Welcome, to yet another tale under the Python Series.....

We present to you, the sequel to Exception Handling 1

Exception Handling The Final Chapter

Presented By

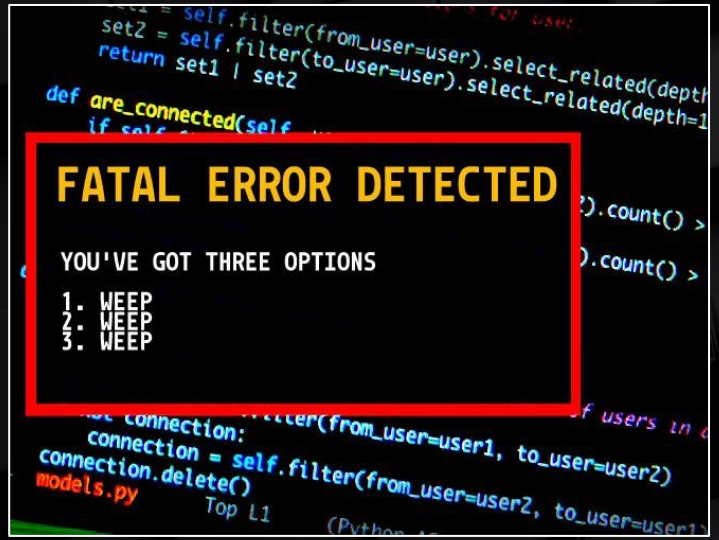
Group 10

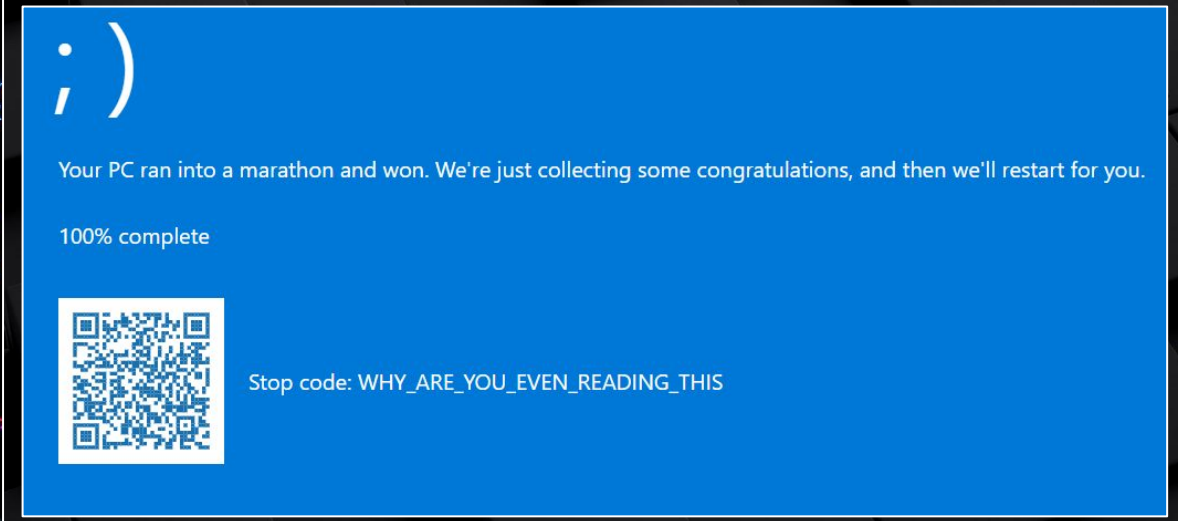
Tanisha
Riya Nain
Anushka Gupta
Asheesh Kumar
Aditya Chaurasiya
Kamaltosh Jha

