CHAPTER 12

Error and Exception Handling

KEY Concepts

• Types of Errors and Exceptions • try - except Blocks • finally Block • Raising Exceptions • Re-raising Exceptions • Built-in and User-defined Exceptions • Handling Invoked Functions • Assertions

12.1 INTRODUCTION TO ERRORS AND EXCEPTIONS

In our programs, we had been getting some or the other errors but we had not mentioned much about them. Basically, there are (at least) two kinds of errors: *syntax errors* and *exceptions*.

The programs that we write may behave abnormally or unexpectedly because of some errors and/or exceptions (Figure 12.1). The two common types of errors that we very often encounter are *syntax errors* and *logic errors*. While logic errors occur due to poor understanding of problem and its solution, syntax errors, on the other hand, arises due to poor understanding of the language. However, such errors can be detected by exhaustive debugging and testing of procedures.

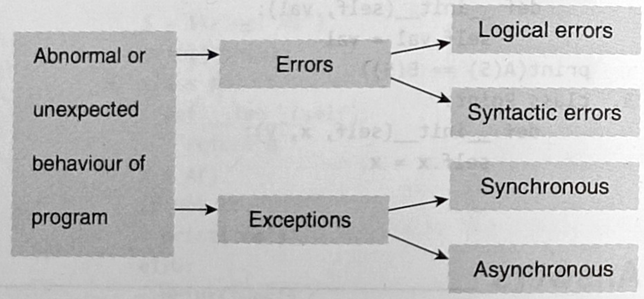


Figure 12.1 Errors and exceptions

But many a times, we come across some peculiar problems which are often categorized as *exceptions*. Exceptions are run-time anomalies or unusual conditions (such as divide by zero, accessing arrays out of its bounds, running out of memory or disk space, overflow, and underflow) that a program may encounter during execution. Like errors, exceptions can also be categorized as synchronous or asynchronous exceptions. While *synchronous* exceptions (like divide by zero, array index out of bound, etc.) can be controlled by the program, *asynchronous* exceptions (like an interrupt from the keyboard, hardware malfunction, or disk failure), on the other hand, are caused by events that are beyond the control of the program.

12.1.1 Syntax Errors

Syntax errors occurs when we violate the rules of Python and they are the most common kind of error that we get while learning a new language. For example, consider the lines of code given below.

```
>>> i=0
>>> if i == 0 print(i)
SyntaxError: invalid syntax
```


In the aforementioned code, we have missed the ':' before the keyword `print`. If you had run this code in a file, then the file name and line number would have also been printed to help you know where the error has occurred. Basically, in this case, the Python interpreter has found that it cannot complete the processing of the instruction because it does not conform to the rules of the language.

Note You will get syntax errors frequently as you start learning a new language.

12.1.2 Logic Error

The other type of error, known as a logic error, specifies all those type of errors in which the program executes but gives incorrect results. Logical error may occur due to wrong algorithm or logic to solve a particular program. In some cases, logic errors may lead to divide by zero or accessing an item in a list where the index of the item is outside the bounds of the list. In this case, the logic error leads to a run-time error that causes the program to terminate abruptly. These types of run-time errors are known as *exceptions*.

Many programmers may think of exception as a fatal run-time error. But programming languages provide an elegant way to deal with these errors so that the program terminates elegantly, not abruptly.

12.1.3 Exceptions

Even if a statement is syntactically correct, it may still cause an error when executed. Such errors that occur at run-time (or during execution) are known as *exceptions*. An exception is an event, which occurs during the execution of a program and disrupts the normal flow of the program's instructions. When a program encounters a situation which it cannot deal with, it raises an exception. Therefore, we can say that an exception is a Python object that represents an error.

When a program raises an exception, it must handle the exception or the program will be immediately terminated. You can handle exceptions in your programs to end it gracefully, otherwise, if exceptions are not handled by programs, then error messages are generated. Let us see some examples in which exceptions occurs.

```
  • >>> 5/0
Traceback (most recent call last):
 File "<pyshell#5>", line 1, in <module>
   5/0
ZeroDivisionError: integer division or modulo by zero
  • >>> var + 10
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    var + 10
NameError: name 'var' is not defined
  • >>> 'Roll No' + 123
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    'Roll No' + 123
TypeError: cannot concatenate 'str' and 'int' objects
```

Programming Tip: Standard exception names are built-in identifiers and not reserved keywords.

