Department of Computer Science and Engineering (UG Studies)

PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Exception Handling in Python

Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. **Errors detected during execution are called *exceptions*** and are not unconditionally fatal: you will soon learn how to handle them in Python programs.

Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Note: Standard exception names are built-in identifiers (not reserved keywords).

A Python program terminates as soon as it encounters an error. **In Python, an error can be a syntax error or an exception.**

Exception: Exceptional story

A program executes some piece of code normally. Sometimes it may take an abnormal path due to runtime errors.

An exception does not always mean it is an error. If we walk through some iterable using the for loop, we should come to an end of the iterable at some time. Then python signals it as *StopIteration*.

Sometimes exception could also be an error. The index specified to get an element of the list may be beyond the list boundary. Then python indicates this as index error.

When an exception occurs, the program will be terminated abnormally. Can we avoid this abnormal termination? Can we get a chance to proceed further? Can we have graceful degradation?
This concept is called exception handling.

A few components and terminologies used with exception handling:

1. try:

The try block lets you test a block of code for errors.
We indicate to the Python runtime that something unusual could happen in the suite of code.

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

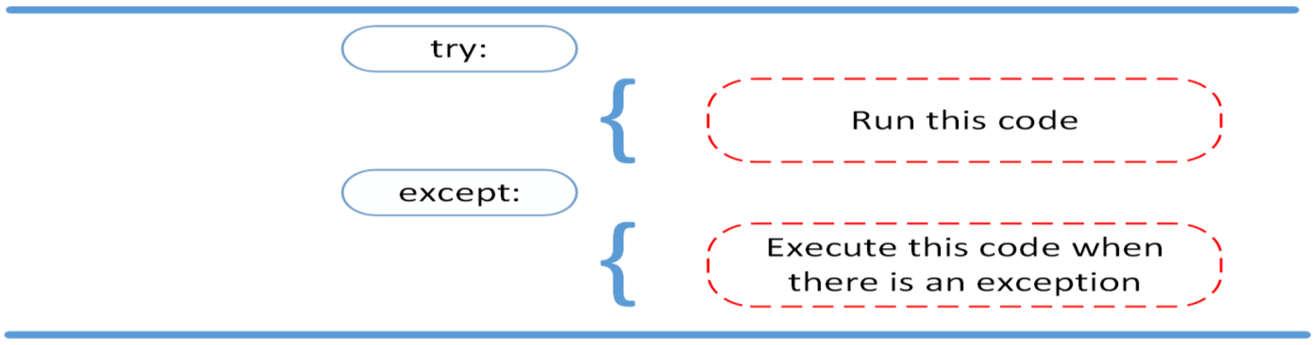
2. except [<type>]:

The except block lets you handle the error.

We should have at least one except block to which the control shall be transferred if and only if something unusual happens in the try suite.

So try block may be followed by one or more except block – all but one of them specifying the name of the exception – one of them may provide a common and default way of handling all exceptions.

The **try** and **except** block in Python is used to catch and handle exceptions. **Python executes code following the try statement as a "normal" part of the program. The code that follows the except statement is the program's response to any exceptions in the preceding try block.**



As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The except block determines how your program responds to exceptions.

Here's an example where you open a file and use a built-in exception:

```
try:
    with open('file1.txt') as file:
        read_data = file.read()

except:       # default except block with no exception specified
    print('Could not open file1.txt')
```

If file1.txt does not exist, this block of code will output the following:

```
Could not open file1.txt
```

This is an informative message, and our program will still continue to run. In the Python docs, you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

Exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. Corresponds to errno ENOENT.
To catch this type of exception and print it to screen, you could use the following code:

```
try:
    with open('file1.txt') as file:
        read_data = file.read()

except FileNotFoundError as fnf_error:
    print(fnf_error)
```

In this case, if *file1.txt* does not exist, the output will be the following:

```
[Errno 2] No such file or directory: 'file1.txt'
```

You can have more than one function call in your try clause and anticipate catching various exceptions. A thing to note here is that the code in the try clause will stop as soon as an exception is encountered.

Warning: Catching Exception hides all errors…even those which are completely unexpected. This is why you should avoid bare except clauses in your Python programs. Instead, you'll want to refer to *specific exception classes* you want to catch and handle.

