



Exception

Week 9

Difference between Syntax error and Exception

- Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. **It leads to the termination of the program.**

```
# initialize the amount variable
amount = 10000

# check that You are eligible to
# purchase Dsa Self Paced or not
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

```
File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
        ^
SyntaxError: invalid syntax
```

Exception

- Exceptions are raised when the program is syntactically correct, but the code resulted in an error.
- This error does not stop the execution of the program, however, it changes the normal flow of the program.
- In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

```
# initialize the amount variable
marks = 10000

# perform division with 0
a = marks / 0
print(a)
```

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

Exception Hierarchy

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        +-- BufferError
        +-- ArithmeticError
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- EnvironmentError
            +-- IOError
            +-- OSError
                +-- WindowsError (Windows)
                +-- VMSError (VMS)
        +-- EOFError
        +-- ImportError
        +-- LookupError
            +-- IndexError
            +-- KeyError
        +-- MemoryError
        +-- NameError
            +-- UnboundLocalError
        +-- ReferenceError
        +-- RuntimeError
            +-- NotImplementedError
        +-- SyntaxError
            +-- IndentationError
                +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
            +-- UnicodeError
                +-- UnicodeDecodeError
                +-- UnicodeEncodeError
                +-- UnicodeTranslateError
    +-- Warning
        +-- DeprecationWarning
        +-- PendingDeprecationWarning
        +-- RuntimeWarning
        +-- SyntaxWarning
        +-- UserWarning
        +-- FutureWarning
        +-- ImportWarning
        +-- UnicodeWarning
        +-- BytesWarning
```

Try and Except Statement – Catching Exceptions

- Try and except statements are used to catch and handle exceptions in Python.
- Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.



Example

- Let us try to access the array element whose index is out of bound and handle the corresponding exception.
- In the example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

```
# Python program to handle simple runtime error
#Python 3

a = [1, 2, 3]
try:
    print ("Second element = %d" %(a[1]))

    # Throws error since there are only 3 elements in array
    print ("Fourth element = %d" %(a[3]))

except:
    print ("An error occurred")
```

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

The Base Exception Class

- The root class for all built-in exceptions. It is represented by the class `BaseException`.
- All other built-in exceptions in Python inherit from this base class.
- Examples of exceptions that inherit from `BaseException` include ``SystemExit``, ``KeyboardInterrupt``, and ``GeneratorExit``.
- It's not typically recommended to directly catch or handle exceptions derived from `BaseException` unless you have a very specific reason to do so.

The Exception class

- `Exception` is the base class for most exceptions in Python.
- It inherits from `BaseException`, so it is a **subclass** of `BaseException`.
- Most user-defined exceptions and built-in exceptions that you commonly encounter, such as `ValueError`, `TypeError`, `IOError`, etc., inherit from `Exception`.
- It's generally more common to catch or handle exceptions derived from `Exception` since these are the exceptions that represent common error conditions that a Python program may encounter during execution.

To easily differentiate Base Exception and Exception classes.

- Base Exception holds exceptions that are usually or most of the time not handled, like ``SystemExit``, ``KeyboardInterrupt``, and ``GeneratorExit``. We do these stuff at command and not due to some logic problem.
- Exception holds the most common exceptions that should be handled like ``ValueError``, ``TypeError``, ``IOError``, which are usually caused by faulty logic.

Catching Specific Exception

- A try statement can have more than one except clause, to specify handlers for different exceptions.
- Note that at most one handler will be executed. For example, we can add `IndexError` in the code. The general syntax for adding specific exceptions are –

```
try:  
    # statement(s)  
except IndexError:  
    # statement(s)  
except ValueError:  
    # statement(s)
```

Example

```
# Program to handle multiple errors with one
# except statement
# Python 3

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")
```

ZeroDivisionError Occurred and Handled

If you comment on the line fun(3), the output will be

NameError Occurred and Handled

The output above is so because as soon as python tries to access the value of b, NameError occurs.

Try with Else Clause

- In python, you can also use the else clause on the try-except block which must be present after all the except clauses.
- The code enters the else block only if the try clause does not raise an exception.