In all the three cases discussed above, we have seen three types of exceptions had occurred. Since they were not handled in the code, an appropriate error message was displayed to indicate what had happened.

The string printed as the exception type (like `TypeError`) is the name of the built-in exception that occurred. However, this is not true for user-defined exceptions.

12.2 HANDLING EXCEPTIONS

We can handle exceptions in our program by using `try` block and `except` block. A critical operation which can raise exception is placed inside the `try` block and the code that handles exception is written in `except` block. The syntax for `try-except` block can be given as,

```
try:
    statements
except ExceptionName:
    statements
```

Programming Tip: Handlers do not handle exceptions that occur in statements outside the corresponding `try` block.

The `try` statement works as follows.

Step 1: First, the *try block* (statement(s) between the `try` and `except` keywords) is executed.

Step 2a: If no exception occurs, the *except block* is skipped.

Step 2b: If an exception occurs, during execution of any statement in the try block, then,

i. Rest of the statements in the `try` block are skipped.

ii. If the exception type matches the exception named after the `except` keyword, the except block is executed and then execution continues after the `try` statement.

iii. If an exception occurs which does not match the exception named in the `except` block, then it is passed on to outer `try` block (in case of nested try blocks). If no exception handler is found in the program, then it is an *unhandled exception* and the program is terminated with an error message (Refer Figure 12.2).

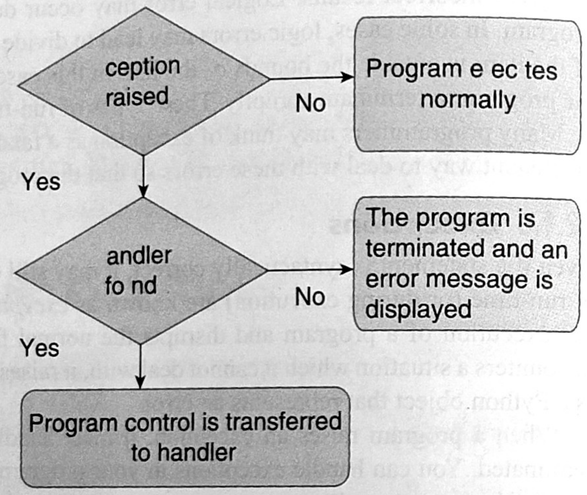


Figure 12.2 Flowchart for Case iii under Step 2b for `try` statements

In the aforementioned program, note that a number was divided by zero, an exception occurred so the control passed to the `except` block.

Example 12.1 Program to handle the divide by zero exception

```
num = int(input("Enter the numerator : "))
deno = int(input("Enter the denominator : "))
try:
  quo = num/deno
  print("QUOTIENT : ", quo)
except ZeroDivisionError:
  print("Denominator cannot be zero")
```

OUTPUT

```
Enter the numerator : 10
Enter the denominator : 0
Denominator cannot be zero
```


Note Exceptions gives you information like what, why, and how something went wrong.

12.3 MULTIPLE EXCEPT BLOCKS

Python allows you to have multiple `except` blocks for a single try block. The block which matches with the exception generated will get executed. A `try` block can be associated with more than one except block to specify handlers for different exceptions. However, only one handler will be executed. Exception handlers only handle exceptions that occur in the corresponding `try` block. We can write our programs that handle selected exceptions. The syntax for specifying multiple `except` blocks for a single `try` block can be given as,

```
try:
   operations are done in this block
   ....................
except Exception1:
   If there is Exception1, then execute this block.
except Exception2:
   If there is Exception2, then execute this block.
   ....................
else:
   If there is no exception then execute this block.
   ....................
```

Programming Tip `try-except` block is same as `try-catch` block. Exceptions are generated using `raise` keyword rather than `throw`.

We will read about the `else` block which is optional a little later. But for now, we have seen that a single `try` statement can have multiple `except` statements to catch different types of exceptions. For example, look at the code given below. The program prompts user to enter a number. It then squares the number and prints its result. However, if we do not specify any number or enter a non-number, then an exception will be generated. We have two `except` blocks. The one matching the case will finally execute. This is very much evident from the output.

Example 12.2 Program with multiple except blocks

```
try:
    num = int(input("Enter the number : "))
    print(num**2)
except (KeyboardInterrupt):
    print("You should have enterd a number..... Program Terminating...")
except (ValueError):
    print("Please check before you enter..... Program Terminating...")
print("Bye")
```

OUTPUT

```
Enter the number : abc
Please check before you enter..... Program Terminating...
Bye
```

Note that after execution of the `except` block, the program control goes to the first statement after the `except` block for that `try` block.

