Exception Handling

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

1. Syntax Errors
2. Runtime Errors

1. 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.

2. Runtime Errors:

Also known as exceptions.
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.

Eg: print(10/0) ==>ZeroDivisionError: division by zero

print(10/"ten") ==>TypeError: unsupported operand type(s) for /: 'int' and 'str'

```
x=int(input("Enter Number:"))
print(x)
```

```
D:\Python_classes>py test.py
```

```
    Enter Number:ten
        ValueError: invalid literal for int() with base 10: 'ten'
```

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.

Eg:

For example our programming requirement is reading data from remote file locating at London. At runtime if london file is not available then the program should not be terminated abnormally. We have to provide local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

```
try:

   read data from remote file locating at london
except FileNotFoundError:
   use local file and continue rest of the program  normally
```

Q. What is an Exception?
Q. What is the purpose of Exception Handling?
Q. What is the meaning of Exception Handling?

Default Exception Handing in Python:

Every exception in Python is an object. For every exception type the corresponding classes are available.

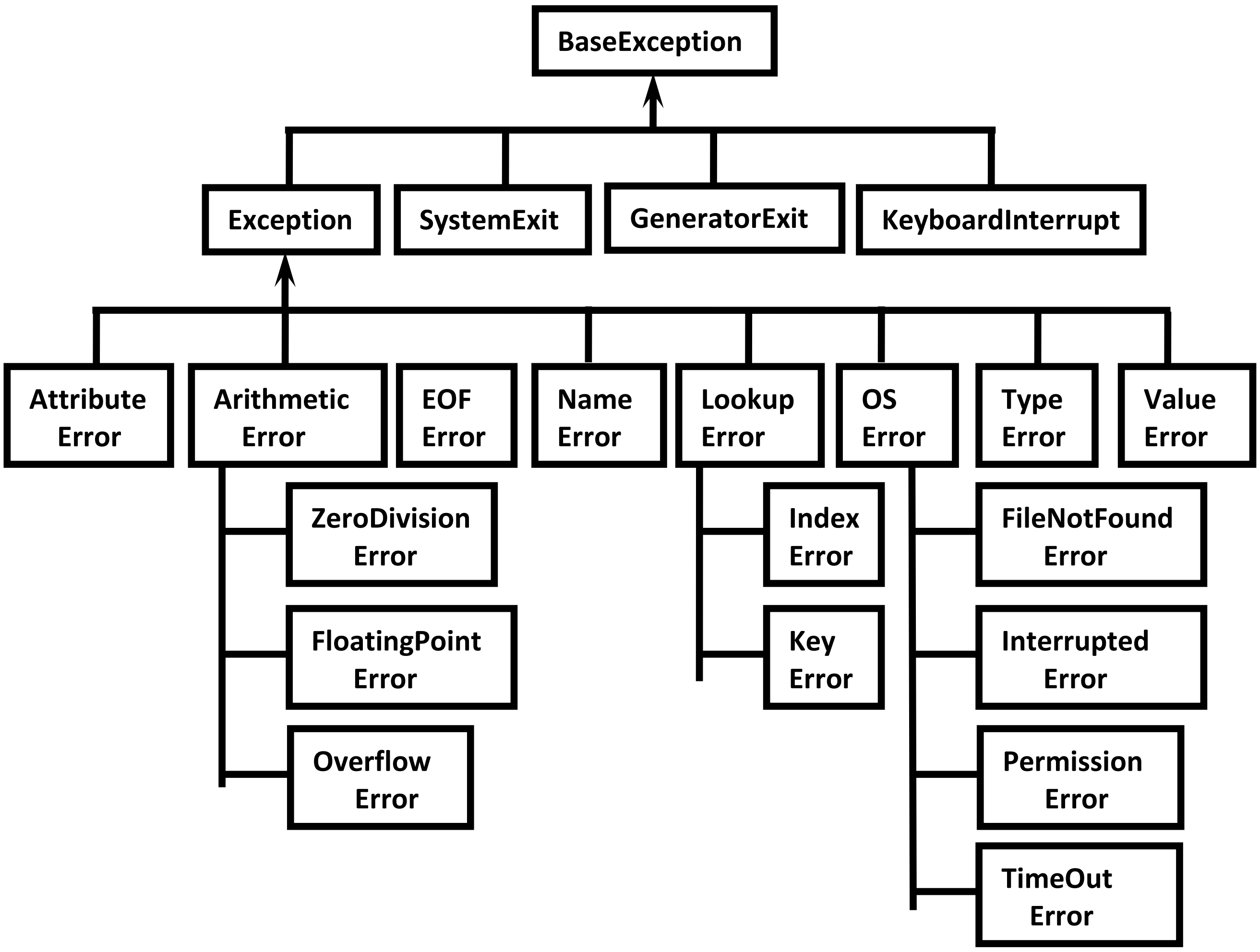
Whevever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.
The rest of the program won't be executed.

