negative" in a monospaced font.

Reason: Negative Age entered.

CONCLUSION:-

In this lab session, we learned about **exception handling** in Python, including how to catch and manage errors using try, except, else, and finally blocks. We explored common errors like ZeroDivisionError and IndexError, and understood how to handle them properly. Additionally, we studied **logical errors**, which occur due to incorrect logic in the code, such as off-by-one errors in loops. By understanding the reasons behind these errors, we can write more robust and error-free code. This session emphasized the importance of debugging and exception handling to improve code quality.