எங்கள் வாழ்வும் எங்கள் வளமும்
மங்காத தமிழ் என்று சங்கே முழங்கு ... *புரட்சிக்கவி*

NOTICE

www.DataScienceInTamil.com

Day 27 - Batch 3 - Python Language

Chapter 015 Exceptions

To watch the recorded Python and Data Science videos in YouTube:

Day 27- Batch 3 - Exceptions - Part 1

https://youtu.be/bt9GDUXn9Qw

Official Website:

https://DataScienceInTamil.com/

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

https://www.DatascienceInTamil.com/#faq

To join DataScienceInTamil Telegram group:

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

https://t.me/joinchat/lUZEsr-zidpjZjEx

To Join the class, please fill the form :

https://forms.gle/QFpLHwAoinFaX2cE6

Join Zoom Meeting (From Sep 26 2022 to Oct 26 2022)

https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFFRUVpTzZ2S1Vsdz09

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language
- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.
- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at info@DataScienceInTamil.com.

➢ All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content /code example, definitions, etc., please use these websites for further reading:

- Book : Python Notes For Professionals
- https://www.w3schools.com
- https://www.geeksforgeeks.org
- https://www.askpython.com
- https://docs.python.org
- https://www.programiz.com/
- https://www.openriskmanagement.com/
- https://pynative.com/python-sets/
- https://www.alphacodingskills.com/
- https://codedestine.com/
- https://appdividend.com/
- https://freecontent.manning.com/
- https://stackoverflow.com/
- https://datagy.io/python-isdigit
- https://www.datacamp.com/community/tutorials/functions-python-tutorial
- https://data-flair.training/blogs/python-function/
- https://problemsolvingwithpython.com/07-Functions-and-Modules/07.07-Positional-and-Keyword-Arguments/

- https://www.tutorialsteacher.com/python/callable-method
- https://www.geeksforgeeks.org/python-exception-handling/
- https://www.javatpoint.com/python-exception-handling
- https://www.tutorialspoint.com/python/python_exceptions.htm
- https://www.w3schools.com/python/python_try_except.asp
- https://www.programiz.com/python-programming/exception-handling
- https://www.datacamp.com/community/tutorials/exception-handling-python
- https://www.geeksforgeeks.org/python-exception-handling/
- https://www.javatpoint.com/python-exception-handling
- https://www.tutorialspoint.com/python/python_exceptions.htm
- https://www.w3schools.com/python/python_try_except.asp
- https://www.programiz.com/python-programming/exception-handling
- https://www.datacamp.com/community/tutorials/exception-handling-python
- www.sentinelone.com/blog/python-stack-trace-understanding-using-debug/
- https://riptutorial.com/python/example/5532/re-raising-exceptions

Exceptions

What to learn

1. **Exceptions and the try:except: and raise statements**
2. **What is an exception**
3. **Catching Exceptions**
4. **‘Else’ will execute only when there is no error in the try block**
5. **After the try..except block, only finally block accepted**
6. **‘raise’ token stops further execution of the except block**
7. **BaseException Class**
8. **Do not catch everything!**
9. **After the ‘raise’, we can not execute any statement / expression.**
10. **Even simple ‘exempt’ works with out Exception class**

11. Re-raising exceptions
12. Catching multiple exceptions
13. Multiple ‘except’ block
14. Exception Hierarchy
15. Else clause
16. Raising Exceptions
17. Creating custom exception types
18. Near-real examples of exception handling
19. Exceptions are Objects too
20. Running clean-up code with finally
21. Chain exceptions with raise from
22. Raise Custom Errors / Exceptions
23. Custom Exception
24. Catch custom Exception
25. Extra notes to understand exceptions

Exceptions and the try:except and raise statements

1. The try:except: statement enables us to catch an exception that is thrown from within a block of code, or from code called from any depth withing that block.

2. The **raise** statement enables us to throw an exception.

3. An exception is a class or an instance of an exception class. If an exception is not caught, it results in a traceback and termination of the program.

4. There is a set of standard exceptions

5. You can define your own exception classes. To do so, create an empty subclass of the class Exception.

6. Defining your own exception will enable you (or others) to throw and then catch that specific exception type while ignore others exceptions.

7. ***inside of raise block will not be executed***

What is an exception

Errors detected during execution are called exceptions and are not unconditionally fatal. Most exceptions are not handled by programs; it is possible to write programs that handle selected exceptions. There are specific features in Python to deal with exceptions and exception logic. Furthermore, exceptions have a rich type hierarchy, all inheriting from the BaseException type.

Catching Exceptions

Below is sequences of try, except, else and finally

```
try:
   5/0
except:
   print("'There is an error in try block , so  the statement
under 'except' will execute")
   raise # if we don’t give raise, the error will not be shown, raise
raise error sound
else:
   print("This 'else' will NOT print, it prints only if there is NO
error in try block")
finally:
    print("This 'finally' block will print , irespective of error in the 'try' or not")
```

ouput

```
Traceback (most recent call last):
  File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line 2, in <module>
    5/0
ZeroDivisionError: division by zero
'There is an error in try block , so  the statement under 'except' will print
This 'finally' block will print , irespective of error in the 'try' or not
```

==========

‘Else’ will execute only when there is NO error in the try block

```
try:
   5/1
except:
   print("'There is NO error in try block , so  the statement
under 'except' will NOT print")
   raise
else:
   print("This 'else' will  print,  reason is,  there is NO error in
try block")
finally:
   print("This 'finally' block will print , irespective of error in
the 'try' block or not")
```

output

```
This 'else' will  print,  reason is,  there is NO error in try block
This 'finally' block will print , irespective of error in the 'try' block
or not
```

```
try:
        x = 5/0
except ZeroDivisionError as e:# in except block we should do
some thing / use pass token
        print("The error is ", e)
        x = 0
        print(x)

finally:
    print("The end")
```

output

```
The error is  division by zero

0

The end

----------
```

```
try:
    print ("Muthu") # use 5/0, then try execute
except:
    raise
    print("FFF") # inside of raise block will not be executed
print("outside of try block will execute")
```

output

```
Muthu

outside of try block will execute
```

```
try:
   print (6/0) # use 5/0, then try execute
except:
   # raise
   print("There is an issue in try - exempt, so this except will execute") # inside of raise
block will not be executed
```

output

```
There is an issue in try - exempt, so this except block will execute
```

After raise- no code will execute

```
try:
    print (6/0) # use 5/0, then try execute
except:
    raise
    print("after raise- no code will execute") # inside of raise block will not be executed
```

output

```
    print (6/0) # use 5/0, then try execute
ZeroDivisionError: division by zero
```

========

After the try..except block, finally block only accepted

Note: below is wrong and will not be executed. After the try..except block, only finally block accepted. But we use print() after the try ..except block

```
try:
       x = 5/0
except ZeroDivisionError as e:
# except ZeroDivisionError: # this also ok (With out ‘e’)
       print("The error is ", e)
       x = 0
       print(x)
print("This is  error")

finally:
    print("The end")
```

output

```
    finally:
```

```
  ^
SyntaxError: invalid syntax
---------
```

The below one is modified code of the above

```
try:
        x = 5/0
except FileNotFoundError as e:
# except ZeroDivisionError as e: # this also ok (With out ‘e’)
        print("The error is ", e)
        x = 0
        print(x)
        print("This is  error")

finally:
    print("The end")
```

output

```
The end
Traceback (most recent call last):
  File
"C:\Users\Melcose\PycharmProjects\pythonProject\Ausralia+Canada.py", line 2, in <module>
    x = 5/0
ZeroDivisionError: division by zero
-----------
try:
        x = 5/0
except FileNotFoundError as e:
    pass
except ZeroDivisionError as e: # this also ok (With out ‘e’)
         print("The error is ", e)
        x = 0
        print(x)
        print("This is  error")
        # raise

finally:
    print("The end")
```

output

```
The error is  division by zero
0
This is  error
The end
```

```
try:
    a = 5/0
except FileNotFoundError as e:
    print("This is not the opt error, so this line will skipped")
except ZeroDivisionError :
    print("This is an opt error and this line will be printed")
    raise
# after raise, nothing will work but finally, so the below line will not be executed

print("outside of try block will execute")
```

output

```
File
"C:\Users\Melcose\PycharmProjects\pythonProject\Ausralia+Canada.py", line 2, in <module>
    a = 5/0
ZeroDivisionError: division by zero
---------
```

