ICS 104 - Introduction to Programming in Python and C

# Files and Exceptions

## Reading Assignment

- Chapter 7 Sections 1, 2 and 5.

# Chapter Learning Outcomes

## At the end of this chapter, you will be able to

- read and write text files
- process collections of data
- raise and handle exceptions

# Reading and Writing Text Files

## Opening a File for Reading

- To access a **file**, you must first **open** it.
- When you open a file, you give the name of the file.
    - If the file is stored in a different directory, the file name is preceded by the directory path.
- You also specify whether the file is to be opened for **reading** or **writing**.

In [ ]:

```
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 **input.txt**.
    - Stores the file object in a variable **infile**
- When opening a file for reading, the file must **exist** or an **exception** is raised.

# Opening a File for Writing

- To open a file for writing, use the following statement

In [ ]:

```
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.

- All operations for accessing a file are made via the file object.

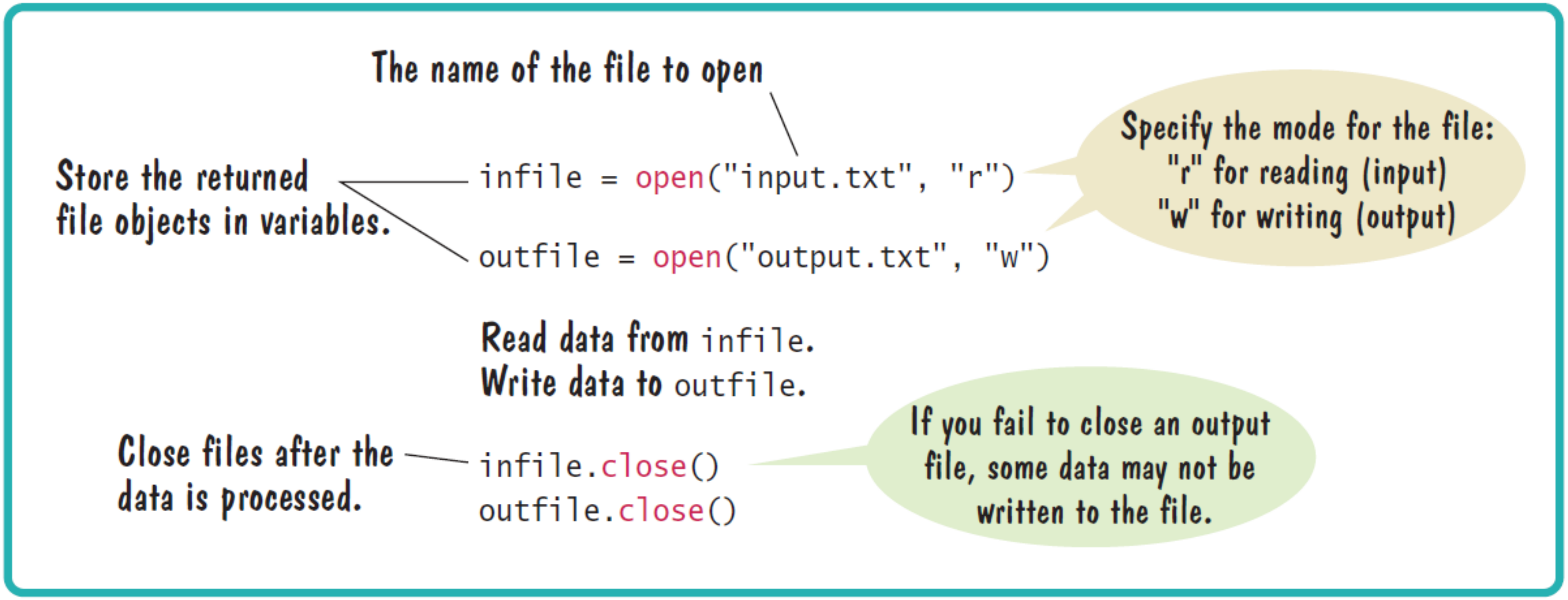
- When you are done processing a file, by sure to close the file using the close method:

In [ ]:

```
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 file.

# Reading and Writing Text Files



The name of the file to open

Store the returned file objects in variables.

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

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

Specify the mode for the file:
"r" for reading (input)
"w" for writing (output)

Read data from `infile`.
Write data to `outfile`.

Close files after the data is processed.

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

If you fail to close an output file, some data may not be written to the file.

# Reading 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.

- When a file is opened, an input marker is positioned at the beginning of the file.
- The **readline** method reads the text, starting at the current position and continuing until the newline character is encountered.

- The input marker is then moved to the next line.
- The **readline** method returns the text that it reads, including the newline character that denotes the end of the line.
    - Consider the input file "input.txt"

In [ ]:

```
# A sample use of readline
infile = open("input.txt","r")
line1 = infile.readline()
print(line1 + "The length of line1 is", len(line1))
line2 = infile.readline()
print(line2 + "The length of line2 is", len(line2))
line3 = infile.readline()
print(line3 + "The length of line3 is", len(line3))
infile.close()
```

- The first call to **readline** returns the string "flying\n".
    - Recall that \n denotes the newline character that indicates the end of the line.
- If you call **readline** a second time, it returns the string "circus".
    - Note that there is no "\n" since this was the last line in the text file, and you have reached the end of file marker.
- Calling **readline** again yields the empty string "" because you have already reached the end of file marker.

# Reading Multiple Lines of a File

- 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:

In [ ]:

```
infile = open("input.txt","r")
line = infile.readline()
print(line, end="")
while line !="" :
    line = infile.readline()
    print(line, end="")
print()
infile.close()
```

- 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:
    - For example, "`value = float(line)`"

## Writing a File

- You can write text to a file that has been opened for writing.
- This is done b applying the **write()** method to the file object.

- For example, we can write the string "Hello, World" to our output file using the statement:

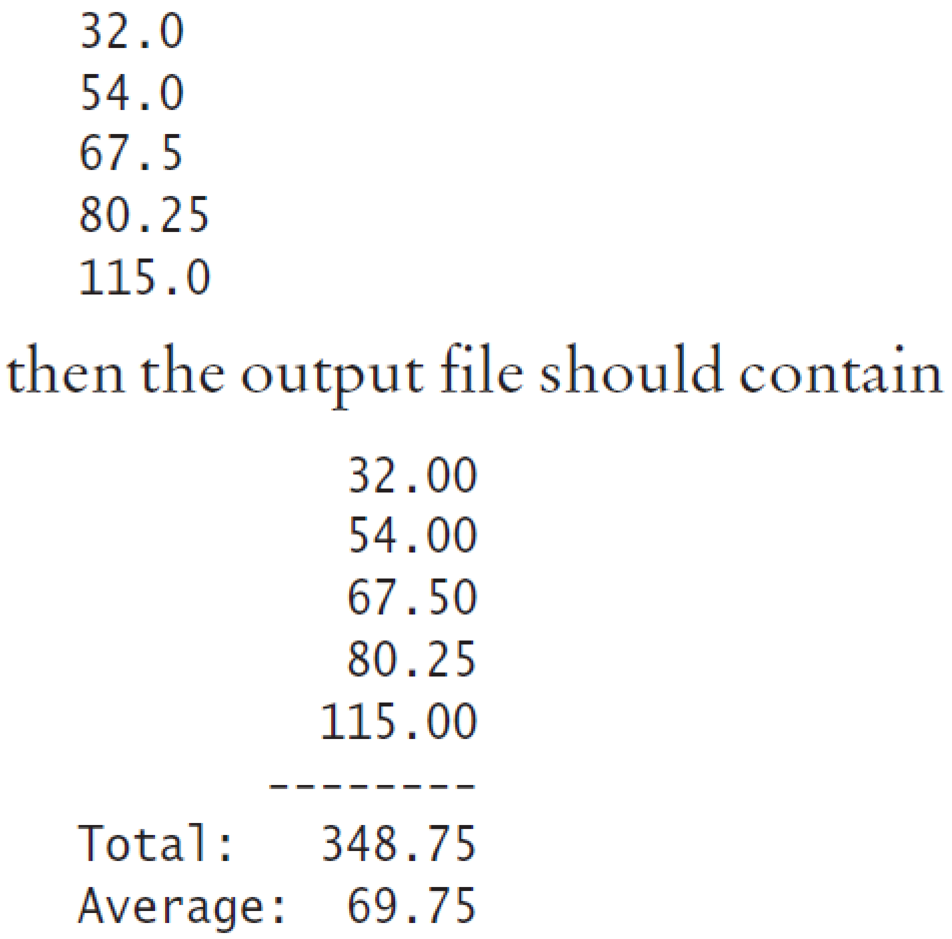
In [ ]:

```
outfile = open("output.txt","w")
outfile.write("Hello, World!\n")
outfile.close()
```

# A File Processing Example

- Suppose you are given a text file that contains a sequence of floating-point values, stored one value per line.
- You need to read the values and write them to a new output file, aligned in a column and followed by their total and average values.

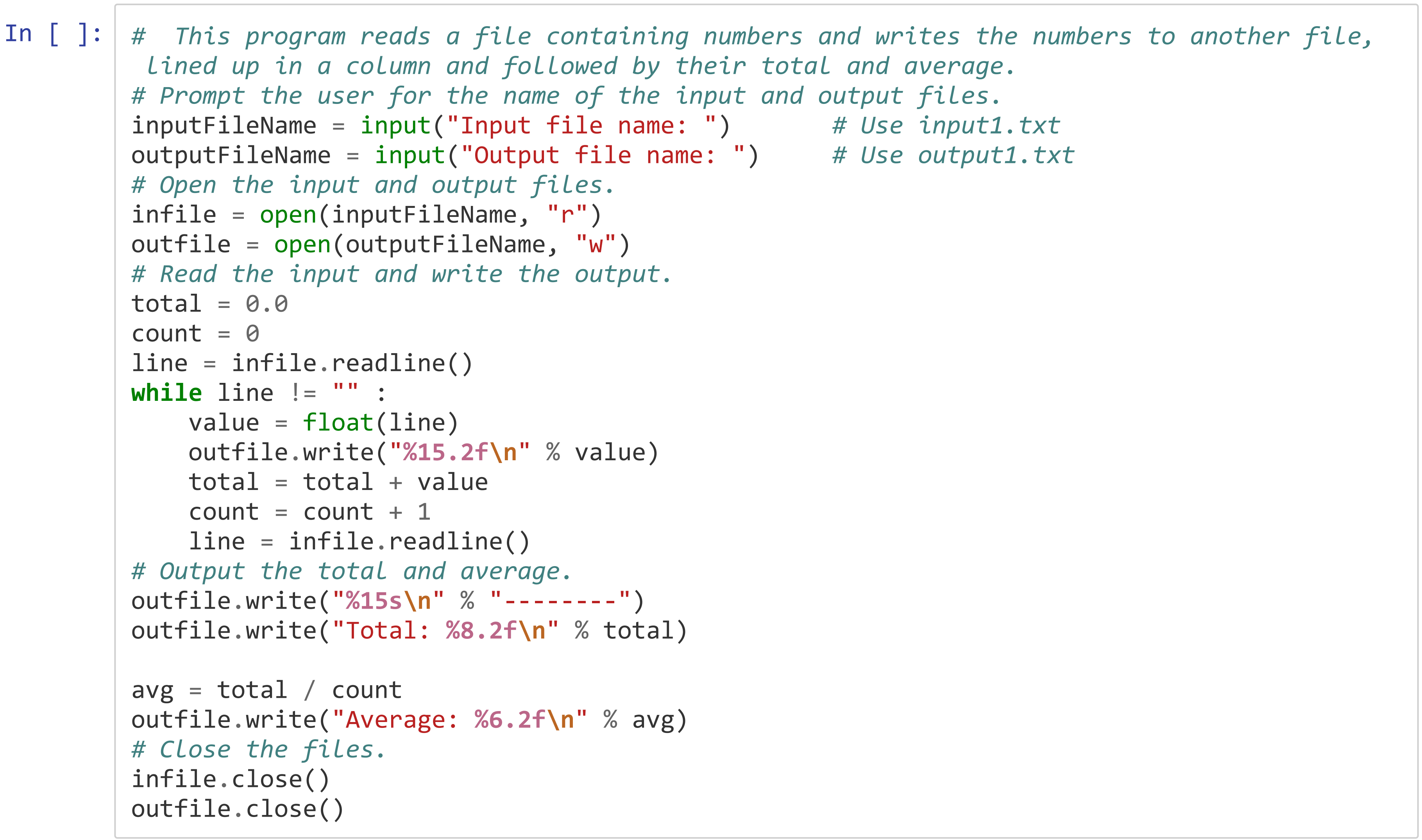
- If the input file has the content:



```
32.0
54.0
67.5
80.25
115.0
```

then the output file should contain

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



