

Chapter Seven

FILES AND EXCEPTIONS

Chapter Goals

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

*In this chapter, you will learn how to write programs that
manipulate files*

Contents

- Reading and Writing Text Files
- Text Input and Output

Reading and Writing Text Files

SECTION 7.1

Reading and Writing Text Files

- Text files are very commonly used to store information
 - They are the most 'portable' types of data files
- Examples of text files include files that are created with a simple text editor, such as Windows Notepad, and Python source code and HTML files

Opening Files: Reading

- To access a file, you must first *open* it
- Suppose you want to read data from a file named `input.txt`, located in the same directory as the program
- To open a file for reading, you must provide the name of the file as the first argument to the `open` function and the string `"r"` as the second argument:

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

Opening Files: Reading (2)

- Important things to keep in mind:
 - When opening a file for reading, the file must exist (and otherwise be accessible) or an exception occurs
 - The file object returned by the open function must be saved in a variable
 - All operations for accessing a file are made via the file object

Opening Files: Writing

- 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

Closing Files: Important

- When you are done 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

Syntax: Opening And Closing 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 From a File

- To read a line of text from a file, call the `readline()` method with the file object that was returned when you opened the file:

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

- 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 end of the line is encountered
 - The input marker is then moved to the next line

Reading From a File (2)

- For example, suppose input.txt contains the lines
flying
circus
- 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\n"`

Reading From a File (3)

- Calling `readline()` again yields the empty string `""` because you have reached the end of the file
- If the file contains a blank line, then `readline()` returns a string containing only the newline character `"\n"`

Reading Multiple Lines From a File

- You repeatedly read a line of text and process it until the sentinel value is reached:
- The sentinel value is an empty string, which is returned by the `readline()` method after the end of file has been reached

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

Converting File Input

- As with the input function, the `readline()` method can only return **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)
```

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

Writing To A File

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

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

- Unlike `print()` when writing text to an output file, *you must explicitly write the newline character to start a new line*
- 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))
```


Example: File Reading/Writing

- 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 value
- If the input file has the contents
32.0
54.0
67.5
80.25
115.0

Example: File Reading/Writing (2)

- The output file will contain

32.00

54.00

67.50

80.25

115.00

Total: 348.75

Average: 69.75

Example One: Self Study

- Open the file total.py

```

inputFileName = input("Input file name: ")
outputFileName = input("Output file name: ")

# 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()

```

Common Error

- Backslashes in File Names
 - When using a String literal for a file name with path information, you need to supply each backslash twice:

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

- A single backslash inside a quoted string is the *escape character*, which means the next character is interpreted differently (for example, `'\n'` for a newline character)
- When a user supplies a filename into a program, the user should not type the backslash twice

Text Input and Output

SECTION 7.2

Text Input and Output

- In the following sections, you will learn how to process text with complex contents, and you will learn how to cope with challenges that often occur with real data
- Reading Words Example:

Mary had a little lamb

input

```
for line in inputFile :  
    line = line.rsplit()
```

output

Mary
had
a
little
lamb

Processing Text Input

- There are times when you want to read input by:
 - Each word
 - Each line
 - A single character
- Python provides methods such: `read()`, `split()` and `strip()` for these tasks

Processing text input is required for almost all types of programs that interact with the user

Text Input and Output

- Python can treat an input file as though it were a container of strings in which each line comprises an individual string
- For example, the following loop reads all lines from a file and prints them:

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

- At the beginning of each iteration, the loop variable line is assigned the value of a string that contains the next line of text in the file
- There is a critical difference between a file and a container:
 - Once you read the file you must close it before you can iterate over it again

An Example of Reading a File

- We have a file that contains a collection of words; one per line:

spam

and

eggs

Removing The Newline (1)

- Recall that each input line ends with a newline (`\n`) character
- Generally, the newline character must be removed before the input string is used
- When the first line of the text file is read, the string line contains

s	p	a	m	\n
---	---	---	---	----

Removing The Newline (2)

- To remove the newline character, apply the `rstrip()` method to the string:

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

- This results in the string:

s p a m

Character Strip Methods

Table 1 Character Stripping Methods

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

Character Strip Examples

Table 2 Character Stripping Examples

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

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

Reading Words (2)

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

Mary

had

a

little

...