Another code

```
try:
        x = 5/0
except FileNotFoundError as e:
    pass

except FloatingPointError:
    pass
    raise

except ZeroDivisionError as e: # this also ok (With out 'e')
        print("The error is ", e)
        x = 0
        print(x)
        print("This is  error")
        raise
```

output

```
This is  error
Traceback (most recent call last):
  File
"C:\Users\Melcose\PycharmProjects\pythonProject\Sudha_Lind
a.py", line 2, in <module>
    x = 5/0
ZeroDivisionError: division by zero
```

What happens if we don’t supply the suitable exception / if we never properly handle the exception

Ans :The outside block will not be executed / the pgm will exit / terminates

```
try:
    a = 5/0

except FileNotFoundError as e:
    print("This is not the opt error, so this line will skipped")

except IndentationError :
    print("This is an opt error and this line will be printed")
    # raise

print("outside of try block will execute")
```

The exception class that is specified - in this case, ZeroDivisionError - catches any exception that is of that class or of any subclass of that exception.

For example, **ZeroDivisionError** is a subclass of **ArithmeticError**:

Note: Since ArthimeticError is super class for ZeroDivisionError, we use Arithmetic error also ..see below

```
try:
        x = 5/0
except ArithmeticError as e: # instead of ZeroDivisionError

        print("Got Arithmetic error", e)
        x = 0
        print(x)
        #raise

finally:
    print("The end")
```

output

```
Got Arithmetic error division by zero #division by zero is from e
0
The end
```

Note: technically, the error is Zero division error, but we provide the parent of ZeroDivisionError , is ArithmeticError – so no issues

```
------------------
try:
        x = 5/0
except ArithmeticError as e:
    pass
    raise
except FileNotFoundError as e: # this also ok (With out ‘e’)
        print("The error is ", e)
        x = 0
        print(x)
        print("This is  error")
        raise

finally:
    print("The end")
```

output
Traceback (most recent call last):

```
  File
"C:\Users\Melcose\PycharmProjects\pythonProject\Ausralia+Ca
nada.py", line 2, in <module>
    x = 5/0
ZeroDivisionError: division by zero
The end
```

Melcose’s Note: Since the first except able to catches the error, it will not execute the following except statement

----

‘raise’ token stops further execution of the except block

```
try:
        x = 5/0
except ArithmeticError as e:
    pass
    print("Before the raise token, excution will happen")
    raise
    print("after the raise token, nothing will execute")
```

```
except FileNotFoundError as e: # this also ok (With out ‘e’)
        print("The error is ", e)
        raise
        x = 0
        print(x)
        print("This is  error")
```

output

```
Before the raise token, excution will happen
Traceback (most recent call last):
  File
"C:\Users\Melcose\PycharmProjects\pythonProject\DataScience
Mel.py", line 2, in <module>
    x = 5/0
ZeroDivisionError: division by zero
```

```
try:
        x = 5/0
except ArithmeticError: # the e is missed , still it works, if we
```

```
'as e", it gives exact error
        print("Got Arithmetic error")
        x = 0
        print(x)

finally:
    print("The end")
```

output

```
Got Arithmetic error division by zero
0
The end
---------------
```

BaseException Class

Note: BaseException is the **super class for ArithemticError** class and ZeroDivision Error class. So we use BaseExcetpion class – see below

```
try:
        x = 5/0
except BaseException as e:

        print("Got Arithmetic error", e)
        x = 0
        print(x)
```

output

```
Got Arithmetic error division by zero
0
```

Note:

```
Object
BaseException
        Exception
                ArithemticError
                        ZeroDivisionError
```

```
print(BaseException.__bases__)
print(Exception.__bases__)
print(ArithmeticError.__bases__)
print(ZeroDivisionError.__bases__)
output
(<class 'object'>,)
(<class 'BaseException'>,)
(<class 'Exception'>,)
(<class 'ArithmeticError'>,)
-----------
```

The exception class that is specified - in this case, ZeroDivisionError - catches any exception that is of that class or of any subclass of that exception.

For example, ZeroDivisionError is a subclass of ArithmeticError:
To find all the hierarchy, use MRO

```
print(ZeroDivisionError.mro())
```

output

```
[<class 'ZeroDivisionError'>, <class 'ArithmeticError'>, <class
'Exception'>, <class 'BaseException'>, <class 'object'>]
```

Do not catch everything!

While it's often tempting to catch every Exception:

```
try:
    very_difficult_function() # this makes error
except Exception:
    print("This statement will be executed")

finally:
    print("The end")
```

output

```
This statement will be executed
The end
--------
```

If we don’t give ‘raise’ it does show the type of exception. So always give the ‘raise’ in except class – (The above code is written with ‘raise’ and see the output and it has the exception details)

```
try:
    very_difficult_function() # this makes error
except Exception:
    print("This statement will be executed")
    raise

finally:
    print("The end"
```

output

```
very_difficult_function() # this makes error
```

```
NameError: name 'very_difficult_function' is not defined
This statement will be executed
The end
--------
```

Note: **after the ‘raise’, we can NOT execute any statement / expression.**

========

The code in except block is excuted if we give super exception classes (Exception and BaseException)

```
try:
    very_difficult_function()
except Exception:
    a = 10
    b = 20
    print("This statement will be executed")
    print(a+b)

finally:
    print("The end")
```

output

```
This statement will be executed
30
The end
```

Even simple ‘exempt’ works with out Exception class

```
try:
    very_difficult_function() # this makes error
except:
    print("This statement will be executed")
    # raise
```

```
finally:
    print("The end")
```

output

```
This statement will be executed
The end
```

The below code gives error, reason we did not give super exception classes (Exception and BaseException), but ArithemeticError or ZeroDivisionError

```
try:
    very_difficult_function() # this makes error
except ArithmeticError as e:
    pass

finally:
    print("The end")
```

output

```
The end # this is from finally code
Traceback (most recent call last):
  File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line 2, in <module>
    very_difficult_function()
NameError: name 'very_difficult_function' is not defined
```

Note: the function “very_difficult_function()” makes error. But the ArithmeticError or ZeroDivisionEror CANNOT catch the exception. So it is an error. To avoid it , use Exception or BaseExcetpion (see the one above pgm)

The ArithmeticError or ZeroDivisionEror will not give the name of the error / exception in the function very_difficult_function(). To find the exact name of the error / exception use Exception or BaseExcetpion – see below

```
try:
    even_ery_difficult_function()
except BaseException as e: # we use the object ‘e’ that fetches the
name of the exception
    print(e)

finally:
    print("The end")
```

output

```
name 'even_ery_difficult_function' is not defined
The end
```

See some more examples how the datatypes are treated inside the try ..except block

```
try:
    5
    True
    complex
    5.6

except BaseException as e:
    print(e)
```

outout

no error. even though we did not assign the above values to any variables. Still it accepts. Reason the datatypes are ok. But if we declare a string without quote it is an error – and it is caught by the Exeception

```
try:
    SampleString
except BaseException as e:
    print(e)
```

output

```
name 'SampleString' is not defined
----------
```

In most cases it's bad practice. It might catch more than intended, such as SystemExit, KeyboardInterrupt and MemoryError - each of which should generally be handled differently than usual system or logic errors. It also means there's no clear understanding for what the internal code may do wrong and how to recover properly from that condition. **If you're not catching every error, you won't know what error occurred or how to fix it.**

This is more commonly referred to as '**bug masking**' **and should be avoided.** Let your program crash instead of silently failing or even worse, failing at deeper level of execution. (Imagine it's a transactional system)

Usually these constructs are used at the very outer level of the program, and will log the details of the error so that the bug can be fixed, or the error can be handled more specifically

Re-raising exceptions

Sometimes you want to catch an exception just to inspect it, e.g. for logging and debugging purposes. After the inspection, you want the exception to continue propagating as it did before.