Note The except block without an exception can also be used to print an error message and then re-raise the exception.

12.4 MULTIPLE EXCEPTIONS IN A SINGLE BLOCK

An except clause may name multiple exceptions as a parenthesized tuple, as shown in the program given below. So whatever exception is raised, out of the three exceptions specified, the same except block will be executed.

Example 12.3 Program having an except clause handling multiple exceptions simultaneously

```
try:
    num = int(input("Enter the number : "))
    print(num**2)
except (KeyboardInterrupt, ValueError, TypeError):
    print("Please check before you enter..... Program Terminating...")
print("Bye")
```

Programming Tip: No code should be present between a list of except blocks.

OUTPUT

```
Enter the number : abc
Please check before you enter..... Program Terminating...
Bye
```

Thus, we see that if we want to give a specific exception handler for any exception raised, we can better have multiple except blocks. Otherwise, if we want the same code to be executed for all three exceptions then we can use the except(list_of_exceptions) format.

12.5 EXCEPT BLOCK WITHOUT EXCEPTION

You can even specify an except block without mentioning any exception (i.e., except:). This type of except block if present should be the last one that can serve as a wildcard (when multiple except blocks are present). But use it with extreme caution, since it may mask a real programming error.

In large software programs, may a times, it is difficult to anticipate all types of possible exceptional conditions. Therefore, the programmer may not be able to write a different handler (except block) for every individual type of exception. In such situations, a better idea is to write a handler that would catch all types of exceptions. The syntax to define a handler that would catch every possible exception from the try block is,

```
try:
   Write the operations here
   ....................
except:
   If there is any exception, then execute this block.
   ....................
else:
   If there is no exception then execute this block.
```

The except block can be used along with other exception handlers which handle some specific types of exceptions but those exceptions that are not handled by these specific handlers can be handled by the except: block. However, the default handler must be placed after all other except blocks because otherwise it would prevent any specific handler to be executed.

Example 12.4 Program to demonstrate the use of except: block

```
try:
    file = pen('File1.txt')
    str = f.readline()
    print(str)
except IOError:
    print("Error occured during Input ..... Program Terminating...")
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error.... Program Terminating...")
```

Programming Tip: When an exception occurs, it may have an associated value, also known as the exception's *argument*.

OUTPUT

```
Unexpected error.... Program Terminating...
```

Note Using except: without mentioning any specific exception is not a good programming practice because it catches all exceptions and does not make the programmer identify the root cause of the problem.

12.6 THE else CLAUSE

The try ... except block can optionally have an *else clause*, which, when present, must follow all except blocks. The statement(s) in the else block is executed only if the try clause does not raise an exception. For example, the codes given below illustrate both the cases. This will help you to visualize the relevance of the else block.

Example 12.5 Programs to demonstrate else block

```
try:
    file = open('File1.txt')
    str = file.readline()
    print(str)
except IOError:
    print("Error occurred during Input
..... Program Terminating...")
else:
    print("Program Terminating
Successfully.....")
```

OUTPUT

```
Hello
Program Terminating Successfully.....
```

```
try:
    file = open('File1.txt')
    str = f.readline()
    print(str)
except:
    print("Error occurred ..... Program
Terminating...")
else:
    print("Program Terminating
Successfully.....")
```

OUTPUT

```
Error occurred.....Program
Terminating...
```

12.7 RAISING EXCEPTIONS

You can deliberately raise an exception using the `raise` keyword. The general syntax for the `raise` statement is,

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

Here, `Exception` is the name of exception to be raised (example, `TypeError`). `args` is optional and specifies a value for the exception argument. If `args` is not specified, then the exception argument is `None`. The final argument, `traceback`, is also optional and if present, is the `traceback` object used for the exception.

For example, the code given below simply creates a variable and prints its value. There was no error in the code but we have deliberately raised an exception.

Example 12.6 Program to deliberately raise an exception

```
try:
    num = 10
    print(num)
    raise ValueError
except:
    print("Exception occurred .... Program Terminating...")
```

OUTPUT

```
10
Exception occurred .... Program Terminating...
```

The only argument to the `raise` keyword specifies the exception to be raised. Recall that, we had earlier said that you can re-raise the exceptions in the `except:` block. This is especially important when you just want to determine whether an exception was raised but don't intend to handle it. The code given below is used to re-raise an exception from the `except:` block.

