Unit 6 - Exception Handling

Objectives:

- ❖ Students will learn the Common Standard Exceptions available in python programming
- ❖ Students can understand how to handle exceptions and different ways of handling exceptions
- ❖ Students will learn how to create who exceptions
- ❖ Students will know the scope of exceptions handling

Module – 1

6.1 Introduction

In our programming experience so far we have encountered several kinds of run-time exceptions, such as division by zero, accessing a list with an out-of-range index, and attempting to convert a non-number to an integer. We have seen these and other run-time exceptions immediately terminate a running program.

Python provides a standard mechanism called exception handling that allows programmers to deal with these kinds of run-time exceptions and many more. Rather than always terminating the program's execution, an executing program can detect the problem when it arises and possibly execute code to correct the issue or mitigate it in some way. This chapter explores handling exceptions in Python.

6.2 What is Exceptions?

An *exception* is a special object that the executing program can create when it encounters an extraordinary situation. Such a situation almost always represents a problem, usually some sort of run-time error.

- Official explanation: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- So ... it is an unexpected event, like an error. However, not all exceptions are errors.
- When an error occurs an exception is raised.
- When an exception occurs, python stops whatever it is doing and goes to the last seen exception handler.
- You can raise exceptions yourself in the code.
- You can create your own exceptions.
- When an exception is handled the program can continue.

Examples of exceptional situations include:

- evaluating the expression lst[i] where lst is a list, and i ≥ len(lst).
- attempting to convert a nonnumeric string to a number, as in int('Fred')
- attempting to read a variable that has not been defined
- attempting to calculate the division with 0 (zero)
- attempting to evaluate the expressions with not suitable operator.

Exceptions represent a standard way to deal with run-time errors. In programming languages that do not support exception handling, programmers must devise their own ways of dealing with exceptional situations. The proper use of Python's exception handling infrastructure leads to code that is logically cleaner and less prone to programming errors. The standard Python library uses exceptions, and programmers can create new exceptions that address issues specific to their particular problems. These exceptions all use a common syntax and are completely compatible with each other.

6.3 Common Standard Exceptions

We have encountered a number of Python's standard exception classes. Below table lists some of the more common exception classes.

Exceptions	Meaning
AttributeError	Object does not contain the specified instance variable or method
ImportError	The import statement fails to find a specified module or name in that module
IndexError	A sequence (list, string, tuple) index is out of range
KeyError	Specified key does not appear in a dictionary
NameError	Specified local or global name does not exist

TypeError	Operation or function applied to an inappropriate type
ValueError	Operation or function applied to correct type but inappropriate value
ZeroDivisionError	Second operand of divison or modulus operation is zero

The following interactive sequence provides an example of each of the exceptions shown in Table

AttributeError

```
>>> from fractions import Fraction
>>> frac = Fraction(1, 2)
>>> frac.numerator
1
>>> frac.numertor
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    frac.numertor
AttributeError: 'Fraction' object has no attribute 'numertor'
>>> l=[3,4]
>>> l.append(5)
>>> l
[3, 4, 5]
>>> l.apped(6)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    l.apped(6)
AttributeError: 'list' object has no attribute 'apped'
```

ImportError

```
>>> from fractions import Fractions
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    from fractions import Fractions
ImportError: cannot import name 'Fractions' from 'fractions' (C:\Users\DELL\AppData\Local\Program
>>> from math import squareroot
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    from math import squareroot
ImportError: cannot import name 'squareroot' from 'math' (unknown location)
```

IndexError

```
>>> seq = [2, 5, 11]
>>> print(seq[1])
5
>>> print(seq[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

KeyError

```
>>> d = {}
>>> d['Fred'] = 100
>>> print(d['Fred'])
100
>>> print(d['Freddie'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Freddie'
```

NameError

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

TypeError

```
>>> 4+'a'
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    4+'a'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> l=[4,5]
>>> l['a']
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    l['a']
TypeError: list indices must be integers or slices, not str
>>> 5 < 'a'
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    5 < 'a'
TypeError: '<' not supported between instances of 'int' and 'str'
```

ValueError