3. builtin exceptions:

There are a number of exceptions which Python knows. We call them built-in exceptions.
Examples:

IndexError Raised when an index is not found in a sequence.
KeyError Raised when the specified key is not found in the dictionary.
NameError Raised when an identifier is not found in the local or global namespace.

These are automatically raised when they happen.

4. raising an exception:

So far, you have only been catching exceptions that are thrown by the Python run-time system. However, it is possible for your program to throw an exception explicitly, using the **raise** statement. The general form of **raise** is shown here:

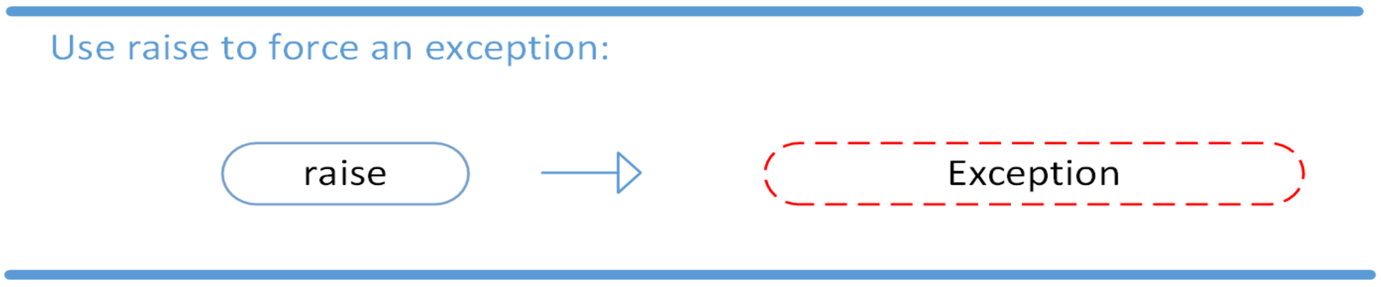
```
raise Exception_name(<message>)
```

Here, **Exception_name** must be a subclass of **Exception.**

This causes creation of an exception object with the message. The object also remembers the place in the code (line number, function name, how this function got called on). The raising of exception causes the program to abort if the raise statement is not in a try block or transfer the control to the end of try block to match the except blocks.

Raising an Exception

We can use raise to throw an exception if a condition occurs. The statement can be complemented with a custom exception.



If you want to throw an error when a certain condition occurs using raise, you could go about it like this:

```
x = 10
if x > 5:
    raise Exception(f'x should not exceed 5. The value of x was: {x}')
```

When you run this code, the output will be the following:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
Exception: x should not exceed 5. The value of x was: 10
```

The program comes to a halt and displays our exception to screen, offering clues about what went wrong.

5. matching of except blocks:

- **The raised or thrown exception object is matched with the except blocks in the order in which they occur in the try-except statement.**
- **The code following the first match is executed**. **It is always the first match and not the best match**.
- **If no match occurs and there is a default except block (with no exception specified), then that block will be executed**.

Example 1:

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). **For example, the following code will print B, C, D in that order:**

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that **the following code will print B, B, B** — the first matching except clause is triggered.

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except B:
        print("B")
    except C:
        print("C")
    except D:
        print("D")
```

Example 2:

In the below code, the last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())

except OSError as err:
    print("OS error: {0}".format(err))

except ValueError:
    print("Could not convert data to an integer.")

except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

6. finally block:

This is optional. This follows all the except blocks. It is part of the try statement. This block shall be executed on both normal flow and exceptional flow.

Let us understand the flow of execution.