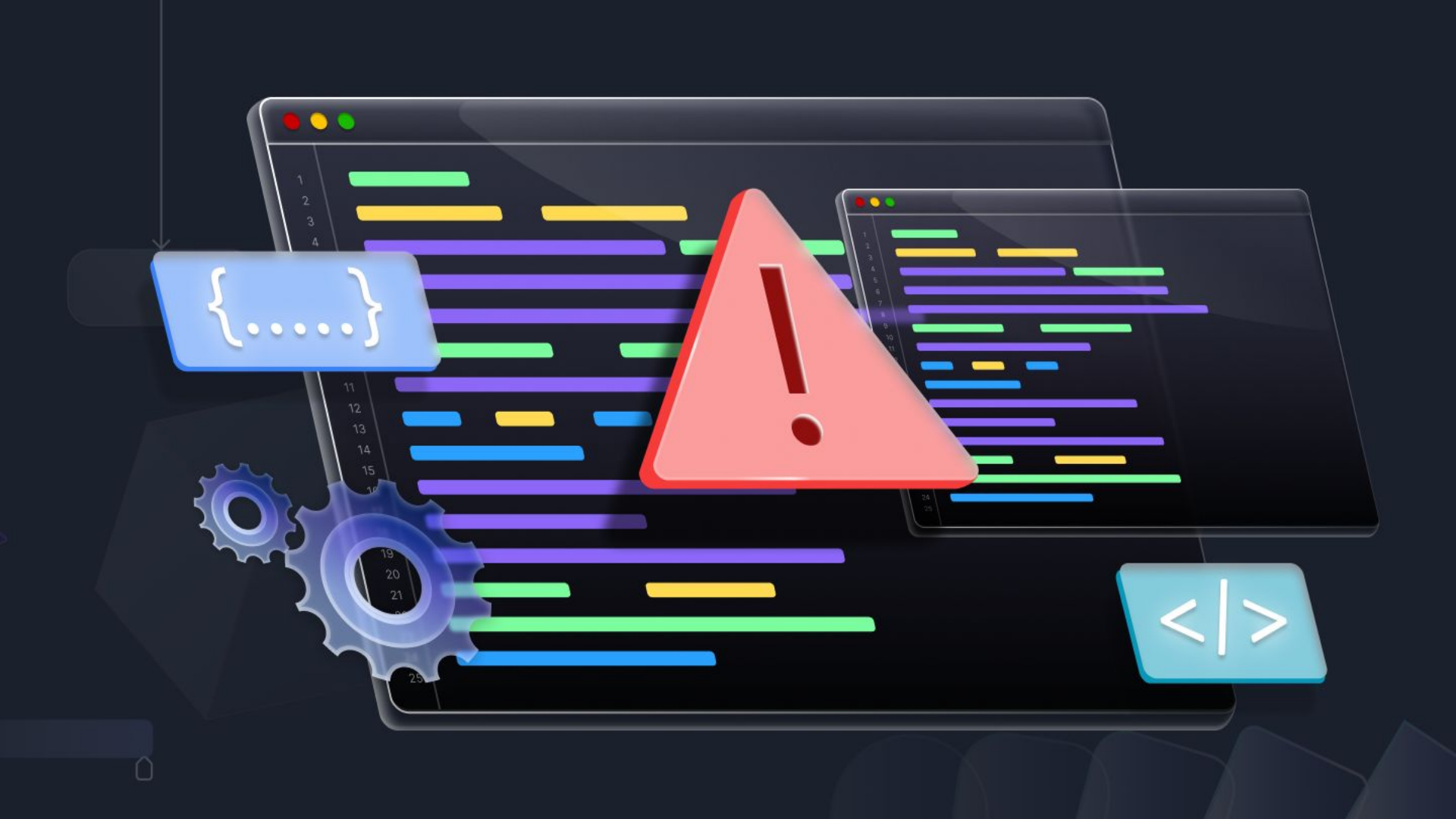
Mysteries yet to be revealed

1. Introduction to Exception Handling
2. Types of Exceptions - Built In Exceptions
3. User Defined Exceptions
4. Mini program
5. Try ,Except keyword
6. Finally and Else keyword
7. The Final Showdown - A Program Depicting the above.

Introduction to Exception Handling








MIKAEL GUSTAFSSON

INTRODUCTION TO EXCEPTION HANDLING

INTRODUCTION TO EXCEPTION HANDLING

An exception is an error which happens at the time of execution of a program.

However, while running a program when an error occurs, Python generates an exception that should be handled to avoid your program to crash.

In Python language, exceptions trigger automatic on errors, or they can be triggered and intercepted by set of code.

Example

Suppose we have to make an online payment using mobile application but we don’t have enough money in our bank account. Now at this time we get an insufficient fund error from the payment application.This can be interpreted as an exception because we get it at run time and it warned us before causing some fault to the underlying system

SITUATIONS THIS CAN BE IMPLEMENTED

Some common examples of such errors are dividing a number by zero, adding two incompatible types, trying to access a non-existent index of a sequence or accessing a file that does not exist.

Exception handling is mechanism through which we can handles the run time error.

Exception Handling is not the ultimate way to handle errors, there are methods to do it with custom ways, later in the slides

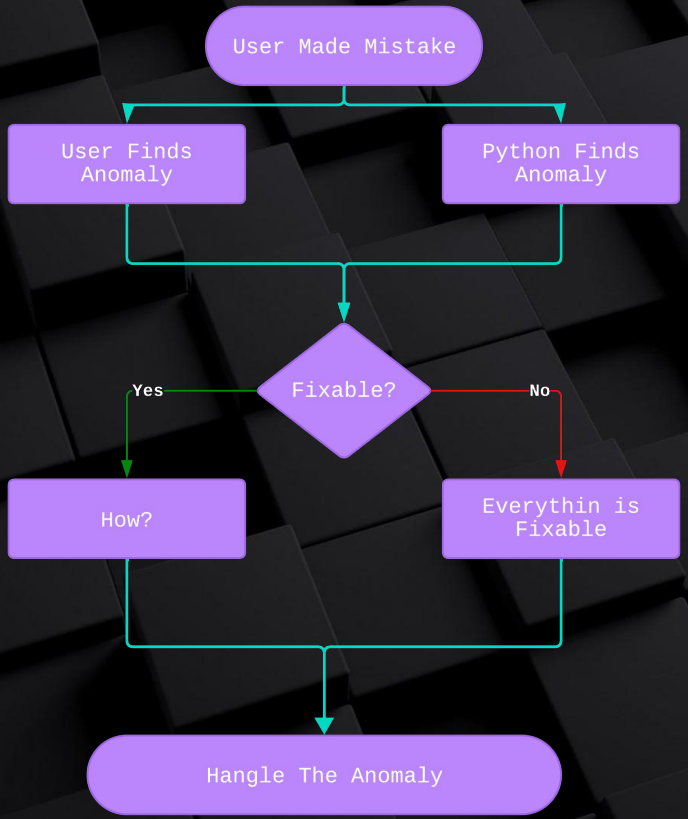
There are some keywords involved

try, except, raise, finally and else

Taken further in the slides.

