PGR107
Python Programming

Lecture 8 – Files & Exceptions

Chapter 7
Files and Exceptions

Chapter Goals

- To read and write text files
- To process collections of data
- To process command line arguments
- To raise and handle exceptions

Reading and Writing Text Files

- **Opening a File:** To access a file, you must first **open** it. When you open a file, you give the name of the file, or, if the file is stored in a different directory, the file name preceded by the directory path. You also specify whether the file is to be opened for **reading** or **writing**. Suppose you want to read data from a file named "**input.txt**", located in the same directory as the program.

```
infile = open("input.txt", "r")
```

- This statement opens the file for reading (indicated by the string argument "**r**") and returns a file object that is associated with the file named input.txt. When opening a file for reading, the file must exist or an exception occurs.

Reading and Writing Text Files

- To open a file for **writing**, you provide the name of the file as the first argument to the open function and the string "**w**" as the second argument:

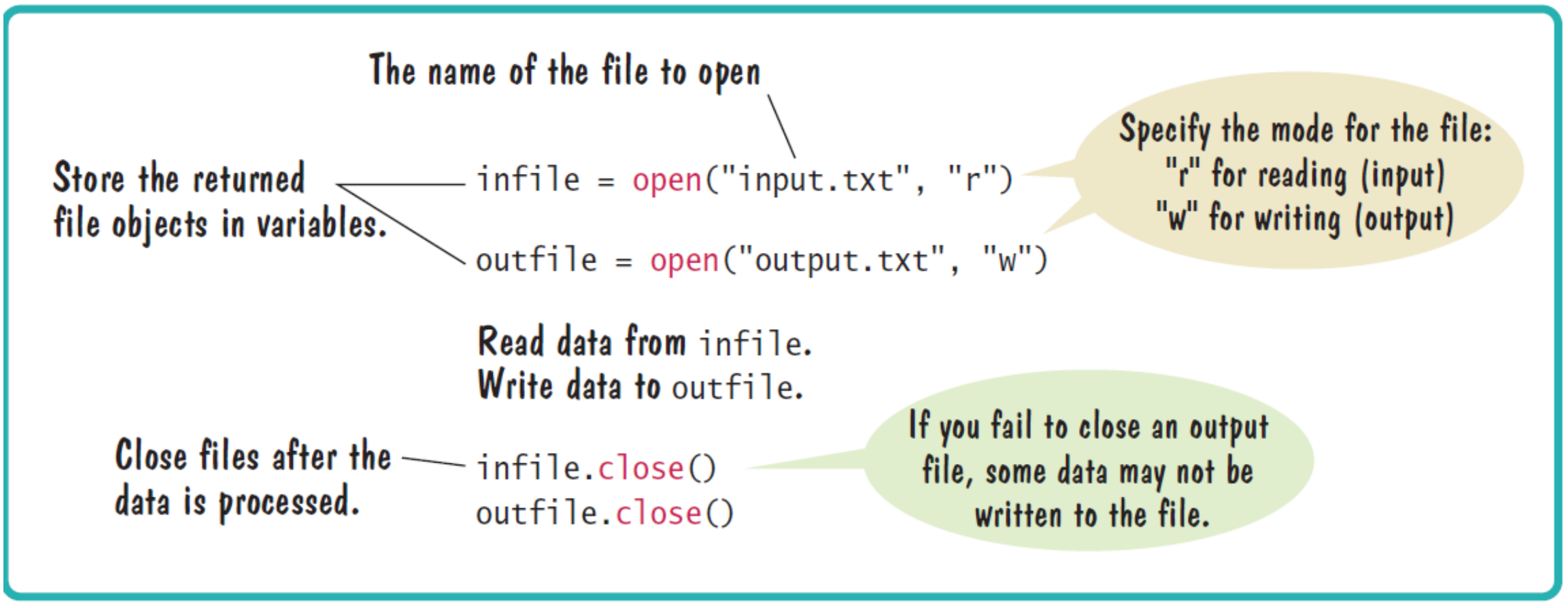
```
outfile = open("output.txt", "w")
```

- If the output file already exists, it is emptied before the new data is written into it. If the file does not exist, an empty file is created. When you are don processing a file, be sure to close the file using the **close** method:

```
infile.close()
outfile.close()
```

- If your program exits without closing a file that was opened for writing, some of the output may not be written to the disk file.