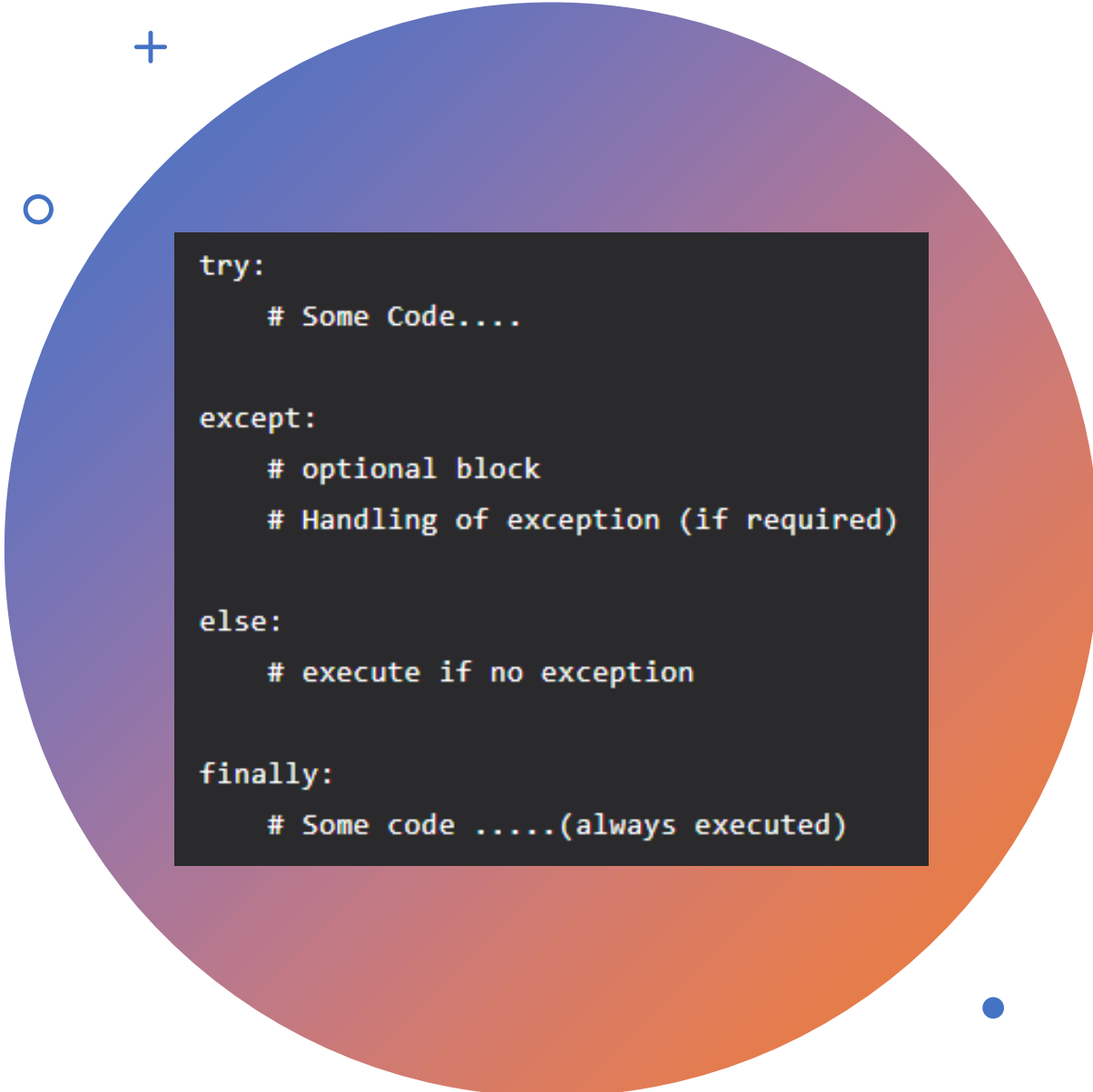
```
# Program to depict else clause with try-except
# Python 3
# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)

# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

-5.0

a/b result in 0

The “Finally”, Keyword in Python



```
try:
    # Some Code....

except:
    # optional block
    # Handling of exception (if required)

else:
    # execute if no exception

finally:
    # Some code .....(always executed)
```

- Python provides a keyword finally, which is always executed after the try and except blocks.
- The final block always executes after normal termination of try block or after try block terminates due to some exception.

Example