- 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

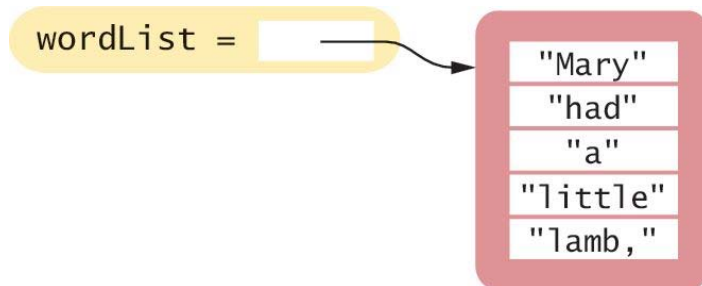
```
line = line.rstrip()  
wordlist = line.split()
```


Reading Words (3)

- 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:

line = M a r y h a d a l i t t l e l a m b ,

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



Reading Words (4)

- Notice that the last word in the line contains a comma
- If we only want to print the words contained in the file without punctuation marks, we can strip those from the substrings using the `rstrip()` method introduced in the previous section:

```
word = word.rstrip(". , ? !")
```

Reading Words: Complete Example

```
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()
```

Example Two: Self Study

- Open the file lyrics.py

Additional String Splitting Methods

Table 3 String Splitting Methods

Method	Returns
<code>s.split()</code> <code>s.split(<i>sep</i>)</code> <code>s.split(<i>sep</i>, <i>maxsplit</i>)</code>	Returns a list of words from string <i>s</i> . If the string <i>sep</i> is provided, it is used as the delimiter; otherwise, any white space character is used. If <i>maxsplit</i> is provided, then only that number of splits will be made, resulting in at most <i>maxsplit</i> + 1 words.
<code>s.rsplit(<i>sep</i>, <i>maxsplit</i>)</code>	Same as <code>split</code> except the splits are made starting from the end of the string instead of from the front.
<code>s.splitlines()</code>	Returns a list containing the individual lines of a string split using the newline character <code>\n</code> as the delimiter.

Additional String Splitting Examples

Table 4 String Splitting Examples

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

Reading Characters

- 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, the read() method returns a string consisting of the next character in the file

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

- If the end of the file is reached, it returns an empty string ""

Algorithm: Reading Characters

```
while char != "" :  
    Process character  
    char = inputFile.read(1)
```


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 an identification number, full name, address, and class year
- 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 the file
    Read the entire record
    Process the record
```

Record Formats: Example

- The organization or format of the records can vary, however, making some formats easier to read than others
- A typical format for such data is to store each field on a separate line of the file with all fields of a single record on consecutive lines:

China

1330044605

India

1147995898

United States

303824646

...

Record Formats: Example

- Reading the data in this format is rather easy
- Because each record consists of two fields, we read two lines from the file for each record

```
line = infile.readline()
while line != "":
    countryName = line.rstrip()
    line = infile.readline()
    population = int(line)
    Process data record
    line = infile.readline()
```

Record Formats: Example 2

- Another common format stores each data record on a single line
- If the record's fields are separated by a specific delimiter ":" you can extract the fields by splitting the line with the `split()` method

China:1330044605

India:1147995898

United States:303824646

...

Record Formats: Example 3

- But what if the fields are not separated by a delimiter?

China 1330044605

India 1147995898

United States 303824646

...

- Because some country names have more than one word, we cannot simply use a blank space as the delimiter because multi-word names would be split incorrectly
- Can we use `rsplit` in this case?