In this case, simply use the raise statement with no parameters.

```
try:
    5/0
except ZeroDivisionError as e:
    print("The name of the exception is ", e)
    raise
```

output is error

```
5/0 # this is from raise

ZeroDivisionError: division by zero # this is from raise

The name of the exception is  division by zero
```

```
try:
    5/0
except ZeroDivisionError:
    raise
```

output

```
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
2, in <module>
    5/0
ZeroDivisionError: division by zero

Process finished with exit code 1
```

The below code gives the same result of as above

```
try:
    5/0
except:
    raise
```

If we use 5/1, then the except block will not execute, no exception will be raised, even if we use ‘raise’

```
try:
    5/1
except ZeroDivisionError as e:
    print("The name of the exception is ", e)
    raise
```

output

```
NO OUTPUT
```

=============

Keep in mind, though, that someone further up in the caller stack can still catch the exception and handle it somehow. The done output could be a nuisance in this case because it will happen in any case (caught or not caught). So it might be a better idea to raise a different exception, containing your comment about the situation as well as the original exception:

```
try:
    5/0
except ZeroDivisionError as e:
    print("Got an ERROR")
    raise ZeroDivisionError ("Got an error") from e
```

output

```
Got an ERROR
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
2, in <module>
    5/0
ZeroDivisionError: division by zero
```

```
The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
5, in <module>
    raise ZeroDivisionError ("Got an error") from e
ZeroDivisionError: Got an error
---------
```

But this has the drawback of reducing the exception trace to exactly this raise while the raise without argument retains the original exception trace

In Python 3 you can keep the original stack by using the raise-from syntax:

```
raise ZeroDivisionError ("Got an error") from e
```

Catching multiple exceptions

There are a few ways to catch multiple exceptions (https://docs.python.org/3/tutorial/errors.html#handling-exceptions

).

The first is by creating a **tuple of the exception types** you wish to catch and handle in the same manner. This example will cause the code to ignore KeyError and AttributeError exceptions.

```
try:
    d = {}
    a = d[1]
    b = d
    AAA

except (KeyError, AttributeError, IndentationError, NameError) as e:
    print("A KeyError or an AttributeError exception has been caught.", e)
    raise
```

output

```
A KeyError or an AttributeError exception has been caught. 1
```

Note: if we don’t give ‘raise’ the try block gives only the key name (1) that casues the error. But DOES NOT give the name of the exception, to see the name of exception and the key name that cases exception , we give ‘raise’ – see below

------

If we give ‘raise’ after ‘except’ block we get the name/s of the exception

```
try:
    d = {}
    a = d[1]
    b = d
```

```
except (KeyError, AttributeError) as e:
    print("A KeyError or an AttributeError exception has been caught.")
    raise
```

output

```
File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line 3, in <module>
    a = d[1]
KeyError: 1
A KeyError or an AttributeError exception has been caught.
```

Multiple ‘except’ block

If you wish to handle different exceptions in different ways, you can provide a separate exception block for each type. In this example, we still catch the KeyError and AttributeError, but handle the exceptions in different manners

```
try:
    d = {}
    a = d[1]
    b = d
except KeyError as e:
    print("A KeyError has occurred. Exception message:", e)
except AttributeError as e:
    print("An AttributeError has occurred. Exception message:",
e)
```

output

```
A KeyError has occurred. Exception message: 1
```

Note: the output does not give the exception name. so use ‘raise’ to see the exception – see below

----

```
try:
    d = {}
    a = d[1]
    b = d
except KeyError as e:
    print("A KeyError has occurred. Exception message:", e)
    raise
except AttributeError as e:
    print("An AttributeError has occurred. Exception message:",
e)
    raise
```

ouput

```
A KeyError has occurred. Exception message: 1
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
3, in <module>
    a = d[1]
KeyError: 1
```

Note: the ‘raise’ gives us name of the exception(KeyError) and the key name(1) that causes the exeception.

Observe: we have 2 ‘except’ with ‘raise’ block. Only the ‘except’ block that has the corresponding error only executed. Other ‘except’ blocks will NOT be executed

Note: if there was an AttributeError, then the second ‘except’ block would have executed

