EXCEPTIONAL HANDLING

EXCEPTION: An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash. Exceptions are convenient in many ways for handling errors and special conditions in a program.

EXCEPTIONAL HANDLING: If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax:
```
try:
  statements;
  statements;
  statements;
except:
  statements;
  statements;
  statements;
```

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
You can also provide a generic except clause, which handles any exception.
After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
The else-block is a good place for code that does not need the try: block's protection.

These are the list of standard expections

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.

EXCEPTIONAL CLAUSE: In Python, exceptions can be handled using a try statement.
A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause. It is up to us, what operations we perform once we have caught the exception.

Syntax: try:
```
    statements;
    statements;
    statements;
  except:
    statements;
    statements;
    statements;
```

E.G:
```
import sys
randomList = ['a', 0, 2]
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Next entry.")
        print()
    print("The reciprocal of",entry,"is",r)
```

Output:
```
The entry is a
Oops! <class 'ValueError'> occured.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

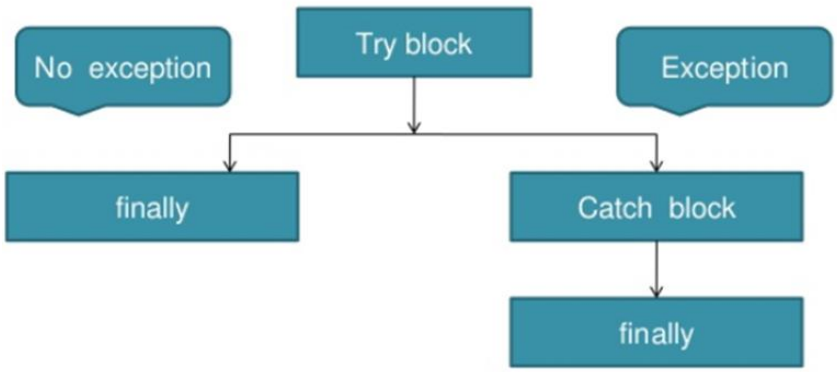
TRY FINAL CLAUSE: You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.

Syntax

```
try:
        statements;
        statements;
        statements;
    finally:
        statements;
        statements;
        statements;
```

E.g:

```
try:
    val = int(raw_input("how many kids have you? "))
except StandardError, e:
    print "Error - ",e
    return "yer wot?"
finally: # A finally block is ALWAYS run, even if you return before it
    print "T time"
print val
return val
```



USER_DEFINED EXCEPTIONS: 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 form this class.

Syntax :

```
class MyError(Exception): # class MyError is derived from super class Exception
 def __init__(self, value):     # Constructor or Initializer
        self.value = value    # __str__ is to print() the value
 def __str__(self):
        return(repr(self.value))
try:
        raise(MyError(3*2))
except MyError as error: # Value of Exception is stored in error
```

```
print('A New Exception occured: ',error.value)
```

Deriving Error from Super Class Exception: Super class Exceptions are created when a module needs to handle several distinct errors. One of the common way of doing this is to create a base class for exceptions defined by that module. Further, various subclasses are defined to create specific exception classes for different error conditions.

E.G:

```
# class Error is derived from super class Exception
class Error(Exception):

   # Error is derived class for Exception, but
   # Base class for exceptions in this module
   pass

class TransitionError(Error):

   # Raised when an operation attempts a state
   # transition that's not allowed.
   def __init__(self, prev, nex, msg):
      self.prev = prev
      self.next = nex

      # Error message thrown is saved in msg
      self.msg = msg
try:
   raise(TransitionError(2,3*2,"Not Allowed"))

# Value of Exception is stored in error
except TransitionError as error:
   print('Exception occured: ',error.msg)
```

Using standard exception as base class: Runtime error is a class is a standard exception which is raised when a generated error does not fall into any category. In a similar way, any exception can be derived from the standard exceptions of Python.

E.g:

```
#NetworkError has base RuntimeError
# and not Exception
class Networkerror(RuntimeError):
   def __init__(self, arg):
      self.args = arg

try:
   raise Networkerror("Error")

except Networkerror as e:
   print (e.args)
```