Eg:

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

D:\Python_classes>py test.py
Hello
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Python's Exception Hierarchy



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

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

Customized Exception Handling by using try-except:

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.

try:
 Risky Code
except XXX:
  Handling code/Alternative Code

without try-except:

```
print("stmt-1")
print(10/0)
print("stmt-3")

Output
stmt-1
ZeroDivisionError: division by zero
```

Abnormal termination/Non-Graceful Termination

with try-except:

```
print("stmt-1")
try:
    print(10/0)
except ZeroDivisionError:
    print(10/2)
print("stmt-3")

Output
stmt-1
5.0
stmt-3
```

Normal termination/Graceful Termination

Control Flow in try-except:

```
try:
            stmt-1
            stmt-2
            stmt-3
except XXX:
        stmt-4
stmt-5
```

<u>case-1:</u> If there is no exception
  1,2,3,5 and Normal Termination

<u>case-2:</u> If an exception raised at stmt-2 and corresponding except block matched
1,4,5 Normal Termination

<u>case-3:</u> If an exception raised at stmt-2 and corresponding except block not matched
1, Abnormal Termination

<u>case-4:</u> If an exception raised at stmt-4 or at stmt-5 then it is always abnormal termination.

Conclusions:

1. within the try block if anywhere exception raised then rest of the try block wont be executed eventhough we handled that exception. Hence we have to take only risky code inside try block and length of the try block should be as less as possible.

2. In addition to try block,there may be a chance of raising exceptions inside except and finally blocks also.

3. If any statement which is not part of try block raises an exception then it is always abnormal termination.

How to print exception information:

<u>try:</u>

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

Output 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 seperate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.

<u>Eg:</u>
```
try:
    -------
    -------
    -------
except ZeroDivisionError:
    perform alternative
    arithmetic operations
```

```
except FileNotFoundError:
    use local file instead of remote file
```

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")

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 2
5.0

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
Can't Divide with Zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
please provide int value only
```

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")

D:\Python_classes>py test.py
```

```
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.

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

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)

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
Plz Provide valid numbers only and problem is:  division by zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
Plz Provide valid numbers only and problem is:  invalid literal for int() with b
ase 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")

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
ZeroDivisionError:Can't divide with zero

D:\Python_classes>py test.py
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:

1. It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarentee for the execution of every statement inside try block always.

2. 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 speciality 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")

Output
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")

Output
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")

Output
try
finally
ZeroDivisionError: division by zero(Abnormal Termination)
```

***** Note:** **There is only one situation where finally block won't be executed ie whenever we are using os._exit(0) function.**

Whenever we are using os._exit(0) function then Python Virtual Machine itself will be shutdown.In this particular case finally won't be executed.

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

Output
try
```

Note:

os._exit(0)

where 0 represents status code and it indicates normal termination
There are multiple status codes are possible.

Control flow in try-except-finally:

```
try:
  stmt-1
  stmt-2
  stmt-3
except:
 stmt-4
```

```
finally:
  stmt-5
stmt6
```

Case-1: If there is no exception
1,2,3,5,6 Normal Termination

Case-2: If an exception raised at stmt2 and the corresponding except block matched
1,4,5,6 Normal Termination

Case-3: If an exception raised at stmt2 but the corresponding except block not matched
1,5 Abnormal Termination

Case-4: If an exception raised at stmt4 then it is always abnormal termination but before that finally block will be executed.

Case-5: If an exception raised at stmt-5 or at stmt-6 then it is always abnormal termination

Nested try-except-finally blocks:

We can take try-except-finally blocks inside try or except or finally blocks.i.e nesting of try-except-finally is possible.

