Lists In Python

Jerry Cain
CS 106AX
October 25, 2024
slides leveraged from those constructed by Eric Roberts

Arrays and Lists

- From the earliest days of computing, programming languages have supported the idea of an ***array,*** which is an ordered sequence of values.
- The individual values in an array are called ***elements***. The number of elements is called the ***length*** of the array.
- Each element is identified by its position number in the array, which is called its ***index***. In Python—as in almost all modern languages, including JavaScript—index numbers begin with 0 and extend up to one less than the length of the array.
- Python implements the array concept in a more general form called a ***list***. Lists support all standard array operations, but also allow insertion and deletion of elements.
- The terms list and array are often used interchangeably in Python, but we'll bias toward the former.

Creating Lists in Python

- The simplest way to create a list is to specify its elements surrounded by square brackets and separated by commas, just as you do in JavaScript. For example, the declaration

```
primes = [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

  creates a list of the first ten prime numbers. **`primes[0]`** refers to the 2, and **`primes[9]`** and **`primes[-1]`** each refer to the 29.

- Python lists, like JavaScript arrays, can store elements of any type, including lists.

```
lectures = ["Monday", "Wednesday", "Friday"]
cardinal = [0xC4, 0x1E, 0x3A]
board = [["X", "", ""], ["", "", "O"], ["", "", ""]]
pascal = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]
```

Cycling through List Elements

- The traditional **`for`** loop pattern we used to cycle over all characters in a string generalizes to all Python lists. The standard **`for`** loop pattern for doing so looks like this:

```
for element in list:
    perform some operation on the element.
```

- As an example, the following function returns the sum of the elements in the list:

```
def sumIntegerList(list):
    sum = 0
    for value in list:
        sum += value
    return sum
```

Python Sequences

- The last few slides should remind you of string operations, which are almost identical.
- Strings and lists are both examples of a more general class of objects in Python called ***sequences***. All sequences support the following operations:
  - The **`len`** function
  - Index numbering beginning at 0
  - Negative index numbering that counts backward from the end
  - Selection of an individual element using square brackets
  - Slicing in all its forms
  - Concatenation using the **`+`** or **`+=`** operator
  - Repetition using the **`*`** operator
  - Inclusion testing using **`in`** operator (e.g., **`if 4 in numbers`**)

Mutable *vs.* Immutable Types

- The most important difference between a list and a string is that you are allowed to change the contents of a list while the characters in a string are fixed.
- Types like strings for which you are not allowed to change the individual components are defined to be ***immutable***.
- Types like lists where the elements are assignable are said to be ***mutable***.
- Immutable types have many advantages in programming:
  - You don't have to worry about whether values will be changed.
  - Values that are immutable can more easily be shared.
  - Immutable objects are easier to use in concurrent programs.
- Despite these advantages, there are still situations in which mutable types like lists are precisely what's needed.

Methods that Return Information

list`.index(`*value*`)` Returns the first index at which *value* appears in *list* or raises an error.
list`.index(`*value*`, `*start*`)` Returns the first index of *value* after the starting position.
list`.count(`*value*`)` Returns the number of times *value* appears in *list*.
list`.copy()` Creates a new list whose elements are the same as the original.

Methods that Add and Remove Elements

list`.append(`*value*`)` Adds *value* to the end of the list.
list`.insert(`*index*`,` *value*`)` Inserts *value* at the specified index, shifting subsequent elements over.
list`.remove(`*value*`)` Removes the first instance of *value*, or raises an error if it's not there.
list`.pop()` Removes and returns the last element of the list.
list`.pop(`*index*`)` Removes and returns the element at the specified index.
list`.clear()` Removes all elements from the list.

Methods that Reorder Elements

list`.reverse()` Reverses the order of elements in the list.
list`.sort()` Sorts the elements of *list* in increasing order.
list`.sort(`*key*`)` Sorts the elements of *list* using *key* to generate the key value.
list`.sort(`*key*`,` *reverse*`)` Sorts in descending order if *reverse* is `True`.

List Methods that Involve Strings

str`.split()` Splits a string into a list of its components using whitespace as separator.
str`.split(`*sep*`)` Splits a string into a list using the specified separator.
str`.splitlines()` Splits a string into separate lines at instances of the newline character.
sep`.join(`*list*`)` Joins the elements of *list* into a string, using *sep* as the separator.

Using Lists for Tabulation

- Lists 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 lists in a different way from 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 a list 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 is useful in decoding letter-substitution ciphers, which is one aspect of the topic of cryptography covered in Assignment #4.

Poe's Cryptogram Puzzle, Revisited