```
inputString = "United States 303824646"  
result = inputString.rsplit(" ",1)  
print(result)
```

Record Formats: Example 3

- One approach for reading records in this format is to read the line, then search for the first digit in the string returned by `readline()`:

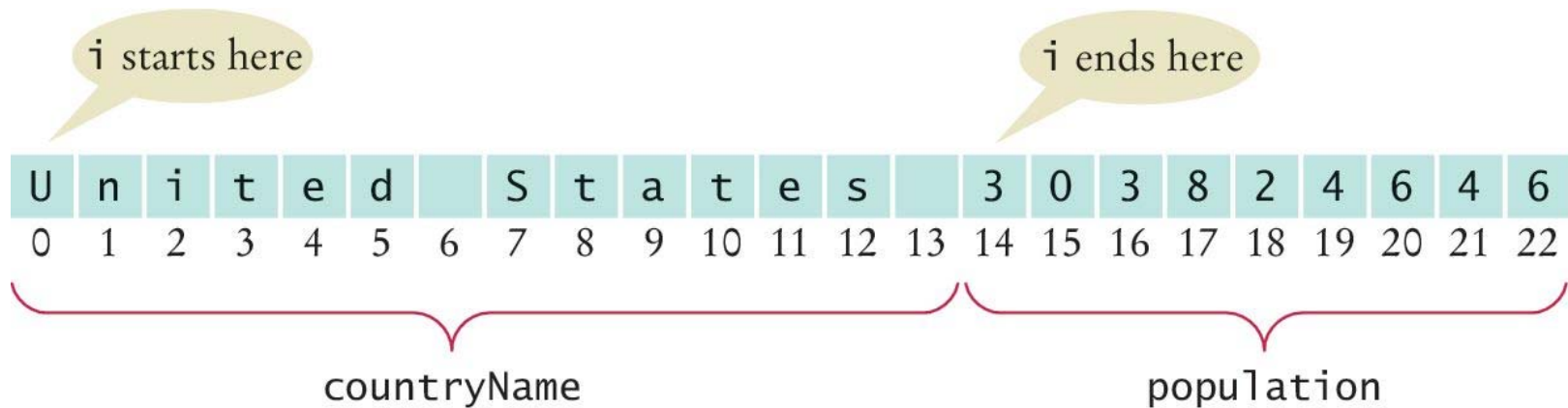
```
i = 0
char = line[0]
while not line[0].isdigit() :
    i = i + 1
```

- You can then extract the country name and population as substrings using the slice operator:

```
countryName = line[0 : i - 1]
population = int(line[i : ])
```

Record Formats: Example 3

- 'Slicing' the string read from file



File Operations

Table 5 File Operations

Operation	Explanation
<code>f = open(filename, mode)</code>	Opens the file specified by the string <i>filename</i> . The <i>mode</i> parameter indicates whether the file is opened for reading ("r") or writing ("w"). A file object is returned.
<code>f.close()</code>	Closes a previously opened file. Once closed, the file cannot be used until it has been reopened.
<code>string = f.readline()</code>	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.
<code>string = f.read(num)</code> <code>string = f.read()</code>	Reads the next <i>num</i> 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.
<code>f.write(string)</code>	Writes the <i>string</i> to a file opened for writing.

A Second Example: Self-Study

- Open the file `items.py`

```

inputFileName = input("Input file: ")
outputFileName = input("Output file: ")

# Open the input and output files.
inputFile = open(inputFileName, "r")
outputFile = open(outputFileName, "w")

# Read the input and write the output.
total = 0.0

for line in inputFile :
    # Make sure there is a colon in the input line, otherwise skip the line.
    if ":" in line :
        # Split the record at the colon.
        parts = line.split(":")