```
# Python program to demonstrate finally
# No exception Exception raised in try block
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')
```

Can't divide by zero
This is always executed

Raising Exception

- The raise statement allows the programmer to force a specific exception to occur.
- The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).
- The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

Disadvantages of Exception Handling

- Performance overhead: Exception handling can be slower than using conditional statements to check for errors, as the interpreter must perform additional work to catch and handle the exception.
- Increased code complexity: Exception handling can make your code more complex, especially if you must handle multiple types of exceptions or implement complex error handling logic.
- Possible security risks: Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it's important to handle exceptions carefully and avoid exposing too much information about your program.
- Overall, the benefits of exception handling in Python outweigh the drawbacks, but it's important to use it judiciously and carefully in order to maintain code quality and program reliability.

Why raise your own exception?

- There are times where an exception is not considered as an exception by Python.
- Creating or raising your own exception can help with your logic.
- It can also assist in finding certain bugs relating to logic errors, which you can force certain things to guide you find them.
- It can improve the robustness and effectivity of your code in handling errors which Python does not natively handle.
- Some mistakes that are unavoidable can be raised an error for the users to know that they should correct themselves.

Raising a Custom Exception example.

```
# Program to depict Raising Exception

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

```
Traceback (most recent call last):
  File "/home/d6ec14ca595b97b9ff8d8034bbf212a9f.py", line 5, in <module>
    raise NameError("Hi there")  # Raise Error
NameError: Hi there
```

Seeing the benefits and usage of Raising your own Exception

Raising your own exceptions can be beneficial in scenarios where you want to handle specific error conditions in your code or enforce certain constraints.

Let's consider an example where you have a function that calculates the area of a rectangle, but you want to ensure that both the length and width of the rectangle are positive numbers. In this case, it makes sense to raise a custom exception if invalid input is provided. Here's an example:

```
1 def calculate_rectangle_area(length, width):
2     """Calculate the area of a rectangle."""
3     if length <= 0 or width <= 0:
4         raise ValueError("Rectangle dimensions must be positive
5                               numbers.")
6     return length * width
7
8 # Example usage:
9 try:
10     length = float(input("Enter the length of the rectangle: "))
11     width = float(input("Enter the width of the rectangle: "))
12     area = calculate_rectangle_area(length, width)
13     print("Area of the rectangle:", area)
14 except ValueError as e:
15     print("Error:", e)
```

```
Enter the length of the rectangle: -3
Enter the width of the rectangle: -4
ERROR!
Error: Rectangle dimensions must be positive numbers.
```

Its ok not to raise your own, but its better to do so!

We directly use the built-in ValueError exception instead of creating a custom one.

The calculate_rectangle_area function raises a ValueError if either the length or width is not a positive number.

When using the calculate_rectangle_area function, if an invalid rectangle is provided, a ValueError is raised with a descriptive error message.

The code that calls calculate_rectangle_area catches and handles the ValueError exception separately, providing a clear indication of what went wrong and why.

This simplified version achieves the same purpose as the previous example but without the use of a custom exception class.

```
1 def calculate_rectangle_area(length, width):
2     """Calculate the area of a rectangle."""
3     if length <= 0 or width <= 0:
4         return None # Return None if dimensions are invalid
5     return length * width
6
7
8 # Example usage:
9 length = float(input("Enter the length of the rectangle: "))
10 width = float(input("Enter the width of the rectangle: "))
11 area = calculate_rectangle_area(length, width)
12
13 if area is not None:
14     print("Area of the rectangle:", area)
15 else:
16     print("Invalid input. Rectangle dimensions must be positive numbers.")
```

```
Enter the length of the rectangle: -3
Enter the width of the rectangle: -4
Invalid input. Rectangle dimensions must be positive numbers.
```