```
>>> print(int('Fred'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Fred'
```

ZeroDivisionError

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

6.4 Catching Exceptions in Python

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong). When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

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:
        You do your operations here;
        ......................
except ExceptionName:
        If there is ExceptionName, then execute this block.
```

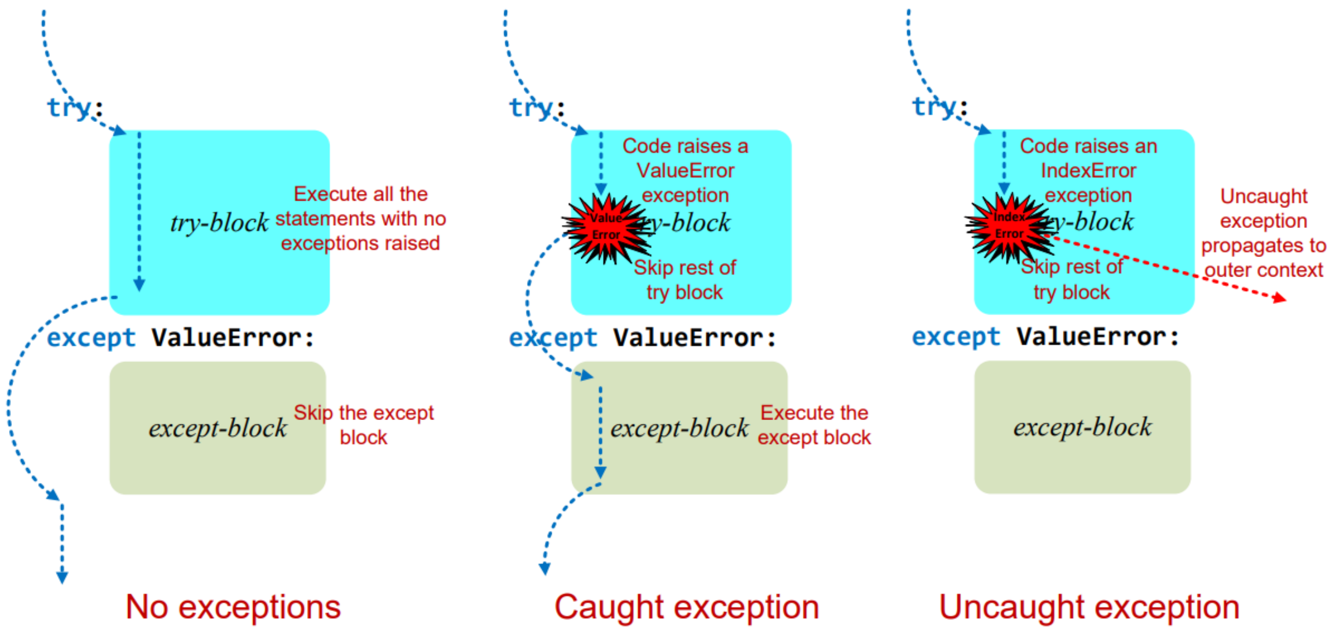
Example:

```
try:
        val = int(input("Please enter a small positive integer: "))
        print('You entered', val)
except ValueError:
        print('Input not accepted')
```

The two statements between try and except constitute the try block. The statement after the except line represents an except block. If the user enters a string unacceptable to the int function, the int function will raise a ValueError exception. At this point the program will not complete the assignment statement nor will it execute the print statement that follows. Instead the program immediately will begin executing the code in the except block. This means if the user enters five, the program will print the message Input not accepted. If the user enters a convertible string like 5, the program will complete the try block and ignore the code in the except block.

Below figure contrasts the possible program execution flows within a try/except statement. We say the except block handles the exception raised in the try block. Another common terminology used to describe the excepting handling process uses the throw/catch metaphor: the executing program throws an exception that an except block catches. The above example program catches only exceptions of type ValueError. If for some reason the code within the try block of Listing (above program) raises a different type of exception, this try statement is unable to catch it. In this case the program will behave as if the try/except statement were not there. Unless other exception handling code is present in the calling environment, the interpreter simply will terminate the program with an error message.

