Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #107
Exceptions - try, except, else, finally, raise

By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M(Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Introduction

Everyday life is full of situations that we don't anticipate. For example, you get up for work in the morning and look for your phone charger, but you can't find it anywhere. You go to the bathroom to shower only to discover that the pipes are frozen. You get in your car, but it won't start. A human is able to cope with such unforeseen circumstances quite easily. Then **why can't the Computer? Why can't the program? Why can't the programmer?**

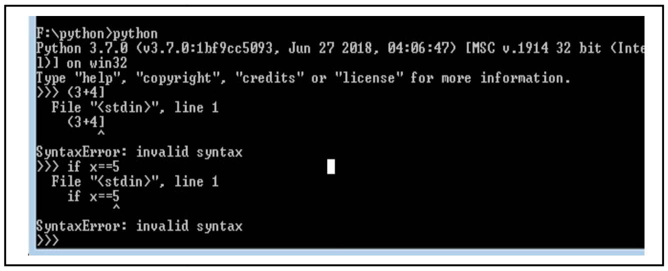
We usually try to write programs that do not produce errors, but the unfortunate truth is that **even programs written by the experienced developers sometimes crash**. Even if a program is perfect, it could still produce errors because of the data coming from outside to a program is malformed. This is a big problem with **server programs, such as web, mail and gaming servers.** Hence it is good to study about errors that can occur during the execution of the program. This is known as Exceptions in Programming and Python language is able cope up with these exceptions.
Note: Exceptions occur occasionally.

Types of Errors:

- **Syntax Errors/Parsing Errors:** Those errors which are due to the **incorrect format of a python statement.** These errors **occur while the statement or program is being translated to machine language and before it is being executed**. **A component of python's interpreter called as parser discovers these errors if is not hybrid interpreter. If it is hybrid, compiler takes care of syntactical errors.**

- **Runtime Errors/Exceptions:** An exception is a type of error that occurs when a syntactically correct code raises an error.

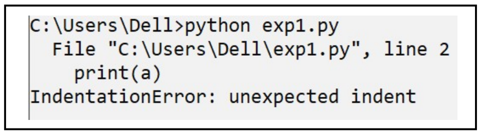
Syntax Error in detail:



```
F:\python>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> (3+4]
  File "<stdin>", line 1
    (3+4]
        ^

SyntaxError: invalid syntax
>>> if x==5
  File "<stdin>", line 1
    if x==5
          ^

SyntaxError: invalid syntax
>>>
```

Consider,

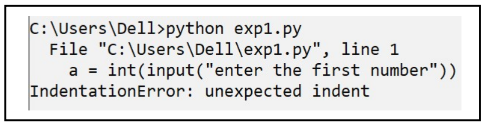
```
a = int(input("enter the first number"))
   print(a)
b = int(input("enter the first number")
print(b)
print(a+b)
```



```
C:\Users\Dell>python exp1.py
  File "C:\Users\Dell\exp1.py", line 2
    print(a)
IndentationError: unexpected indent
```

Again,

```
   a = int(input("enter the first number"))
print(a)
b = int(input("enter the first number")
print(b)
print(a+b)
```



```
C:\Users\Dell>python exp1.py
  File "C:\Users\Dell\exp1.py", line 1
    a = int(input("enter the first number"))
IndentationError: unexpected indent
```

Exception in detail:

It is an event that occurs during the execution of a program which disrupts of the normal flow of execution of the program.

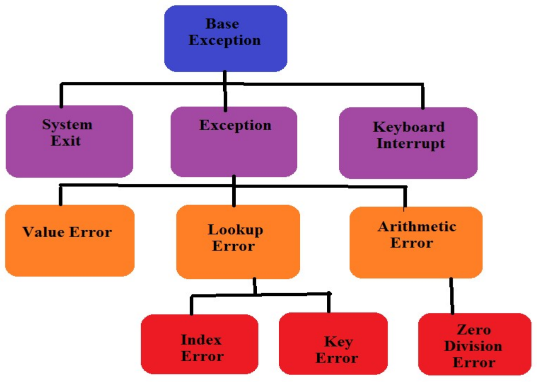
Example: Every morning we leave home (or the hostel) – come to the college directly – attend all classes – go home in the evening. Occasionally we get distracted for a movie which is released yesterday or a shooting is happening on the way and there is traffic jam, or a strong nice aroma is there on the way in some bajji shop and you will stop the car/bike or you will get down from bus to buy it. This, you are changing your path for that day but not for every day. Therefore, it is an exception and it occurs during the runtime.