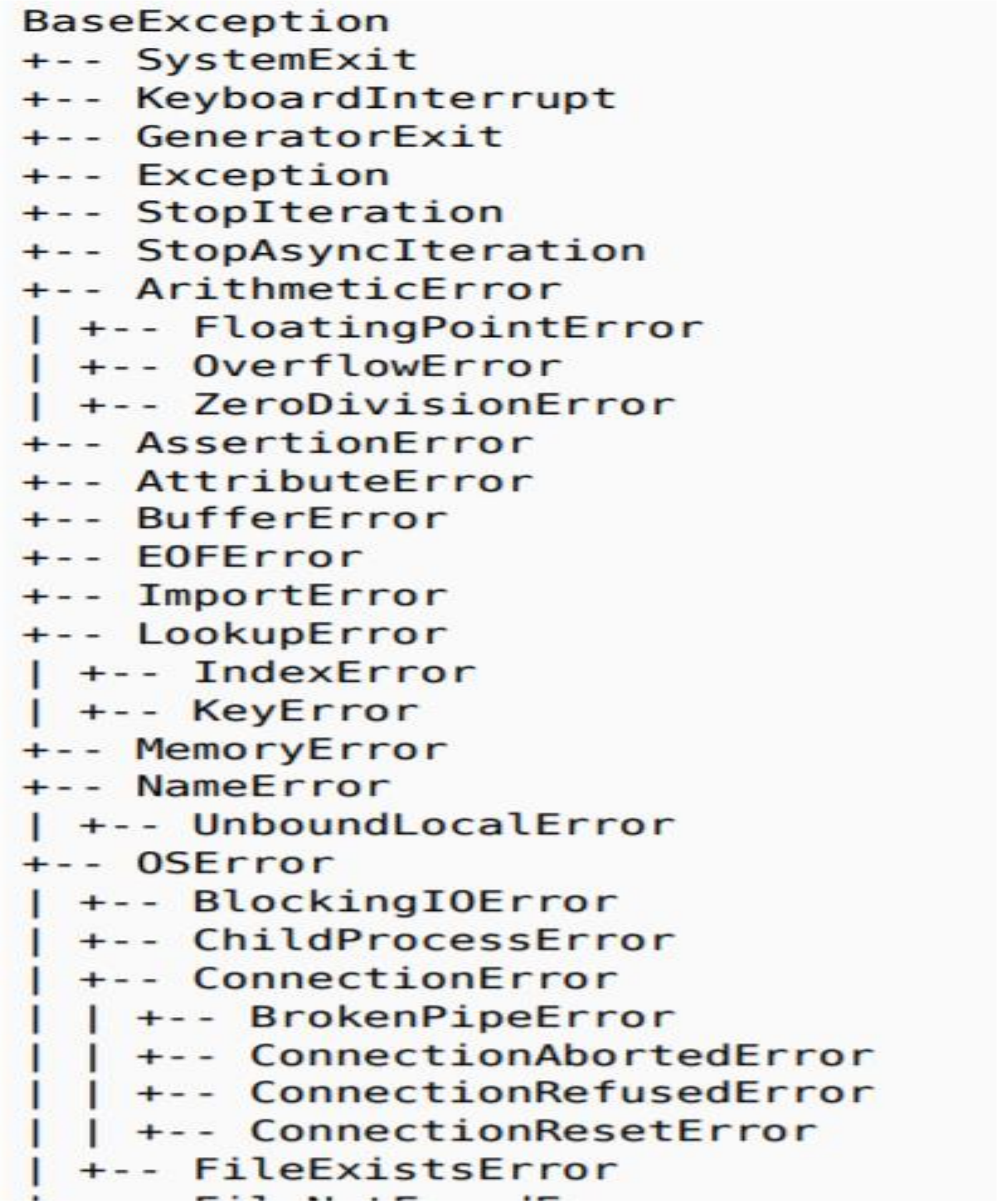
===========

Exception Hierarchy

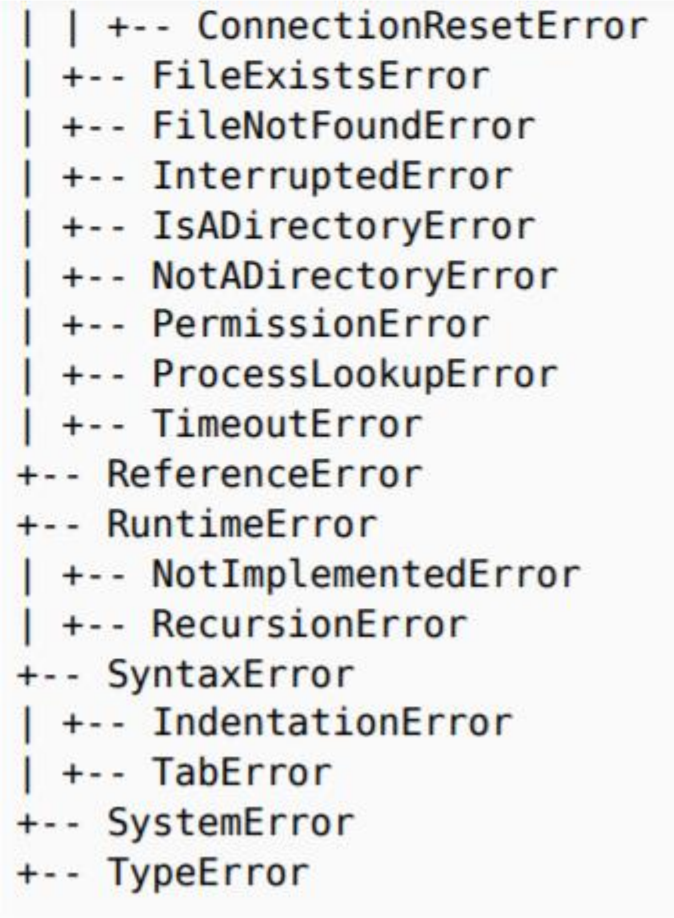
Exception handling occurs based on an exception hierarchy, determined by the inheritance structure of the exception classes.

For example, IOError and OSError are both subclasses of EnvironmentError. Code that catches an IOError will not catch an OSError. However, code that catches an EnvironmentError will catch both IOErrors and OSErrors.

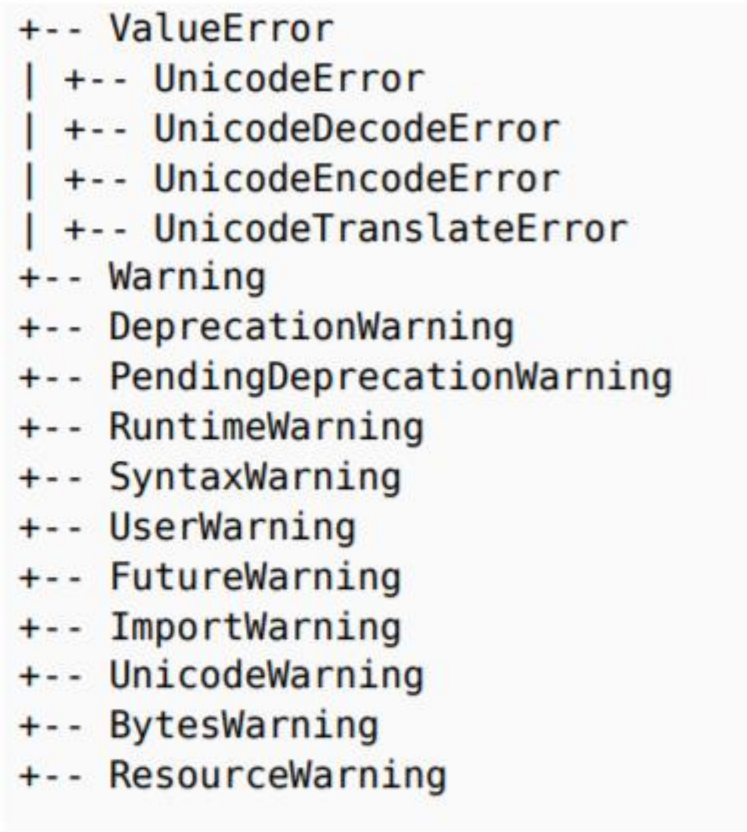
The hierarchy of built-in exceptions:



```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StopAsyncIteration
+-- ArithmeticError
| +-- FloatingPointError
| +-- OverflowError
| +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
+-- LookupError
| +-- IndexError
| +-- KeyError
+-- MemoryError
+-- NameError
| +-- UnboundLocalError
+-- OSError
| +-- BlockingIOError
| +-- ChildProcessError
| +-- ConnectionError
| | +-- BrokenPipeError
| | +-- ConnectionAbortedError
| | +-- ConnectionRefusedError
| | +-- ConnectionResetError
| +-- FileExistsError
| +-- FileNotFoundError
```



```
| | +-- 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
```

Else clause

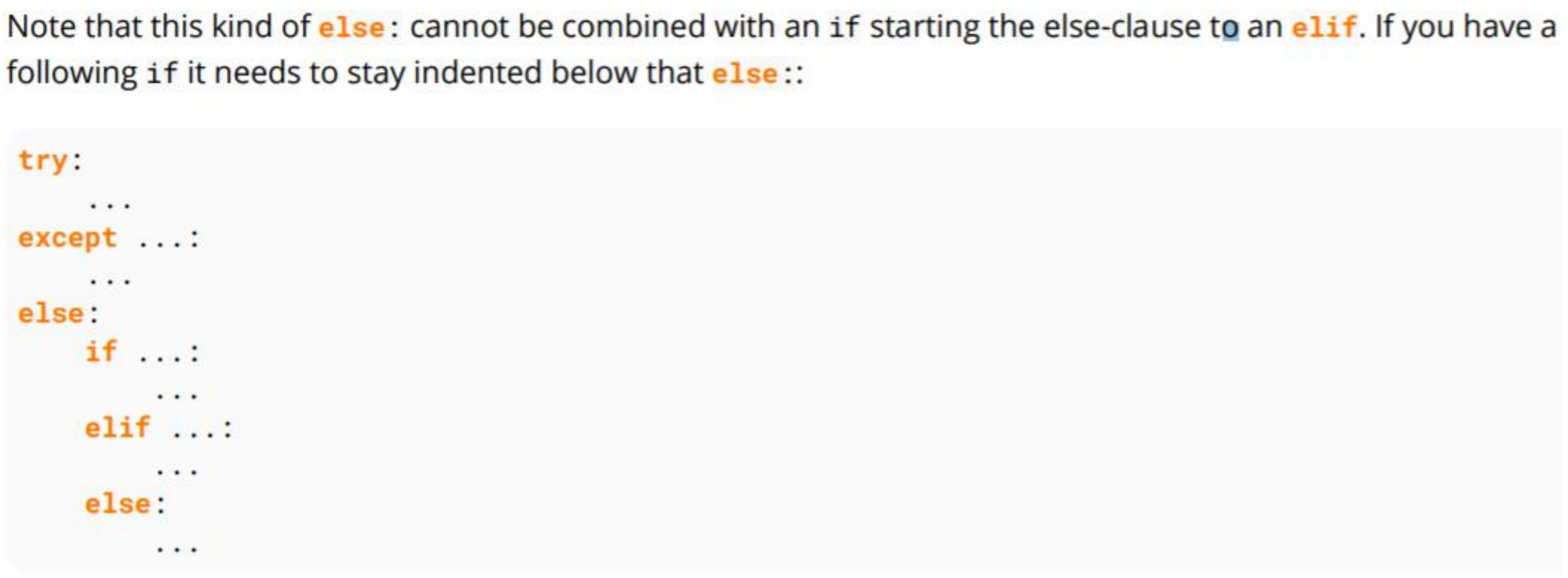
Code in an else block will only be **run if no exceptions were raised by the code in the try block.** This is useful if you have some code you don’t want to run if an exception is thrown, but you don’t want exceptions thrown by that code to be caught.

```
try:
    data = {1:"One", 2:"Two"}
    print (data[1])

except KeyError as e:
    print("Key is not found")

else:
    raise ValueError()
```

ouput

```
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
9, in <module>
    raise ValueError()
```

```
ValueError
One
--------
```