Reading and Writing Text Files



Reading and Writing Text Files

- **Reading from a File**: To read a line of text from a file, call the **readline method** on the file object that was returned when you opened the file:

```
line = infile.readline()
```

- Reading multiple lines of text from a file is very similar to reading a sequence of values with the input function. You repeatedly read a line of text and process it until the sentinel value is reached:

```
line = infile.readline()
while line != "" :
    Process the line.
    line = infile.readline()
```

Reading and Writing Text Files

- As with the input function, the readline method can return only strings. If the file contains numerical data, the strings must be converted to the numerical value using the int or float function:

```
value = float(line)
```

- Note that the newline character at the end of the line is ignored when the string is converted to a numerical value.

Reading and Writing Text Files

- **Writing to a File:** You can write text to a file that has been opened for writing. This is done by applying the write method to the file object. For example, we can write the string "Hello, World!" to our output file using the statement:

```
outfile.write("Hello, World!\n")
```

- You can also write formatted strings to a file with the write method:

```
outfile.write("Number of entries: %d\nTotal: %8.2f\n" % (count, total))
```

Reading and Writing Text Files

- **A File Processing Example:**

```
32.0
54.0
67.5
80.25
115.0
```

input.txt

```
                32.00
                54.00
                67.50
                80.25
               115.00
              --------
Total:         348.75
Average:        69.75
```

output.txt

Reading and Writing Text Files

- **Backslashes in File Names:** When you specify a file name as a string literal, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

```
infile = open("c:\\homework\\input.txt", "r")
```

- A single backslash inside a quoted string is an **escape character** that is combined with the following character to form a special meaning, such as **\n** for a **newline character**. The \\ combination denotes a single backslash.

- When a program user supplies a file name to a program, however, the user should not type the backslash twice.

Text Input and Output

- **Iterating over the Lines of a File:** To read the lines of text from the file, you can iterate over the file object using a **for loop**.

```
for line in infile :
    print(line)
```

You can iterate over a file object to read the lines of text in the file.

- Suppose we have an input file that contains a collection of words, stored one per line:

```
spam
and
eggs
```

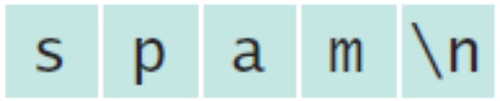
```
spam

and

eggs
```

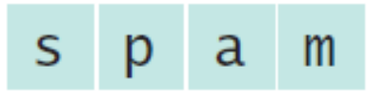
Text Input and Output

- When the first line of the text file is read, the string line contains:



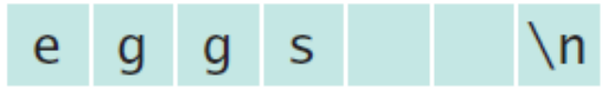
- To remove the newline character, apply the **rstrip method** to the string.

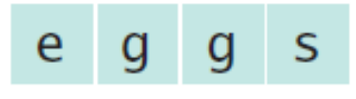
```
line = line.rstrip()
```



Use the rstrip method to remove the newline character from a line of text.

- By default, the rstrip method creates a new string in which all white space (**blanks, tabs, and newlines**) at the end of the string has been removed. For example, if there are two blank spaces following the word eggs in the third line of text, the rstrip method will remove not only the newline character but also the blank spaces:





Text Input and Output

- To remove specific characters from the end of a string, you can pass a string argument containing those characters to the rstrip method. For example, if we need to remove a **period** or a **question mark** from the end of string, we can use the command:

```
line = line.rstrip(".?")
```

Method	Returns
s.lstrip() *s*.lstrip(*chars*)	A new version of *s* in which white space (blanks, tabs, and newlines) is removed from the left (the front) of *s*. If provided, characters in the string *chars* are removed instead of white space.
s.rstrip() *s*.rstrip(*chars*)	Same as lstrip except characters are removed from the right (the end) of *s*.
s.strip() *s*.strip(*chars*)	Similar to lstrip and rstrip, except characters are removed from the front and end of *s*.