The **Exceptions are of two categories:**

- Built-In exception
- User defined exception

Built-In exception:

Python3 defines **63 built-in exceptions**. The below figure shows a small view of the **Exception tree.**



Let us try to understand few of these in detail as per the below table mentioned.

Exception Name	Description
KeyboardInterrupt	Raised when user enters Ctrl-C, the interrupt key
OverflowError	Raised when a floating-point expression evaluates to a value that is too large
ZeroDivisionError	Raised when attempting to divide by 0
IOError	Raised when an I/O operation fails for an I/O related reason
IndexError	Raised when a sequence index is outside the range of valid indexes
NameError	Raised when attempting to evaluate an Unassigned identifier
TypeError	Raised when an operation or function is Applied to an object of the wrong type
ValueError	Raised when an operation or function has an argument of the right type builtin correct value

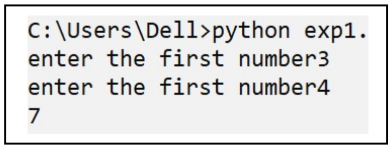
The Exception – ValueError

Consider the below code to add two integers entered by the user.

a = int(input("enter the first number"))
b = int(input("enter the first number"))
print(a+b)

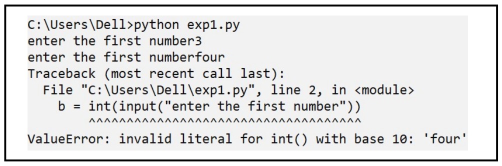
Case 1: When user enters 3 and 4 respectively,

output is 7



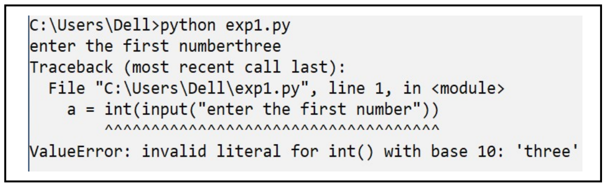
```
C:\Users\Dell>python exp1.
enter the first number3
enter the first number4
7
```

Case 2: When user enters 3 and four, what happens?



```
C:\Users\Dell>python exp1.py
enter the first number3
enter the first numberfour
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 2, in <module>
    b = int(input("enter the first number"))
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'four'
```

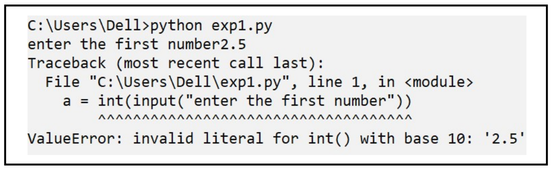
Case 3: When user enters three, press enter, can you then really enter 4? No



```
C:\Users\Dell>python exp1.py
enter the first numberthree
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 1, in <module>
    a = int(input("enter the first number"))
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'three'
```

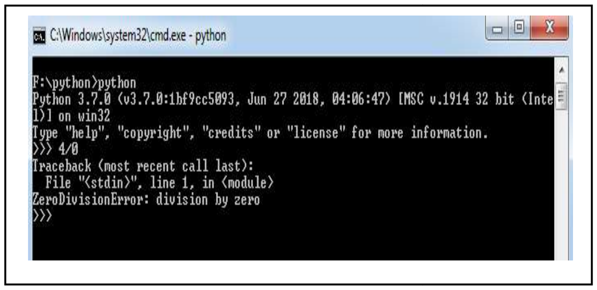
In the logic provided above to add two integers entered by the user, we assumed that users are really good and they follow the rule and hence they enter only integers. If they enter anything other than integers, we are in trouble and code stops running abruptly. These are exceptions. One more case is given below.