```python
#  This program reads a file containing numbers and writes the numbers to another file, lined up in a column and followed by their total and average.
# Prompt the user for the name of the input and output files.
inputFileName = input("Input file name: ")        # Use input1.txt
outputFileName = input("Output file name: ")     # Use output1.txt
# Open the input and output files.
infile = open(inputFileName, "r")
outfile = open(outputFileName, "w")
# Read the input and write the output.
total = 0.0
count = 0
line = infile.readline()
while line != "" :
    value = float(line)
    outfile.write("%15.2f\n" % value)
    total = total + value
    count = count + 1
    line = infile.readline()
# Output the total and average.
outfile.write("%15s\n" % "--------")
outfile.write("Total: %8.2f\n" % total)

avg = total / count
outfile.write("Average: %6.2f\n" % avg)
# Close the files.
infile.close()
outfile.close()
```

# 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.

In [ ]:

```
infile = open("input.txt","r")
for line in infile:
    print(line)
infile.close()
```

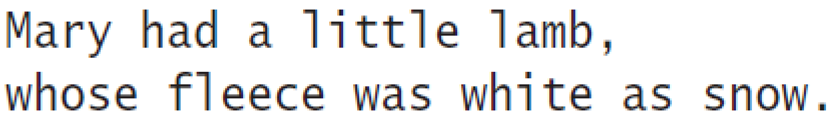
- Note, when the lines of input are printed to the terminal, they are displayed with a blank line between each word:
- To remove the newline character, apply the **rstrip** method to the string.
    - The `rstrip()` method removes all trailing white spaces (tabs, spaces and newlines) from the end of the string when called without an argument.
        - If we supply an argument, it will remove the trailing characters in the argument.

In [ ]:

```
infile = open("input.txt","r")
for line in infile:
    line = line.rstrip()
    print(line)
infile.close()
```

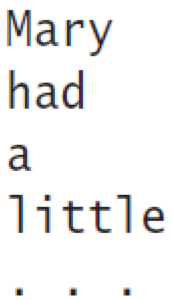
# Reading Words

- 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.
```

- that we would like to print to the terminal, one word per line



```
Mary
had
a
little
. . .
```

- 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:

In [ ]:

```
infile = open("7.2.2.txt","r")
for line in infile:
    wordList = line.split()
    print(wordList)
infile.close()
```

In [ ]:

```
infile = open("7.2.2.txt","r")
for line in infile:
    wordList = line.split()
    for word in wordList:
        print(word)
infile.close()
```

# Reading Words - Student Activity

- Notice that the last word in the last output contains punctuation marks.
- If you want to print the words contained in the file without punctuation marks, which function we can use?

In [ ]:

```
infile = open("7.2.2.txt","r")
for line in infile:
    wordList = line.split()
    for word in wordList:
        word = word.rstrip(".,?!")
        print(word)
infile.close()
```

# 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)
- 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 "".

In [ ]:

```
inputFile = open("input.txt","r")
char = inputFile.read(1)
while char !="":
    char = inputFile.read(1)
    print(char)
inputFile.close()
```

# Reading Records

- A text file can contain a collection of **data records** in which each record consists of multiple fields.
- For example, a file containing student data may consist of records composed of fields for an identification number, full name, address, and class year.

- A file containing bank account transactions may contain records composed of the transaction date, description, and amount fields.
- When working with text files that contain data records, you generally have to read the entire record before you can process it:

- For each record in file
    - Read the entire record.
    - Process the record.

# 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 the alternatives.
    - It needs to report the error to another part of the program.

- In Python, **exception handling** provides a flexible mechanism for passing control from the point of **error** detection to a handler that can deal with the error.

## Raising Exceptions

- When you detect an error condition, your job is really easy.
- You just **raise** an appropriate exception, and you are done.

- For example, suppose someone tries to withdraw too much money from a bank account:

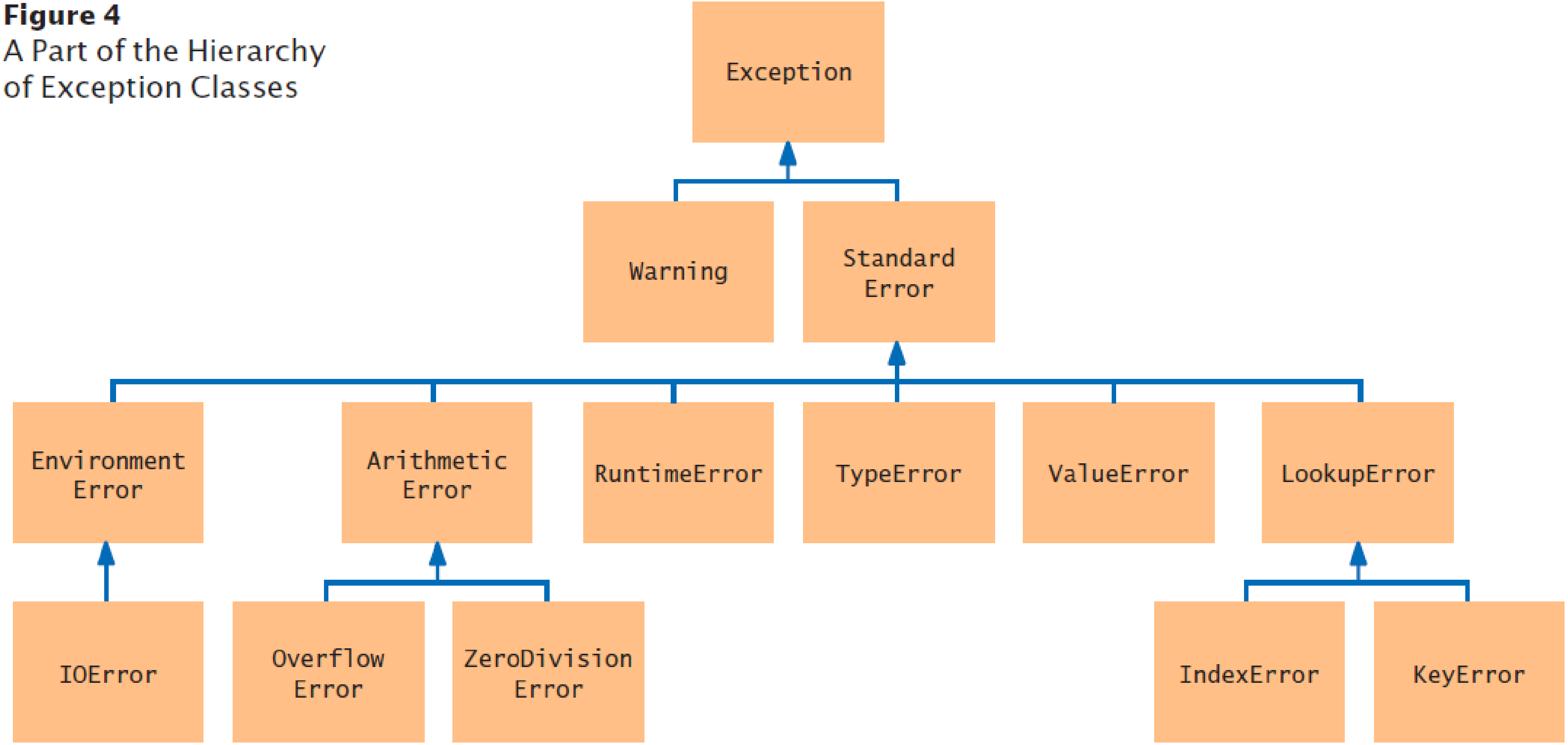
In [ ]:

```
# if amount > balance:
    # Now what?
```

- First look for an appropriate exception.
- The Python library provides a number of standard exceptions to signal all sorts of exceptional conditions.

# Standard Exceptions

**Figure 4**
A Part of the Hierarchy of Exception Classes



- Look around for an exception type that might describe your situation?

- How about the ArithematicError exception? Is it an arithmetic error to have a negative balance?