Seeing the difference with raising and not raising

```
1 def calculate_rectangle_area(length, width):
2     """Calculate the area of a rectangle."""
3     if length <= 0 or width <= 0:
4         raise ValueError("Rectangle dimensions must be positive
5             numbers.")
6     return length * width
7
8 # Example usage:
9 try:
10     length = float(input("Enter the length of the rectangle: "))
11     width = float(input("Enter the width of the rectangle: "))
12     area = calculate_rectangle_area(length, width)
13     print("Area of the rectangle:", area)
14 except ValueError as e:
15     print("Error:", e)
```

```
Enter the length of the rectangle: -3
Enter the width of the rectangle: -4
ERROR!
Error: Rectangle dimensions must be positive numbers.
```

```
1 def calculate_rectangle_area(length, width):
2     """Calculate the area of a rectangle."""
3     if length <= 0 or width <= 0:
4         return None # Return None if dimensions are invalid
5     return length * width
6
7
8 # Example usage:
9 length = float(input("Enter the length of the rectangle: "))
10 width = float(input("Enter the width of the rectangle: "))
11 area = calculate_rectangle_area(length, width)
12
13 if area is not None:
14     print("Area of the rectangle:", area)
15 else:
16     print("Invalid input. Rectangle dimensions must be positive
17         numbers.")
```

```
Enter the length of the rectangle: -3
Enter the width of the rectangle: -4
Invalid input. Rectangle dimensions must be positive numbers.
```

Capturing the exception instance

`except Exception` is a generic way to catch any exception that is derived from the base `Exception` class in Python.

When you use `except Exception`, you're telling Python to catch any exception that occurs, regardless of its type.

This can be useful for handling unexpected errors or for providing a general error-handling mechanism.

However, it's important to be cautious when using such a broad exception handler because it might catch exceptions that you didn't anticipate, potentially hiding bugs in your code.

```
1 try:
2     # Code that may raise an exception
3     result = 10 / 0
4 except Exception:
5     print("An error occurred")
```

Capturing the exception is a difference maker!

Being able to capture the instance of the exception makes it easier to understand what the exception is!

This prevents the cases of just raising an exception and prompting whatever is to be prompted. It does not give any details of what was captured.

Not being able to determine what was captured can hide bugs in your code.

Explicitly knowing the exception would be of great help!

This approach also not require you to make exceptions for all possible exceptions!

```
1 try:
2     # Code that may raise an exception
3     result = 10 / 0
4 except Exception as e:
5     print(f"An error occurred: {e}")
```

An error occurred: division by zero

Capturing the exception instance is beauty!

As show in the image, the exception e captures the error and mentions what it is without explicitly identifying it.

Also, no problem is shown, as the finally and else works respectively on how they should work.

```
1 list = [1, 2, 3]
2
3 try:
4     # Code that may raise an exception
5     print(list[3])
6 except Exception as e:
7     print(f"An error occurred: {e}")
8
9 else:
10    print("I will not be printed :(")
11
12 finally:
13    print("\nI always get printed ;)")
14    print("Your code will continue to work. I don't
        care about errors :P")
```

```
An error occurred: list index out of range
```

```
I always get printed ;)
```

```
Your code will continue to work. I don't care about errors
:P
```


The captured implicitly defined error can be used for your logic!

```
1 try:
2     # Some code that may raise an exception
3     result = 10 / 0
4 except ZeroDivisionError as e:
5     # Capturing the exception and using its information
6     print("An error occurred:", e)
7     print("Exception type:", type(e).__name__)
8     print("Exception message:", e.args[0])
9     # You can use the exception's information for your code logic
10    if "division by zero" in str(e):
11        print("Handling division by zero...")
12        # Your code logic here to handle the division by zero
13        case
14    else:
15        print("Handling other types of errors...")
16        # Your code logic here to handle other types of errors
17 else:
18     # This block executes if no exception occurred
19     print("No exception occurred")
```