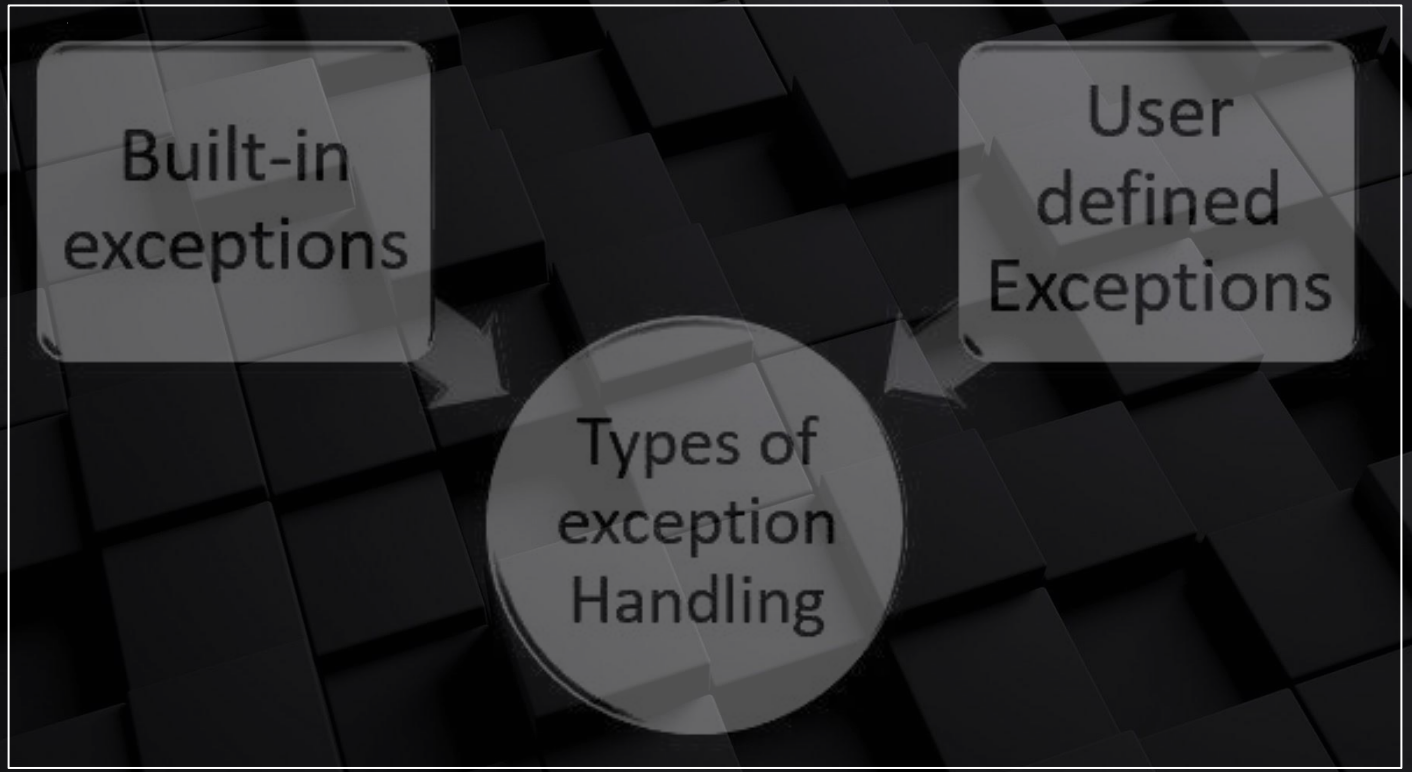
There are types of Exceptions as well. Some are already there in python and also a way by which we can create our own.

The Underlying Process



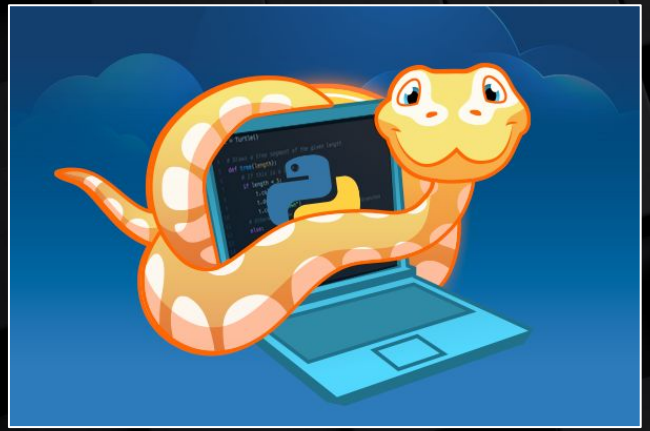
TYPES OF EXCEPTIONS

TYPES OF EXCEPTIONS



Built-In Exceptions

Python has a number of built-in exceptions, such as the well-known errors SyntaxError, NameError, and TypeError. These Python Exceptions are thrown by standard library routines or by the interpreter itself. They are built-in, which implies they are present in the source code at all times.



User Defined Exceptions

User-defined exceptions are ones that the developer writes in their code to address particular errors. These exceptions are defined by creating a new class that inherits from the Exception class or one of its subclasses.

Built-in Exceptions

Built-in Exceptions

While writing code in Python, errors can occur due to various reasons, such as invalid input, syntax errors, or unexpected behavior.

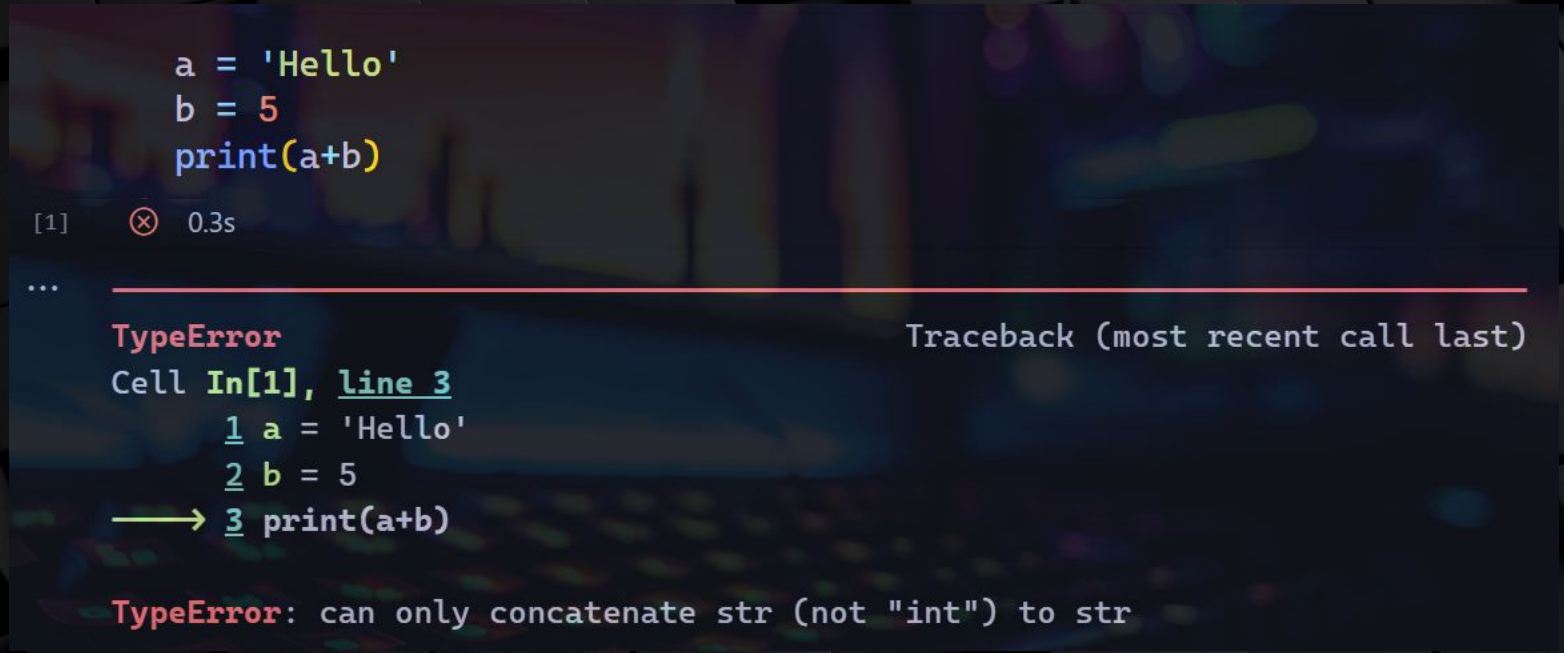
To handle such errors, Python provides a set of built-in exceptions. These exceptions are predefined in Python and can be raised by the interpreter or by the user's code.

