Files

Files

Eric Roberts
CSCI 121
October 12, 2018

File Systems and the Multics Project

- Today, everyone who works with computers is familiar with the idea of a ***hierarchical file system*** in which information is stored in a tree of ***files*** and ***directories***.
- The hierarchical file system model was invented for a system called ***Multics*** (Multiplexed Information and Computing System), developed jointly by MIT under the direction of Fernando Corbató, Bell Labs, and General Electric in the 1960s.

Fernando Corbató (1926–)

Reading Data from Files

- Applications that work with arrays and array lists often need to work with lists that are too large to enter by hand. In many cases, it is easier to read the values of a list from a data file.
- A ***file*** is the generic name for any named collection of data maintained on the various types of permanent storage media attached to a computer. In most cases, a file is stored on a hard disk, but it can also be stored on a removable medium, such as a CD or flash memory drive.
- Files can contain information of many different types. The most common type of file—and the only type we'll consider in this class—is a ***text file***, which contains character data of the sort you find in a string.

Text Files *vs.* Strings

Although text files and strings both contain character data, it is important to keep in mind the following important differences between text files and strings:

1. ***The information stored in a file is permanent.*** The value of a string variable persists only as long as the variable does. Local variables disappear when the method returns, and instance variables disappear when the object goes away, which typically does not occur until the program exits. Information stored in a file exists until the file is deleted.
2. ***Files are usually read sequentially.*** When you read data from a file, you usually start at the beginning and read the characters in order, either individually or in groups that are most commonly individual lines. Once you have read one set of characters, you then move on to the next set of characters until you reach the end of the file.

Reading Text Files

- The standard paradigm for reading a text file begins uses the following code to ***open*** the file and associate it with a variable used as its ***file handle***:

```
with open(filename) as variable:
    Code to read the file using variable as the handle.
```

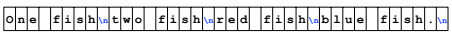
- The **`with`** statement, which you will learn more about later in the semester, ensures that the resources associated with the file are released when Python reaches the end of the **`with`** body.
- Python offers several strategies for reading data from a file:
  - Reading the entire file as a string using the **`read`** method.
  - Reading lines from a files using **`readline`** or **`readlines`**.
  - Using the file handle as an iterator.
  - Using the **`read`** method together with **`splitlines`**.

Reading an Entire File as a String

- In many ways, the simplest strategy for reading a file is to use the **`read`** method, which reads the entire file as a string, with embedded newline characters (**`\n`**) to mark the ends of lines.
- For example, if **`Seuss.txt`** is the file

```
One fish
two fish
red fish
blue fish.
```

calling **`read`** reads the entire file into a string like this:



- A potential downside to this approach is that reading an entire file into a string can require a large amount of memory.

Reading One Line at a Time

- Python offers several methods for reading a line from a file:
  - The **readline** method reads the next line with its newline.
  - The **readlines** method reads lines (with newlines) into a list.
  - The file handle can be used as an iterator.
- Of these, the last strategy is almost certainly the best and leads to the following idiom:

```
with open(filename) as f:
    for line in f:
        Code to process the line.
```

- The problem with all of these strategies is that the newline character is included as part of the line, which is almost never what you want as a client.

Finding the Longest Line in a File

- As long as you are working with files of modest size, the simplest way to read a file as a list of lines is to combine the **read** method from the file class with the **splitlines** method from the string class, as follows:

```
with open(filename) as f:
    lines = f.read().splitlines()
    Code to process the lines of the file.
```

- The advantage of this approach is that **splitlines** strips the newline characters from the end of each line.

Writing Text Files

- Python makes it possible to write new files using a code pattern that mirrors the one for reading:

```
with open(filename, mode) as variable:
    Code to write the file using variable as the handle.
```

- The difference between this pattern and the one for reading a file is the inclusion of a *mode* parameter, which is either the string **"w"** to write a new file (or overwrite an existing one) or **"a"** to its current contents.
- The body of the **with** statement includes calls to the **write** and **writelines** methods to write string data to the file.

Exception Handling

- When you are opening a file for reading, it is possible that the file does not exist. Python handles this situation—and many other errors or events that occur during execution—using a mechanism called ***exception handling***, which has become a standard feature of modern programming languages.
- In Python, an ***exception*** is an instance of a class that is part of hierarchy of exception classes. This hierarchy contains many exception types used for different purposes. File operations, for example, use the exception class **IOError**.
- If the **open** function encounters an error, such as a missing file, it reports the error by ***raising an exception*** using **IOError** as its exception type. Raising an exception terminates execution unless your program includes a **try** statement to handle that exception, as described on the next slide.

