Module 5:

Exception Handling

By Ninad Gaikwad

Reference - “Core Python Programming”
Dr. R. Nageshwara Rao
Dreamtech Press

Types of Errors

- Errors in a program are classified in three types:
  - Compile time errors
  - Runtime errors
  - Logical errors

Compile time errors

- Syntactical errors found in code due to which program fails to compile
- Detected by python compiler
- Forgetting to write colon in an if statement after condition
- Eg.

  if x ==1

  ‘’’Output

  File "<string>", line 1

  if x ==1

  ^

  SyntaxError: invalid syntax ‘’’

Runtime Errors

- When PVM cannot execute the bite code , it flags a runtime error
- Eg Insufficient memory to store something
- Not detected by compiler, detected by PVM at runtime
- Eg.

```
def concat(a,b):
    print(a+b)

concat('Hai', 25)
```

‘’Output

```
Traceback (most recent call last):
  File "<string>", line 4, in <module>
File "<string>", line 2, in concat
TypeError: can only concatenate str (not "int") to str ’”
```

Logical Errors

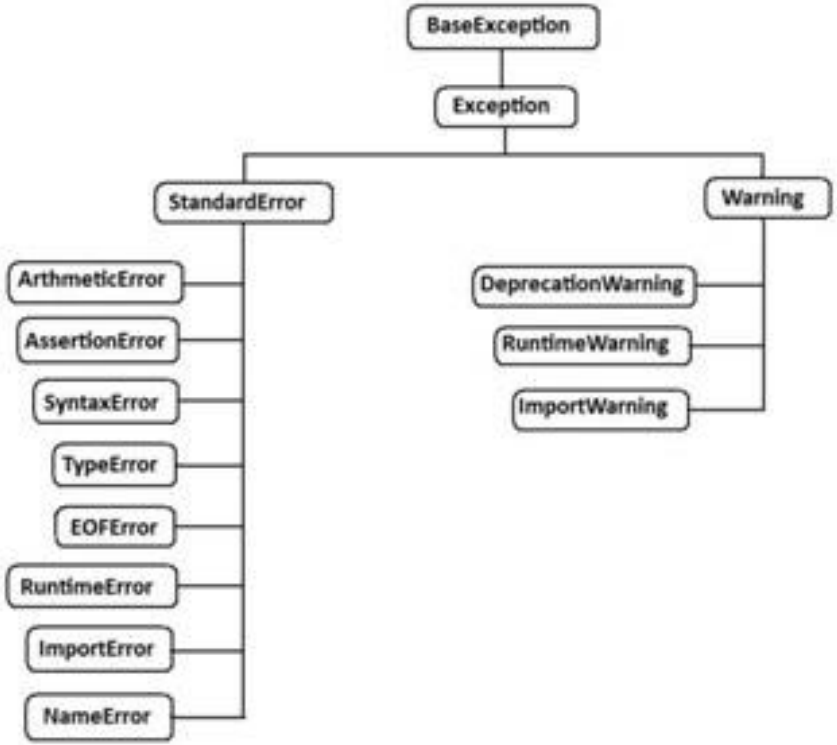
- Depict flaws in logic of the program
- Not detected by compiler or PVM, programmer is solely responsible for them
- Eg.

```
def increment(sal):
    sal = sal * 15/100 # should be sal + sal*15/100
    return sal
```

- When there is an error in a program, due to sudden termination, following things can be suspected
  - Important data in files or databases used in program may be lost
  - Software may be corrupted
  - Program terminates abruptly giving error to user making user losing trust in software

16.1 Exceptions

- Runtime errors which can be handled by the programmer
- Exceptions that cannot be handled are called errors
- The programmer can guess the occurrence of the exception and do something to eliminate the harm caused by it
- Exceptions are represented as classes in python
- There are built in functions for which BaseException is the base class
- Exception hierarchy in python



16.2 Exception Handling

- Purpose of error handling is to make program robust
- Robust program does not terminate in the middle
- On occurrence of error the program displays appropriate message
- Steps to handle exceptions:
    a. Use try block where there is a possibility of occurrence of errors
    b. Use an except block where the programmer displays the exception details to the user
    c. Use finally block to perform clean up operations like closing files and terminating running processes
- Types of exceptions: Exception, ArithmeticError, AssertionError, AttributeError, EOFError, FloatingPointError, GeneratorExit, IOError, Import Error, IndexError, KeyError, KeyboardInterrupt, NameError, NorImplementedError, OverflowError, RuntimeError, StopIteration, SyntaxError, IndentationError, SystemExit, TypeError, UnboundLocalError, ValueError, ZeroDivisionError

16.2 Exception Handling

- Eg.

```
try:
    f=open("myfile","w")
    a, b = [int(x) for x in input("Enter two numbers ").split()]
    c = a/b
    f.write("writing %d into file" %c)
except ZeroDivisionError:
    print("Division by zero occurred")
finally:
    f.close()
    print("File closed")
‘’’ Output
 C:\Users\Administrator\Desktop>python demo.py
Enter two numbers 10 2
File closed

C:\Users\Administrator\Desktop>python demo.py
Enter two numbers 10 0
Division by zero occurred
File closed ‘’’
```