When a program encounters a condition which may cause error or a fault during runtime, the Python Interpreter raises some predefined exceptions. These exceptions are part of the Python language and provide a way to handle errors and unexpected situations.

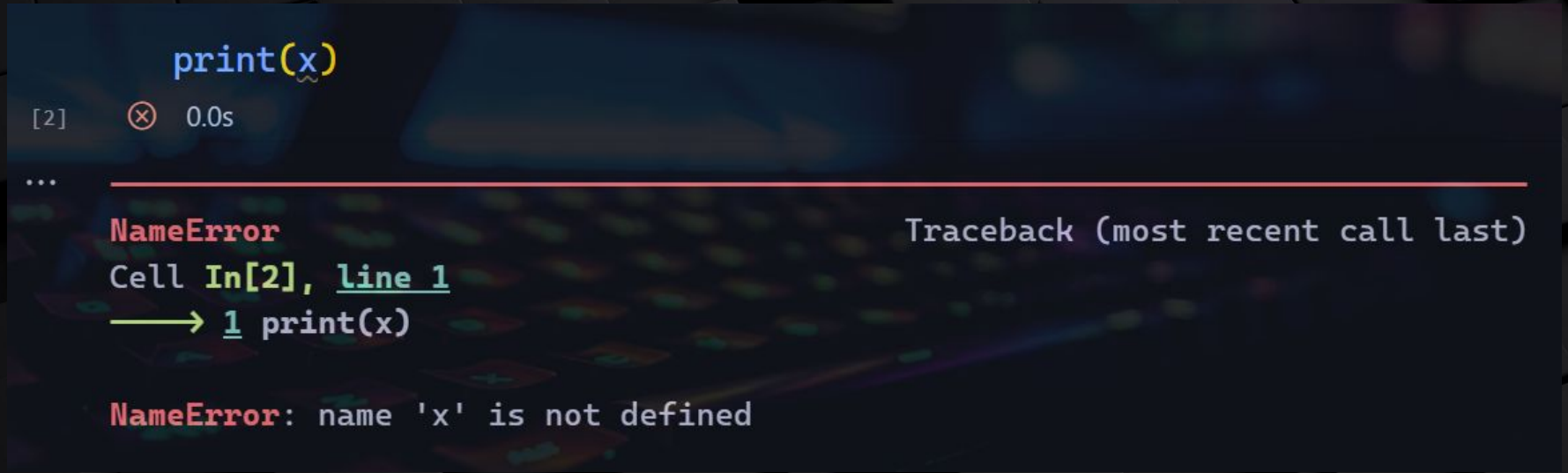
These exceptions help in debugging and handling errors gracefully. They can be caught using try-except blocks to handle specific errors or perform appropriate actions when an exception occurs.

These are as follows…….

1. **TYPE ERROR:**
   This exception is raised when an operation or function is applied to an object of inappropriate type.
   For example, trying to concatenate a string and an integer will result in a TypeError.



NameError – This exception is raised when a name or variable is not defined. For example, if we try to access a variable that has not been defined, a NameError will be raised.



ValueError – When an operation or function is applied to an object with an inappropriate value, this exception is thrown. A ValueError will be raised, for instance, if an attempt is made to convert a string to an integer when the string does not represent an acceptable integer.

```
a = 'abc'
b = int(a)
```

⊗ 0.0s

```
ValueError                                Traceback (most recent call last)
Cell In[3], line 2
      1 a = 'abc'
----> 2 b = int(a)

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

IndexError – When an index is outside of bounds, this exception is thrown. An IndexError will be raised, for instance, if an element in a list is attempted to be accessed using an index that does not exist.

```
a = [1, 2, 3]
print(a[3])
```

⊗ 0.0s

```
IndexError                                Traceback (most recent call last)
Cell In[4], line 2
      1 a = [1, 2, 3]
----> 2 print(a[3])

