22 Exception Handling

"Expect an exception and prepare for it"

Contents

What may go Wrong?

- While creating and executing a Python program things may go wrong at two different stages—during compilation and during execution.
- Errors that occur during compilation are called **Syntax Errors**. Errors that occur during execution are called **Exceptions**.

Syntax Errors

- If things go wrong during compilation:

  Means - Something in the program is not as per language grammar
  Reported by - Interpreter/Compiler
  Action to be taken - Rectify program

- Examples of syntax errors:

```
print 'Hello'          # ( ) is missing
d = 'Nagpur'
a = b + float(d)       # d is a string, so it cannot be converted to float
a = Math.pow(3)        # pow( ) needs two arguments
```

- Other common syntax error are:
  - Leaving out a symbol, such as a colon, comma or brackets
  - Misspelling a keyword
  - Incorrect indentation
  - Empty if, else, while, for, function, class, method
  - Missing :
  - Incorrect number of positional arguments
- Suppose we try to compile the following piece of code:

```
basic_salary = int ( input('Enter basic salary') )
if basic_salary < 5000
    print('Does not qualify for Diwali bonus')
```

We get the following syntax error:

```
File 'c:\Users\Kanetkar\Desktop\Phone\src\phone.py', line 2
    if basic_salary < 5000
```

```
                ^
SyntaxError: invalid syntax
```

- ^ indicates the position in the line where an error was detected. It occurred because : is missing after the condition.
- Filename and line number are also displayed to help you locate the erroneous statement easily.

Exceptions

- If things go wrong during execution (runtime):

  Means - Something unforeseen has happened
  Reported by - Python Runtime
  Action to be taken - Tackle it on the fly

- Examples of Runtime errors:

  Memory Related - Stack/Heap overflow, Exceeding bounds
  Arithmetic Related - Divide by zero
  Others - Attempt to use an unassigned reference, File not found

- Even if the program is grammatically correct, things may go wrong during execution causing exceptions.

```
a = int(input('Enter an integer: '))
b = int(input('Enter an integer: '))
c = a / b
```

If during execution of this script we give value of b as 0, then following message gets displayed:

```
Exception has occurred: ZeroDivisionError
division by zero
File 'C:\Users\Kanetkar\Desktop\Phone\src\trial.py', line 3, in
<module> c = a / b
# blah blah... rest of the stack trace showing how we landed here
```

- Another example of exception:

```
a, b = 10, 20
c = a / b * d
```

```
File 'c:\Users\Kanetkar\Desktop\Phone\src\phone.py', line 2, in <module>  c = a / b * d
NameError: name 'd' is not defined
# blah blah... rest of the stack trace showing how we landed here
```

- The stack trace prints the names of the files, line numbers starting from the first file that got executed, up to the point of exception.
- The stack trace is useful for the programmer to figure out where things went wrong. However, a user is likely to get spooked looking at it, thinking something is very wrong. So we should try and tackle the exceptions ourselves and provide a graceful exit from the program, instead of printing the stack trace.

How to deal with Exceptions?

- **try** and **except** blocks are used to deal with an exception.
- Statement(s) which you suspect may go wrong at runtime should be enclosed within a **try** block.
- If while executing statement(s) in **try** block, an exceptional condition occurs it can be tackled in two ways:
  (a) Pack exception information in an object and raise an exception.
  (b) Let Python Runtime pack exception information in an object and raise an exception.

  In the examples in previous section Python Runtime raised exceptions **ZeroDivisionError** and **NameError**.

  Raising an exception is same as throwing an exception in C++/Java.
- Two things that can be done when an exception is raised:
  (a) Catch the raised exception object in **except** block.
  (b) Raise the exception further.
- If we catch the exception object, we can either perform a graceful exit or rectify the exceptional situation and continue execution.
- If we raise the exception object further - Default exception handler catches the object, prints stack trace and terminates.
- There are two ways to create exception objects:
  (a) From ready-made exception classes (like **ZeroDivisionError**)
  (b) From user-defined exception classes

- Advantages of tackling exceptions in OO manner:
  - More information can be packed into exception objects.
  - Propagation of exception objects from the point where they are raised to the point where they are tackled is managed by Python Runtime.
- Python facilitates exception handling by providing:
  - Keywords **try**, **except**, **else**, **finally**, **raise**.
  - Readymade exception classes.

How to use *try - except*?

