Exception Handling

In programming development, there may be some cases where the programmer does not want to go, those cases are called errors.

In general errors are of two types

i) Compile-time error

These errors resulting out of violation of programming language syntax rule, ie writing syntactically incorrect statement

print(“A” + 2)

will result into compile time error because of invalid syntax.

ii) Run-time error

These errors occur during runtime because of unexpected situations, for e.g. accesses to resources that do not exist or because it gets out of an unexpected range etc. Such errors are called Exceptions. It can be handled through exception handling routine of Python.

So we can summarize Exception as:

It is an exceptional event that occur during runtime and causes normal program flow to disrupted.

For instance consider the following code:

```
>>> print(3/0)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(3/0)
ZeroDivisionError: division by zero
```

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

This message is generated by default exception handler of Python. The default exception handler does the following upon occurrence of an Exception:

i) Prints out exception description

ii) Prints the stack trace ie, hierarchy of methods where the exception occurred.

iii) Causes the program to terminate.

Exception Handling in Python

Exception Handling in Python involves the use of **try** and **except** clause in the following format

```
try:

  # write here the code that may generate an exception

except:

  #write code here about what to do when the exception has occurred
```

```
try:

        x = int("XII")

except:

        print("Error converting XII to a number")
```

the output of above code

- **Error converting XII to a number**

Consider one more example

```
try:

        my_file = open("myfine.txt", "r")
```

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

```
    print(myfile.read())
except:
        print("Error opening in file")
```

Now the above code may give the error message because of two reasons

i) The file did not exist or
ii) There was no data in the file

But the above code did not tell which caused the error.

To resolve the above issue we can use second argument with except block

```
try:
    # code here the code that may generate an exception
except <ExceptionName>, <exArgument>:
    #handler error here
```

The except clause can use this additional argument to print the associated error-message of this exception as <exArgument>.message.

```
try:
        my_file = open("myfine.txt", "r")
    print(myfile.read())
exceptIOError e:
        print("Exception : ", e.message)
```

here IOError is built in class of Python to manage input/ouput Exception related with files. In this way Python has given built-in library classes to handle almost every exception situation. Even though we can create our own Exception classes also.

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

Some built in exception classes in Python are as follows

EOFError
IOError
NameError
IndexError
ImportError
TypeError
ValueError
ZeroDivisionError
OverflowError
KeyError

Handling Multiple Errors

Multiple types of errors may be captured and processed differently. It can be useful to provides a more exact error message to the used than a simple message. In order to capture and process different type of exceptions, there must be multiple exception block – each one pertaining to different type of exception.

```
try:
    #:
except <ExceptionName1>:
    #
except <ExceptionName2>:
    #
except <ExceptionName3>:
    #
else:
```

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

```
    #if there is no exception then the statements in this
block get executed.
```

The finally block

You can use a finally: block along with a try: block, just like you use except: block

syntax

```
try:
    #:
[except <ExceptionName1>:
    #]
finally:
    #statement that will always run
```

You can use finally: block with try: block, without using except: block, even you can combine all three together,

```
try:
    fh = open("poems.txt", "r+")
    fh.write("Adding new lines")
finally:
```

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

```
    print("Error : can\'t find file or read data")

    # this block will always be execute, whether the try:
    raised an exception or #not.
```

You may also combine finally: with except: clause. In such a combination, the except: block will get executed only in case an exception is raised and finally: block will get executed ALWAYS in the end.

```
try:

    fh = open("poems.txt", "r+")

    fh.write("Adding new lines")

except:

    print("Exception Occured")

finally:

    print("Finally says good bye")
```

Raising/Forcing an Exception

In Python, you can use the **raise** keyword to raise/force an exception. That means, you as programmer can force an exception to occur through **raise** keyword. It can also pass a custom message to your exception handling module.

```
raise <exception>([ Message])
```

Consider the following code that forces a ZeroDivisionError exception

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

```
try:
        n = int(input("Enter numerator "))
        d = int(input("Enter denominator "))
        if d == 0:
                raise ZeroDivisionError(str(n) + "/0 is not
possible ")
        print (n / d)
except ZeroDivisionError as e:
        print("Exception : ", e)
```

The output of above code is :

Enter numerator 23

Enter denominator 0

Exception : 23/0 is not possible

Raising and Creating User-defined Exception

A user – define exception can also be raised through raise statement as per following syntax:

raise <user-define-exception>

Before you raise a user define exception, you need to define a custom Exception class as follows

```
class <user-define-exception>(Exception):
    pass
```

Then you need to create user-defined exceptions as objects of above defined class, eg.

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

```
class MyError(Exception):
    pass
myError1 = MyError("Custom Message")
```

Now consider the following code:

```
class MyError(Exception):
    pass
try:
    marks = float(input("Enter marks : "))
    if marks <0:
        raise MyError("Negative marking not allowed")
    elif marks >100:
        raise MyError("Maximum marks are 100")
    elif marks > 40:
        print("Promoted to next class")
    else:
        print("Your re-examination is scheduled")
except MyError as e:
    print(e, "; Marks entered :", marks)
```

The Output of above code is

```
Enter marks : 65
Promoted to next class
```

This is proprietary content of SSi Indore, Unauthorized use of this is restricted

```
>>>

Enter marks : -32

Negative marking not allowed ; Marks entered : -32.0

>>>

Enter marks : 112

Maximum marks are 100 ; Marks entered : 112.0

>>>

Enter marks : 33

Your re-examination is scheduled
```

This is proprietary content of SSi Indore, Unauthorized use of this is restricted