```
try:
          ----------
          ----------
          ----------
     try:
           -------------
                --------------
                --------------
   except:
          --------------
          --------------
          --------------
   ------------
except:
        -----------
        -----------
        -----------
```

General Risky code we have to take inside outer try block and too much risky code we have to take inside inner try block. Inside Inner try block if an exception raised then inner

except block is responsible to handle. If it is unable to handle then outer except block is responsible to handle.

Eg:

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

Output
outer try block
Inner try block
Inner except block
Inner finally block
outer finally block
```

Control flow in nested try-except-finally:

```
try:
        stmt-1
        stmt-2
        stmt-3
        try:
                stmt-4
                stmt-5
                stmt-6
        except X:
                stmt-7
        finally:
                stmt-8
        stmt-9
except Y:
        stmt-10
finally:
        stmt-11
stmt-12
```

<u>case-1:</u> If there is no exception
1,2,3,4,5,6,8,9,11,12 Normal Termination

<u>case-2:</u> If an exception raised at stmt-2 and the corresponding except block matched
1,10,11,12 Normal Termination

<u>case-3:</u> If an exceptiion raised at stmt-2 and the corresponding except block not matched
1,11,Abnormal Termination

<u>case-4:</u> If an exception raised at stmt-5 and inner except block matched
1,2,3,4,7,8,9,11,12 Normal Termination

<u>case-5:</u> If an exception raised at stmt-5 and inner except block not matched but outer except block matched

1,2,3,4,8,10,11,12,Normal Termination

<u>case-6:</u>If an exception raised at stmt-5 and both inner and outer except blocks are not matched

1,2,3,4,8,11,Abnormal Termination

<u>case-7:</u> If an exception raised at stmt-7 and corresponding except block matched
1,2,3,.,.,.,8,10,11,12,Normal Termination

<u>case-8:</u> If an exception raised at stmt-7 and corresponding except block not matched
1,2,3,.,.,.,8,11,Abnormal Termination

<u>case-9:</u> If an exception raised at stmt-8 and corresponding except block matched
1,2,3,.,.,.,.,10,11,12 Normal Termination

<u>case-10:</u> If an exception raised at stmt-8 and corresponding except block not matched
1,2,3,.,.,.,.,11,Abnormal Termination

<u>case-11:</u> If an exception raised at stmt-9 and corresponding except block matched
1,2,3,.,.,.,.,8,10,11,12,Normal Termination

<u>case-12:</u> If an exception raised at stmt-9 and corresponding except block not matched
1,2,3,.,.,.,.,8,11,Abnormal Termination

<u>case-13:</u> If an exception raised at stmt-10 then it is always abonormal termination but before abnormal termination finally block(stmt-11) will be executed.

case-14: If an exception raised at stmt-11 or stmt-12 then it is always abnormal termination.

Note: If the control entered into try block then compulsary finally block will be executed. If the control not entered into try block then finally block won't be executed.

else block with try-except-finally:

We can use else block with try-except-finally blocks.
else block will be executed if and only if there are no exceptions inside try block.

```
try:
        Risky Code
except:
        will be executed if exception inside try
else:
        will be executed if there is no exception inside try
finally:
        will be executed whether exception raised or not raised and handled or not
handled
```

Eg:

```
try:
        print("try")
        print(10/0)--->1
except:
        print("except")
else:
        print("else")
finally:
        print("finally")
```

If we comment line-1 then else block will be executed b'z there is no exception inside try. In this case the output is:

```
try
else
finally
```

If we are not commenting line-1 then else block won't be executed b'z there is exception inside try block. In this case output is:

try
except
finally

Various possible combinations of try-except-else-finally:

1. Whenever we are writing try block, compulsory we should write except or finally block.i.e without except or finally block we cannot write try block.

2. Wheneever we are writing except block, compulsory we should write try block. i.e except without try is always invalid.

3. Whenever we are writing finally block, compulsory we should write try block. i.e finally without try is always invalid.

4. We can write multiple except blocks for the same try,but we cannot write multiple finally blocks for the same try

5. Whenever we are writing else block compulsory except block should be there. i.e without except we cannot write else block.

6. In try-except-else-finally order is important.

7. We can define try-except-else-finally inside try,except,else and finally blocks. i.e nesting of try-except-else-finally is always possible.