Note that this kind of `else:` cannot be combined with an `if` starting the else-clause to an `elif`. If you have a following `if` it needs to stay indented below that `else:`:

```
try:
    ...
except ...:
    ...
else:
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
```

Raising Exceptions

If your code encounters a condition it doesn't know how to handle, such as an incorrect parameter, it should raise the appropriate exception

```
def even_the_odds(odds):
    if odds % 2 !=1:
        raise ValueError ("Did not get an odd number")
    return odds + 1

a = even_the_odds(2)
print(a)
```

output

```
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
6, in <module>
    a = even_the_odds(2)
  File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line 3, in even_the_odds
    raise ValueError ("Did not get an odd number")
ValueError: Did not get an odd number
```

Note: if we pass 3, we get the answer 4

======

Creating custom exception type

Purpose: user friendly message can be given, instead of system generated message

Create a class inheriting from Exception:

```
class FooException(Exception):
    pass
try:
    raise FooException ("Insert descritpion here")
except FooException:
    print("A Foo Exception was raised")
```

output

```
A Foo Exception was raised
```

=====

or another exception type:

```
class NegativeError(ValueError):
    pass

def foo (x):
    if x < 0:
        raise NegativeError("Cannot process negative numbers")
try:
    result = foo(int(input("Enter a positive intger: ")))
```

```
except NegativeError:
    print("You entered a negative number")
else:
    print("The result was  ", str(result))
```

=============

Real near examples of exception handling

User input

Imagine you want a user to enter a number via input. You want to ensure that the input is a number. You can use try/except for this:

```
while True:
    try:
        nb = int(input("Enter a number:  "))
        break
    except ValueError:
        print("This is not a number, try again")
```

Dictionaries

Imagine you are iterating over a list of consecutive integers, like range(n), and you have a list of dictionaries d that contains information about things to do when you encounter some particular integers, say skip the d[i] next ones.

The below code need additional explanation

```
d = [{7: 3}, {25: 9}, {38: 5}]
for i in range(len(d)):
    do_stuff(i)
    try:
        dic = d[i]
        i  = i + dic[i]
    except KeyError:
        i = i +1
```

A KeyError will be raised when you try to get a value from a dictionary for a key that doesn’t exist.
=============

Exceptions are Objects too

Exceptions are just regular Python objects that inherit from the built-in BaseException. A Python script can use the raise statement to interrupt execution, causing Python to print a stack trace of the call stack at that point and a representation of the exception instance. For example:

```
def failing_function():
    raise ValueError ("Example Error")

failing_function()
```

output

```
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
4, in <module>
    failing_function()
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
2, in failing_function
    raise ValueError ("Example Error")
ValueError: Example Error
===============
```

which says that a ValueError with the message 'Example error!' was raised by our failing_function(), which was executed in the interpreter

Calling code can choose to handle any and all types of exception that a call can raise:

```
try:
    def failing_function():
except ValueError as e:
    print("Handled the error", e)
```

output

```
File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py",
line 3
    except ValueError as e:
IndentationError: expected an indented block
```

Note: This error was handled by Python itself

===========

Below code need explanation

```
try:
    failing_function():
except ValueError as e:
    print("Caught Exception ", (repr(e)))
```

output

```
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
2
    failing_function():
```

```
               ^
SyntaxError: invalid syntax
```

===============

A complete list of built-in Python exceptions along with their descriptions can be found in the Python Documentation: https://docs.python.org/3.5/library/exceptions.html
. And here is the full list arranged hierarchically: Exception Hierarchy.

===============

Running clean-up code with finally

Sometimes, you may want something to occur regardless of whatever exception happened, for example, if you have to clean up some resources.

The finally block of a try clause will happen regardless of whether any exceptions were raised.

```
# resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except ValueError as e:
    print(e)
finally:
    print("Free the resource")
```

Output

```
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
3, in <module>
    do_stuff(resource)
NameError: name 'do_stuff' is not defined
Free the resource
```

This pattern is often better handled with context managers (using the with statement).

==============

Chain exceptions with raise from

In the process of handling an exception, you may want to raise another exception. For example, if you get an IOError while reading from a file, you may want to raise an application-specific error to present to the users of your library, instead.

You can chain exceptions to show how the handling of exceptions proceeded:

```
try:
    5/0
```

```
except ZeroDivisionError as e:
    # print(e)
    raise ValueError("Divison Failed") from e
```

output

```
Traceback (most recent call last):
  File
"C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line
2, in <module>
    5/0
ZeroDivisionError: division by zero

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line 5, in <module>
    raise ValueError("Divison Failed") from e
ValueError: Divison Failed

Process finished with exit code 1
```

===========

Raise Custom Errors / Exceptions

Python has many built-in exceptions which force your program to output an error when something in it goes wrong.

However, sometimes you may need to create custom exceptions that serve your purpose.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived from this class.

Custom Exception

Here, we have created a user-defined exception called CustomError which is derived from the Exception class. This new exception can be raised, like other exceptions, using the raise statement with an optional error message.

```
class CustomError(Exception):
    pass

x = 1
if x ==1:
    raise CustomError("This is custom error")
```

output

```
Traceback (most recent call last):
  File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line 6, in <module>
    raise CustomError("This is custom error")
__main__.CustomError: This is custom error
============
class FiftyNotesNowAvaialbe(Exception):
    pass

x = 1
if x ==1:
    raise FiftyNotesNowAvaialbe("Fifty notes are not availabel in this branch")
================
```

Catch custom Exception

This example shows how to catch custom Exception

```
class CustomError(Exception):
    pass

try:
    raise CustomError ("Can you catch me")
except CustomError as e:
    print("Catched Custom Error: {} ". format(e))
except Exception as e:
    print ("Generic Exception {}". format(e))
```

output

```
Catched Custom Error: Can you catch me
```

Note: we give 2 exceptions, one is custom exception and an another one generic exception. Since the first ‘except’ block catches the custom exception the second ‘except’ block will not execute. See below ,, we modified the code such that the first ‘except’ block is not able to catch the custom error / exception. So it goes to second ‘except’ block where it was caught .

```
class CustomError(Exception):
    pass

try:
    raise CustomError ("Can you catch me")
except ArithmeticError as e:
    print("Catched Custom Error: {} ". format(e))
except Exception as e:
    print ("Generic Exception {}". format(e))
```

output

```
Generic Exception Can you catch me
==========
```

Extra notes to understand exceptions

If there is an exception in try block and if we give the exact corresponding Exception in except block, then the except block will execute

```
try:
    1/0
except ZeroDivisionError as e:
    print("Except block caught the exception")
```

output

```
Except block caught the exception
```

============

If there is an exception in try block and if we NOT give the exact corresponding Exception in except block, then the except block will NOT execute, the system generate the correct exception based on the try block code

```
try:
    1/0
except AttributeError as e:
    print ("Except block will not execute")
```

output

```
Traceback (most recent call last):
  File "C:\Users\ABC\PycharmProjects\pythonProject\CIPACA.py", line 2, in <module>
    1/0
ZeroDivisionError: division by zero
```

==========

If there is no error in try block, the except block will not be executed and else block will be executed

```
try:
    1/1
except Exception as e:
    print("Except block")
else:
    print("Prints only when there is no exception in try block")
```

output

```
Prints only when there is no exception in try block
```

=======

Now try block has 2 errors.

1/0 and using raise we create AttributeError

But except block catches only 1/0 exception. There is no except block to catch the AttributeError(by raise).

```
try:
    1/0
    raise AttributeError
except ZeroDivisionError as e:
    print("Except block will execute")
```

output

```
Except block will execute
==========
```

```
try:
    1/0
    raise AttributeError
except ZeroDivisionError as e:
    print("First Except block will execute for 1/0")

try:
    raise AttributeError
except:
    print("This is from second except block execute for AttributeError(raise"))
```

output

```
First Except block will execute for 1/0
This is from second except block execute for AttributeError(raise)
```

=============

Each exception must be handled individually in each try block

```
try:
    1/0

except ZeroDivisionError as e:
    print("First Except block will execute for 1/0")

try:
    AAA
except NameError as e:
    print("Error is, ", e)
```

output

```
First Except block will execute for 1/0
Error is,  name 'AAA' is not defined
```

========================

Additional notes and codes

EXCEPTION HANDLING

ERRORS

- Errors are the problems in a program due to which the program will stop the execution

Errors can be of various types:

- **Syntax Error (Parsing errors)**
- **Out of Memory Error (RAM & Heap)**
- **Recursion Error (Stack & Function calls)**
- **Exceptions**

Syntax Error:

- Error is caused by the wrong syntax in the code.
- It leads to the termination of the program.