1. **Normal flow**

   `try block – finally block if any – code following try block`

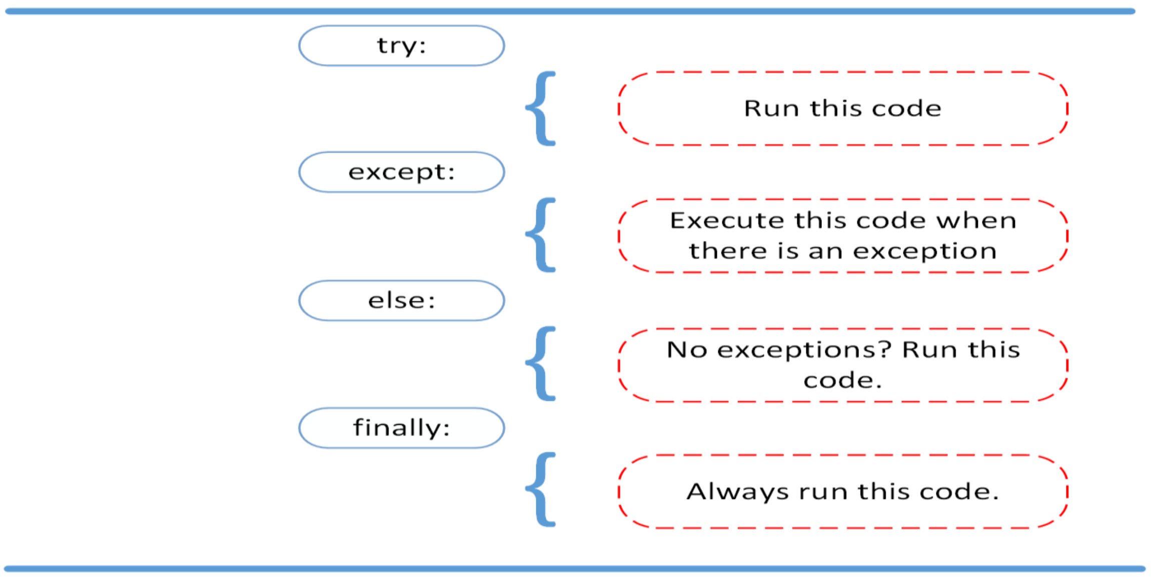
2. **Exceptional flow**

   `try block – exit the try block on an exception – find the first matching except block – execute the matched except block – finally block – code following try block.`

Observe there is no mechanism in any language including Python to go back to the try block – no way to resume at the point of exception.

Cleaning up after using by finally block

Imagine that you always had to implement some sort of action to clean up after executing your code.
Python enables you to do so using the finally clause.



try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

Have a look at the following example:

```
import sys

def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')

try:
    linux_interaction()  # this function can only run on Linux systems
except AssertionError as error:
    print(error)
else:
    try:
        with open('file1.txt') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

In the previous code, everything in the finally clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses. Running the previous code on a Windows machine would output the following:

```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

7. user defined exception:

It is not possible for a language to specify all possible unusual cases. So the users can also specify exception as a class which inherits from a class called **Exception**.

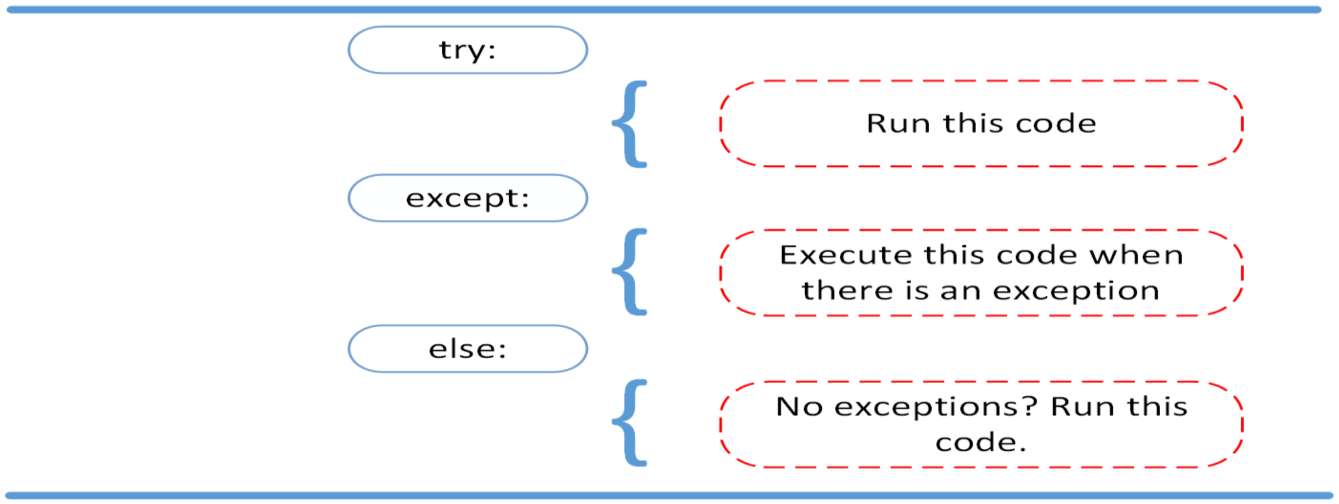
Often you'll need to throw something other than one of the existing exceptions; they aren't always the best choice. Let's take a look at creating your own types of exceptions. You do this by subclassing one of the existing Exception subclasses.