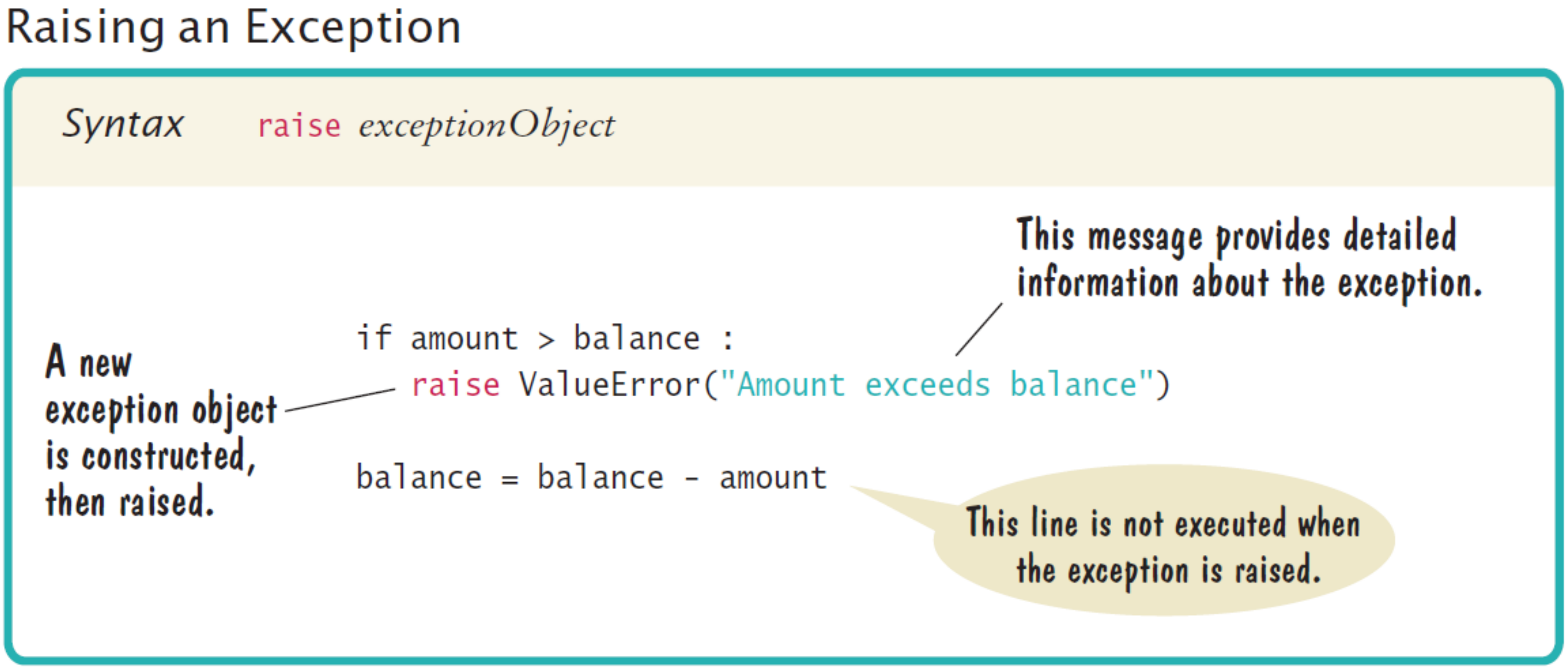
- No, Python can deal with negative numbers.

- Is the amount to be withdrawn an illegal value?
    - Indeed it is. It is just too large.
    - Therefore, let's raise a ValueError exception.

In [ ]:

```
amount = 100
balance = 50
if amount > balance:
    raise ValueError("Amount exceeds balance")
```

# Raising Exceptions



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.

- When you raise an exception, execution does not continue with the next statement but with an *exception handler*.
    - Every exception should be handled somewhere in your program.
    - If an exception has no handler, an error message is printed, and your program terminates.