Example 12.7 Program to re-raise an exception

```
try:
    raise NameError
except:
    print("Re-raising the exception")
    raise
```

Programming Tip: Avoid using except block without any exception.

OUTPUT

```
Re-raising the exception
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 2, in <module>
    raise NameError
NameError
```

12.8 INSTANTIATING EXCEPTIONS

Python allows programmers to instantiate an exception first before raising it and add any attributes (or arguments) to it as desired. These attributes can be used to give additional information about the error. To

instantiate the exception, the `except` block may specify a variable after the exception name. The variable then becomes an exception instance with the arguments stored in `instance.args`. The exception instance also has the `__str__()` method defined so that the arguments can be printed directly without using `instance.args`.

Note The contents of the argument vary based on exception type.

Example 12.8 Program to understand the process of instantiating an exception

```
try:
    raise Exception('Hello', 'World')
except Exception as errorObj:
    print(type(errorObj))      # the exception instance
    print(errorObj.args)       # arguments stored in .args
    print(errorObj)            # __str__ allows args to be printed directly
    arg1, arg2 = errorObj.args
    print('Argument1 =', arg1)
    print('Argument2 =', arg2)
```

OUTPUT

```
<type 'exceptions.Exception'>
('Hello', 'World')
('Hello', 'World')
Argument1 = Hello
Argument2 = World
```

Note If you raise an exception with arguments but do not handle it, then the name of the exception is printed along with its arguments.

Example 12.9 Program to raise an exception with arguments

```
try:
    raise Exception('Hello', 'World')
except ValueError:
    print("Program Terminating...")
```

OUTPUT

```
Exception: ('Hello', 'World')
```

12.9 HANDLING EXCEPTIONS IN INVOKED FUNCTIONS

Till now, we have seen that exception handlers have handled exceptions if they occur in the `try` block. But, exceptions can also be handled inside functions that are called in the `try` block as shown in the program given below.

Example 12.10 Program to handle exceptions from an invoked function

```
def Divide(num, deno):
    try:
        quo = num/deno
    except ZeroDivisionError:
        print("You cannot divide a number by zero... Program Terminating...")
Divide(10,0)
```

OUTPUT

```
You cannot divide a number by zero... Program Terminating...
```

Basically, a large program is usually divided into n number of functions. The possibility that the invoked function may generate an exceptional condition cannot be ignored. Figure 12.3 shows the scenario when the function invoked by the `try` block throws an exception which is handled by the `except` block in the calling function. The syntax for such a situation can be given as,

```
function_name(arg list):
   ----------
   ----------
try
   -----------
   function_name()     // function call
   -----------

except ExceptionName:
   -----------
   // Code to handle exception
   ------------
```

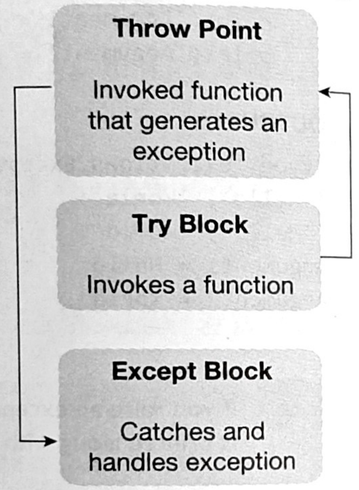


Figure 12.3 Function invoked by the `try` block throws an exception which is handled by the `except` block

Note Irrespective of the location of the exception, the try block is always immediately followed by the catch block.

The program given in the following example generates a `divide by zero exception` from a called function. The `main module` has a `try` block from which a function `Divide()`is invoked. In `Divide()`,the exception occurs which is thrown and is handled by the `except` block defined in the `main module` immediately followed by the `try` block.

Example 12.11 Program to handle exception in the calling function

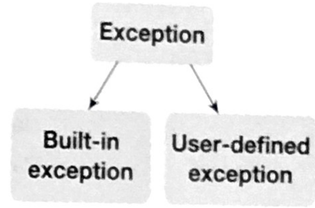
```
def Divide(num, deno):
    return num/deno
try:
    Divide(10,0)
except ZeroDivisionError:
    print("You cannot divide a number by zero... Program Terminating...")
```

OUTPUT

```
You cannot divide a number by zero... Program Terminating...
```


Note Python allows programmers to raise an exception in a deeply nested `try` block or in a deeply nested function call.