1	try: print("try")	✗
2	except: print("Hello")	✗

3	else:
	    print("Hello")
		✗

4	finally: print("Hello")	✗

5	try:
	    print("try")
	except:
	    print("except")
		✓

6	try:
	    print("try")
	finally:
	    print("finally")
		✓

7	try: print("try")	✓

	except:
	        print("except")
	else:
	        print("else")	

8	try:
	        print("try")
	else:
	        print("else")
		✗

9	try:
	        print("try")
	else:
	        print("else")
	finally:
	        print("finally")
		✗

10	try:
	      print("try")
	except XXX:
	      print("except-1")
	except YYY:
	      print("except-2")
		✓

11	try:
	        print("try")
	except :
	     print("except-1")
	else:
	        print("else")
	else:
	        print("else")
		✗

12	try:
	        print("try")
	except :
	        print("except-1")
	finally:
	        print("finally")
	finally:
	        print("finally")
		✗

13	try:
	        print("try")
	        print("Hello")
	except:
	        print("except")
		✗

14	try: print("try") except:	✗

	print("except")
	print("Hello")
	except:
	    print("except")	

15	try:
	    print("try")
	except:
	    print("except")
	print("Hello")
	finally:
	    print("finally")
		✗

16	try:
	    print("try")
	except:
	    print("except")
	print("Hello")
	else:
	    print("else")
		✗

17	try:
	    print("try")
	except:
	    print("except")
	try:
	    print("try")
	except:
	    print("except")
		✓

18	try:
	    print("try")
	except:
	    print("except")
	try:
	    print("try")
	finally:
	    print("finally")
		✓

19	try:
	    print("try")
	except:
	    print("except")
	if 10>20:
	    print("if")
	else:
	    print("else")
		✗

20	try: print("try")	✓

	try:
	    print("inner try")
	  except:
	    print("inner except block")
	  finally:
	  print("inner finally block")
	except:
	    print("except")	

21	try:
	    print("try")
	
	except:
	    print("except")
	    try:
	      print("inner try")
	    except:
	      print("inner except block")
	    finally:
	  print("inner finally block")
		✓

22	try:
	    print("try")
	except:
	    print("except")
	finally:
	    try:
	    print("inner try")
	    except:
	      print("inner except block")
	    finally:
	      print("inner finally block")
		✓

23	try:
	    print("try")
	except:
	    print("except")
	try:
	    print("try")
	else:
	    print("else")
		✗

24	try:
	    print("try")
	    try:
	      print("inner try")
	except:
	    print("except")
		✗

25	try:
	        print("try")
	else:
	        print("else")
	except:
	        print("except")
	finally:
	        print("finally")
		✗

Types of Exceptions:

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

1. Predefined Exceptions:

Also known as in-built exceptions

The exceptions which are raised automatically by Python virtual machine whenver a particular event occurs, are called pre defined 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
```

2. User Defined Exceptions:

Also known as Customized Exceptions or Programatic Exceptions

Some time 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.

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(predefined exception class name):
      def __init__(self,arg):
            self.msg=arg
```

Eg:

```python
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:

```python
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>60:
   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!!!")

D:\Python_classes>py test.py
```

```
Enter Age:90
__main__.TooYoungException: Plz wait some more time you will get best match soon!!!

D:\Python_classes>py test.py
Enter Age:12
__main__.TooOldException: Your age already crossed marriage age...no chance of g
etting marriage

D:\Python_classes>py test.py
Enter Age:27
You will get match details soon by email!!!
```

Note:

raise keyword is best suitable for customized exceptions but not for pre defined exceptions

PYTHON LOGGING

Logging the Exceptions:

It is highly recommended to store complete application flow and exceptions information to a file. This process is called logging.

The main advantages of logging are:

1. We can use log files while performing debugging
2. We can provide statistics like number of requests per day etc

To implement logging, Python provides one inbuilt module logging.

logging levels:

Depending on type of information, logging data is divided according to the following 6 levels in Python.
 table

1. CRITICAL==>50==>Represents a very serious problem that needs high attention
2. ERROR===>40===>Represents a serious error
3. WARNING==>30==>Represents a warning message, some caution needed. It is alert to the programmer
4. INFO===>20===>Represents a message with some important information
5. DEBUG===>10==>Represents a message with debugging information
6. NOTSET==>0==>Rrepresents that the level is not set.

By default while executing Python program only WARNING and higher level messages will be displayed.

How to implement logging:

To perform logging, first we required to create a file to store messages and we have to specify which level messages we have to store.

We can do this by using basicConfig() function of logging module.