Character Stripping Examples

Statement	Result	Comment
`string = "James\n"` `result = string.rstrip()`	J a m e s	The newline character is stripped from the end of the string.
`string = "James \n"` `result = string.rstrip()`	J a m e s	Blank spaces are also stripped from the end of the string.
`string = "James \n"` `result = string.rstrip("\n")`	J a m e s	Only the newline character is stripped.
`name = " Mary "` `result = name.strip()`	M a r y	The blank spaces are stripped from the front and end of the string.
`name = " Mary "` `result = name.lstrip()`	M a r y	The blank spaces are stripped only from the front of the string.

Text Input and Output

- **Reading Words:** Sometimes you may need to read the individual words from a text file. For example, suppose our input file contains two lines of text:

```
Mary had a little lamb,
whose fleece was white as snow.
```

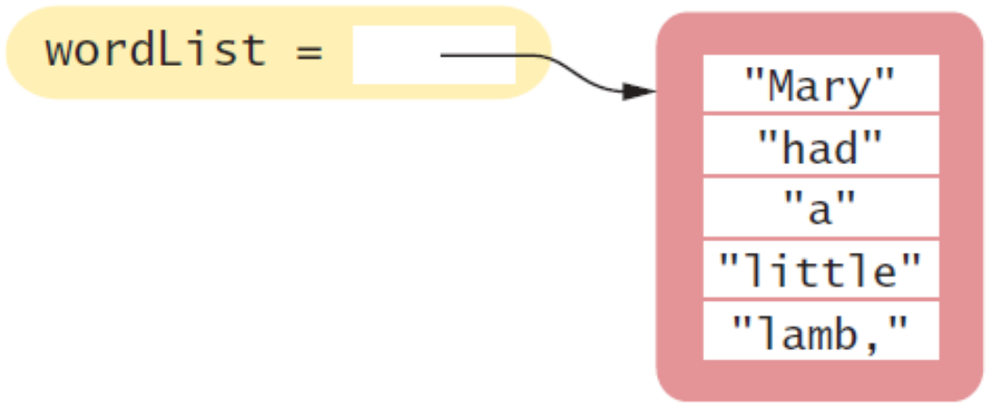
- Because there is no method for reading a word from a file, you must first read a line and then split it into individual words. This can be done using the **split method**:

```
wordList = line.split()
```

Text Input and Output

- The split method returns the list of substrings that results from splitting the string at each blank space. For example, if line contains the string

- it will be split into 5 substrings that are stored in a list in the same order in which they occur in the string:




```
for word in wordList :
   print(word)
```

Text Input and Output

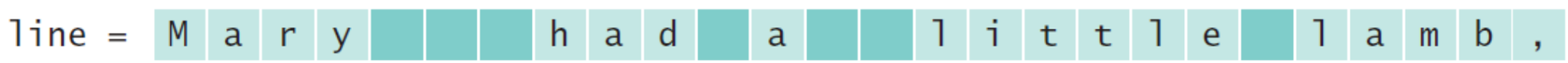
- The complete solution for our original task is:

```
inputFile = open("lyrics.txt", "r")
for line in inputFile :
   line = line.rstrip()
   wordList = line.split()
   for word in wordList :
      word = word.rstrip(".,?!")
      print(word)

inputFile.close()
```

Text Input and Output

- The split method treats consecutive blank spaces as a single delimiter. Thus, if the string contains multiple spaces between some or all of the words,



- **line.split ()** would still result in the same five substrings:

```
"Mary" "had" "a" "little" "lamb,"
```