# Handling Exceptions

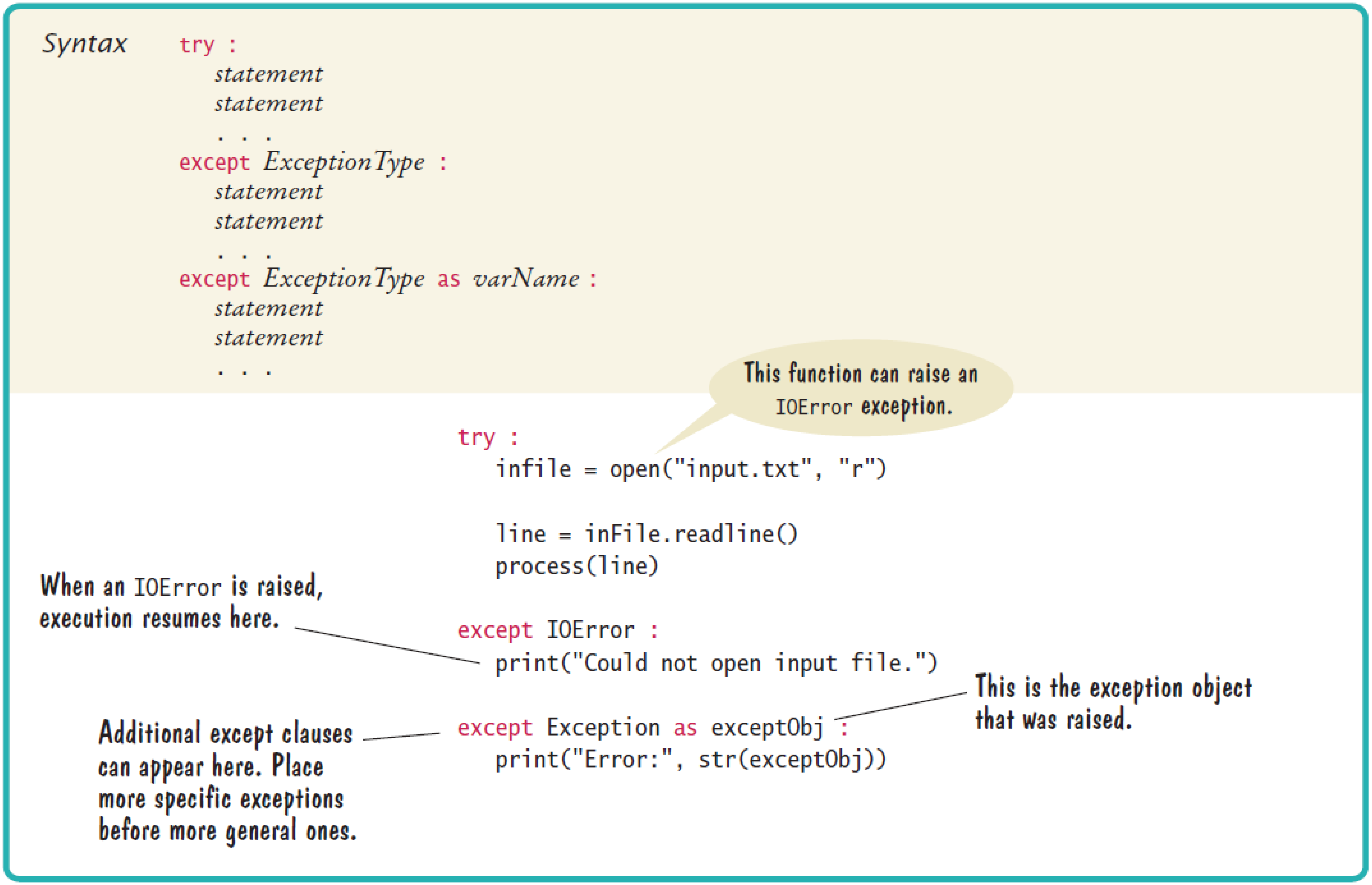
- You handle exceptions with the **try/except** statement.
- Place the statement into a location of your program that knows how to handle a particular exception.

- The **try block** contains one or more statements that may cause an exception of the kind that you are willing to handle.
- Each except clause contains the handler for an exception type.

In [ ]:

```
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))
```

# General Syntax for Handling Exceptions



*Syntax*

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

```
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))
```

This function can raise an IOError exception.

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.

# 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.

- For example, it is important to close an output file to ensure that all output is written to the file.

In [ ]:

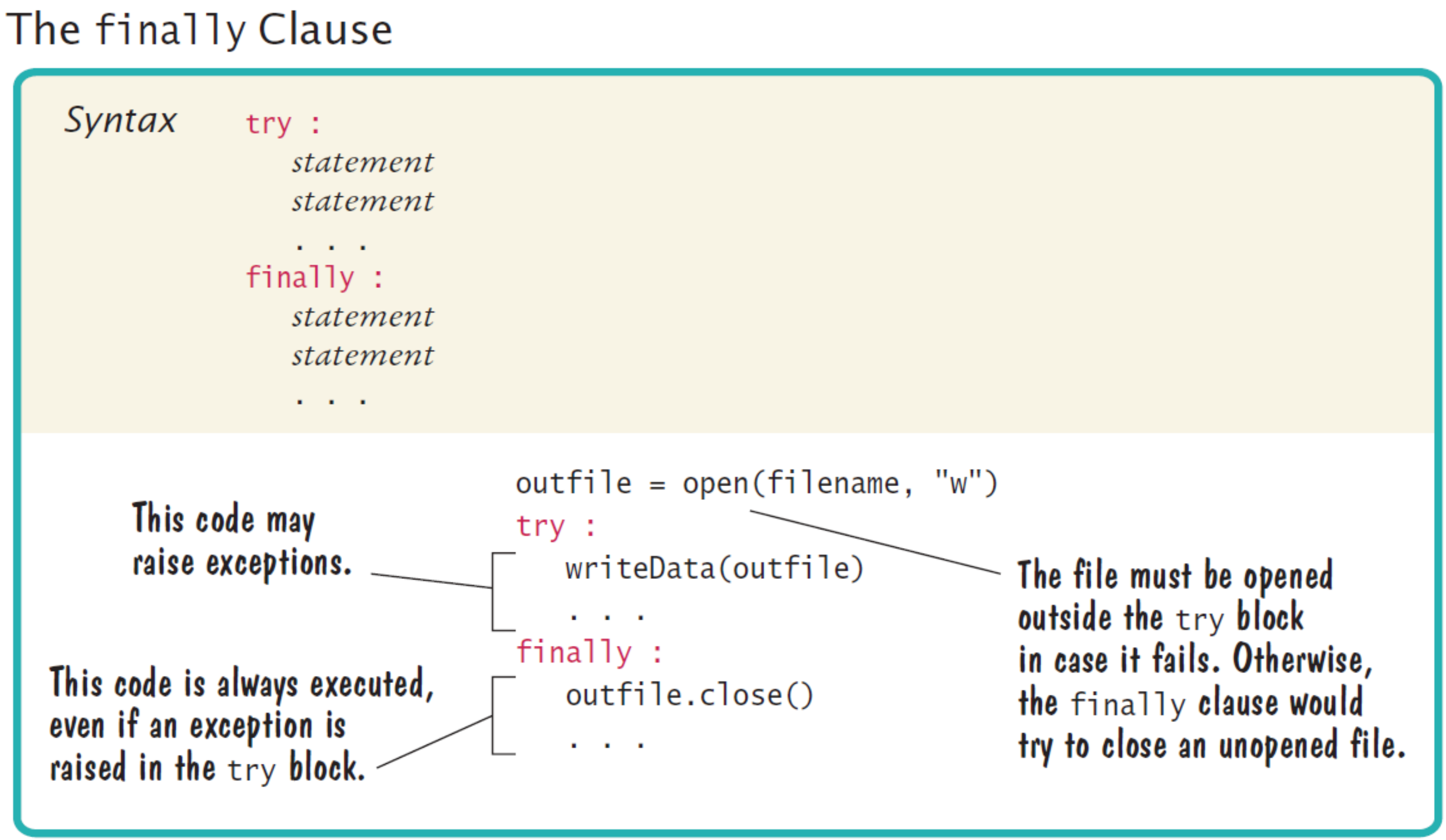
```
filename= input("Enter filename: ")
outfile = open(filename, "w")
try:
    outfile.write("Hello World\n")
    value = 1 / 0
    outfile.close()
except ArithmeticError as exception:
    print("Error:",str(exception))
```

- Since **ArithmeticError** exception is raised, the call to close is never executed.
- You solve this problem by placing the call to **close** inside a **finally clause**:

In [ ]:

```
filename= input("Enter filename: ")
outfile = open(filename, "w")
try:
    outfile.write("Hello World\n")
    value = 1 / 0
except ArithmeticError as exception:
    print("Error:",str(exception))
finally:
    outfile.close()
```

# Syntax of the Finally Clause



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.

- The `finally` block is always executed after leaving the try statement.
- In case if some exception was not handled by except block, it is re-raised after execution of finally block.

# Summary

- When opening a file, you supply the name of the file stored on disk and the mode in which the file is to be opened.
- Close all files when you are done processing them.
- Use the readline() method to obtain lines of text from a file.
- Write to a file using the write() method or the pring() function.

# Summary

- You can iterate over a file object to read the lines of text in the file.
- Use the rstrip() method to remove the newline character from a line of text.
- Use the split() method to split a string into individual words.
- Read one or more characters with the read() method.

# Summary

- To signal an exception condition, use the `raise` statement to raise an exception object.
- When you raise an exception, processing continues in an exception handler.
- Place the statements that can cause an exception inside a try block, and the handler inside an except clause.
- Once a try block is entered, the statements in a finally clause are guaranteed to be executed, whether or not an exception is raised.