        # Extract the two data fields.
        item = parts[0]
        price = float(parts[1])

        # Increment the total.
        total = total + price

        # Write the output.
        outputFile.write("%-20s%10.2f\n" % (item, price))

# Write the total price.
outputFile.write("%-20s%10.2f\n" % ("Total:", total))

# Close the files.
inputFile.close()
outputFile.close()

```

Processing Text Files Example

PAGE 360

Steps to Processing Text Files

- Problem Statement:
- Read two country data files, `worldpop.txt` and `worldarea.txt`
- Write a file `world_pop_density.txt` that contains country names and population densities with the country names aligned left and the numbers aligned right

Afghanistan	50.56
Akrotiri	127.64
Albania	125.91
Algria	14.18
American Samoa	288.92
...	

Six Steps to Processing Text Files

1. Understand the Processing Task
 - i. Process the data “on the go” or store data and then process?
2. Determine which files you need to read and write
3. Choose a mechanism for obtaining the file names
4. Choose between iterating over the file or reading individual lines
 - i. If all data is on one line, normally use line input
5. With line-oriented input, extract required data
 - i. Examine the line and plan for whitespace, delimiters...
6. Use functions to factor out common tasks

Processing Text Files: Pseudocode

Step 1: Understand the Task

- While there are more lines to be read
 - Read a line from each file
 - Extract the country name
 - population = number following the country name in the line from the first file
 - area = number following the country name in the line from the second file
 - If area != 0
 - density = population / area
 - Print country name and density

Afghanistan	50.56
Akrotiri	127.64
Albania	125.91
Algria	14.18
American Samoa	288.92

Step Two

- Determine the file you need to read and write
 - There are two input files:
 - worldpop.txt
 - worldarea.txt
 - There is one output file:
 - world_pop_density.txt

Step Three

- Choose a mechanism for obtaining the file names
- We have three options:
 - Hardcode the filename as a constant
 - Prompt the user
 - Use a command line argument
- We will use hard-coded files names
 - The file names are constant in this example

Step Four

- Choose between iterating over the file or reading individual lines
- Generally if the data is grouped on lines we iterate over the file
- If the data is spread over several line, we read the individual lines
- In this example we will read individual lines since we are reading from two input files

Step Five

- Extract the data into the individual fields
 - Use spilt or rsplit to extract the data elements

Step Six

- Use functions to factor out common tasks
- Find the repetitive tasks and develop functions to handle them

Example Code: Self-Study

- Open the file `population.py`

Exception Handling

SECTION 7.5

Exception Handling

- There are two aspects to dealing with run-time program errors:
 - 1) Detecting Errors
 - 2) Handling Errors
- The open function can detect an attempt to read from a non-existent file
 - The open function cannot handle the error
 - There are multiple alternatives, the function does not know which is the correct choice
 - The function reports the error to another part of the program to be handled
- Exception handling provides a flexible mechanism for passing control from the point of the error to a **handler** that can deal with the error

Detecting Errors

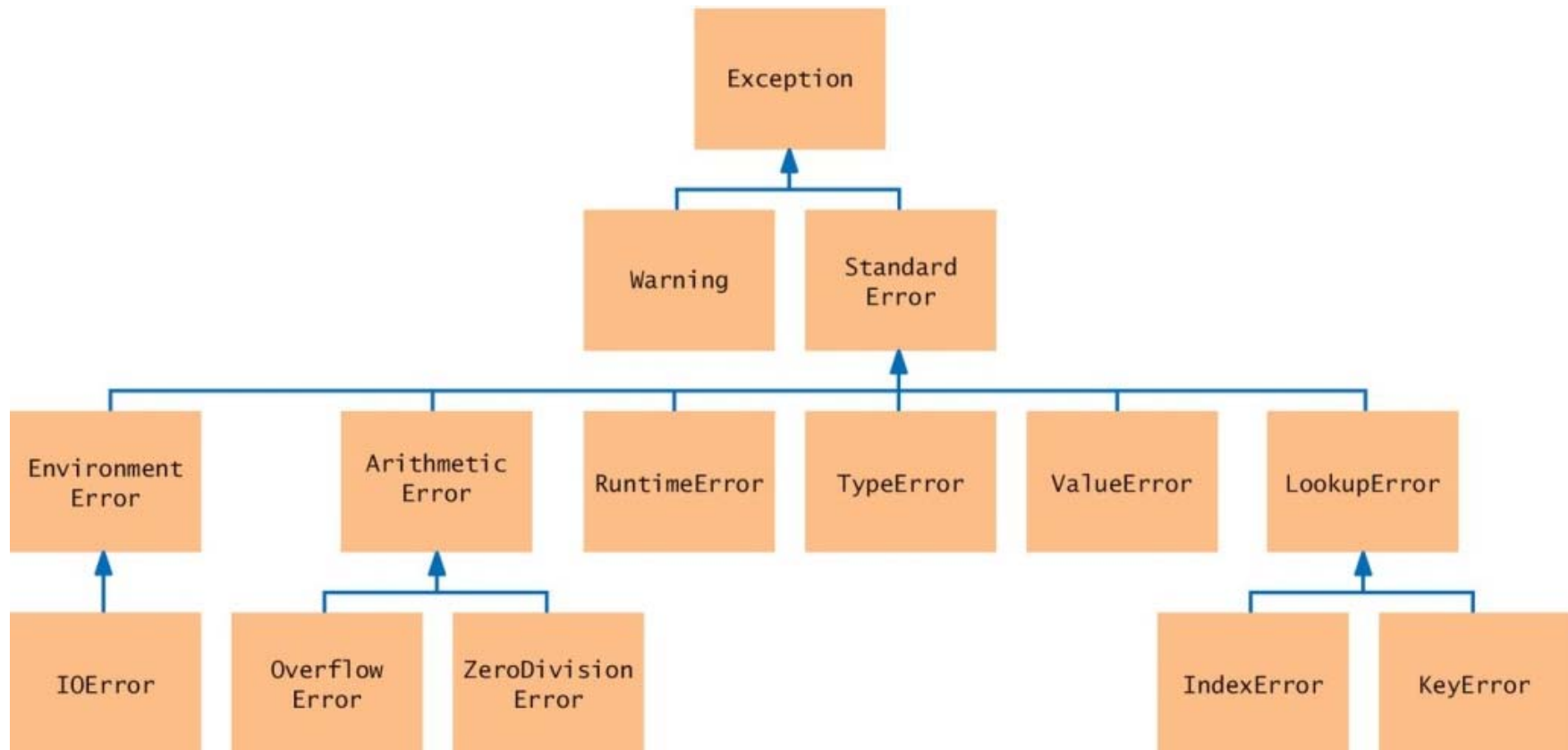
- What do you do if someone tries to withdraw too much money from a bank account?
- You can **raise** an exception
- When you **raise** an exception, execution does not continue with the next statement
 - It transfers to the **exception handler**

Use the `raise` statement
to signal an exception

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

Exception Classes (a subset)

- Look for an appropriate exception



Syntax: Raising an Exception

Syntax `raise exceptionObject`

A new exception object is constructed, then raised.

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

This message provides detailed information about the exception.

This line is not executed when the exception is raised.

Handling Exceptions

- Every exception should be handled somewhere in the program
- This is a complex problem
 - You need to handle each possible exception and react to it appropriately
- Handling recoverable errors can be done:
 - Simply: exit the program
 - User-friendly: Ask the user to correct the error

Handling Exceptions: Try-Except

- 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

Syntax: Try-Except

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)
```