Text Input and Output

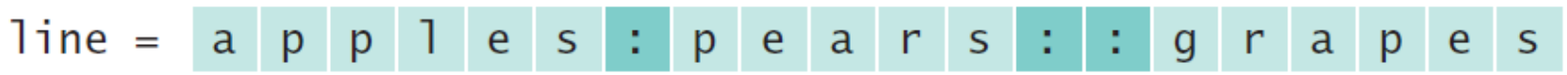
- By default, the split method uses white space characters as the delimiter. You can also split a string using a different delimiter. For example, if the words were separated by a colon instead of blank spaces,


```
substrings = line.split(":")
```

```
"apples" "pears" "oranges" "grapes"
```

Text Input and Output

- Note that when a delimiter is passed as an argument, consecutive delimiters are not treated as a single one, as was the case when no argument was supplied. Thus, the string




```
substrings = line.split(":")
```

```
"apples" "pears" "" "grapes"
```

Text Input and Output

- Additional methods for splitting strings:

Method	Returns

`s.split()`
`s.split(sep)`
`s.split(sep, maxsplit)`
	Returns a list of words from string *s*. If the string *sep* is provided, it is used as the delimiter; otherwise, any white space character is used. If *maxsplit* is provided, then only that number of splits will be made, resulting in at most *maxsplit* + 1 words.

`s.rsplit(sep, maxsplit)`	Same as `split` except the splits are made starting from the end of the string instead of from the front.
`s.splitlines()`	Returns a list containing the individual lines of a string split using the newline character `\n` as the delimiter.

String Splitting Examples

Statement	Result	Comment
`string = "a,bc,d"` `string.split(",")`	`"a" "bc" "d"`	The string is split at each comma.
`string = "a b  c"` `string.split()`	`"a" "b" "c"`	The string is split using the blank space as the delimiter. Consecutive blank spaces are treated as one space.
`string = "a b  c"` `string.split(" ")`	`"a" "b" "" "c"`	The string is split using the blank space as the delimiter. With an explicit argument, the consecutive blank spaces are treated as separate delimiters.
`string = "a:bc:d"` `string.split(":", 1)`	`"a" "bc:d"`	The string is split into 2 parts starting from the front. The split is made at the first colon.
`string = "a:bc:d"` `string.rsplit(":", 1)`	`"a:bc" "d"`	The string is split into 2 parts starting from the end. The split is made at the last colon.

Text Input and Output

- **Reading Characters:** Instead of reading an entire line, you can read individual characters with the **read method**.

- The **read method** takes a single argument that specifies the number of characters to read. The method returns a string containing the characters. When supplied with an argument of 1,

```
char = inputFile.read(1)
```

Read one or more characters with the read method.

- The **read method** returns a string consisting of the next character in the file. Or, if the end of the file is reached, it returns an empty string "".

Text Input and Output

- **Reading the Entire File:** There are two methods for reading an entire file. The call **inputFile.read ()** returns a string with all characters in the file. The **readlines method** reads the entire contents of a text file into a list:

```
inputFile = open("sample.txt", "r")
listOfLines = inputFile.readlines()
inputFile.close()
```

- Each element in the list returned by the **readlines method** is a string containing a single line from the file (including the newline character). Once the contents of the file are in the list, you can access lines in the list by position, as in **listOfLines[2]**. You can also iterate over the entire list:

```
for line in listOfLines :
   text = line.rstrip()
   print(text)
```

Text Input and Output

Operation	Explanation
f = open(*filename*, *mode*)	Opens the file specified by the string *filename*. The *mode* parameter indicates whether the file is opened for reading ("r") or writing ("w"). A file object is returned.
f.close()	Closes a previously opened file. Once closed, the file cannot be used until it has been reopened.
string = *f*.readline()	Reads the next line of text from an input file and returns it as a string. An empty string "" is returned when the end of file is reached.
string = *f*.read(*num*) string = *f*.read()	Reads the next *num* characters from the input file and returns them as a string. An empty string is returned when all characters have been read from the file. If no argument is supplied, the entire contents of the file is read and returned in a single string.
f.write(*string*)	Writes the *string* to a file opened for writing.