Note that program execution creates a *stack* as one function calls another. When a function at the bottom of the stack raises an exception, it is propagated up through the call stack so that the function may handle it. If no function handles it while moving towards top of the stack, the program terminates and a `traceback` is printed on the screen. The `traceback` helps the programmer to identify what went wrong in the code.



12.10 BUILT-IN AND USER-DEFINED EXCEPTIONS

Table 12.1 lists some standard exceptions that are already defined in Python. These built-in exceptions force your program to output an error when something in it goes wrong.

Table 12.1 Built-in exceptions

Exception	Description
`Exception`	Base class for all exceptions
`StopIteration`	Generated when the `next()` method of an iterator does not point to any object
`SystemExit`	Raised by `sys.exit()` function
`StandardError`	Base class for all built-in exceptions (excluding `StopIteration` and `SystemExit`)
`ArithmeticError`	Base class for errors that are generated due to mathematical calculations
`OverflowError`	Raised when the maximum limit of a numeric type is exceeded during a calculation
`FloatingPointError`	Raised when a floating point calculation could not be performed
`ZeroDivisionError`	Raised when a number is divided by zero
`AssertionError`	Raised when the assert statement fails
`AttributeError`	Raised when attribute reference or assignment fails
`EOFError`	Raised when end-of-file is reached or there is no input for `input()` function
`ImportError`	Raised when an import statement fails
`KeyboardInterrupt`	Raised when the user interrupts program execution (by pressing Ctrl+C)
`LookupError`	Base class for all lookup errors
`IndexError`	Raised when an index is not found in a sequence
`KeyError`	Raised when a key is not found in the dictionary
`NameError`	Raised when an identifier is not found in local or global namespace (referencing a non-existent variable)
`UnboundLocalError`, `EnvironmentError`	Raised when an attempt is made to access a local variable in a function or method when no value has been assigned to it.

Contd

Table 12.1 *Contd*

Exception	Description
`IOError`	Raised when input or output operation fails (for example, opening a file that does not exist)
`SyntaxError`	Raised when there is a syntax error in the program
`IndentationError`	Raised when there is an indentation problem in the program
`SystemError`	Raised when an internal system error occurs
`ValueError`	Raised when the arguments passed to a function are of invalid data type or searching a list for a non-existent value
`RuntimeError`	Raised when the generated error does not fall into any of the above category
`NotImplementedError`	Raised when an abstract method that needs to be implemented in an inherited class is not implemented
`TypeError`	Raised when two or more data types are mixed without coercion

Besides these, Python allows programmers to create their own exceptions by creating a new exception class. The new exception class is derived from the base class `Exception` which is pre-defined in Python. The program given below explains this concept.

Example 12.12 Program to define a user-defined exception

```
class myError(Exception):
    def __init__(self, val):
        self.val = val
    def __str__(self):
        return repr(self.val)
try:
    raise myError(10)
except myError as e:
    print('User Defined Exception Generated with value', e.val)
```

OUTPUT

```
User Defined Exception Generated with value 10
```

In the above program, the `__init__()` method of `Exception` class has been overridden by the new class. The customized exception class can be used to perform any task. However, these classes are usually kept simple and have only limited attributes to provide information about the error to be extracted by handlers for the exception. Note that creating your own exception class or defining a user defined exception is known as *custom exception*.

Note An exception can be a string, a class, or an object. Most of the exceptions raised by Python are classes, with an argument that is an instance of the class.

Moreover, when creating a module that can raise different exceptions, a better approach would be to create a base class for exceptions defined by that module, and subclasses to create specific exception classes for different error conditions.

Note 'as' is a keyword that allows programmers to name a variable within an except statement.

Example 12.13 Program to create sub-classes of Exception class to handle exceptions in a better customized way

```
class Error(Exception):
    def message(self):
        raise NotImplementedError()
class InputError(Error):
    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg
    def message(self):
        print("Error in input in expression"),
        print(self.expr)
try:
    a = input("Enter a : ")
    raise InputError("input(\"Enter a : s\")", "Input Error")
except InputError as ie:
    ie.message()
```

OUTPUT

```
Enter a : 10
Error in input in expression input("Enter a : s")
```

Although there is no naming convention for naming a user-defined exception, it is better to define exceptions with names that end in `"Error"` to make it consistent with the naming of the standard exceptions.