16.3 The Except Block

- Used to catch the exception that is raised by the try block
  Eg. except ExceptionClass:
- We can catch exception as an object that contains some description about the exception
  Eg. except ExceptionClass as obj:
- To catch multiple exceptions we can write multiple except blocks
  Eg. except (ExceptionClass1, ExceptionClass2, …):
- To catch any type of exception where we are not bothered about exception type, write exception block without mentioning the exception
  Eg. except:
- When we need to use the try block alone we need to follow it with finally block
  Eg.

```
try:
        x=int(input('Enter a number'))
        y = 1/x
finally:
        print('we are not catching the exception')
print('the inverse is :' , y)
```

16.4 The assert statement

- Useful to ensure that the given condition is true
- If it is not true it raises AssertionError
- Syntax: assert condition, message
- Eg

```
try:
    x = int(input('enter number between 5 and 10: '))
    assert x>=5 and x<=10
    print("the number entered ",x)
except AssertionError:
    print('the condition is not fulfilled')
```

‘’’ output
enter number between 5 and 10: 12
the condition is not fulfilled ‘’’

OR

16.4 The assert statement

```
try:
    x = int(input('enter number between 5 and 10: '))
    assert x>=5 and x<=10, "your input is not correct"
    print("the number entered ",x)
except AssertionError as obj:
    print(obj)
‘’Output
enter number between 5 and 10: 12
your input is not correct ‘’’
```

16.5 User defined exceptions

- User defined exception should be a class which is a subclass to the built-in ‘Exception’ class
- This class has a constructor where msg is defined
- Eg.

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

- A programmer has to raise his exception when he suspects its possibility
- Eg.

```
raise MyException(‘message’)
```

- A programmer can insert a try block and catch the exception using the except block
- Eg.

```
try:
        Code
except MyException as me:
        print(me)
```

16.5 User defined exceptions

- Example # creating your own class as a sub class to exception class

```
class MyException(Exception):
    def __init__(self, arg):
        self.msg = arg
# raising the exception
def check(dict):
    for k, v in dict.items():
        print('name = {:15s} balance = {:10.2f}'.format(k, v))
        if v<2000:
            raise MyException('Balance is less in account of '+k)
bank = {'raj':2200,'vani':8500,'ajay':1990}
try:
    check(bank)
except MyException as me:
    print(me)
“”” Output
name = raj            balance =    2200.00
name = vani           balance =    8500.00
name = ajay           balance =    1990.00
Balance is less in account of ajay “””
```

16.6 Logging the Exceptions

- We can store the errors raised by the program in files
- This helps programmers to quantify and locate the errors
- Python provides “logging” module to create a log file that can store all error messages
- Depending on the seriousness of the error they are classified into 6 levels:

Level	Numeric Value	Description
CRITICAL	50	Very serious error that needs high attention
ERROR	40	Serious error
WARNING	30	Warning message. Some caution needed
INFO	20	Message with some important information
DEBUG	10	Message with debugging information
NOTSET	0	Level is not set

16.6 Logging the Exceptions

- The errors above the specified level of seriousness are saved others are not saved
- Eg.

```
import logging
# store messages into mylog.txt file
logging.basicConfig(filename='mylog.txt', level= logging.ERROR)

# these are stored in the file
logging.error("There is an error in the program")
logging.critical("There is a problem in design")

# but these are not saved
logging.warning("The project is going slow")
logging.info("You are a junior programmer")
logging.debug("Line 10 contains syntax error")
“”” Output (in mylog.txt)
There is an error in the program
There is a problem in design
```

16.6 Logging the Exceptions

- import logging module to save exceptions in the file
- We use exception method to log messages in the file
- Eg.

```
import logging

# store logging messages in log.txt file
logging.basicConfig(filename='log.txt', level= logging.ERROR)
try:
    a=int(input('Enter a number'))
    b=int(input('Enter another number'))
    c=a/b

except Exception as e:
    logging.exception(e)
else:
    print('the result of division is ',c)
```

16.6 Logging the Exceptions

“””Output **(CMD)**
C:\Users\Administrator\Desktop>python demo.py
Enter a number10
Enter another number20
the result of division is 0.5
C:\Users\Administrator\Desktop>python demo.py
Enter a number10
Enter another number0
C:\Users\Administrator\Desktop>python demo.py
Enter a number10
Enter another numberab
(Log file)
ERROR:root:division by zero
Traceback (most recent call last):
  File "C:\Users\Administrator\Desktop\demo.py", line 8, in <module>
    c=a/b
ZeroDivisionError: division by zero
ERROR:root:invalid literal for int() with base 10: 'ab'
Traceback (most recent call last):
  File "C:\Users\Administrator\Desktop\demo.py", line 7, in <module>
    b=int(input('Enter another number'))
ValueError: invalid literal for int() with base 10: 'ab'

Experiment 10:

- Write a program to demonstrate exception handling using try,multiple except and finally.
- Write a python program to create user defined exceptions