Figure The possible program execution flows through a `try/except` statement.



6.5 Handling Multiple Exceptions

A try statement can have multiple except blocks. Each except block must catch a different type of exception object. Below example program offers three except blocks. Its try statement specifically can catch ValueError, IndexError, and ZeroDivisionError exceptions.

Syntax:

```
try:
        You do your operations here; Example:
        ......................
except Exception I:
        If there is ExceptionI,
        then execute this block.
except Exception II:
        If there is ExceptionII,
        then execute this block.
        ......................
```

Example:

```
try:
   x=input("Enter Index Value :")
   b=int(x)       #Try to add different datatype values
   ab=[4,5,6]
   print(ab[b])  # Try to access outoff index value
   print(3/0)     # Try to divide by zero
except ValueError:
   print('Cannot convert integer')
except IndexError:
   print('List index is out of range')
except ZeroDivisionError:
   print('Division by zero not allowed')
```

The except clause with multiple exceptions

If we need the exact code to handle more than one exception type, we can associate multiple types with a single except block by listing each exception type within a tuple

Syntax:

```
try:
        You do your operations here;
        ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
        If there is any exception from the given exception list, then execute this block
        .......................
```

Example:

```
try:
   x=input("Enter Value:")
  a=int(x)
  print a
  b=a+'rkv'
  print b
except (ValueError, TypeError):
  print ('Problem in the Code')
```

Catching All Exceptions

If we want our programs not to crash, we need to handle all possible exceptions that can arise. This is particularly important when we use libraries that we did not write. A program may execute code that only under very rare circumstances raises an exception. This situation may be so rare that it evades our thorough testing and appears only after we deploy the application to users. We need a handler that can catch any exception. The type Exception matches any exception type that a programmer would reasonably want to catch.

To catch all exceptions, write an exception handler for Exception

Example-1:

```
try:
   a=int('x')
except Exception :
  print('Problem in the Code')
```

Example-2:

```
try:
   a=int('x')
except:
  print('Problem in the Code')
```

Catching Exception Objects

The except blocks "catch" exception objects. We can inspect the object that an except block catches if we specify the object's name with the as keyword. Below example program shows how to use the as keyword to get access to the exception object raised by code in the try block.

The except blocks "catch" exception objects. We can inspect the object that an except block catches if we specify the object's name with the as keyword.

Example

```
try:
     a=int('x')
except ValueError as a:
     print(a)
```

.

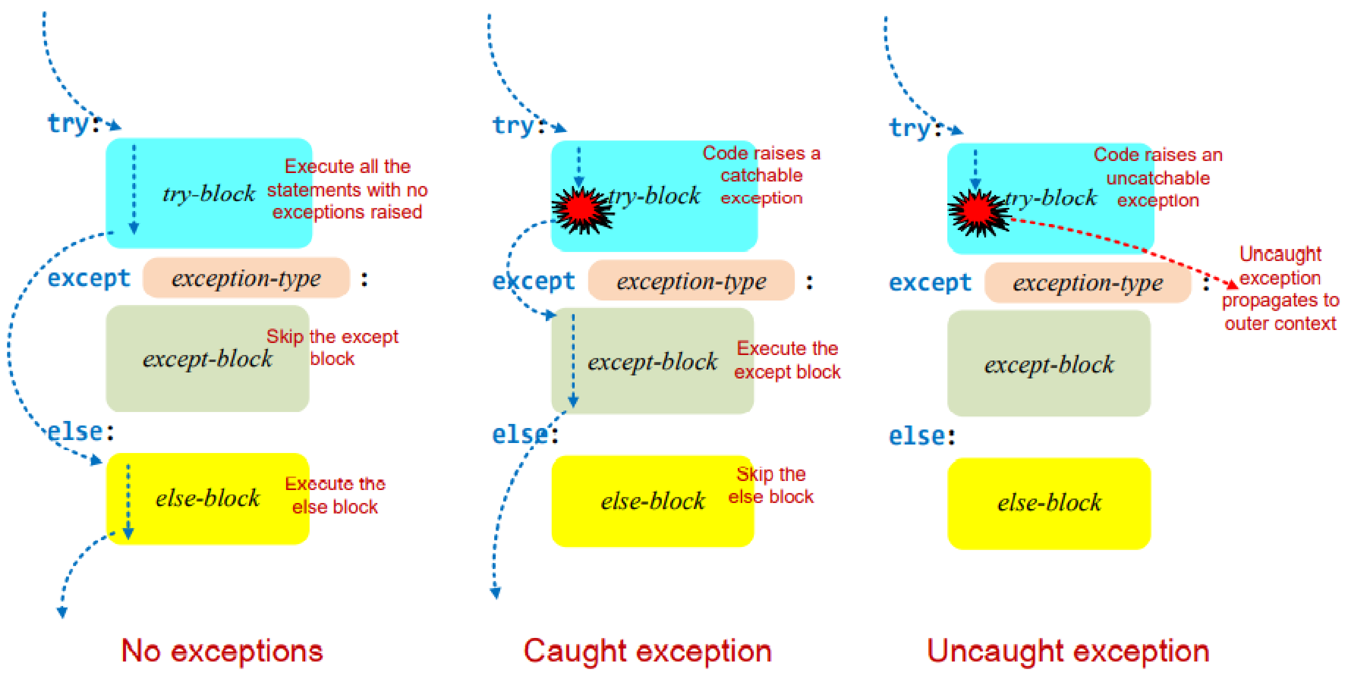
Output

```
>>>
invalid literal for int() with base 10: 'x'
```