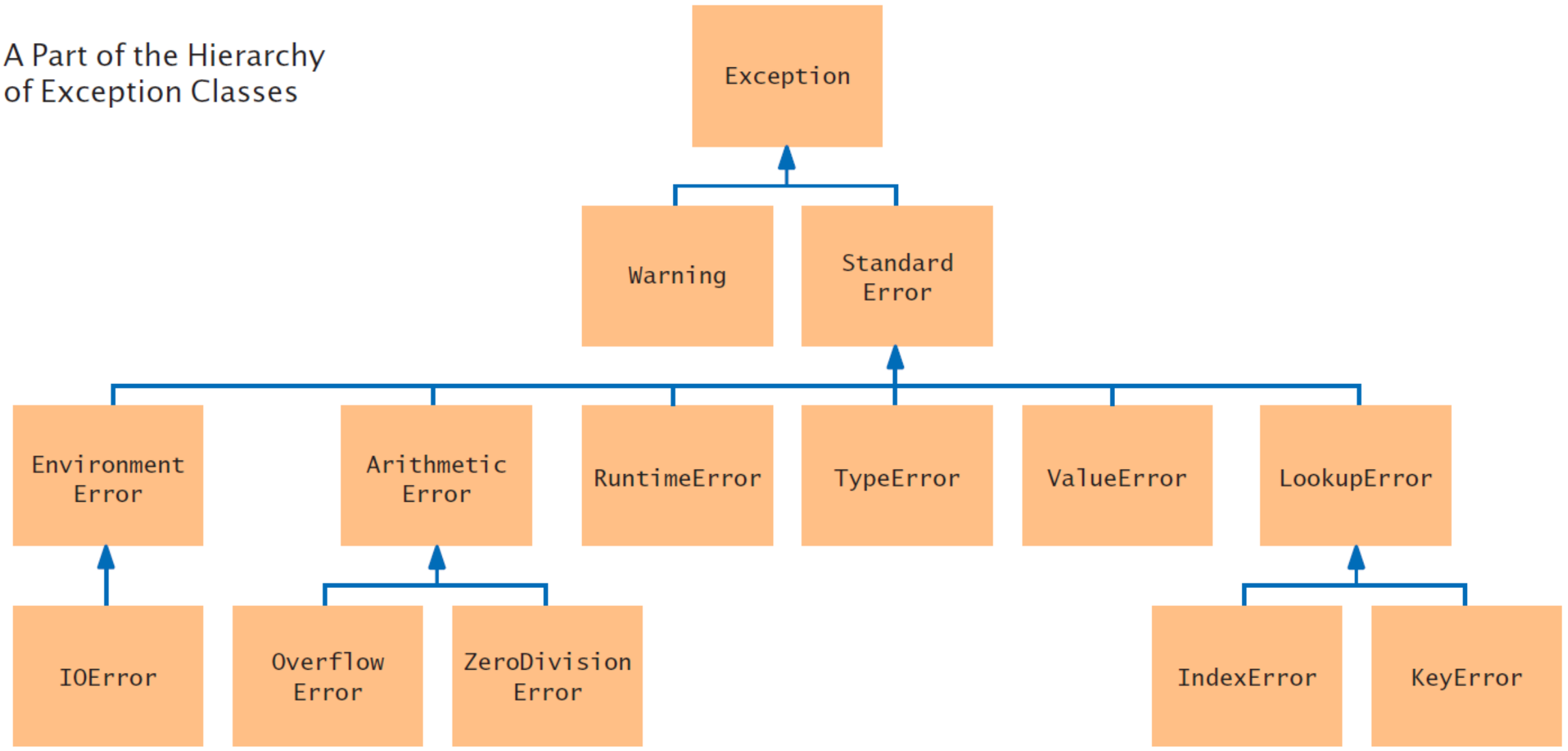
Exception Handling

- There are two aspects to dealing with program errors: detection and handling.

- For example, the open function can detect an attempt to read from a non-existent file. However, it cannot handle that error. A satisfactory way of handling the error might be to terminate the program, or to ask the user for another file name. The open function cannot choose between these alternatives. It needs to report the error to another part of the program.