Case 4: When user enters any float value, what happens? Try other inputs



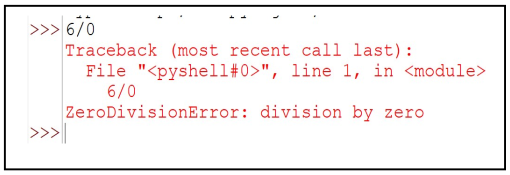
```
C:\Users\Dell>python exp1.py
enter the first number2.5
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 1, in <module>
    a = int(input("enter the first number"))
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: '2.5'
```

The Exception – ZeroDivisionError:



C:\Windows\system32\cmd.exe - python

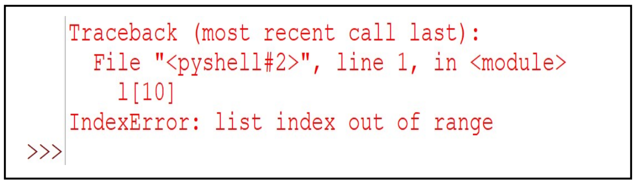
```
F:\python>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 4/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```



```
>>> 6/0
    Traceback (most recent call last):
      File "<pyshell#0>", line 1, in <module>
        6/0
    ZeroDivisionError: division by zero
>>>
```

The Exception – IndexError:

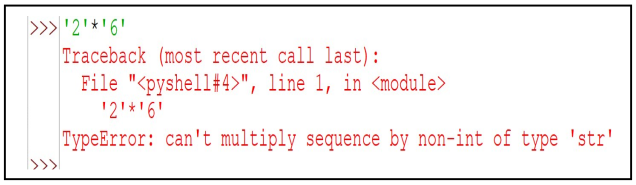
>>>lst=[1,4,5]
>>>lst[10]



```
    Traceback (most recent call last):
      File "<pyshell#2>", line 1, in <module>
        l[10]
    IndexError: list index out of range
>>>
```

The Exception – TypeError:

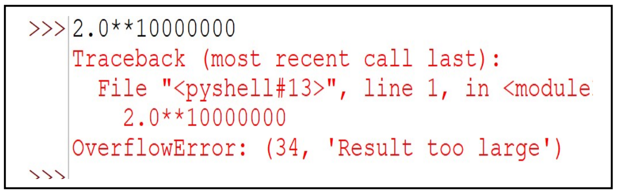
>>> '2'*'6'



```
>>> '2'*'6'
    Traceback (most recent call last):
      File "<pyshell#4>", line 1, in <module>
        '2'*'6'
    TypeError: can't multiply sequence by non-int of type 'str'
>>>
```

The Exception - OverflowError

>>> 2.0**10000000



```
>>> 2.0**10000000
    Traceback (most recent call last):
      File "<pyshell#13>", line 1, in <module
        2.0**10000000
    OverflowError: (34, 'Result too large')
>>>
```

In all above examples, **when an exception occurred, the program has been terminated.**

Think about this!

- **Can we avoid abrupt termination?**
- **Can we get a chance to proceed further?**
- **Can we have graceful degradation?**

Yes. It is possible to have a facility like this. **This is called as Exception handling.**

Exception handling

Providing a **graceful termination of the program by handling the runtime errors** using few **keywords** such as **try, except, else, raise** and **finally** is known as exception handling. It allows you to gracefully manage these situations/errors instead of letting the program crash. Also, this is a **way to deal with errors or exceptional situations** that may occur during the execution of a program.

try:

The code which might throw an exception, put those lines of code inside the suite of this block. It lets you test a block of code for errors indicating to the Python runtime that something unusual could happen in the suite of this.

except [<type>]:

At least one except block is mandatory to which the control must be transferred if and only if something unusual happens in the try suite. **If the try is unsuccessful**, none of the statement below that line will be executed in the suite of try and control is transferred to except block.

else:

Using this, you can instruct a program to execute a certain block of code only in the absence of exceptions in try or if no exceptions were raised in the try block. If **try is successful, statement/s in else block will be executed**.

finally:

This block **always gets executed either exception is raised or not in the try block**.

raise:

If something unusual happens in the try block or based on some condition, we can **explicitly mention the exception to be thrown.** This is done by raising an exception.

Syntax: raise Exceptionname([message])

This causes **creation of an exception object with the message if message is specified.** The object also **remembers the place in the code** (line number, function name, how this function got called on). The **raising of exception causes the program to abort if not in a try block** or **transfer the control to the matching except blocks if raised inside try block.**

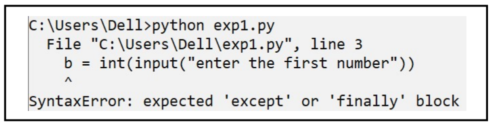
Let us handle few Exceptions using the above constructs one by one and try to understand exception handling constructs in detail.

Consider this code,