EXAMPLE:

```
amount = 10000

if(amount > 2999)

print("You are eligible to purchase ")
```

OUTPUT:

```
File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 2

  if(amount > 2999)

            ^

SyntaxError: invalid syntax

Process finished with exit code 1
```

EXCEPTIONS

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print(c)
print("successful completion")
```

OUTPUT:

```
Enter a:5
Enter b:0
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 3, in <module>
    c = a/b
ZeroDivisionError: division by zero
Process finished with exit code 1
```

STACK TRACE (stack traceback, backtrace, or traceback):

- Python prints a stack trace when your code throws an exception.
- Stack trace holds a lot of valuable information that helps us to identify the source of the problem.
- A stack trace report contains the function calls made in your code right before the error occurred.

- A stack trace prints all the calls prior to the function that raised an exception.
- The last line of a stack trace prints the most valuable information as here the error gets printed.

EXAMPLE:

```
def FS(name):
    print('Hello, ' + nam)

FS('DATA SCIENCE IN TAMIL MEMEBERS')
```

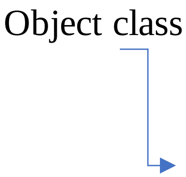
Output:

```
Traceback (most recent call last):

  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 4, in <module>

    FS('DATA SCIENCE IN TAMIL MEMEBERS')

  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 2, in FS

    print('Hello, ' + nam)

NameError: name 'nam' is not defined

Process finished with exit code 1
```

EXCEPTIONS

- Exceptions are raised when the program is syntactically correct, but the code resulted in an error.
- An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.
- Whenever an exception occurs, the program stops the execution, and thus the further code is not executed.
- An exception is the run-time errors that are unable to handle to Python script.
- An exception is a Python object that represents an error.
- Python provides a way to handle the exception so that the code can be executed without any interruption.
- If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

- Python3 has 63 **built-in exceptions** that enable our program to run without interruption and give the output.

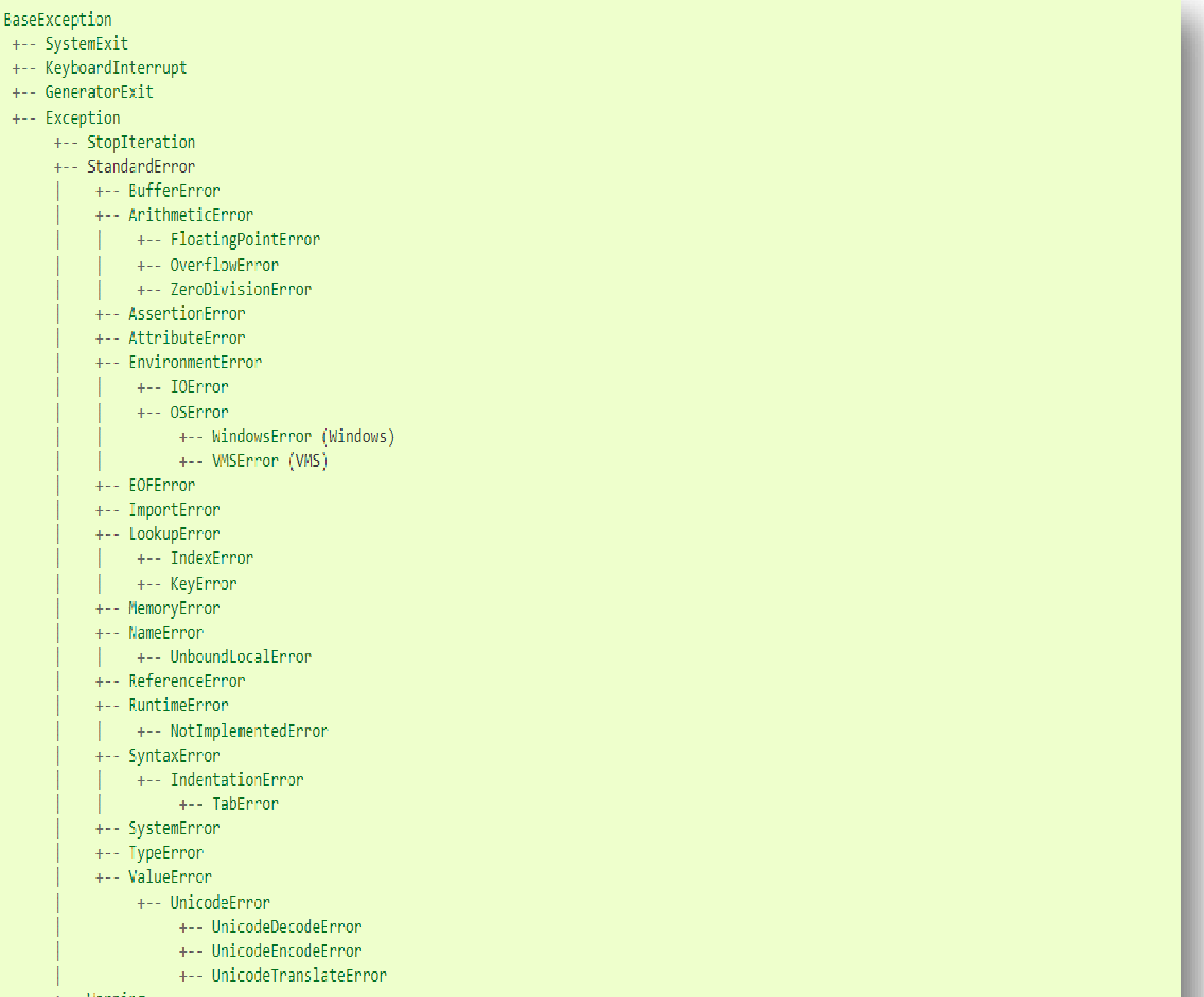


Object class

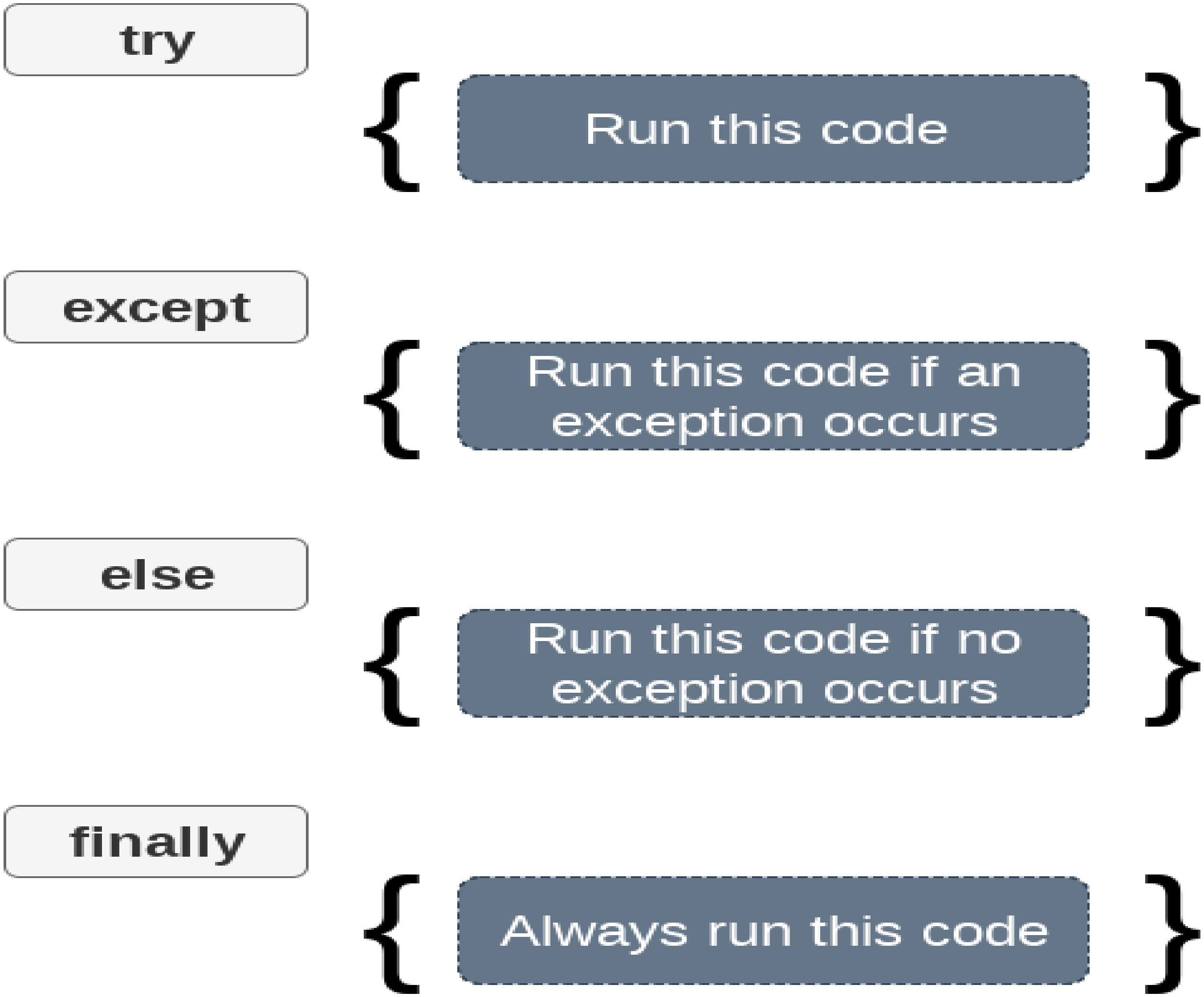
BaseException class

NOTE: Exceptions are objects created from built in BaseException class.

Exception handling in Python