IndexError: list index out of range
```

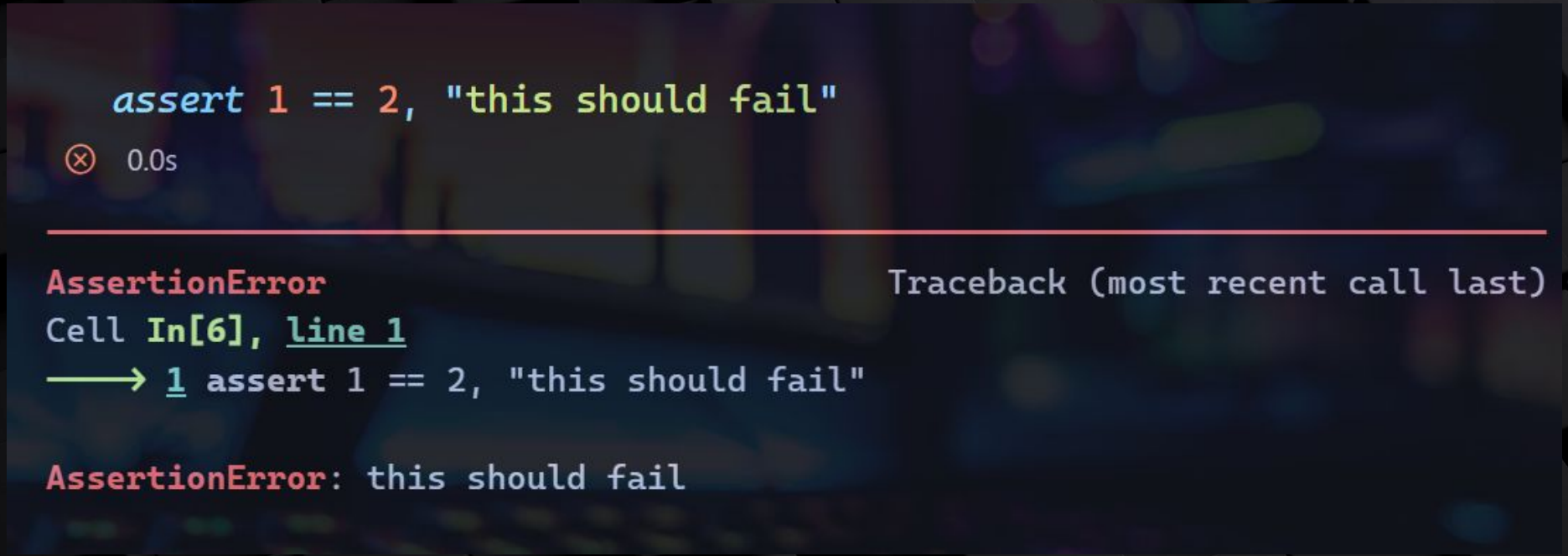
KeyError – This exception is raised when a key is not found in a dictionary. For example, trying to access a value in a dictionary with a key that does not exist will result in a KeyError.

```
a = {'a' : 1, 'b' : 2}
print(a['c'])
```

⊗ 0.0s

```
KeyError                                  Traceback (most recent call last)
Cell In[5], line 2
      1 a = {'a' : 1, 'b' : 2}
----> 2 print(a['c'])

KeyError: 'c'
```

AssertionError – This exception is raised when an assertion fails. Assertions are used to check that certain conditions are met, and if they are not, an AssertionError is raised.



IndentationError – Raised when indentation is incorrect:

```
for i in range(1, 5):
print(i)
```

⊗ 0.0s

```
  Cell In[7], line 2
    print(i)
    ^
IndentationError: expected an indented block after 'for' statement on line 1
```

ZeroDivisionError: When the second operator in a division is zero, an error is raised.

```
x = 5
y = 0
z = x/y
```

⊗ 0.0s

```
ZeroDivisionError                         Traceback (most recent call last)
Cell In[11], line 3
      1 x = 5
      2 y = 0
----> 3 z = x/y

ZeroDivisionError: division by zero
```

Overflow Error: This error is raised when an arithmetic operation exceeds the limits to be represented

```
j = 5.0
for i in range(1, 100):
    j = j**i
    print(j)
```

0.0s

```
5.0
25.0
15625.0
5.960464477539062e+16
7.52316384526264e+83
```

```
OverflowError                               Traceback (most recent call last)
Cell In[12], line 3
      1 j = 5.0
      2 for i in range(1, 100):
----> 3     j = j**i
      4     print(j)

OverflowError: (34, 'Result too large')
```

There are more…..

Built-In Exceptions in Python

Exception	Description
ArithmeticError	Raised when an error occurs in numeric calculations
AssertionError	Raised when an assert statement fails
AttributeError	Raised when attribute reference or assignment fails
Exception	Base class for all exceptions
EOFError	Raised when the input() method hits an "end of file" condition (EOF)
FloatingPointError	Raised when a floating point calculation fails
GeneratorExit	Raised when a generator is closed (with the close() method)
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indentation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete

Built-In Exceptions in Python (Contd.)

MemoryError	Raised when a program runs out of memory
NameError	Raised when a variable does not exist
NotImplementedError	Raised when an abstract method requires an inherited class to override the
OSError	Raised when a system related operation causes an error
OverflowError	Raised when the result of a numeric calculation is too large
ReferenceError	Raised when a weak reference object does not exist
RuntimeError	Raised when an error occurs that do not belong to any specific exceptions
StopIteration	Raised when the next() method of an iterator has no further values
SyntaxError	Raised when a syntax error occurs
TabError	Raised when indentation consists of tabs or spaces
SystemError	Raised when a system error occurs
SystemExit	Raised when the sys.exit() function is called
TypeError	Raised when two different types are combined

Built-In Exceptions in Python (Contd.)

UnboundLocalError	Raised when a local variable is referenced before assignment
UnicodeError	Raised when a unicode problem occurs
UnicodeEncodeError	Raised when a unicode encoding problem occurs
UnicodeDecodeError	Raised when a unicode decoding problem occurs
UnicodeTranslateError	Raised when a unicode translation problem occurs
ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero

User Defined Exceptions

- **What are User-defined exceptions?**

  User-defined exceptions are exceptions made by a user to address particular scenarios in their code. They are derived from the Exception class or one of its subclasses.

- **Need for User-defined exceptions?**

  - Unique exceptions that are suited to the particular requirements of a project.
  - Custom exceptions give programmers the ability to deliver thorough error messages that can aid in the runtime diagnosis of problems and facilitate their resolution.
  - They offer a way to arrange and encapsulate error handling logic, custom exceptions can improve the readability and maintainability of code.

User Defined Exceptions Syntax

```
class CustomError(Exception):
    pass

raise CustomError("Error Message!")
```

[1]

```
CustomError                                 Traceback (most recent call last)
Cell In[1], line 4
      1 class CustomError(Exception):
      2     pass
----> 4 raise CustomError("Error Message!")

