Error Handling in Python

In any programming language there are 2 types of errors are possible.

1. Syntax Errors
2. Runtime Errors

Syntax Errors:

The errors which occurs because of **invalid syntax** are called syntax errors.

Eg 1:

```
x=10
if x==10
   print("Hello")
```

```
SyntaxError: invalid syntax
```

Eg 2:

```
print "Hello"
```

```
        SyntaxError: Missing parentheses in call to 'print'
```

Note:

Programmer is responsible to correct these syntax errors.

Once all syntax errors are corrected then only program execution will be started.

Runtime Errors:

- While executing the program if something goes wrong because of end user input or programming logic or memory problems etc then we will get Runtime Errors.
- Also known as **Exceptions**.

Eg: 1

```
print(10/0)      ==> ZeroDivisionError: division by zero
print(10/"ten") ==>TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Eg: Without Handling exception.

```
num = int(input("Enter a number: "))
print(num*num)
```

```
D:\PythonApps>python test.py
Enter a number: 10
100

D:\PythonApps>python test.py
Enter a number: ten
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    num = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: 'ten'
```

Eg: With Handling ValueError.

```
while True:
    try:
        num = int(input("Enter a number: "))
        print(num*num)
        break
    except ValueError:
        print('Please Enter only number')
```

```
D:\PythonApps>python test.py
Enter a number: ten
Please Enter only number
Enter a number: 10
100
```

Note: Exception Handling concept applicable for Runtime Errors but not for Syntax Errors

What is Exception?

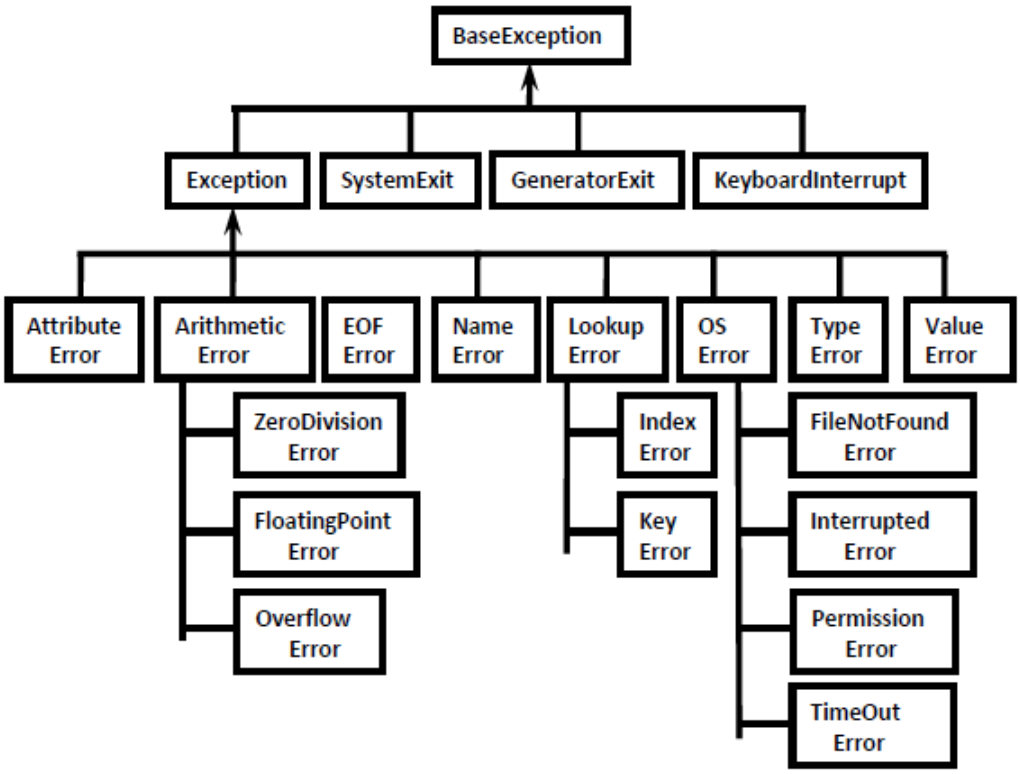
- An unwanted and unexpected event that disturbs normal flow of program is called exception.

Eg:

ZeroDivisionError
TypeError
ValueError
FileNotFoundError
EOFError
SleepingError
TyrePuncturedError

- It is highly recommended to handle exceptions.
- The main objective of exception handling is Graceful Termination of the program (i.e we should not block our resources and we should not miss anything)
- Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Python's Exception Hierarchy



- Every Exception in Python is a class.
- All exception classes are children of **BaseException**. i.e every exception class extends **BaseException** either directly or indirectly.
- Hence **BaseException** acts as root for Python Exception Hierarchy.

Note:
Most of the times being a programmer we have to concentrate **Exception** and its **child classes**.

How to handle the Exceptions:

- Exceptions are handled by using **try-except:** block.
- It is highly recommended to handle exceptions.
- The code which may raise exception is called **risky code** and we have to take risky code inside **try** block.
- The corresponding handling code we have to take inside **except** block.

Syntax:

```
try:
        Risky Code
except ExceptionName:
        Handling code/Alternative Code
```

Eg: without try-except:

```
print("Hello")
print(10/0)
print("Hi")
```

Output

Hello

ZeroDivisionError: division by zero

Abnormal termination/Non-Graceful Termination

Eg: with try-except:

```
print("Hello")

try:
    print(10/0)
except ZeroDivisionError:
    print("Can't divide a number by 0")

print("Hi")
```

```
Hello
Can't divide a number by 0
hi
```

Normal termination/Graceful Termination

How to print exception information:

```
try:
  print(10/0)
except ZeroDivisionError as msg:
  print("exception raised and its description is:",msg)
```

```
exception raised and its description is: division by zero
```

try with multiple except blocks:

- The way of handling exception is varied from exception to exception.
- Hence for every exception type a separate except block we have to provide.
  i.e try with multiple except blocks is possible and recommended to use.

Eg:

```
try:
   -------
   -------
   -------
except ZeroDivisionError:
        perform alternative
        arithmetic operations
except FileNotFoundError:
        use local file instead of remote file
```

Case 1:

If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

Eg:

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError :
   print("Can't Divide with Zero")
except ValueError:
   print("please provide int value only")
```

Case 2:

If try with multiple except blocks available then the order of these except blocks is important.
Python interpreter will always consider from top to bottom until matched except block identified.

Eg:

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ArithmeticError :
   print("ArithmeticError")
except ZeroDivisionError:
   print("ZeroDivisionError")
```

```
Enter First Number: 10
Enter Second Number: 0
ArithmeticError
```

Single except block that can handle multiple exceptions:

- We can write a single except block that can handle multiple different types of exceptions.

Syntax:

```
except (Exception1,Exception2,Exception3,..):
or
except (Exception1,Exception2,exception3,..) as msg :
```

Note: Parenthesis are mandatory and this group of exceptions internally considered as **tuple**.

Eg:

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except (ZeroDivisionError,ValueError) as msg:
   print("Plz Provide valid numbers only and problem is: ",msg)
```

```
Enter First Number: 10
Enter Second Number: 0
Plz Provide valid numbers only and problem is:  division by zero

Enter First Number: 10
Enter Second Number: ten
Plz Provide valid numbers only and problem is:  invalid literal for int() with base 10: 'ten'
```

Default except block:

- We can use default except block to handle any type of exceptions.
- In default except block generally we can print normal error messages.

Syntax:

```
except:
    statements
```

Eg:

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError:
   print("ZeroDivisionError:Can't divide with zero")
except:
   print("Default Except:Plz provide valid input only")
```

Enter First Number: 10
Enter Second Number: 0
ZeroDivisionError: Can't divide with zero

Enter First Number: 10
Enter Second Number: ten
Default Except: Plz provide valid input only

Note:
If try with multiple except blocks available then default except block should be last, otherwise we will get SyntaxError.
Eg:

```
try:
   print(10/0)
except:
   print("Default Except")
except ZeroDivisionError:
   print("ZeroDivisionError")
```

```
SyntaxError: default 'except:' must be last
```

Note:

The following are various possible combinations of except blocks

1. except ZeroDivisionError:
1. except ZeroDivisionError as msg:
3. except (ZeroDivisionError,ValueError) :
4. except (ZeroDivisionError,ValueError) as msg:
5. except :

finally block:

- It is not recommended to maintain **clean up code** (Resource De-allocating Code or Resource Releasing code) inside try block because there is no guarantee for the execution of every statement inside try block always.

- It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.

- Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but **finally block**.

- Hence the main purpose of finally block is to maintain **clean up code**.

```
try:
    Risky Code
except:
    Handling Code
finally:
    Cleanup code
```

- The specialty of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

Case-1: If there is no exception

```
try:
  print("try")
except:
  print("except")
finally:
  print("finally")
```

```
try
finally
```

Case-2: If there is an exception raised but handled:

```
try:
  print("try")
  print(10/0)
except ZeroDivisionError:
  print("except")
finally:
  print("finally")
```

```
try
except
finally
```

Case-3: If there is an exception raised but not handled:

```
try:
   print("try")
   print(10/0)
except NameError:
   print("except")
finally:
   print("finally")
```

```
try
finally
ZeroDivisionError: division by zero
```

Note:

There is only one situation where finally block won't be executed

i.e., whenever we are using os._exit(0) function.

Whenever we are using os._exit(0) function then PVM itself will be shutdown.

In this particular case finally won't be executed.