- **try** block - Enclose in it the code that you anticipate will cause an exception.
- **except** block - Catch the raised exception in it. It must immediately follow the **try** block.

```
try :
    a = int(input('Enter an integer: '))
    b = int(input('Enter an integer: '))
    c = a / b
    print('c =', c)
except ZeroDivisionError :
    print('Denominator is 0')
```

Given below is the sample interaction with the program:

```
Enter an integer: 10
Enter an integer: 0
Denominator is 0
```

- If no exception occurs while executing the **try** block, control goes to first line beyond the **except** block.
- If an exception occurs during execution of statements in **try** block, an exception is raised and rest of the **try** block is skipped. Control now goes to the **except** block. Here, if the type of exception raised matches the exception named after **except** keyword, that **except** block is executed.
- If an exception occurs which does not match the exception named in **except** block, then the default exception handler catches the exception, prints stack trace and terminates execution.

- When exception is raised and **except** block is executed, control goes to the next line after **except** block, unless there is a **return** or **raise** in **except** block.

Nuances of *try* and *except*

- **try** block:
  - Can be nested inside another **try** block.
  - If an exception occurs and if a matching except handler is not found in the **except** block, then the outer **try**'s **except** handlers are inspected for a match.
- **except** block:
  - Multiple **except** blocks for one **try** block are ok.
  - At a time only one **except** block goes to work.
  - If same action is to be taken in case of multiple exceptions, then the except clause can mention these exceptions in a tuple

```
try :
    # some statements
except (NameError, TypeError, ZeroDivisionError) :
    # some other statements
```

  - Order of **except** blocks is important - Derived first, Base last.
  - An empty **except** is like a catchall—catches all exceptions.
  - An exception may be re-raised from any **except** block.
- Given below is a program that puts some of the **try**, **except** nuances to a practical stint:

```
try :
    a = int(input('Enter an integer: '))
    b = int(input('Enter an integer: '))
    c = a / b
    print('c =', c)
except ZeroDivisionError as zde :
    print('Denominator is 0')
    print(zde.args)
    print(zde)
except ValueError :
    print('Unable to convert string to int')
except :
```

```
    print('Some unknown error')
```

Given below is the sample interaction with the program:

```
Enter an integer: 10
Enter an integer: 20
c = 0.5

Enter an integer: 10
Enter an integer: 0
Denominator is 0
('division by zero',)
division by zero

Enter an integer: 10
Enter an integer: abc
Unable to convert string to int
```

- If an exception occurs, the type of exception raised is matched with the exceptions named after **except** keyword. When a match occurs, that **except** block is executed, and then execution continues after the last **except** block.
- If we wish to do something more before doing a graceful exit, we can use the keyword **as** to receive the exception object. We can then access its argument either using its **args** variable, or by simply using the exception object.
- **args** refers to arguments that were used while creating the exception object.

User-defined Exceptions

- Since all exceptional conditions cannot be anticipated, for every exceptional condition there cannot be a class in Python library.
- In such cases we can define our own exception class as shown in the following program:

```
class InsufficientBalanceError(Exception) :
    def __init__(self, accno, cb) :
        self.__accno = accno
        self.__curbal = cb
```

```
    def get_details(self) :
        return { 'Acc no' : self.__accno,
                 'Current Balance' : self.__curbal}

class Customers :
    def __init__(self) :
        self.__dct = { }

    def append(self, accno, n, bal) :
        self.__dct[accno] = { 'Name' : n, 'Balance' : bal }

    def deposit(self, accno, amt) :
        d = self.__dct[accno]
        d['Balance'] = d['Balance'] + amt
        self.__dct[accno] = d

    def display(self) :
        for k, v in self.__dct.items( ) :
            print(k, v)
        print( )

    def withdraw(self, accno, amt) :
        d = self.__dct[accno]
        curbal = d['Balance']
        if curbal - amt < 5000 :
            raise InsufficientBalanceError(accno, curbal)
        else :
            d['Balance'] = d['Balance'] - amt
            self.__dct[accno] = d

c = Customers( )
c.append(123, 'Sanjay', 9000)
c.append(101, 'Sameer', 8000)
c.append(423, 'Ajay', 7000)
c.append(133, 'Sanket', 6000)
c.display( )
c.deposit(123, 1000)
c.deposit(423, 2000)
c.display( )

try :
    c.withdraw(423, 3000)
    print('Amount withdrawn successfully')
    c.display( )
    c.withdraw(101, 5000)
```

```
    print('Amount withdrawn successfully')
    c.display( )
except InsufficientBalanceError as ibe :
    print('Withdrawal denied')
    print('Insufficient balance')
    print(ibe.get_details( ))
```