CustomError: Error Message!
```

Example

```python
class NotEvenAnError(Exception):
    def __init__(self, code) -> None:
        self.err_code = code

    def __str__(self) -> str:
        return repr(f"Error Code: {self.err_code}")

if (True):
    raise NotEvenAnError(3+4)
```

[6] ⊗ 0.0s

```
NotEvenAnError                            Traceback (most recent call last)
Cell In[6], line 9
      6         return repr(f"Error Code: {self.err_code}")
      8 if (True):
----> 9     raise NotEvenAnError(3+4)

NotEvenAnError: 'Error Code: 7'
```

Exception Handling Keywords are not the ultimate way to handle error !

Managing Edge Cases without Exception Handling
Example : Calculator Program

➔ Welcome screen and operation choice

```
print("!*** WELCOME TO BASIC CALCULATOR ***!")
print("Where you can add as well as subtract")
while True:
    operation = input("Choose operation (add/subtract): ").lower()
    if operation not in ['add', 'subtract']:
        print("Invalid operation. Please enter 'add' or 'subtract'.")
        continue  # Restart the loop
```

➔ Getting and Validating Input operands

```
# Get user input for the numbers
num1 = input("Enter first number: ")
num2 = input("Enter second number: ")

# Validate input as numbers
if not num1.isdigit() or not num2.isdigit():
    print("Invalid input. Please enter valid numbers.")
    continue  # Restart the loop
```

➔ Performing operations based on choice

```
# Perform the selected operation
if operation == 'add':
    result = float(num1) + float(num2)
    print("the sum is " + str(result))
elif operation == 'subtract':
    result = float(num1) - float(num2)
    print("the subtract is " + str(result))
```

➔ Continue or Exit based on input

```
# Ask the user if they want to continue
user_choice = input("Continue? (yes/no): ").lower()
# Check if the user wants to continue
if user_choice ≠ 'yes':
    print("Exiting the calculator.")
    break
```

OUTPUT - 1

```
!*** WELCOME TO BASIC CALCULATOR ***!
Where you can add as well as subtract
Choose operation (add/subtract): add
Enter first number: 12
Enter second number: 13
the sum is 25.0
Continue? (yes/no): yes
```

OUTPUT - 2

- Choose operation (add/subtract): subtract
- Enter first number: 100
- Enter second number: 50
- the subtract is 50.0
- Continue? (yes/no): no
  -
- Exiting the calculator.

Keywords related to Exception Handling

TRY -EXCEPT

- The try block lets you test a block of code for errors.

- The except block lets you handle the error.

CODE 1 OF TRY - EXCEPT

```
# A function that prints inverse
def inverse(num) → None:
    x = 1/num
    print("The result is: {}".format(x))
```

[4] ✓ 0.0s

NO EXCEPTION OCCURS

```
try:
    inverse(5)
except ZeroDivisionError as err:
    print("Handling run-time error: ", err)
```

[5] ✓ 0.0s

```
The result is: 0.2
```

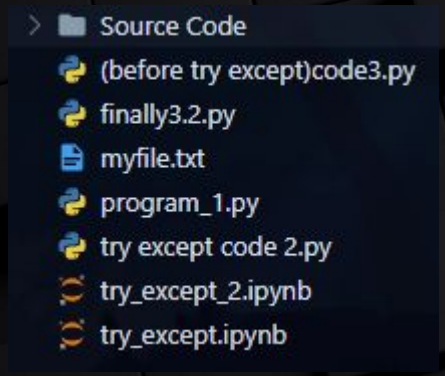
EXCEPTION OCCURS

```
try:
    inverse(0)
except ZeroDivisionError as err:
    print("Handling run-time error: ", err)
```

[3] ✓ 0.0s

```
Handling run-time error:  division by zero
```

CODE 2 OF TRY - EXCEPT Case I



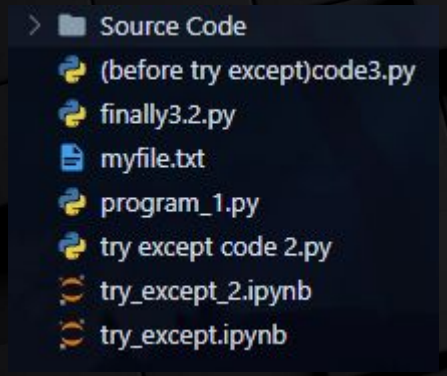
Folder Contents

```
try:
    file = open('myfile.txt')
    print('File exists')
    print(file.readline())

except FileNotFoundError:
    print('File does not exist')

except Exception:
    print('An error occurred')
```

[1] ✓ 0.0s

```
File exists
Hello Everyone, I am a File
```

CODE 2 OF TRY - EXCEPT Case II



Folder Contents

```
try:
    file = open('yourfile.txt')
    print('File exists')
    print(file.readline())

except FileNotFoundError:
    print('File does not exist')

except Exception:
    print('An error occurred')
```

[2] ✓ 0.0s

```
File does not exist
```

Finally

The finally block lets you execute code, regardless of the result of the try- and except blocks.

CODE OF Finally

```
def foo(index_value) → int:
    try:
        list1 = ['Maths',
                 'Science',
                 'Stats',
                 'Operational Research']
        print(list1[index_value])
        return 1
    except:
        print("Invalid Input")
        return 0

    finally:
        print("this part will be executed always")

myFavSubject = foo(0)
print("My favourite subject is {}".format(myFavSubject))
```

OUTPUT

- Maths
- this part will be executed always
- My favourite subject is 1

Try with else clause

```
def group(a, b) → None:
    try:
        c = ((a + b) / (a - b))
    except ZeroDivisionError:
        print("This results in infinity")
    else:
        print(c)
    finally:
        print("this part is always executed")
```

[2] ✓ 0.0s

Case I