Note Many standard modules define their own exceptions to report errors that may occur in functions they define.

12.11 THE finally BLOCK

The `try` block has another optional block called `finally` which is used to define clean-up actions that must be executed under all circumstances. The `finally` block is always executed before leaving the `try` block. This means that the statements written in `finally` block are executed irrespective of whether an exception has occurred or not. The syntax of finally block can be given as,

```
try:
   Write your operations here
   ......................
   Due to any exception, operations written here will be skipped
finally:
   This would always be executed.
   ......................
```

Let us see with the help of a program how `finally` block will behave when an exception is raised in the `try` block and is not handled by `except` block.

Example 12.14 Program with `finally` block that leaves the exception unhandled

```
try:
    print("Raising Exception.....")
    raise ValueError
finally:
    print("Performing clean up in Finally......")
```

OUTPUT

```
Raising Exception.....
Performing clean up in Finally......
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 4, in <module>
    raise ValueError
ValueError
```

From the above code, we can conclude that when an exception occurs in the `try` block and is not handled by an `except` block or if the exception occurs in the `except` or `else` block, then it is re-raised after executing the `finally` block. The `finally` block is also executed when any block of the `try` block is exited via a `break`, `continue` or `return` statement.

Now, let us see the flow of control in a program that has `try`, `except`, as well as `finally` block in the program given below.

Example 12.15 Program to illustrate the use of `try`, `except` and `finally` block all together

```
try:
    print("Raising Exception.....")
    raise ValueError
except:
    print("Exception caught.....")
finally:
    print("Performing clean up in Finally......")
```

OUTPUT

```
Raising Exception.....
Exception caught.....
Performing clean up in Finally......
```

From the output, you can see that the `finally` block is executed when exception occurs and also when an exception does not occur.

In real world applications, the `finally` clause is useful for releasing external resources like file handles, network connections, memory resources, etc. regardless of whether the use of the resource was successful.

Note You cannot have an `else` block with a `finally` block.

If you place the `finally` block immediately after the `try` block and followed by the `execute` block (may be in case of a nested `try` block), then if an exception is raised in the `try` block, the code in `finally` will be

executed first. The `finally` block will perform the operations written in it and then re-raise the exception. This exception will be handled by the `except` block if present in the next higher layer of the `try-except` block. This is shown in the program given below.

Example 12.16 Program having `finally` block to re-raise the exception that will be handled by an outer `try-except` block

```
try:
   print("Dividing Strings....")
   try:
      quo = "abc" / "def"
   finally:
      print("In finally block.....")
except TypeError:
   print("In except block.. handling TypeError...")
```

Programming Tip: `finally` block can never be followed by an except block.

OUTPUT

```
Dividing Strings....
In finally block.....
In except block.. handling TypeError...
```

12.12 PRE-DEFINED CLEAN–UP ACTION

In Python, some objects define standard clean-up actions that are automatically performed when the object is no longer needed. The default clean-up action is performed irrespective of whether the operation using the object succeeded or failed. We have already seen such an operation in file handling. We preferred to open the file using `with` keyword so that the file is automatically closed when not in use. So, even if we forget to close the file or the code to close it is skipped because of an exception, the file will still be closed. Consider the code given below, which opens a file to print its contents on the screen.

```
file = open('File1.txt')
str = file.readline()
print(str)
```

The code is perfectly alright except for one thing that it does not close the file after use. So the file is opened for an indeterminate amount of time after the code has finished executing. This may not be a big issue when writing small and simple programs, but can be a problem for large applications. Therefore, the `with` statement allows objects like files to be cleaned up when not in use. The better version of the code given above is therefore,

```
with open('File1.txt') as file:
    for line in file:
        print(line)
```

OUTPUT

```
Hello
# Welcome to the world of Programming
```

```
Python is a very simple and interesting language
Happy Reading
```

In the aforementioned program, after printing the contents of the file there are no more statements to execute. So just before the program completes its execution, the file is closed. The file would have closed even if any problem had occurred while executing the code.

Programming Tip: Many standard modules define exceptions in a separate file known as exceptions.py or errors.py.

12.13 RE-RAISING EXCEPTION

Python allows programmers to re-raise an exception. For example, an exception thrown from the `try` block can be handled as well as re-raised in the `except` block using the keyword `raise`. The code given below illustrates this concept.