```
a = int(input("enter the first number"))
b = int(input("enter the first number"))
print(a+b)
```

Now need to decide what code must be inside try block. Whichever lines might throw an exception, put it inside a try block. Let us say first line.

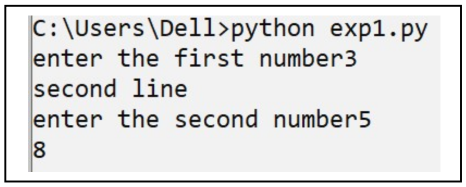
```
try:
   a = int(input("enter the first number")) #indentation as it is inside try.
   print("second line")
b = int(input("enter the second number")) # no indentation
print(a+b)
```



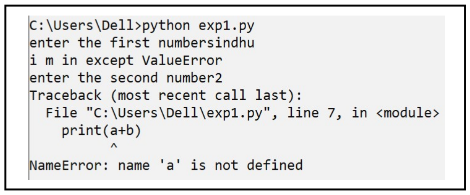
```
C:\Users\Dell>python exp1.py
  File "C:\Users\Dell\exp1.py", line 3
    b = int(input("enter the first number"))
    ^
SyntaxError: expected 'except' or 'finally' block
```

Resulted in Syntax Error. Let us add except block. We know that the statement in try block might throw ValueError. So we will handle that exception in except block. Also, observe that there are only two statements in the try block and except handles only one type of Error – ValueError. Now let us run this code.

```
try:
    a = int(input("enter the first number"))
    print("second line")
except ValueError:
    print("i m in except ValueError")
b = int(input("enter the second number"))
print(a+b)
print("ended")
```



```
C:\Users\Dell>python exp1.py
enter the first number3
second line
enter the second number5
8
```

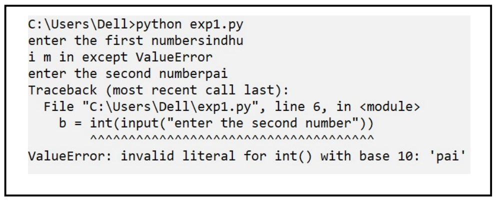


```
C:\Users\Dell>python exp1.py
enter the first numbersindhu
i m in except ValueError
enter the second number2
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 7, in <module>
    print(a+b)
          ^
NameError: name 'a' is not defined
```

In the first output, input from the user was fine. There was no Exception anywhere in the code. This is the normal flow.

But in the second output, user entered something unusual and this resulted in ValueError. Saw was caught by except clause. Hence except block got executed immediately. Observe that once there was exception in try block, none of the statement below that line is executed. But matching except is executed. Then control was outside the try except block. User entered number again. But print(a+b) resulted in Exception which was not handled as we have not specified this in a try except block.

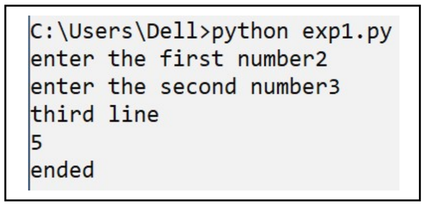
Also taking second number from the user can also throw an exception. But this is not inside try block. So it will not be handled properly and results in abrupt termination of the code execution.



```
C:\Users\Dell>python exp1.py
enter the first numbersindhu
i m in except ValueError
enter the second numberpai
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 6, in <module>
    b = int(input("enter the second number"))
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'pai'
```

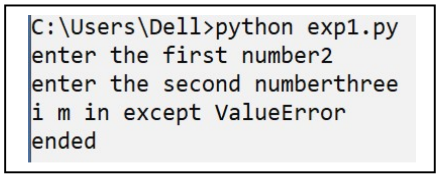
Now can we change the above code to handle ValueError for all lines of code. If no exception thrown in try block, let us add else block to perform addition and print it.

```
try:
    a = int(input("enter the first number"))
    b = int(input("enter the second number"))
    print("third line")
```



```
C:\Users\Dell>python exp1.py
enter the first number2
enter the second number3
third line
5
ended
```

```
except ValueError:
   print("i m in except ValueError")
else:    #if try is successful, this block gets
          #executed
   print(a+b)
print("ended")
```