```
import os
try:
   print("try")
   os._exit(0)
except NameError:
   print("except")
finally:
   print("finally")
```

```
try
```

Note:

os._exit(0)

where 0 represents status code and it indicates normal termination

There are multiple status codes are possible.

Types of Exceptions:

- In Python there are 2 types of exceptions are possible.
  1. Predefined Exceptions
  2. User Defined Exceptions

Predefined Exceptions:

- Also known as in-built exceptions
- The exceptions which are raised automatically by **Python Virtual Machine** whenever a particular event **occurs, are called predefined exceptions.**

Eg 1:

Whenever we are trying to perform Division by zero, automatically Python will raise **ZeroDivisionError**.

```
print(10/0)
```

Eg 2:

Whenever we are trying to convert input value to int type and if input value is not int value then Python will raise ValueError automatically

```
x=int("ten")===>ValueError
```

User Defined Exceptions:

- Sometimes we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions are called **User Defined Exceptions or Customized Exceptions**.

- Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "**raise**" keyword.

- Also known as **Customized Exceptions or Programmatic Exceptions**

Eg:

InSufficientFundsException
InvalidInputException
TooYoungException
TooOldException

How to Define and Raise Customized Exceptions:

Every exception in Python is a class that extends **Exception** class either directly or indirectly.

Syntax:

```
class classname(Exception):
    def __init__(self,arg):
        self.msg=arg
```

Eg:

```
class TooYoungException(Exception):
    def __init__(self,arg):
        self.msg=arg
```

TooYoungException is our class name which is the child class of Exception

We can raise exception by using **raise** keyword as follows

```
raise TooYoungException("message")
```

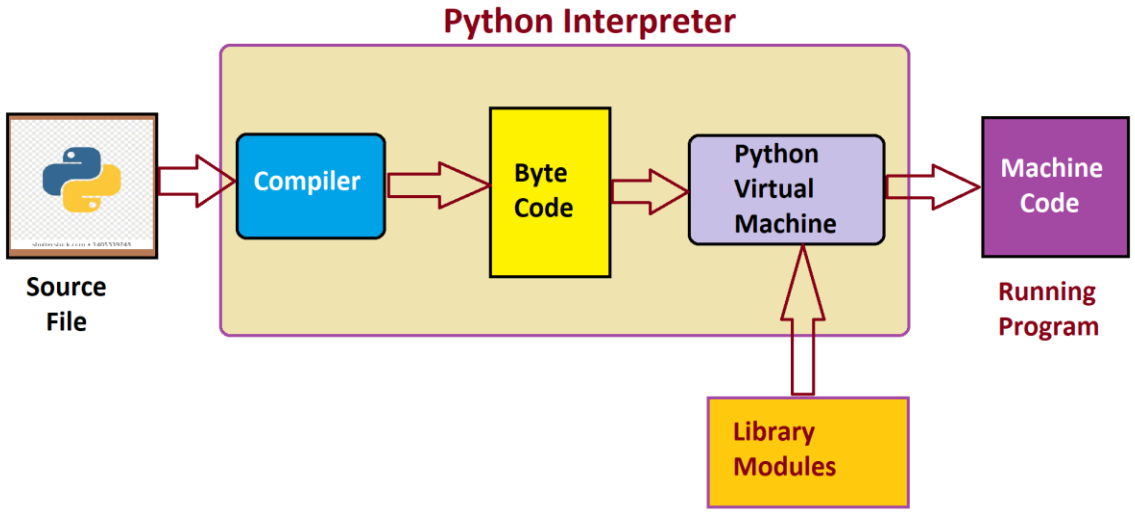
Eg:

```
class TooYoungException(Exception):
    def __init__(self,arg):
        self.msg=arg

class TooOldException(Exception):
    def __init__(self,arg):
        self.msg=arg

age=int(input("Enter Age:"))
if age>50:
    raise TooYoungException("Plz wait some more time you will get best match soon!!!")
elif age<18:
    raise TooOldException("Your age already crossed marriage age...no chance of getting marriage")
else:
    print("You will get match details soon by email!!!")
```

Note:

raise keyword is best suitable for **customized exceptions** but not for predefined exceptions

Python Internal Flow



Compiler:

- Compiler is a program that convert source code (.py file) into Byte Code.

Byte Code:

- The bytecode is a **low-level platform-independent** representation of your source code.
- In Python, the bytecode is stored in a **.pyc** file and these are stored in a folder named **__pycache__.**
- This folder is automatically created when you try to **import** another file that you created.

PVM:

- After compilation, the bytecode is sent for execution to the PVM.
- The PVM is an **interpreter** that is responsible to run the bytecode to Machine Code.
- Generally, PVM is specific to the target machine.
- The default implementation of PVM is **CPython** which is written in the **C programming language**.

Example:

mymath.py

```
def double(num):
    print(num+num)
```

first.py

```
import mymath

print(mymath.double(10))
```

>python first.py