Example 12.17 Program to re-raise the exception

```
try:
   f = open("Abc123.txt")  # opening a non-existent file
except:
   print("File does not exist")
   raise    # re-raise the caught exception
```

OUTPUT

```
File does not exist
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 2, in <module>
    f = open("Abc123.txt")  # opening a non-existent file
IOError: [Errno 2] No such file or directory: 'Abc123.txt'
```

Note To re-raise, use the `raise` keyword without any arguments.

12.14 ASSERTIONS IN PYTHON

An *assertion* is a basic check that can be turned on or off when the program is being tested. You can think of `assert` as a `raise-if` statement (or a raise-if-not statement). Using `assert` statement, an expression is tested, and if the result of the expression is `False` then an exception is raised. The `assert` statement is intended for debugging statements. It can be seen as an abbreviated notation for a conditional `raise` statement.

In Python, assertions are implemented using `assert` keyword. Assertions are usually placed at the start of a function to check for valid input, and after a function call to check for valid output.

When Python encounters an `assert` statement, the expression associated with it is calculated and if the expression is `False`, an `AssertionError` is raised. The syntax for `assert` statement is:

```
assert expression[, arguments]
```

If the expression is `False` (also known as assertion fails), Python uses `ArgumentExpression` as the argument for the `AssertionError`. `AssertionError` exceptions can be caught and handled like any other exception using the `try-except` block. However, if the `AssertionError` is not handled by the program, the program will be terminated and an error message will be displayed. In simple words, the `assert` statement, is semantically equivalent to writing,


```
assert <expression>, <message>
```

The above statement means if the `expression` evaluates to `False`, an exception is raised and `<message>` will be printed on the screen.

Consider the program given below. The program prompts a user to enter the temperature in `Celsius`. If the temperature is greater than 32 degree `Fahrenheit`, then an `AssertionError` is raised. Since the exception is not handled, the program is abruptly terminated with an error message.

Note `assert` statement should be used for trapping user-defined constraints.

Example 12.18 Program to use the `assert` statement

```
c = int(input("Enter the temperature in Celsius: "))
f = (c * 9/5) + 32
assert(f<=32), "Its freezing"
print("Temperature in Fahrenheit = ", f)
```

OUTPUT

```
Enter the temperature in Celsius: 100
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 3, in <module>
    assert(f<=32), "Its freezing"
AssertionError: Its freezing
```

Key points to remember

1. Do not catch exceptions that you cannot handle.
2. User defined exceptions can be very useful if some complex or specific information has to be stored in exception instances.
3. Do not create new exception classes when the built-in exceptions already have all the functionality you need.

Programming Tip: When we are developing a large program, it is a good practice to place all the user-defined exceptions that the program may raise in a separate file.

PROGRAMMING EXAMPLES

Program 12.1 Write a program that prompts the user to enter a number and prints its square. If no number is entered (Ctrl + C is pressed), then a `KeyboardInterrupt` is generated.

```
num = int(input("Enter the numerator : "))
deno = int(input("Enter the denominator : "))
try:
  quo = num/deno
  print("QUOTIENT : ", quo)
except ZeroDivisionError:
  print("Denominator cannot be zero")
```

OUTPUT

```
Enter the numerator : 10
Enter the denominator : 0
Denominator cannot be zero
```

Program 12.2 Write a program that opens a file and writes data to it. Handle exceptions that can be generated during the I/O operations.

```
try:
   with open('myFile.txt','w') as file:
      file.write("Hello, Good Morning !!!")
except IOError:
   print("Error working with file")
else:
   print("File Writing Successful ......")
```

Programming Tip: `assert` should not be used to catch divide by zero errors because Python traps such programming errors itself.

OUTPUT

```
File Writing Successful ......
```

Program 12.3 Write a program that deliberately raises a user-defined `SocketError` with any number of arguments and derived from class `Runtime`.

```
class SocketError(RuntimeError):
   def __init__(self, *arg):  # * because any number of arguments can be passed
      self.args = arg
try:
   raise SocketError('Socket', 'Establishment', 'Error')
except SocketError as e:
   print(e.args)
```

OUTPUT

```
('Socket', 'Establishment', 'Error')
```

Program 12.4 Write a program that prompts the user to enter a number. If the number is positive or zero print it, otherwise raise an exception.

```
try:
   num = int(input("Enter a number : "))
   if num >= 0:
      print(num)
   else:
      raise ValueError("Negative number not allowed")
except ValueError as e:
   print(e)
```