6.6 The try Statement's Optional else Block

The Python try statement supports an optional else block. Its behavior is reminiscent of the while statement's else block. If the code within the try block does not produce an exception, no except blocks trigger, and the program's execution continues with code in the else block. Below figure contrasts the possible program execution flows within a try/else statement. The else block, if present, must appear after all of the except blocks. Since the code in the else block executes only if the code in the try block does not raise an exception, why not just append the code in the else block to the end of the code within the try block and eliminate the else block altogether? The code restructured in this way may not behave identically to the original code. Consider example program which demonstrates the different behavior.

Figur The possible program execution flows through a `try/else` statement.



Example -1:

```
try:
    print('try code')
except:
    print('exception handling code')
else:
    print('no exception raised code')
    x = int('a') # Raises an exception
```

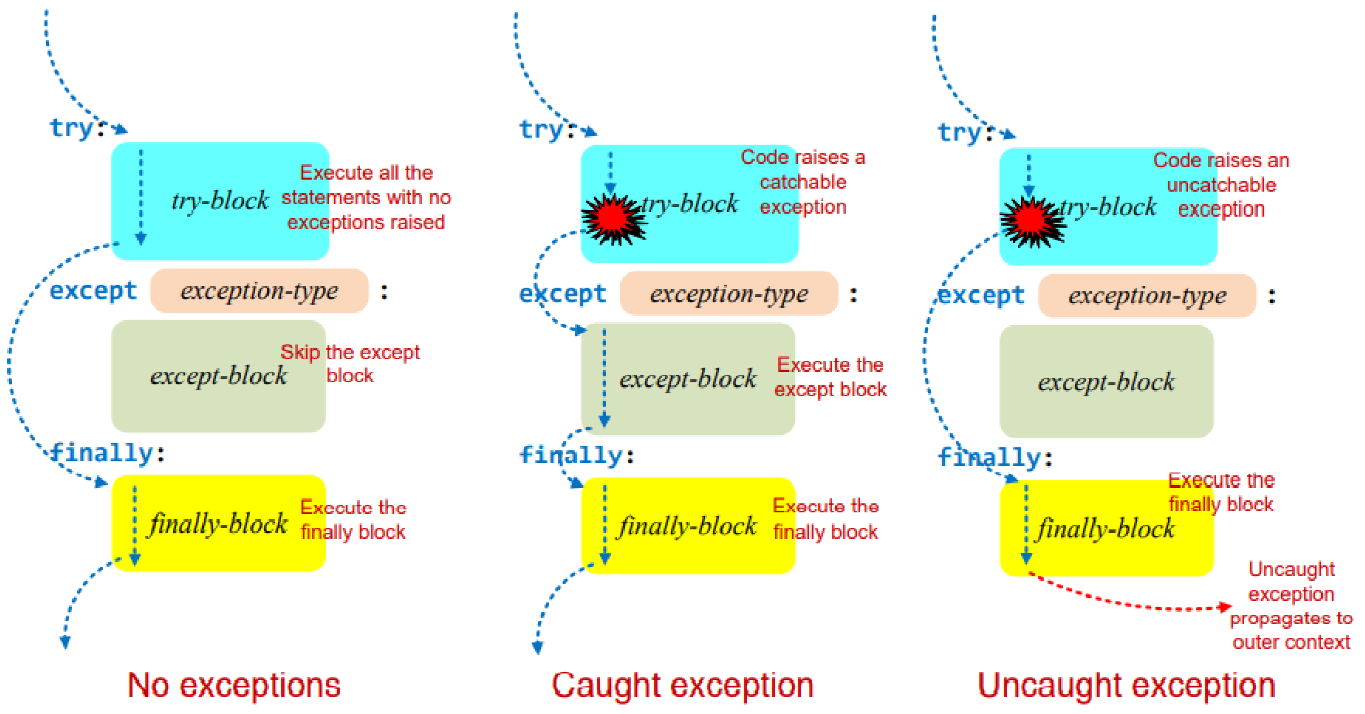
Example-2:

```
try:
     x=int(input("Enter a Number :"))
except ValueError:
     print("The Input was not avalid Integer. Try Again")
else:
     print("Dividing by 50 ",x,"will give  you :",50/x)
```