The following code is from the file 4_exception.py.
It shows how to make our own exception object.

```
# user defned exception
class MyException(Exception):
      def __init__(self, str):
            self.str = str
      def __str__(self):
            return self.str

# check whether n is between 1 and 100
n = int(input("enter a number:"))
try:
      if not 1 <= n <= 100 :
            raise MyException("number not in range")
      print("number is fine : ", n)

except MyException as e:
      print(e)

print("thats all")
```

8. the else Clause

In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.

Look at the following example:

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    print('Executing the else clause.')
```

9. exception propagation:

We say that a try block has dynamic scope. Any exception raised in the block or any function called from there are considered part of the try block. Any exception raised within these functions called from there will cause the control to be propagated backwards to the except blocks of this try block.
Let us go through a few examples.

The following code is taken from the file 2_exception_intro.py.

You will observe that the program aborts as soon as any one of these statements is executed. You get know a few builtin exceptions in this program.

```
# example 1
# print("res : ", 10 / 0)  # ZeroDivisionError: division by zero

# example 2
# print(myvar)            # NameError: name 'myvar' is not defned

# example 3
# open("unknown.txt")  #FileNotFoundError: [Errno 2] No such fle or directory:'unknown.txt'
```

Let us observe some code from the file 2_exception.py

```
m = 10
#n = 2
n = 0
try:
      print("one")
      print("res : ", m / n)
      print("two")

except Exception as e:
      print(e)

print("three")
```

In the above code, *print("two")* statement is executed only if the earliest statement does not raise an exception.

On an exception, object e of the class `DivisionByException` is created. This class is the derived class of class `Exception`. The variable is of type Exception. An object of base class can always receive an object of the derived `class`. `print(e)` calls `e.__str__()` and displays the resulting string.

The code below shows how to have multiple except blocks and how the ordering matters.

Try putting default except block as the first and see what happens.

```
m = 10
#n = 2
n = 0
try:
      print("one")
      print("res : ", m / n)
      print("two")
      print(myvar)
      print("three")

except NameError as e:
      print("no such name : ", e)

#except Exception as e:
# print("all other exceptions : ", e)

except:
      print("all exceptions")
```

Also observe that on an exception:

- first match is executed
- one and only one except block is executed
- there is no way to go back to the try block i.e., no resume at the point of exception.

Here are the key takeaways:

- **A try clause is executed up until the point where the first exception is encountered.**
- **Inside the except clause, or the exception handler, you determine how the program responds to the exception.**
- **You can anticipate multiple exceptions and differentiate how the program should respond to them.**
- **Avoid using bare except clauses.**

Summing Up

After seeing the difference between syntax errors and exceptions, you learned about various ways to raise, catch, and handle exceptions in Python. In this article, you saw the following options:

- **raise allows you to throw an exception at any time.**
- **assert enables you to verify if a certain condition is met and throw an exception if it isn't.**
- **In the try clause, all statements are executed until an exception is encountered.**
- **except is used to catch and handle the exception(s) that are encountered in the try clause.**
- **else lets you code sections that should run only when no exceptions are encountered in the try clause.**

- **finally enables you to execute sections of code that should always run, with or without any previously encountered exceptions.**

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

References:

1. **18_comprehension_exception.pdf – Prof. N S Kumar, Dept. of CSE, PES University.**
2. **https://realpython.com/python-exceptions/**
3. **https://docs.python.org/**

____________________________________****END****____________________________________

List of Standard Exceptions

Sr.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError**

	Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.
15	**IndexError** Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**IOError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the

	Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Declaring multiple exceptions

The python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions.

Syntax

```
try:

    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
    #block of code
```

Example 1:

```
try:

    a=10/0;

except (ArithmeticError, StandardError):

    print("Arithmetic Exception")

else:

    print("Successfully Done")
```

Output:

```
 Arithmetic Exception
```

Example 2:

```
except (RuntimeError, TypeError, NameError):
    pass
```

Example 3:

```
# Program to handle multiple errors with one except statement
try :
    a = 3
    if a < 4 :
        # throws ZeroDivisionError for a = 3
        b = a/(a-3)
    # throws NameError if a >= 4
    print("Value of b = ", b)

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

Output:

```
Error Occurred and Handled
```