```
group(2.0, 3.0)
```

[3] ✓ 0.0s

```
-5.0

this part is always executed
```

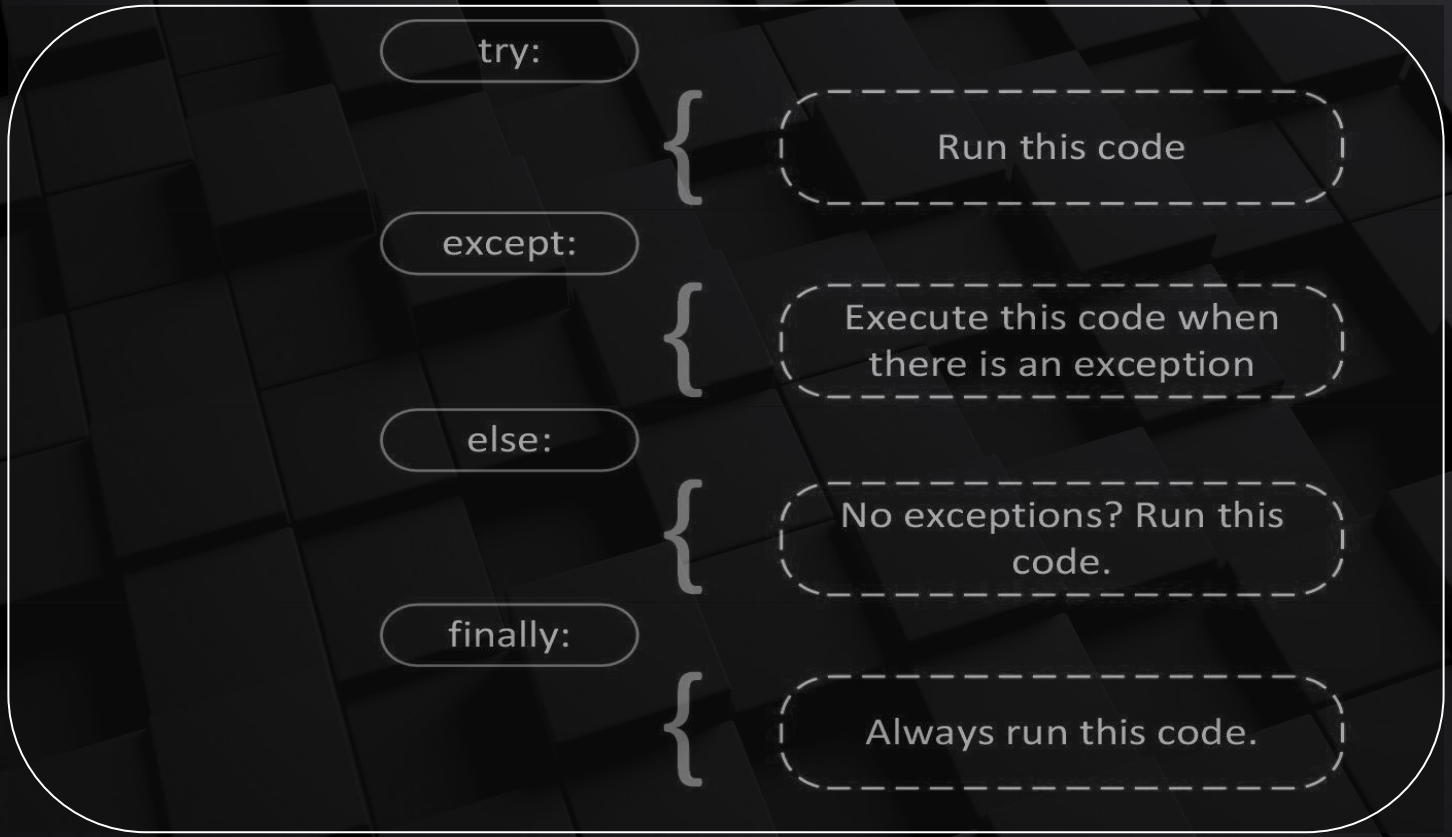
Case II

```
group(3.0, 3.0)
```

[4] ✓ 0.0s

```
This results in infinity

this part is always executed
```

Exception Handling in One Picture



Program to depict the usage of Exception Handling

A simple and short example, for easy and faster explanation

Example Program : Checking Armstrong Number

- ➔ This is one of the programs we created as practicals 😎

- ➔ The example presented here depicts a program that simply uses keywords
  - ◆ try , except, raise, else, finally

Here we need to check whether the number entered by the User is an Armstrong number or not.

We did this program without using these keywords, we simply used if-else with warnings where some exceptions could have occurred

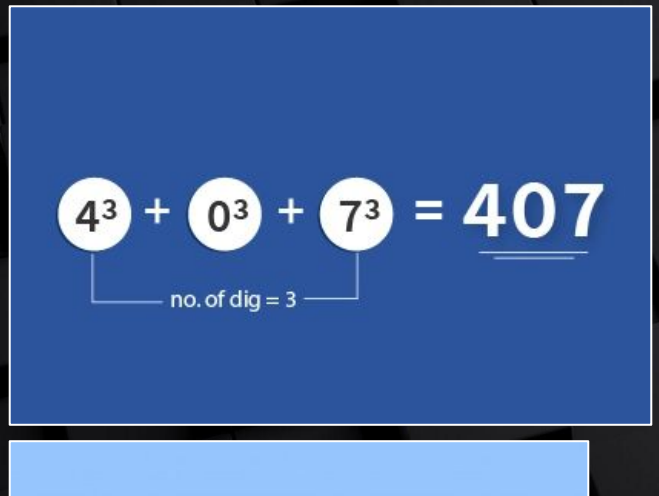
Now lets see how we can use the exception handling feature to make it user friendly

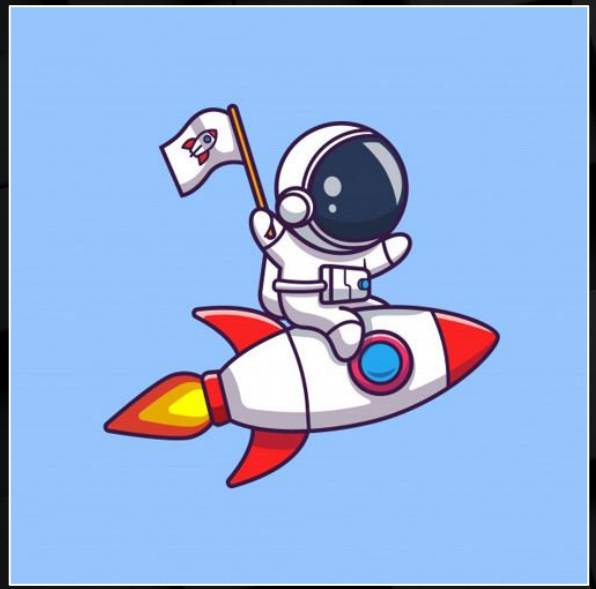
A Quick Recap

- What are Armstrong Numbers?
- Why Named So?
- Also Known as
- Real Life Applications
- Example

$4^3 + 0^3 + 7^3 = 407$

no. of dig = 3





Example 1

a) Our Own Class