```
C:\Users\Dell>python exp1.py
enter the first number2
enter the second numberthree
i m in except ValueError
ended
```

Now, just think about these points!

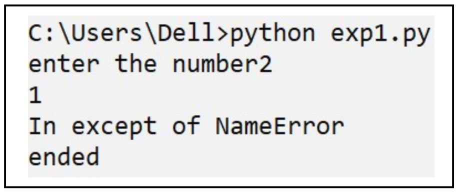
- Can we include multiple except blocks to handle different types of exceptions?
- If there are multiple except blocks, in which order it must be specified? Exception thrown in try will match with which except block?
- Can we have one generic exception which can handle all types of exceptions?
- Is it possible to create a user defined type for exception? If yes, can we specify that in the except clause?
- Is there any except block which can handle all types of exceptions? Like builtin and user defined exceptions – Both!

Let us say, you are going back home every day from the college. That is normal. Some day, it might rain and you might get drenched in rain or you might have to use the umbrella which is there in your bag. Some other day, may be you want to go to pizza hut with friends and reach back home. Similarly, within the try suite, there could be any one of the number of exceptional cases happening – each of which might require different ways of handling. **Hence, multiple except blocks are supported.**

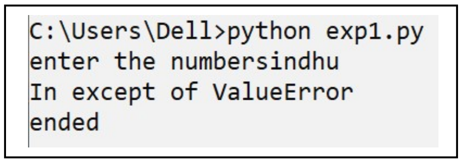
So **try block may be followed by one or more except blocks** – all of them specifying the name of the exception – one of them may provide a generic and only one providing the default way of handling all exceptions. But **only one except block gets executed at any point of time** for that specific try block.

Example_code_1: Handling different exceptions separately in different way.

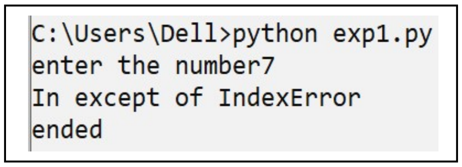
```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print(c) #NameError
   print("last line in try")
except NameError:
   print("In except of NameError")
except ValueError:
   print("In except of ValueError")
except IndexError:
   print("In except of IndexError")
else:
   print("all fine in try block")
print("ended")
```



```
C:\Users\Dell>python exp1.py
enter the number2
1
In except of NameError
ended
```



```
C:\Users\Dell>python exp1.py
enter the numbersindhu
In except of ValueError
ended
```



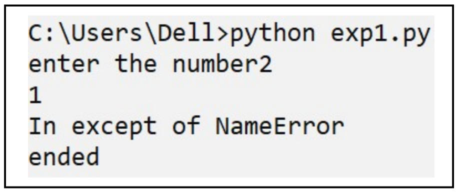
```
C:\Users\Dell>python exp1.py
enter the number7
In except of IndexError
ended
```

Think about this!

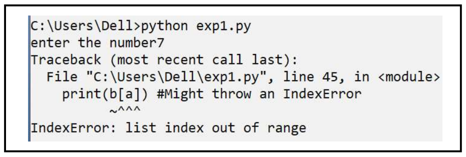
- **Do you ever get the output of last statement of try block in the above code?**
- **Will the above code ever executes else block?**

Example_code_2: But what if some exception was not handled in any one the except blocks? See this code. The except block for indexError removed.

```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print(c) #NameError
   print("last line in try")
except NameError:
   print("In except of NameError")
except ValueError:
   print("In except of ValueError")
else:
   print("all fine in try block")
print("ended")
```



```
C:\Users\Dell>python exp1.py
enter the number2
1
In except of NameError
ended
```



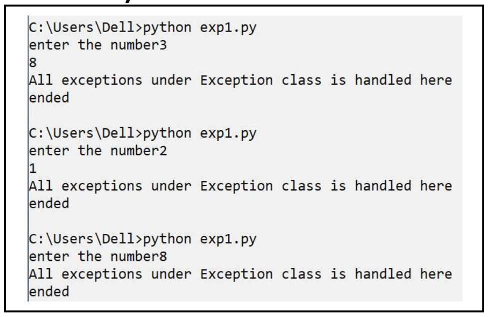
```
C:\Users\Dell>python exp1.py
enter the number7
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 45, in <module>
    print(b[a]) #Might throw an IndexError
          ~^^^
IndexError: list index out of range
```

Observe that when indexError occurred, code abruptly terminated with exception on the terminal and none of the statements below that got executed -> “ended” is not printed at all.

If we are not very sure of the specific Exception name which might be thrown by the try block, we can use the parent name as well in the except block.

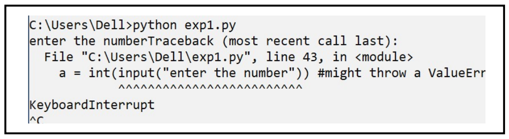
Example_code_3:

```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print(c) #NameError
   print("last line in try")
except Exception:   #generic class
   print("All exceptions under Exception class is handled here")
except NameError:
   print("In except of NameError")
except ValueError:
   print("In except of ValueError")
else:
   print("all fine in try block")
print("ended")
```



```
C:\Users\Dell>python exp1.py
enter the number3
8
All exceptions under Exception class is handled here
ended

C:\Users\Dell>python exp1.py
enter the number2
1
All exceptions under Exception class is handled here
ended

C:\Users\Dell>python exp1.py
enter the number8
All exceptions under Exception class is handled here
ended
```

All exceptions for which Exception class is the super class, are handled in the same except block which is the **First match. Not the Best match.** In the Exception tree mentioned, Exception is the parent of all other Exceptions like NameError, ValueError and LookupError etc.. IndexError and KeyError are the sub classes of LookupError in turn.But if we have any other exception which is not the child of Exception, the execution of the code abruptly terminated.



```
C:\Users\Dell>python exp1.py
enter the numberTraceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 43, in <module>
    a = int(input("enter the number")) #might throw a ValueErr
            ^^^^^^^^^^^^^^^^^^^^^^^^^
KeyboardInterrupt
^C
```

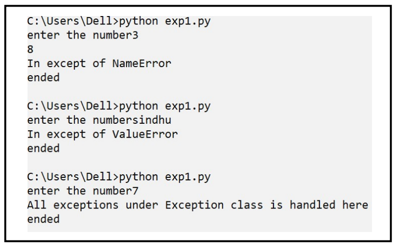
Try it out ->Using except block with BaseException to make sure that keyboardInterrupt(ctrl+c) from the user is also handled.

But using the generic exception followed by specific exception is not a good idea as exception thrown in the try is matching with the first else block not the best else block.

So, all the specific exceptions must be handled first and then the generic ones.

Example_code_4:

```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print(c) #NameError
   print("last line in try")
except NameError:
   print("In except of NameError")
except ValueError:
   print("In except of ValueError")
except Exception:   #moved to the end of the try except block
   print("All exceptions under Exception class is handled here")
        #IndexError is handled here
else:
   print("all fine in try block")
print("ended")
```

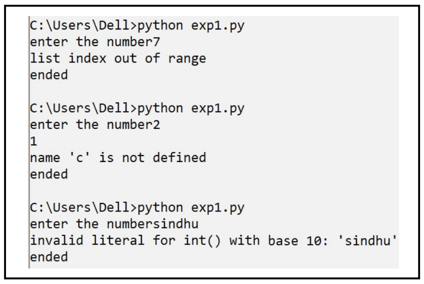


```
C:\Users\Dell>python exp1.py
enter the number3
8
In except of NameError
ended

C:\Users\Dell>python exp1.py
enter the numbersindhu
In except of ValueError
ended

C:\Users\Dell>python exp1.py
enter the number7
All exceptions under Exception class is handled here
ended
```

Now, given only one except block, can we find the type of exception that is thrown in try block? – Yes.

Example_code_5: All exceptions are handled in the same generic except block

```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print(c) #NameError
   print("last line in try")
except Exception as e:
   print(e)
else:
   print("all fine in try block")
print("ended")
```

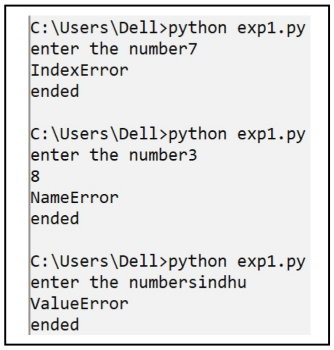


```
C:\Users\Dell>python exp1.py
enter the number7
list index out of range
ended

C:\Users\Dell>python exp1.py
enter the number2
1
name 'c' is not defined
ended

C:\Users\Dell>python exp1.py
enter the numbersindhu
invalid literal for int() with base 10: 'sindhu'
ended
```

This code prints the object of exception. The keyword **as** is **used to create the object** here.

Example_code_6: Using the object name, we can get the specific type of exception

```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print(c) #NameError
   print("last line in try")
except Exception as e:
   print(e.__class__.__name__)
else:
   print("all fine in try block")
print("ended")
```