```
logging.basicConfig(filename='log.txt',level=logging.WARNING)
```

The above line will create a file log.txt and we can store either WARNING level or higher level messages to that file.

After creating log file, we can write messages to that file by using the following methods.

logging.debug(message)
logging.info(message)
logging.warning(message)
logging.error(message)
logging.critical(message)

Q.Write a Python program to create a log file and write WARNING and higher level messages?

```
import logging
logging.basicConfig(filename='log.txt',level=logging.WARNING)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

log.txt:

```
WARNING:root:This is warning message
ERROR:root:This is error message
CRITICAL:root:This is critical message
```

Note:

In the above program only WARNING and higher level messages will be written to log file. If we set level as DEBUG then all messages will be written to log file.

```
import logging
logging.basicConfig(filename='log.txt',level=logging.DEBUG)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

log.txt:

```
DEBUG:root:This is debug message
INFO:root:This is info message
```

```
WARNING:root:This is warning message
ERROR:root:This is error message
CRITICAL:root:This is critical message
```

Note: We can format log messages to include date and time, ip address of the client etc at advanced level.

How to write Python program exceptions to the log file:

By using the following function we can write exceptions information to the log file.

logging.exception(msg)

Q. Python Program to write exception information to the log file

```
import logging
logging.basicConfig(filename='mylog.txt',level=logging.INFO)
logging.info("A New request Came:")
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError as msg:
   print("cannot divide with zero")
   logging.exception(msg)
except ValueError as msg:
   print("Enter only int values")
   logging.exception(msg)
logging.info("Request Processing Completed")


D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 2
5.0

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
cannot divide with zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
Enter only int values
```

mylog.txt:

```
INFO:root:A New request Came:
INFO:root:Request Processing Completed
INFO:root:A New request Came:
ERROR:root:division by zero
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(x/y)
ZeroDivisionError: division by zero
INFO:root:Request Processing Completed
INFO:root:A New request Came:
ERROR:root:invalid literal for int() with base 10: 'ten'
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    y=int(input("Enter Second Number: "))
ValueError: invalid literal for int() with base 10: 'ten'
INFO:root:Request Processing Completed
```

PYTHON DEBUGGING BY USING ASSERTIONS

Debugging Python Program by using assert keyword:

The process of identifying and fixing the bug is called debugging.
Very common way of debugging is to use print() statement. But the problem with the print() statement is after fixing the bug,compulsory we have to delete the extra added print() statments,otherwise these will be executed at runtime which creates performance problems and disturbs console output.

To overcome this problem we should go for assert statement. The main advantage of assert statement over print() statement is after fixing bug we are not required to delete assert statements. Based on our requirement we can enable or disable assert statements.

Hence the main purpose of assertions is to perform debugging. Usully we can perform debugging either in development or in test environments but not in production environment. Hence assertions concept is applicable only for dev and test environments but not for production environment.

Types of assert statements:

There are 2 types of assert statements

1. Simple Version
2. Augmented Version

1. Simple Version:

assert conditional_expression

2. Augmented Version:

assert conditional_expression,message

conditional_expression will be evaluated and if it is true then the program will be continued.
If it is false then the program will be terminated by raising AssertionError.
By seeing AssertionError, programmer can analyze the code and can fix the problem.

Eg:

```
def squareIt(x):
    return x**x
```

```
assert squareIt(2)==4,"The square of 2 should be 4"
assert squareIt(3)==9,"The square of 3 should be 9"
assert squareIt(4)==16,"The square of 4 should be 16"
print(squareIt(2))
print(squareIt(3))
print(squareIt(4))

D:\Python_classes>py test.py
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    assert squareIt(3)==9,"The square of 3 should be 9"
AssertionError: The square of 3 should be 9

def squareIt(x):
    return x*x
assert squareIt(2)==4,"The square of 2 should be 4"
assert squareIt(3)==9,"The square of 3 should be 9"
assert squareIt(4)==16,"The square of 4 should be 16"
print(squareIt(2))
print(squareIt(3))
print(squareIt(4))

Output
4
9
16
```

Exception Handling vs Assertions:

Assertions concept can be used to alert programmer to resolve development time errors.

Exception Handling can be used to handle runtime errors.