When an IOError is raised,
execution resumes here.

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

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

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

This is the exception object
that was raised.

Try-Except: An Example

```
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))
```

`open()` can raise an `IOError` exception

`int()` can raise a `ValueError` exception

Execution transfers here if file cannot be opened

Execution transfers here if the string cannot be converted to an int

If either of these exceptions is raised, the rest of the instructions in the try block are skipped

Example

- If an IOError exception is raised, the **except** clause for the IOError exception is executed
- If a ValueError exception occurs, then second **except** clause is executed
- If any other exception is raised it will not be handled by any of the **except** blocks

Output Messages

- When the body of this handler is executed, it prints the message included with the exception

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

- For example, if the string passed to the int() function was "35x2", then the message included with the exception will be:
invalid literal for int() with base 10: '35x2'

Output Messages (2)

- To obtain the message, we must have access to the exception object itself
- You can store the exception object in a variable with the as syntax:

```
except ValueError as exception :
```

- When the handler for ValueError is executed, **exception** is set to the exception object. In our code, we then obtain the message string by calling **str(exception)**

Source of Output Messages

- When you raise an exception, you can provide your own message string. For example, when you call

```
raise ValueError("Amount exceeds balance")
```

- The message of the exception, "Amount exceeds balance", is the string provided as the argument to the constructor

The **finally** Clause

- The **finally** clause is used when you need to take some action whether or not an exception is raised
- Here is a typical situation
 - It is important to always close an output file whether or not an exception was raised (to ensure that all output is written to the file)
 - Place the call to `close()` inside a finally clause:

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

Syntax: The Finally Clause

Syntax

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

**This code may
raise exceptions.**

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

```
outfile = open(filename, "w")
```

```
try :  
    writeData(outfile)  
    . . .  
finally :  
    outfile.close()  
    . . .
```

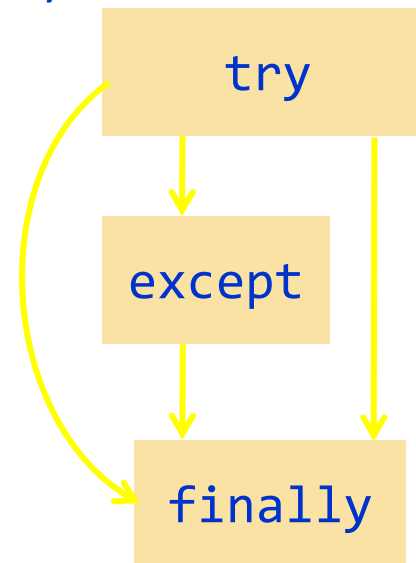
**The file must be opened
outside the try block
in case it fails. Otherwise,
the finally clause would
try to close an unopened file.**

Programming Tip

- Throw exceptions early
 - When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix
- Catch exceptions late
 - Conversely, a method should only catch an exception if it can really remedy the situation
 - Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler

Programming Tip

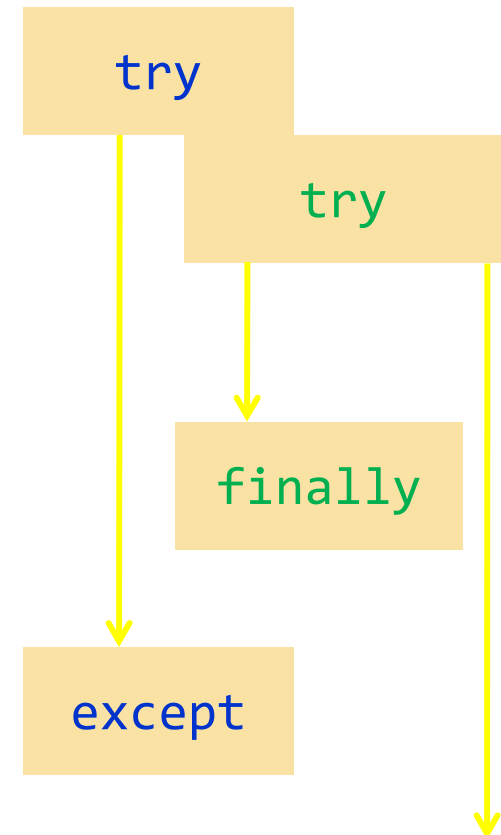
- Do not use `except` and `finally` in the same `try` block
 - The `finally` clause is executed whenever the try block is exited in any of three ways:
 1. After completing the last statement of the `try` block
 2. After completing the last statement of a `except` clause, if this try block caught an exception
 3. When an exception was raised in the `try` block and not handled



Programming Tip

- It is better to use two (nested) try clauses to control the flow

```
try :  
    outfile = open(filename, "w")  
    try :  
        # Write output to outfile  
    finally :  
        out.close() # Close resources  
except IOError :  
    # Handle exception
```



The `with` Statement

- Because a try/finally statement for opening and closing files is so common, Python has a special shortcut:

```
with open(filename, "w") as outfile :  
    Write output to outfile
```

- This `with` statement opens the file with the given name, sets `outfile` to the file object, and closes the file object when the end of the statement has been reached or an exception is raised

Handling Input Errors

SECTION 7.6

Handling Input Errors

- File Reading Application Example
 - Goal: Read a file of data values
 - First line is the count of values
 - Remaining lines have values
 - Risks:
 - The file may not exist
 - The `open()` function will raise an exception when the file does not exist
 - The file might have data in the wrong format
 - When there are fewer data items than expected, or when the file doesn't start with the count of values, the program will raise a `ValueError` exception
 - Finally, when there are more inputs than expected, a `RuntimeError` exception should be raised

```
3
1.45
-2.1
0.05
```

Handling Input Errors: main()

- Outline for method with all exception handling

```
done = False
while not done :
    try:
        # Prompt user for file name
        data = readFile(filename) # May raise exceptions
        # Process data
        done = true;
    except IOError:
        print("File not found.")
    except ValueError :
        print("File contents invalid.")
    except RuntimeError as error:
        print("Error:", str(error))
```

Handling Input Errors: readFile()

- Creates the file object and calls the readData() function
- No exception handling (no except clauses)
- `finally` clause closes file in all cases (exception or not)

```
def readFile(filename) :  
    inFile = open(filename, "r") # May throw exceptions  
    try  
        return readData(inFile)  
    finally  
        in.close()
```

Handling Input Errors: readData()

- No exception handling (no try or except clauses)
- `raise` creates an `ValueError` exception and exits
- `RuntimeError` exception can occur

```
def readData(inFile) :  
    line = inFile.readline()  
    numberOfValues = int(line) # May raise a ValueError exception.  
    data = []  
    for i in range(numberOfValues) :  
        line = inFile.readline()  
        value = int(line) # May raise a ValueError exception.  
        data.append(value)  
    # Make sure there are no more values in the file.  
    line = inFile.readline()  
    # Extra data in file  
    if line != "" :  
        raise RuntimeError("End of file expected.")  
    return data
```

Once Scenario

1. main calls `readFile`
2. `readFile` calls `readData`
3. `readData` calls `int`
4. There is no integer in the input, and `int` raises a `ValueError` exception
5. `readData` has no `except` clause; it terminates immediately
6. `readFile` has no `except` clause; it terminates immediately after executing the `finally` clause and closing the file
7. The `IOError except` clause is skipped
8. The `ValueError except` clause is executed

Example Code: Self-Study

- Open the file `analyzedata.py`

Summary: File Input/Output

- 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 `print()` function

Summary: Processing Text Files

- 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: Exceptions (1)

- To signal an exceptional 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

Summary: Exceptions (2)

- Raise an exception as soon as a problem is detected
 - Handle it only when the problem can be handled
- When designing a program, ask yourself what kinds of exceptions can occur
- For each exception, you need to decide which part of your program can competently handle it