```
C:\Users\Dell>python exp1.py
enter the number7
IndexError
ended

C:\Users\Dell>python exp1.py
enter the number3
8
NameError
ended

C:\Users\Dell>python exp1.py
enter the numbersindhu
ValueError
ended
```

The finally block:

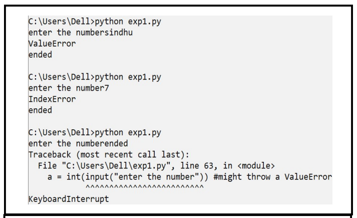
This block of code is always executed. **It is part of the try statement** which is an optional block that follows the except blocks. This block shall be executed on both **normal flow and exceptional flow**.

i) Normal flow: **try block – else block if there - finally block if there** – then continues.

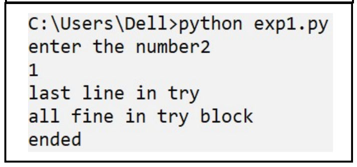
ii) Exceptional flow: **try block – exit the try block on an exception – find the first matching except block – execute the matched except block – finally block** – then continues.

Example_7: Usage of finally block

```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print("last line in try")
except Exception as e:
   print(e.__class__.__name__)
else:
   print("all fine in try block")
finally:
   print("ended")
```



```
C:\Users\Dell>python exp1.py
enter the numbersindhu
ValueError
ended

C:\Users\Dell>python exp1.py
enter the number7
IndexError
ended

C:\Users\Dell>python exp1.py
enter the numberended
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 63, in <module>
    a = int(input("enter the number")) #might throw a ValueError
            ^^^^^^^^^^^^^^^^^^^^^^^^^
KeyboardInterrupt
```

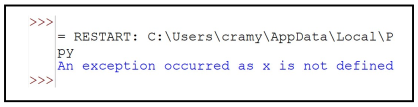


```
C:\Users\Dell>python exp1.py
enter the number2
1
last line in try
all fine in try block
ended
```

Note: Word "ended" always printed no matter whether there is an exception or not.

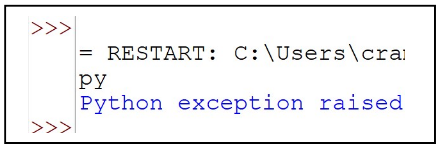
Example_code_8: This code returns NameError. Handled gracefully.

```
try:
        print(x)
except:   #default block
        print("An exception occurred as x is not defined ")
```



```
>>>
= RESTART: C:\Users\cramy\AppData\Local\P
py
An exception occurred as x is not defined
>>>
```

Example_code_9: This code returns ZeroDivisionError. Handled gracefully.



```
>>>
= RESTART: C:\Users\cra
py
Python exception raised
>>>
```

```
try:
        y = 1 / 0
except:   #default block
        print("Python exception raised")
```

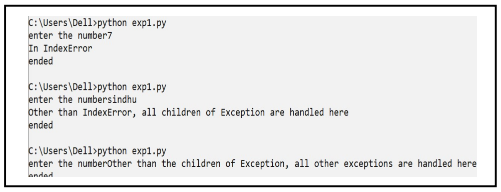
Example_code_8 and 9, both are using except block which is the default block.

Default except block – except:

This can handle any exception that is raised in the try block. This includes all **builtin exceptions and User defined exceptions.** Hence, to end the code gracefully, always it is a good idea to add this at the end of any try except block.

Example_code_10:

```
try:
   a = int(input("enter the number")) #might throw a ValueError
   b = [3,6,1,8]
   print(b[a]) #Might throw an IndexError
   print("last line in try")
except IndexError:
   print("in IndexError")
except Exception:
   print("Other than IndexError, all children of Exception are handled here")
except:
   print("Other than the children of Exception, all other exceptions are handled here")
else:
   print("all fine in try block")
finally:
   print("ended")
```

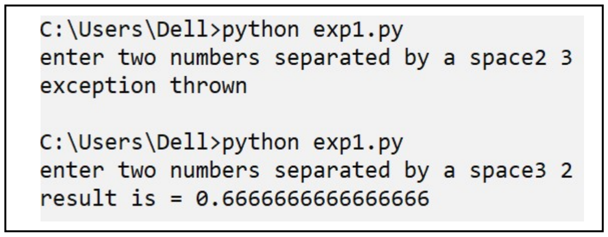


```
C:\Users\Dell>python exp1.py
enter the number7
In IndexError
ended

C:\Users\Dell>python exp1.py
enter the numbersindhu
Other than IndexError, all children of Exception are handled here
ended

C:\Users\Dell>python exp1.py
enter the numberOther than the children of Exception, all other exceptions are handled here
ended
```

A given number if it is divided by zero, an inbuilt exception named ZeroDivisionError is thrown. But if you have your constraint for which you want to raise inbuilt exception, is it possible? Yes – Here comes the usage of raising an exception.

Example_code_11: Raising an Exception for dividing a number by 2