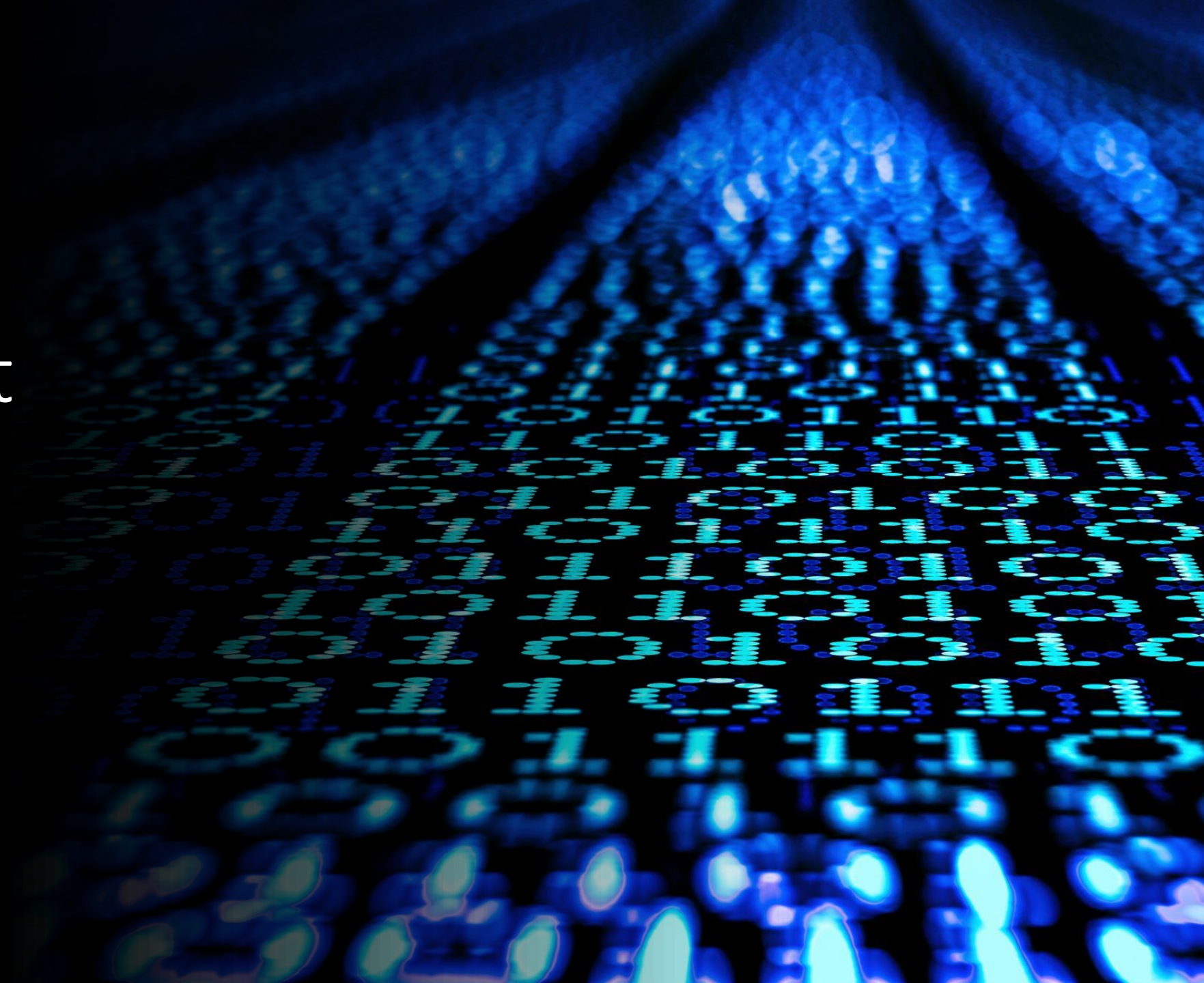
```
An error occurred: division by zero
Exception type: ZeroDivisionError
Exception message: division by zero
Handling division by zero...
```

Summary

```
1 ▾ try:
2     # You attempt your operation here.
3 ▾ except ExceptionI:
4     # If an ExceptionI is raised, execute this block.
5 ▾ except ExceptionII:
6     # If ExceptionII is raised, execute this block.
7 ▾ else:
8     # If no excetpion, execute this block.
9 ▾ finally:
10    # Execute this block with or without an exception.
```



Non-Zero Exit Code (NZEC) Errors



What is NZEC error?

- While coding in various competitive sites, many people must have encountered NZEC errors.
- NZEC (non zero exit code) as the name suggests occurs when your code is failed to return 0.
- When a code returns 0 it means it is successfully executed otherwise it will return some other number depending on the type of error.
- When the program ends and it is supposed to return “0” to indicate if finished fine and is not able to do so it causes NZEC.
- Of course, there are more cases associated with NZEC.

Why does NZEC occur?

- In python, generally, multiple inputs are separated by commas and we read them using `input()` or `int(input())`, but most of the online coding platforms while testing give input separated by space and in those cases, `int(input())` is not able to read the input properly and shows error like NZEC.

```
1  nums = list(map(int, input().split()))
```

NZEC is a Runtime Error!