The **try** Statement

- Python uses the **try** statement to indicate an interest in handling an exception. In its simplest form, the syntax for the **try** statement is

```
try:
    Code in which exceptions may occur.
except type:
    Code to handle the exception.
```

  where *type* is the class name of the exception being handled.
- The range of statements in which the exception can be caught includes not only the statements enclosed in the **try** body but also any functions those statements call. If the exception occurs inside some other functions, any nested stack frames are removed until control returns to the **try** statement itself.

Requesting an Existing File

- The following function repeatedly asks the user to supply the name of an existing file until the file can be opened for input:

```
def getExistingFile(prompt="Input file: "):
    while True:
        filename = input(prompt)
        try:
            with open(filename):
                return filename
        except IOError:
            print("Can't open that file")
```

- If the **open** call succeeds, the body of the **with** statement simply returns the filename without reading any data. If an **IOError** exception occurs, the **except** clause prints an error message and returns to the **while** loop to try again.

Choosing Files Interactively

- The Python package used to implement **`pgl.py`** also supports a mechanism to choose files interactively, which is available through the **`filechooser.py`** library module.
- This library exports two functions—**`chooseInputFile`** and **`chooseOutputFile`**—for selecting a file.
- Both functions bring up a file dialog that allows the user to select a file.
  - Clicking **Open** or **Save** returns the full pathname of the file.
  - Clicking **Cancel** returns the empty string.
- The following paradigm shows the use of **`chooseInputFile`**:

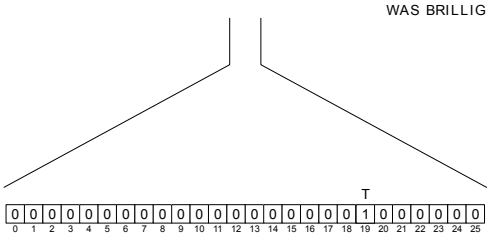
```
filename = chooseInputFile()
with open(filename) as f:
    Code to read the file.
```

Using Arrays for Tabulation

- Arrays turn out to be useful when you have a set of data values and need to count how many values fall into each of a set of ranges. This process is called ***tabulation***.
- Tabulation uses arrays in a slightly different way from those applications that use them to store a list of data. When you implement a tabulation program, you use each data value to compute an index into an array of integers that keeps track of how many values fall into that category.
- The example of tabulation used in the text is a program that counts how many times each of the 26 letters appears in a sequence of text lines. Such a program would be very useful in decoding a letter-substitution ciphers, such as the one from Edgar Allan Poe's short story "The Gold Bug" described earlier in class.

Implementation Strategy

The basic idea behind the program to count letter frequencies is to use an array with 26 elements to keep track of how many times each letter appears. As the program reads the text, it increments the array element that corresponds to each letter.



Code to Count Letter Frequencies

```
# File: CountLetterFrequencies.py

"""
This program counts the frequencies of letters in a sequence of lines
that the user enters on the console.
"""

def CountLetterFrequencies():
    counts = createFrequencyTable()
    print("Enter input lines, ending with a blank line.")
    while True:
        line = input()
        if line == "": break
        updateFrequencyTable(counts, line)
    printFrequencyTable(counts)

def createFrequencyTable():
    """
    Creates an empty frequency table, which is a list of 26 elements
    indicating the counts for each letter of the alphabet.
    """
    return [ 0 ] * 26
```

Code to Count Letter Frequencies

```
def updateFrequencyTable(counts, source):
    """
    Updates the frequency table by scanning the source string and
    recording the number of times each letter is encountered in
    the element of the counts array that corresponds to the index
    of the letter in the alphabet.
    """
    for ch in source:
        if ch.isalpha():
            counts[ord(ch.upper()) - ord("A")] += 1

def printFrequencyTable(counts):
    """
    Prints a frequency table using the data from counts, which is
    a 26-element array of integers, one for each letter.  Letters whose
    count is 0 are not included in the display.
    """
    for i in range(len(counts)):
        count = counts[i]
        if count > 0:
            ch = chr(ord("A") + i)
            print(ch + ": " + str(count))

if __name__ == "__main__":
    CountLetterFrequencies()
```

Adding a File Chooser

```
# File: CountLetterFrequenciesInFile.js

"""
This program counts the frequencies of letters in a file.
"""

from filechooser import chooseInputFile
from CountLetterFrequencies import createFrequencyTable
from CountLetterFrequencies import updateFrequencyTable
from CountLetterFrequencies import printFrequencyTable

def CountLetterFrequenciesInFile():
    filename = chooseInputFile()
    if (filename != ""):
        with open(filename) as f:
            counts = createFrequencyTable()
            for line in f:
                updateFrequencyTable(counts, line)
            printFrequencyTable(counts)

# Startup code

if __name__ == "__main__":
    CountLetterFrequenciesInFile()
```