```
 try:
   a, b = input("enter two numbers separated by a space").split()
   a, b = int(a), int(b)
   if a == 2:
    raise ZeroDivisionError
   else:
    c = b/a
except ZeroDivisionError:
   print("exception thrown")
else:
   print("result is =", c)
```



```
C:\Users\Dell>python exp1.py
enter two numbers separated by a space2 3
exception thrown

C:\Users\Dell>python exp1.py
enter two numbers separated by a space3 2
result is = 0.6666666666666666
```

Similar way, can I create and raise User Defined Exceptions? – Yes.

User-defined Exception/Custom exception:

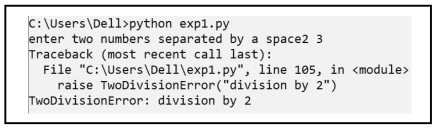
These exceptions **need to be derived from the Exception class, either directly or indirectly.** Most of the exceptions are named as names that end in “Error” similar to the naming of the builtin/standard exceptions in python.

Consider **user defined exception for dividing a number by 2.**

```
# custom exception type created

class TwoDivisionError(Exception):
   def __init__(self, msg):
      self.msg = msg
   def __str__(self):
      return self.msg
```

Example_code_12:

```
#Driver code
try:
  a, b = input("enter two numbers separated by a space").split()
  a, b = int(a), int(b)
  if a == 2:
   raise TwoDivisionError("division by 2")
  else:
   c = b/a
except ZeroDivisionError:
   print("exception thrown")
else:
   print("result is =", c)
```

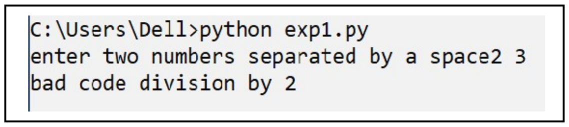


```
C:\Users\Dell>python exp1.py
enter two numbers separated by a space2 3
Traceback (most recent call last):
  File "C:\Users\Dell\exp1.py", line 105, in <module>
    raise TwoDivisionError("division by 2")
TwoDivisionError: division by 2
```

Here TwoDivisionError is not handled by except block. Can we add it now?

Example_code_13:

```
try:
  a, b = input("enter two numbers separated by a space").split()
  a, b = int(a), int(b)
  if a == 2:
   raise TwoDivisionError("division by 2")
  else:
   c = b/a
except ZeroDivisionError:
   print("exception thrown")
except TwoDivisionError as t:
   print("bad code",t)
else:
   print("result is =", c)
```

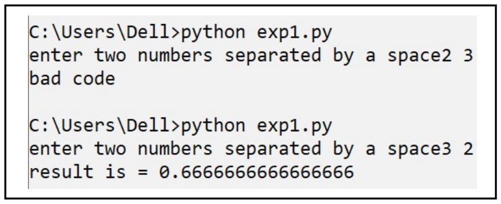


```
C:\Users\Dell>python exp1.py
enter two numbers separated by a space2 3
bad code division by 2
```

Can we replace this with a default except block ?

Example_code_14: Usage of default except block to handle user defined Exception

```
try:
  a, b = input("enter two numbers separated by a space").split()
  a, b = int(a), int(b)
  if a == 2:
    raise TwoDivisionError("division by 2") # This creates  the object but cannot use any variable in in the default except block
  else:
    c = b/a
except ZeroDivisionError:
    print("exception thrown")
except:
    print("bad code")
else:
    print("result is =", c)
```



```
C:\Users\Dell>python exp1.py
enter two numbers separated by a space2 3
bad code

C:\Users\Dell>python exp1.py
enter two numbers separated by a space3 2
result is = 0.6666666666666666
```

What happens if we use except Exception: block just before the default block? Think!

Few points to note:

- There is no mechanism in any language including Python to go back to the try block – no way to resume at the point of exception.
- Always the specific exceptions must be handled first, then the generic exceptions and then the default except block.
- Having default except block in the beginning itself when we have numerous except blocks, results in Syntax Error. Try it!
- Having Generic except block in the beginning itself when we have numerous except blocks, doesn't result in any Error. But if any exception occurred in the try block, it will be handled by the first matched except block not the best match.
- If exception is not handled properly in any of the except blocks, that results in abrupt termination of the program.

-END-