Exception Handling

A Part of the Hierarchy of Exception Classes



Exceptions

- **SyntaxError:** Python cannot parse program
- **NameError:** local or global name not found
- **AttributeError:** attribute reference fails
- **TypeError**: operand does not have correct type
- **ValueError:** operand type ok, but value is illegal
- **IOError:** IO system reports malfunction (e.g. file not found)

Error Messages - Easy

- Trying to access beyond the limits of a list
  test = [1, 2, 3] then test [4] → IndexError

- Trying to convert an inappropriate type
  int (test) → TypeError

- Referencing a non-existent variable
  a → NameError

- Mixing data types without appropriate coercion
  ‘3’/ 4 → TypeError

- Forgetting to close parenthesis, quotation, etc.
  a = len ([1, 2, 3]
  print (a) → SyntaxError

Exception Handling

Raising an Exception

Syntax `raise` *exceptionObject*

```
if amount > balance :
    raise ValueError("Amount exceeds balance")

balance = balance - amount
```

This message provides detailed information about the exception.

A new exception object is constructed, then raised.

This line is not executed when the exception is raised.

Exception Handling

- The ***try*** block lets you test a block of code for errors

- The ***Except*** block lets you handle the error.

- The ***finally*** block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

- You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error.

- For example, Print one message if the try block raises a ***NameError*** and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Exception Handling

- You can use the ***else*** keyword to define a block of code to be executed if no errors were raised:

- For example, in the following code, the ***try*** block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Exception Handling

- **Handling Exceptions:** Every exception should be handled somewhere in your program.

```
try :
   filename = input("Enter filename: ")
   infile = open(filename, "r")
   line = infile.readline()
   value = int(line)
   . . .
except IOError :
   print("Error: file not found.")

except ValueError as exception :
   print("Error:", str(exception))
```

Place the statements that can cause an exception inside a try block, and the handler inside an except clause.

Exception Handling

- If either of these exceptions is actually raised, then the rest of the instructions in the try block are skipped. Here is what happens for the various exception types:

  ✓ If an **IOError** exception is raised, then the except clause for the IOError exception is executed.

  ✓ If a **ValueError** exception occurs, then the second except clause is executed.

  ✓ If any other exception is raised, it will not be handled by any of the except clauses of this try block. It remains raised until it is handled by another try block.

Exception Handling

Syntax

```
try :
   statement
   statement
   . . .
except ExceptionType :
   statement
   statement
   . . .
except ExceptionType as varName :
   statement
   statement
   . . .
```

This function can raise an IOError exception.

```
try :
   infile = open("input.txt", "r")

   line = inFile.readline()
   process(line)

except IOError :
   print("Could not open input file.")

except Exception as exceptObj :
   print("Error:", str(exceptObj))
```

When an IOError is raised, execution resumes here.

This is the exception object that was raised.

Additional except clauses can appear here. Place more specific exceptions before more general ones.

Exception Handling

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b= ", a+b)

except ValueError:
    print("Could not convert to a number.")

except ZeroDivisionError:
    print("Can't divide by zero")

except:
    print("Something went very wrong.")
```

Exception Handling

- **The finally Clause**: Occasionally, you need to take some action whether or not an exception is raised. The **finally construct** is used to handle this situation

```
outfile = open(filename, "w")
writeData(outfile)
outfile.close()    # May never get here.
```

- Now suppose that one of the methods or functions before the last line raises an exception. Then the call to close is never executed! You solve this problem by placing the call to close inside a finally clause:

```
outfile = open(filename, "w")
try :
   writeData(outfile)

finally :
   outfile.close()
```

Exception Handling

The `finally` Clause

Syntax

```
try :
   statement
   statement
   . . .
finally :
   statement
   statement
   . . .
```

```
outfile = open(filename, "w")
try :
   writeData(outfile)
   . . .
finally :
   outfile.close()
   . . .
```

This code may raise exceptions.

This code is always executed, even if an exception is raised in the `try` block.

The file must be opened outside the `try` block in case it fails. Otherwise, the `finally` clause would try to close an unopened file.

Exception Handling

- Use the **finally clause** whenever you need to do some clean up, such as closing a file, to ensure that clean up happens no matter how the method exits.

End of Chapter 7