```
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
```



try

{ Run this code }

except

{ Run this code if an exception occurs }

else

{ Run this code if no exception occurs }

finally

{ Always run this code }

try and except statement:

- Try and except statements are used to catch and handle exceptions in Python.
- Statements that can cause(raise) exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.
- The **try** block must be followed with the **except** statement.

EXAMPLE 1:

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
   c = a/b
   print(c)
except:
   print("cannot perform division")
print("successful completion")
```

OUTPUT:

```
Enter a:5

Enter b:0

cannot perform division
```

```
successful completion

Process finished with exit code 0
```

EXAMPLE 2:

```
a=[1,2,3]
try:
   print(a[1])
   print(a[3])
except:
   print("AN ERROR OCCURRED")
```

OUTPUT:

```
AN ERROR OCCURRED

Process finished with exit code 0
```

SPECIFY EXCEPTION TYPE:

EXAMPLE 1

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
try:
    c = a/b
    print(c)
except ArithmeticError:
    print("cannot perform division")
print("successful completion")
```

OUTPUT:

```
Enter a:8

Enter b:0

cannot perform division

successful completion

Process finished with exit code 0
```

EXAMPLE 2

```
try:
   fileptr=open("file1.txt","r")
except IOError:
    print("file not found")
```

OUTPUT:

```
file not found

Process finished with exit code 0
```

EXAMPLE 3

```
try:
   a=5
   b='0'
   print (a+b)
except TypeError:
   print('Unsupported operation')
```

OUTPUT:

```
Unsupported operation
```

EXCEPTION STATEMENT WITH EXCEPTION VARIABLE:

- Exception variable holds built in information about exception.

```
try:
   a = ['a', 'b', 'c']
   print(a[4])
```

```
except LookupError as e:
    print(e)
    print("Index Error list index out of range")
```

OUTPUT:

```
list index out of range

Index Error list index out of range

Process finished with exit code 0
```

MULTIPLE ‘except ‘BLOCK:

- A try statement can have more than one except clause, to specify handlers for different exceptions.
- At most one handler will be executed.

SYNTAX:

```
try:

    # statement(s)

except Exceptiontype1:

    # statement(s)

except Exceptiontype2:

    # statement(s)

……
```

EXAMPLE 1:

```
try:
    x=int(input("enter a no"))
    y=1/x
    print(y)
except ZeroDivisionError:
    print("divide by zero error")
except ValueError:
    print("enter int value only")
except: # NOTE:UNAMED EXCPETION TO BE SPECIFIED AT LAST ONLY
    print("error")
 print("END")
```

OUTPUT:

CASE 1:

```
enter a no 0

divide by zero error

Process finished with exit code 0
```

CASE 2:

```
enter a no12.3

enter int value only

Process finished with exit code 0
```

EXAMPLE 2:

```
try:
   d={}
   a=d[1] #CASE 1
   #b=c   #CASE 2
except KeyError as e:
   print("key error occurred",e)
except NameError as e:
   print("Name error occurred",e)
```

OUTPUT:

CASE 1:

```
key error occurred 1

Process finished with exit code 0
```

CASE 2:

```
Name error occurred name 'c' is not defined

Process finished with exit code 0
```

<u>CATCHING MULTIPLE EXCEPTIONS:</u>

- Handle two or more exceptions by creating a tuple of the exception types.

SYNTAX:

```
try:
    # statement(s)
except (Exceptiontype1, Exceptiontype2,..):
    # statement(s)
```

EXAMPLE:

```
try:
   d={}
   #a=d[1] #CASE 1:
   b=c       #CASE 2:
except (KeyError,NameError) as e:
   print("error occurred",e)
```

OUTPUT:

CASE 1:

```
key error occurred 1

Process finished with exit code 0
```

CASE 2:

```
Name error occurred name 'c' is not defined

Process finished with exit code 0
```

NOTE:

What happens if we don’t specify the suitable exception type?

The program will terminate by displaying stack trace.

EXCEPTIONS AND FUNCTIONS:

- If an exception is raised inside a function it can be handled either inside the function or outside the function.

```
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) #CASE 1:
    fun(5) #CASE 2:
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

OUTPUT:

CASE 1:

```
ZeroDivisionError Occurred and Handled

Process finished with exit code 0
```

CASE 2:

```
NameError Occurred and Handled

Process finished with exit code 0
```

--

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.

EXAMPLE:

```
try:
   print (float('DataScienceInTamil'))  #CASE 1
  # print(float(19)) #CASE 2
except ValueError:
    print ('ValueError: could not convert string to float: \'DataScienceInTamil\'')
else:
    print ('Success, no error!')
```

OUTPUT:

CASE 1:

```
ValueError: could not convert string to float: 'DataScienceInTamil'
```

```
Process finished with exit code 0
```

CASE 2:

```
19.0
```

```
Success, no error!
```

```
Process finished with exit code 0
```

EXCEPTION HANDLING INSIDE THE FUNCTION:

```
def Div(a,b):
   try:
       c=((a+b)/(a-b))
   except ZeroDivisionError:
       print ("Divide by 0 error")
   else:
       print (c)
Div(2.0, 3.0)#CASE 1
#Div(3.0, 3.0)#CASE 2
```

OUTPUT:

CASE 1:

```
-5.0

Process finished with exit code 0
```

CASE 2:

```
Divide by 0 error

Process finished with exit code 0
```

--

finally:

- Occurs after try …except block
- Finally, block 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.
- Provides a guarantee of execution.
- Used to release external resources.

EXAMPLE:

```
try:
  k = 5//0 # raises divide by zero exception. #CASE 1
  k=5//1 # no exception #CASE 2
  print(k)
except ZeroDivisionError:
  print("Can't divide by zero")
finally:
  print('This is always executed')
```

OUTPUT:

CASE 1:

```
Can't divide by zero

This is always executed

Process finished with exit code 0
```

CASE 2:

```
5

This is always executed

Process finished with exit code 0
```

NOTE:

For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

EXAMPLE 2:

```
try:
  fileptr=open("file1.txt","r")
except IOError:
   print("file not found")
else:
  print("File opened successfully")
finally:
  print("file closed ")
  fileptr.close()
```

OUTPUT:

CASE : If file doesn’t exist

> ***file not found***

CASE: If file exist

> ***File opened successfully***
>
> ***file closed***

--

RAISING EXCEPTION(raise statement):

- An exception can be (thrown)raised forcefully by using the raise clause.
- raise instruction raises the specified exception as it was raised in a normal way.