On execution of this program we get the following output:

```
123 {'Name': 'Sanjay', 'Balance': 9000}
101 {'Name': 'Sameer', 'Balance': 8000}
423 {'Name': 'Ajay', 'Balance': 7000}
133 {'Name': 'Sanket', 'Balance': 6000}

123 {'Name': 'Sanjay', 'Balance': 10000}
101 {'Name': 'Sameer', 'Balance': 8000}
423 {'Name': 'Ajay', 'Balance': 9000}
133 {'Name': 'Sanket', 'Balance': 6000}

Amount withdrawn successfully
123 {'Name': 'Sanjay', 'Balance': 10000}
101 {'Name': 'Sameer', 'Balance': 8000}
423 {'Name': 'Ajay', 'Balance': 6000}
133 {'Name': 'Sanket', 'Balance': 6000}

Withdrawal denied
Insufficient balance
{'Acc no': 101, 'Current Balance': 8000}
```

- Each customer in a Bank has data like account number, name and balance amount. This data is maintained in nested directories.
- If during withdrawal of money from a particular account the balance goes below Rs. 5000, then a user-defined exception called **InsufficientBalanceError** is raised.
- In the matching **except** block, details of the withdrawal transaction that resulted into an exception are fetched by calling **get_details( )** method present in **InsufficientBalanceError** class and displayed.
- **get_details( )** returns the formatted data. If we wish to get raw data, then we can use **ibe.args** variable, or simply **ibe**.

```
print(ibe.args)
print(ibe)
```

else Block

- The **try** .. **except** statement may also have an optional **else** block.
- If it is present, it must occur after all the **except** blocks.
- *Control goes to* **else** block if no exception occurs during execution of the **try** block.
- Program given below shows how to use the **else** block.

```
try :
    lst = [10, 20, 30, 40, 50]
    for num in lst :
        i = int(num)
        j = i * i
        print(i, j)
except NameError:
    print(NameError.args)
else:
    print('Total numbers processed', len(lst))
    del(lst)
```

We get the following output on executing this program:

```
10 100
20 400
30 900
40 1600
50 2500
Total numbers processed 5
```

- Control goes to **else** block since no exception occurred while obtaining squares.
- If we replace one of the elements in **lst** to 'abc', then a **NameError** will occur which will be caught by **except** block. In this case **else** block doesn't go to work.

finally Block

- **finally** block is optional.
- Code in **finally** always runs, no matter what! Even if a **return** or **break** occurs first.
- **finally** block is placed after **except** blocks (if they exist).
- **try** block must have **except** block and/or **finally** block.
- **finally** block is commonly used for releasing external resources like files, network connections or database connections, irrespective of whether the use of the resource was successful or not.

Exception Handling Tips

- Don't catch and ignore an exception.
- Don't catch everything using a catchall **except**, distinguish between types of exceptions.
- Make exception handling optimally elaborate; not too much, not too little.

P</> Programs

Problem 22.1

Write a program that infinitely receives positive integer as input and prints its square. If a negative number is entered then raise an exception, display a relevant error message and make a graceful exit.

Program

```
try:
    while True :
        num = int(input('Enter a positive number: '))
        if num >= 0 :
            print(num * num)
        else :
            raise ValueError('Negative number')
except ValueError as ve :
    print(ve.args)
```

Output

```
Enter a positive number: 12
144
Enter a positive number: 34
1156
Enter a positive number: 45
2025
Enter a positive number: -9
('Negative number',)
```

Problem 22.2

Write a program that implements a stack data structure of specified size. If the stack becomes full and we still try to push an element to it, then an **IndexError** exception should be raised. Similarly, if the stack is empty and we try to pop an element from it then an **IndexError** exception should be raised.

Program

```
class Stack :
   def __init__(self, sz) :
         self.size = sz
         self.arr = [ ]
         self.top = -1

     def push(self, n) :
         if self.top + 1 == self.size :
              raise IndexError('Stack is full')
         else :
              self.top += 1
              self.arr = self.arr + [n]

     def pop(self) :
         if self.top == -1 :
              raise IndexError('Stack is empty')
         else :
              n = self.arr[self.top]
              self.top -= 1
```

```
            return n

    def printall(self) :
        print(self.arr)

s = Stack(5)
try :
    s.push(10)
    n = s.pop( )
    print(n)
    n = s.pop( )
    print(n)
    s.push(20)
    s.push(30)
    s.push(40)
    s.push(50)
    s.push(60)
    s.printall( )
    s.push(70)
except IndexError as ie :
    print(ie.args)
```

Output

```
10
('Stack is empty',)
```

Tips

- A new element is added to the stack by merging two lists.
- **IndexError** is a readymade exception class. Here we have used it to raise a stack full or stack empty exception.

Problem 22.3

Write a program that implements a queue data structure of specified size. If the queue becomes full and we still try to add an element to it, then a user-defined **QueueError** exception should be raised. Similarly, if the queue is empty and we try to delete an element from it then a **QueueError** exception should be raised.