OUTPUT

```
Enter a number : -1
Negative number not allowed
```


Program 12.5 Write a number game program. Ask the user to enter a number. If the number is greater than number to be guessed, raise a `ValueTooLarge` exception. If the value is smaller the number to be guessed then, raise a `ValueTooSmall` exception and prompt the user to enter again. Quit the program only when the user enters the correct number.

```
class ValueTooSmallError(Exception):
   def display(self):
      print("Input value is too small")
class ValueTooLargeError(Exception):
   def display(self):
      print("Input value is too large")
max = 100
while 1:
   try:
      num = int(input("Enter a number: "))
      if num == max:
         print("Great you succeeded....")
         break
      if num < max:
          raise ValueTooSmallError
      elif num > max:
          raise ValueTooLargeError
   except ValueTooSmallError as s:
      s.display()
   except ValueTooLargeError as l:
      l.display()
```

OUTPUT

```
Enter a number: 20
Input value is too small
Enter a number: 102
Input value is too large
Enter a number: 100
Great you succeeded....
```

Program 12.6 Write a program that prints the first 30 numbers. Each number should be printed after a fixed short interval of time. Make use of a timer which prints each number when the timer goes off and exception is generated.

```
class TimeUp(Exception):
   pass
def message(c):
   start_timer = 0
   stop_timer = 10000
   count = start_timer
   try:
      while True:
```

```
        count += 1
        if count == stop_timer:
            raise TimeUp
    except TimeUp as t:
        print(c, end = " ")
for i in range(31):
    message(i)
```

OUTPUT

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

Program 12.7 Write a program which infinitely prints natural numbers. Raise the StopIteration exception after displaying first 20 numbers to exit from the program.

```
def display(n):
    while True:
        try:
            n = n+1
            if n == 21:
                raise StopIteration
        except StopIteration:
            break
        else:
            print(n, end = " ")
i = 0
display(i)
```

OUTPUT

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Program 12.8 Write a program that randomly generates a number. Raise a user-defined exception if the number is below 0.1.

```
import random
class RandomError(Exception):
    pass
try:
    num = random.random()
    if num < 0.1:
        raise RandomError
except RandomError as e:
    print("Random Error Generated ....")
else:
    print("%.3f"%num)
```

Programming Tip: You should only catch exceptions that you are willing to handle.

OUTPUT

```
0.696 (# Any random number will be generated)
```


Program 12.9 Write a program that validates name and age as entered by the user to determine whether the person can cast vote or not.

```
class invalidAge(Exception):
    def display(self):
        print("Sorry !!! Age cannot be below 18... You cannot vote .....")
class invalidName(Exception):
    def display(self):
        print("Please enter a valid name....")
try:
    name = input("Enter the name : ")
    if len(name) == 0:
        raise invalidName
    age = int(input("Enter the age : "))
    if age < 18:
        raise invalidAge
except invalidName as n:
    n.display()
except invalidAge as e:
    e.display()
else:
    print(name, " Congratulation !!! you can vote")
```

Programming Tip: Code in `else` block is executed if no exception was raised in the `try` block.

OUTPUT

```
Enter the name : Goransh
Enter the age : 10
Sorry !!! Age cannot be below 18... You cannot vote .....
```

Summary

- To handle an exception means to prevent it from causing the program to crash. Exceptions are handled using `try-except` block.
- Exceptions can be categorized as synchronous or asynchronous exceptions.
- *Synchronous* exceptions (like divide by zero, array index out of bound, etc.) can be controlled by the program.
- *Asynchronous* exceptions (like an interrupt from the keyboard, hardware malfunction, or, disk failure), are caused by events that are beyond the control of the program.
- Logical error may occur due to wrong algorithm or logic to solve a particular program.
- Exception is a Python object that represents an error.
- When a program raises an exception, it must handle the exception or the program will be immediately terminated.
- You can handle exceptions in your programs to end it gracefully, otherwise if exceptions are not handled by programs, then error messages are generated.
- Python allows you to have multiple `except` blocks for a single `try` block. The block which matches with the exception generated will get executed.
- After execution of the `except` block, the program control goes to the first statement after the `except` block for that `try` block.
- The statement(s) in the `else` block is executed only if the `try` clause does not raise an exception.
- You can deliberately raise an exception by using the `raise` keyword.
- Python allows programmers to create their own exceptions by creating a new exception class. The new exception