SYNTAX:raise Exception_class<value>

- It simulates raising actual exceptions.
- It 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.
- It is useful in in that scenario where we need to raise an exception to stop the execution of the program.

Example: there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

HINTS:

1.To raise an exception, the raise statement is used. The exception class name follows it.

2.An exception can be provided with a value that can be given in the parenthesis.

3.To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

4.We can pass the value to an exception to specify the exception type.

Example 1:

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

OUTPUT:

CASE 1:

Enter the age:10

```
The age is not valid

CASE 2:

Enter the age:19

the age is valid
```

Example 2 Raise the exception with message

```
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
        raise ValueError("That is  a negative number!")
except ValueError as e:
    print("ERROR",e)
```

```
OUTPUT:
Enter a positive integer: -8
ERROR That is  a negative number!
```

EXAMPLE 2.1:

```
x = "hello"

if not type(x) is int:

  raise TypeError("Only integers are allowed")
```

OUTPUT:

```
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 3, in <module>
    raise TypeError("Only integers are allowed")
TypeError: Only integers are allowed
```

Example 3

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b == 0:
        raise ArithmeticError
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

OUTPUT:

```
Enter a:5
Enter b:0
The value of b can't be 0
```

Re raising exception and Exception chaining:

- raise instruction can be used to immediately reraise the same exception as currently handled.

  EXAMPLE:

  ```
  try:
     raise NameError("hi")
  except NameError:
     print("an exception")
     raise
  ```

  OUTPUT:

  ```
  an exception
  Traceback (most recent call last):
    File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 2, in <module>
      raise NameError("hi")
  NameError: hi

  Process finished with exit code 1
  ```

- Sometimes there is a need to catch an exception just to inspect it, e.g. for logging purposes.
- After the inspection,exception can continue propagating as it did before.
- In this case, simply use the raise statement with no parameters.

EXAMPLE 1:

```
try:
    5/0
except:
    print("error")
    raise
```

OUTPUT:

```
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 2, in <module>
    5/0
ZeroDivisionError: division by zero
error
Process finished with exit code 1
```

Note: Use it when there is no need to take any action in response to an exception.

EXAMPLE 2:

```
def ex():
    try:
        int('DSIT')
    except ValueError:
        print("didnt work")
        raise
ex()
```

```
OUTPUT:
didnt work
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 7, in <module>
    ex()
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 3, in ex
    int('DSIT')
ValueError: invalid literal for int() with base 10: 'DSIT'
```

- Exception chaining happens automatically when an exception is raised inside an except or finally section.

- ‘raise’ statement allows an optional ‘from’ which enables chaining exceptions.

EXAMPLE:

```
def F():
    raise NameError
try:
    F()
except NameError as e:
    raise NameError("failure") from e
```

OUTPUT:

```
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 4, in <module>
    F()
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 2, in F
    raise NameError
NameError
The above exception was the direct cause of the following exception:
```

```
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 6, in <module>
    raise NameError("failure") from e
NameError: failure
Process finished with exit code 1
```

- Exception chaining can be disabled by using ‘none’ keyword.

EXAMPLE:

```
def F():
    raise NameError
try:
    F()
except NameError as e:
    raise NameError("failure") from None
```

OUTPUT:

```
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 6, in <module>
    raise NameError("failure") from None
NameError: failure

Process finished with exit code 1
```

Custom Exception/User Defined Exception:

- Users can create custom exception that serves their needs.
- Such exceptions can be created by defining a new class.
- User defined exception class has to be derived from built in ‘Exception’ class.

EXAMPLE 1:

```
class MyException(Exception):
    pass
x=1
if x==1:
    raise MyException
```

OUTPUT:

```
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 5, in <module>
    raise MyException
__main__.MyException
```

```
Process finished with exit code 1
```

EXAMPLE 2:

```
class MyException(Exception):
    pass
x=1
if x==1:
    raise MyException("my own exception message")
```

OUTPUT:

```
Traceback (most recent call last):
  File "C:\Users\Dell\PycharmProjects\pythonProject1\ex.py", line 5, in <module>
    raise MyException("my own exception")
__main__.MyException: my own exception

Process finished with exit code 1
```

EXAMPLE 3:

```
        class MyException(Exception):
            def __init__(self,data):
                self.data=data
            def __str__(self):
```

```
        return repr(self.data)
try:
    raise MyException("hai welcome")
except MyException as e:
    print(e.data)
```

OUTPUT:

```
hai welcome
```

DRAWBACKS OF EXCEPTION HANDLING:

Though exception handling is very useful in catching and handling exceptions in Python, it also has several disadvantages.

- It can trap only run-time errors.
- When you use try-except, the program will lose some performance and slow down a bit.
- The size of the code increases when you use multiple try, except, else and finally blocks.
- The concept of try-catch might be a little difficult to understand for beginners.
- It is useful only in exceptional error cases.

COMMON EXCEPTIONS AND CAUSES:

Exception	Cause of Error
ArithmeticError	Raised when numerical computation fails.
FloatingPointError	Raised when floating point calculation fails.
AssertionError	Raised in case of failure of the Assert statement.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numerical values.
OverflowError	Raised when result of an arithmetic operation is very large to be represented.
IndexError	Raised when an index is not found in a sequence.
ImportError	Raised when the imported module is not found.
IndentationError	Raised when indentation is not specified properly.
KeyboardInterrupt	Raised when the user hits interrupt key.
RuntimeError	Raised when a generated error does not fall into any category.

Exception	Cause of Error
SyntaxError	Raised when there is an error in Python syntax.
IOError	Raised when Python cannot access a file correctly on disk.
KeyError	Raised when a key is not found in a dictionary.
ValueError	Raised when an argument to a function is the right type but not in the right domain.
NameError	Raised when an identifier is not found in the local or global namespace.
TypeError	Raised when an argument to a function is not in the right type.

Exception

Base class for all exceptions
StopIteration Raised when the next() method of an iterator does not point to any object.
SystemExit Raised by the sys.exit() function.
StandardError Base class for all built-in exceptions except StopIteration and SystemExit.
ArithmeticError Base class for all errors that occur for numeric calculation.
OverflowError Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError Raised when a floating point calculation fails.

ZeroDivisionError Raised when division or modulo by zero takes place for all numeric types.
AssertionError Raised in case of failure of the Assert statement.
AttributeError Raised in case of failure of attribute reference or assignment.
EOFError Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError Raised when an import statement fails.
KeyboardInterrupt Raised when the user interrupts program execution, usually by pressing Ctrl+c.

LookupError

Base class for all lookup errors.

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.

UnboundLocalError

Raised when trying to access a local variable in a function or method but no value has been assigned to it.

EnvironmentError

Base class for all exceptions that occur outside the Python environment.

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.
IOError Raised for operating system-related errors.
SyntaxError Raised when there is an error in Python syntax.
IndentationError Raised when indentation is not specified properly.
SystemError Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.

SystemExit

Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

TypeError

Raised when an operation or function is attempted that is invalid for the specified data type.

ValueError

Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

RuntimeError

Raised when a generated error does not fall into any category.

NotImplementedError

Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

===

Q n:

Write a try:except: statement that attempts to open a file for reading and catches the exception thrown when the file does not exist.

Question: How do you find out the name of the exception that is thrown for an input/output error such as the failure to open a file ?

Normal Pgm

```
infile = open("c:/Pythan/abc.txt" 'r')
content = infile.read()
for lines in content:
    print(lines)
```

output

```
Traceback (most recent call last):
  File "C:\Users\Melcose\PycharmProjects\pythonProject\DataScienceMel.py", line 1, in <module>
    infile = open("c:/Pythan/abc.txt" 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'c:/Pythan/abc.txtr'
```

Pgm with try-except

```
try:
    infile = open("c:/Pythan/abc.txt" 'r')
    content = infile.read()
    for lines in content:
        print(lines)
except IOError:
    print("the file path is recoginzed")
```

Ans

If the file path is there, it will open, else the except part will work