6.7 The *try* Statement's *finally* Block

A try statement may include an optional finally block. Code within a finally block always executes whether the try block raises an exception or not. A finally block usually contains "clean-up code" that must execute due to activity initiated in the try block. Below Figure contrasts the possible program execution flows within a try/except/finally statement.

Figure The possible program execution flows through a `try/except/finally` statement.



Example

```
try:
     x = int(input("Please enter a number: "))
except ValueError:
     print("The input was not a valid integer. Please try again...")
else:
     print("Dividing 50 by", x,"will give you :", 50/x)
finally:
     print("Already did everything necessary.")
```

6.8 Raising Exceptions

We have seen how to write code that reacts to exceptions. We know that certain functions, like open and int can raise exceptions. Also, statements that attempt to divide by zero or print an undefined variable will raise exceptions. The following statement raises a ValueError exception:

```
x = int('x')
```

The following statement is a more direct way to raise a ValueError exception:

```
raise ValueError()
```

The raise keyword raises an exception. Its argument, if present, must be an exception object. The class constructor for most exception objects accepts a string parameter that provides additional information to handlers:

```
raise ValueError('45')
```

In the interactive shell:

```
>>> raise ValueError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError
>>> raise ValueError('45')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 45
>>> raise ValueError('This is a value error')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: This is a value error
```

This means we can write a custom version of the int function that behaves similar to the built-in int function.

Syntax:

raise [Exception [, args [, traceback]]]

Example:

```
try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise KeyError("That is not a positive sfsdfsd number!")
except KeyError as ve:
     print(ve)
```

6.9 Exception structure

try:

normal statements which may fail

except exception-type:

error handling statements for this specific exception

except (exception-type1, exception-type2):

error handling statements for these specific exceptions

except exception-type as errname:

error handling statements for this specific exception getting instance via errname

except:

general error handling statements catching all exception types

else:

statements executed if no exception

finally:

clean-up statements always executed

6.10 Questions

1. What is exception? Describe exceptions with examples.
2. How to handle exception? Explain the execution flows through a try/except statement when no exception, caught exception and uncaught exception with example
3. Explain try statement …. else block with example? Also explain the execution flows through a try/else statement when no exception, caught exception and uncaught exception with example
4. Explain try statement …. finally block with example? Also explain the execution flows through a try/finally statement when no exception, caught exception and uncaught exception with example
5. What is raise exception? What is the use with it?
6. Describe the try-except structure of error handling?