Program

```
class QueueError(Exception) :
    def __init__(self, msg, front, rear ) :
        self.errmsg = msg + ' front = ' + str(front) + ' rear = ' + str(rear)

    def get_message(self) :
        return self.errmsg

class Queue :
    def __init__(self, sz) :
        self.size = sz
        self.arr = [ ]
        self.front = self.rear = -1

    def add_queue(self, item) :
        if self.rear == self.size - 1 :
            raise QueueError('Queue is full.', self.front, self.rear)
        else :
            self.rear += 1
            self.arr = self.arr + [item]

            if self.front == -1 :
                self.front = 0

    def delete_queue(self) :
        if self.front == -1 :
            raise QueueError('Queue is empty.', self.front, self.rear)
        else :
            data = self.arr[self.front]
            if ( self.front == self.rear ) :
                self.front = self.rear = -1
            else :
                self.front += 1
            return data

    def printall(self) :
        print(self.arr)

q = Queue(5)
try :
```

```
    q.add_queue(11)
    q.add_queue(12)
    q.add_queue(13)
    q.add_queue(14)
    q.add_queue(15) # oops, queue is full
    q.printall( )
    i = q.delete_queue( )
    print('Item deleted = ', i)
    i = q.delete_queue( )
    print('Item deleted = ', i)
    i = q.delete_queue( )
    print('Item deleted = ', i)
    i = q.delete_queue( )
    print('Item deleted = ', i)
    i = q.delete_queue( )
    print('Item deleted = ', i)
    i = q.delete_queue( ) # oops, queue is empty
    print('Item deleted = ', i)
except QueueError as qe :
    print(qe.get_message( ))
```

Output

```
[11, 12, 13, 14, 15]
Item deleted =  11
Item deleted =  12
Item deleted =  13
Item deleted =  14
Item deleted =  15
Queue is empty. front = -1 rear = -1
```

Problem 22.4

Write a program that receives an integer as input. If a string is entered instead of an integer, then report an error and give another chance to user to enter an integer. Continue this process till correct input is supplied.

Program

```
while True :
   try :
      num = int(input('Enter a number: '))
      break
   except ValueError :
      print('Incorrect Input')
print('You entered: ', num)
```

Output

```
Enter a number: aa
Incorrect Input
Enter a number: abc
Incorrect Input
Enter a number: a
Incorrect Input
Enter a number: 23
You entered:  23
```

Exercises

[A] State whether the following statements are True or False:

(a) The exception handling mechanism is supposed to handle compile time errors.

(b) It is necessary to declare the exception class within the class in which an exception is going to be thrown.

(c) Every raised exception must be caught.

(d) For one **try** block there can be multiple **except** blocks.

(e) When an exception is raised, an exception class's constructor gets called.

(f) **try** blocks cannot be nested.

(g) Proper destruction of an object is guaranteed by exception handling mechanism.

(h) All exceptions occur at runtime.

(i) Exceptions offer an object-oriented way of handling runtime errors.

(j) If an exception occurs, then the program terminates abruptly without getting any chance to recover from the exception.

(k) No matter whether an exception occurs or not, the statements in the **finally** clause (if present) will get executed.

(l) A program can contain multiple **finally** clauses.

(m) **finally** clause is used to perform cleanup operations like closing the network/database connections.

(n) While raising a user-defined exception, multiple values can be set in the exception object.

(o) In one function/method, there can be only one **try** block.

(p) An exception must be caught in the same function/method in which it is raised.

(q) All values set up in the exception object are available in the **except** block that catches the exception.

(r) If our program does not catch an exception then Python Runtime catches it.

(s) It is possible to create user-defined exceptions.

(t) All types of exceptions can be caught using the **Exception** class.

(u) For every **try** block there must be a corresponding **finally** block.

[B] Answer the following questions:

(a) If we do not catch the exception thrown at runtime then who catches it?

(b) Explain in short most compelling reasons for using exception handling over conventional error handling approaches.

(c) Is it necessary that all classes that can be used to represent exceptions be derived from base class **Exception**?

(d) What is the use of a **finally** block in Python exception handling mechanism?

(e) How does nested exception handling work in Python?

(f) Write a program that receives 10 integers and stores them and their cubes in a dictionary. If the number entered is less than 3, raise a user-defined exception **NumberTooSmall**, and if the number entered is more than 30, then raise a user-defined exception **NumberTooBig**. Whether an exception occurs or not, at the end print the contents of the dictionary.

(g) What's wrong with the following code snippet?

```
try :
    # some statements
except :
    # report error 1
except ZeroDivisionError :
    # report error 2
```

(h) Which of these keywords is not part of Python's exception handling vocabulary—**try**, **catch**, **throw**, **except**, **raise**, **finally**, **else**?

(i) What will be the output of the following code?

```
def fun( ) :
    try :
        return 10
    finally :
        return 20

k = fun( )
print(k)
```