- NZEC error is a runtime error and occurs mostly when negative array index is accessed or the program which we have written is utilizing more memory space than the allocated memory for our program to run.

NZEC example

```
x,y = map(int, input())
```

- Doing it the wrong way!
- The program will prompt you to enter some input.
- You need to input exactly two values separated by whitespace (such as space, tab, or newline).
- The `map(int, ...)` function will convert each character of the input into an integer.
- However, if you input more than two characters or less than two characters, you'll get an error. This is because `map(int, ...)` is trying to unpack the input into exactly two integers, but it won't work if the input doesn't have exactly two elements to unpack.

Why is it wrong?

- For example, if you input "12", it will give you a ValueError because it's trying to unpack a single value into two variables.
- Instead, you could input something like "1 2", where it will assign x the value 1 and y the value 2.

```
# input().split() will split the input string into a
# list of substrings using whitespace as the
# delimiter.
# map(int, ...) will then convert each substring to
# an integer.
# We use try-except blocks to handle potential errors
# :
# ValueError: Raised when the input cannot be
# converted to integers, such as when there's only
# one value or non-integer characters are present.
# EOFError: Raised when there's no input provided at
# all.
```



```
x,y = map(int, input().split())
```

The correct way

to delimit input by white spaces:

Input: "12" (a single string without space)

Outcome: This input will cause a `ValueError` because `map(int, input().split())` expects two values separated by space, but it only receives one value. The code will raise an exception, leading to an NZEC error.

To avoid the NZEC error, provide input in the correct format, such as "1 2", where there are two integers separated by space.

Possible reason of getting NZEC error:

Possible reason of getting NZEC error:

- Infinite Recursion – or if you are run out of stack memory.
- Make sure your input and output both are exactly same as the test cases. It is advisable to test your program using a computer code which matches your output with the specified outputs exactly.
- Another common reason of getting this error is when you make basic programming mistakes like dividing by 0.
- Check for the values of your variables, they can be vulnerable to integer flow.
- Directly trying to compute factorial above 20, if you are – find another way to do that.

```
1 try:
2     x, y = map(int, input().split())
3     print("x:", x)
4     print("y:", y)
5 except ValueError:
6     print("Input should contain two integers separated
7         by space.")
8 except EOFError:
9     print("No input provided.")
```

NZEC Example

- A simple program where you have to read 2 integers and print them(in the input file both integers are in same line). Suppose you have two integers as shown below: 23 45 Instead of using :

```
n = int(input())  
k = int(input())
```

```
n, k = raw_input().split(" ")  
n = int(n)  
k = int(k)
```

- **Incorrect code**
- When you run the above code in IDE with the above input you will get error:-

```
n = int(input())  
k = int(input())  
print(n, " ", k)
```

```
Traceback (most recent call last):  
  File "b712edd81d4a972de2a9189fac8a83ed.py", line 1, in  
    n = int(input())  
File "", line 1  
  2 3  
   ^  
SyntaxError: unexpected EOF while parsing
```

Some prominent reasons for NZEC error

- Infinite Recursion or if you have run out of stack memory.
- Input and output are NOT exactly same as the test cases.
- As the online platforms, test your program using a computer code which matches your output with the specified outputs exactly.
- This type of error is also shown when your program is performing basic programming mistakes like dividing by 0.
- Check for the values of your variables, they can be vulnerable to integer flow.
- There could be some other reasons also for the occurrence NZEC error.

NZEC Summary

- Program Exit Codes: When a program runs, it usually exits with a status code. This status code helps indicate whether the program executed successfully or encountered an error.
- Zero vs. Non-Zero Exit Code: Conventionally, a status code of zero indicates successful execution, while any non-zero code suggests an error or abnormal termination.
- Reasons for NZEC: NZEC typically happens due to runtime errors in the code. This could be anything from an unhandled exception causing the program to crash, to not handling input/output correctly according to the platform's requirements.
- Common Causes:
 - Accessing elements beyond the bounds of an array.
 - Division by zero.
 - Trying to read input when there's no input available.
 - Incorrect handling of input/output formats.
 - Running out of memory or exceeding time limits (though these might not always result in NZEC specifically).
- Debugging NZEC: To debug NZEC errors, it's essential to carefully examine your code, paying close attention to how it handles inputs and outputs, as well as potential boundary cases that might cause runtime errors.