```
class NotArmstrongError(Exception):

    def __init__(self, value) → None:
        self.value = value

    def __str__(self) → str:
        return(repr(self.value))
```

b) try block

```
try:
    # number entered by the user
    number = int(input("Enter your number to check! \n"))

    # number of digits
    n = len(str(number))

    sum = 0
    for i in str(number):
        sum += int(i)**n

    if sum != number:
        raise NotArmstrongError(number)
```

c) except block(s)

```
except ValueError as val_err:
    print("Please enter a number!")
    print(val_err)
    continue
except NotArmstrongError as arm_err:
    print('''Unfortunately,
          It is not an Armstrong Number''')
    print("Your number was", arm_err)
```

d) else and finally block

```
else:
    print("Congratulations")
    print(f"{number} is an Armstrong Number")
finally:
    print("Check Complete!")
```

Program 1 Output - Case I

```
➢ Enter your number to check!
➢ 370
➢ Congratulations
➢ 370 is an Armstrong Number
➢ Check Complete!

➢ Do you want to test another number? Enter 1
  for Yes, and 0 for No

➢ 1
```

Program 1 Output - Case II

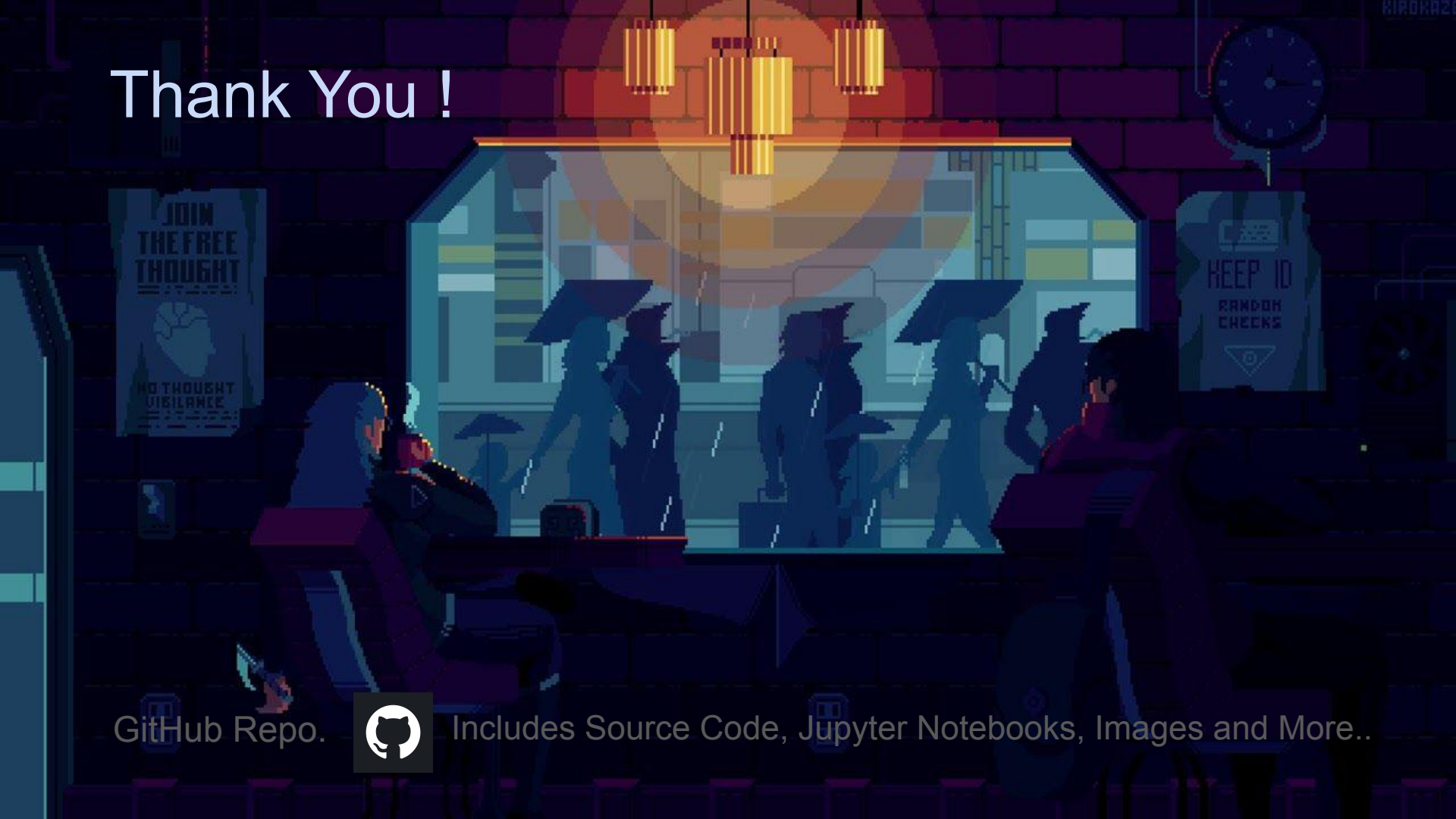
```
➢ Enter your number to check!
➢ 360
➢ Unfortunately,
  It is not an Armstrong Number

➢ Your number was 360
➢ Check Complete!

➢ Do you want to test another number? Enter 1
  for Yes, and 0 for No

➢ 1
```

Program 1 Output - Case III

```
Enter your number to check!
abcd
Please enter a number!
invalid literal for int() with base 10: 'abcd'
Check Complete!
Enter your number to check!
67
Unfortunately,
                It is not an Armstrong Number
Your number was 67
Check Complete!
Do you want to test another number? Enter 1 for Yes, and 0
for No
0
Thank you and Good Bye!
```


The
End

Thank You !



GitHub Repo. Includes Source Code, Jupyter Notebooks, Images and More..