```
53‡‡†305))6*;4826)4‡•)4‡);806*;48†8¶
60))85;1‡(;:‡*8†83(88)5*†;46(;88*96*
?;8)*‡(;485);5*†2:*‡(;4956*2(5*-4)8¶
8*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡
1;48†85;4)485†528806*81(‡9;48;(88;4(
‡?34;48)4‡;161;:188;‡?;
```

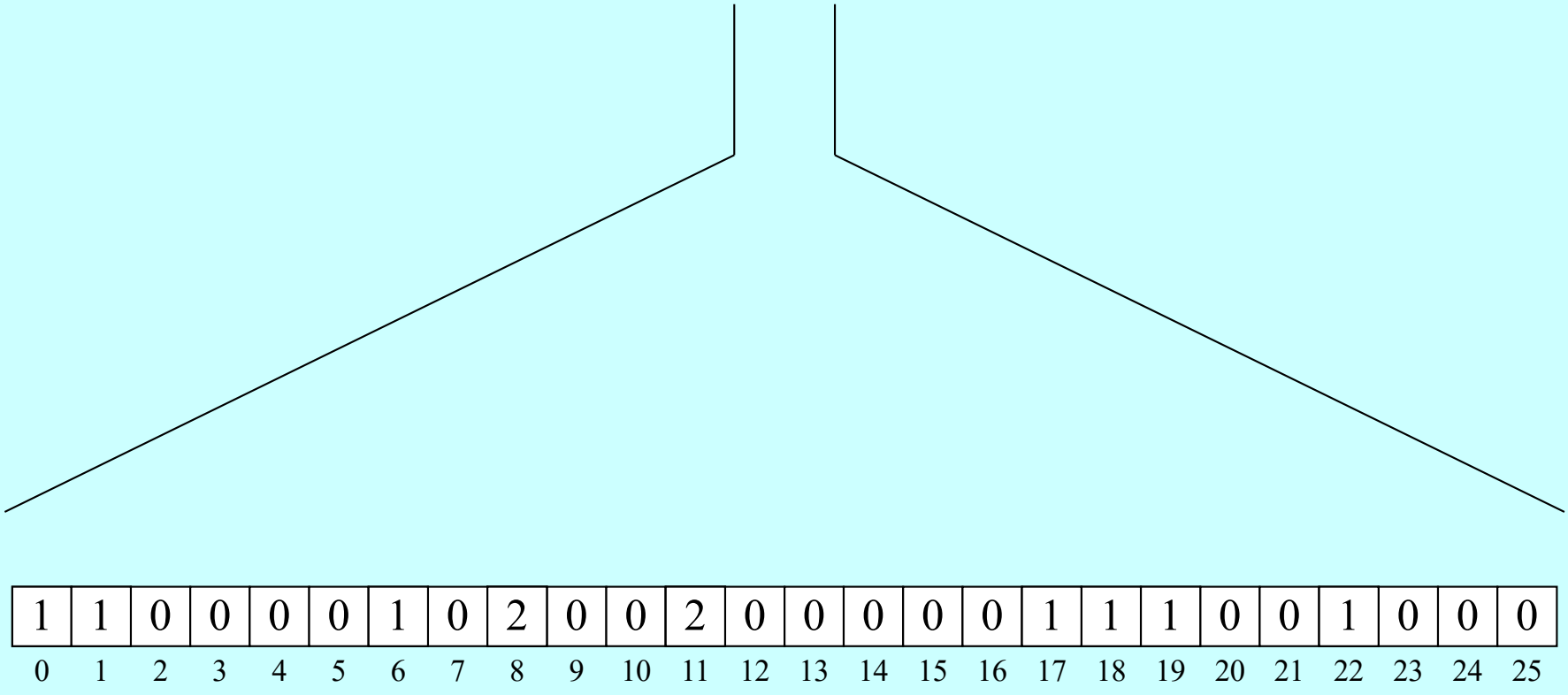
```
AGOODGLASS;6NTHEBISHOPS;HOSTEL6NTHEDEV
6LSSEATFORTYONEDEGREES;ANDTH6RTEENM6N
UTES;NORTHEAST;ANDBYNORTHMA6NBRANCH;SEV
ENTH6MBEAST;6DES;HOTFROMTHEL6FTEYE
FTHEDEATH'SHEAD;6NEFROMTHETREETHR
OUGHTHES;HOTF6TYFEETOUT
```

8	33
;	26
4	19
‡	16
)	16
*	13
5	12
6	11
(	10
†	8
1	8
0	6
9	5
2	5
:	4
3	4
?	3
¶	2
-	1
•	1

Implementation Strategy

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

TWAS BRILLIG



The `chr` and `ord` Functions

- Python includes two built-in functions to simplify conversion between an integer and the corresponding Unicode character. They're cousins to the `String.fromCharCode` function and *str*.`charCodeAt` method you've seen in JavaScript.
- The `chr` function takes an integer and returns a one-character string containing the Unicode character with that code.
  - `chr(32)` → `" "` (the space character)
  - `chr(65)` → `"A"` (an uppercase A)
  - `chr(960)` → `"π"` (the Greek letter pi)
- The `ord` function takes a one-character string and returns the value of that character in Unicode.
  - `ord(" ")` → `32`
  - `ord("A")` → `65`
  - `ord("π")` → `960`

Program 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
```

Program to Count Letter Frequencies

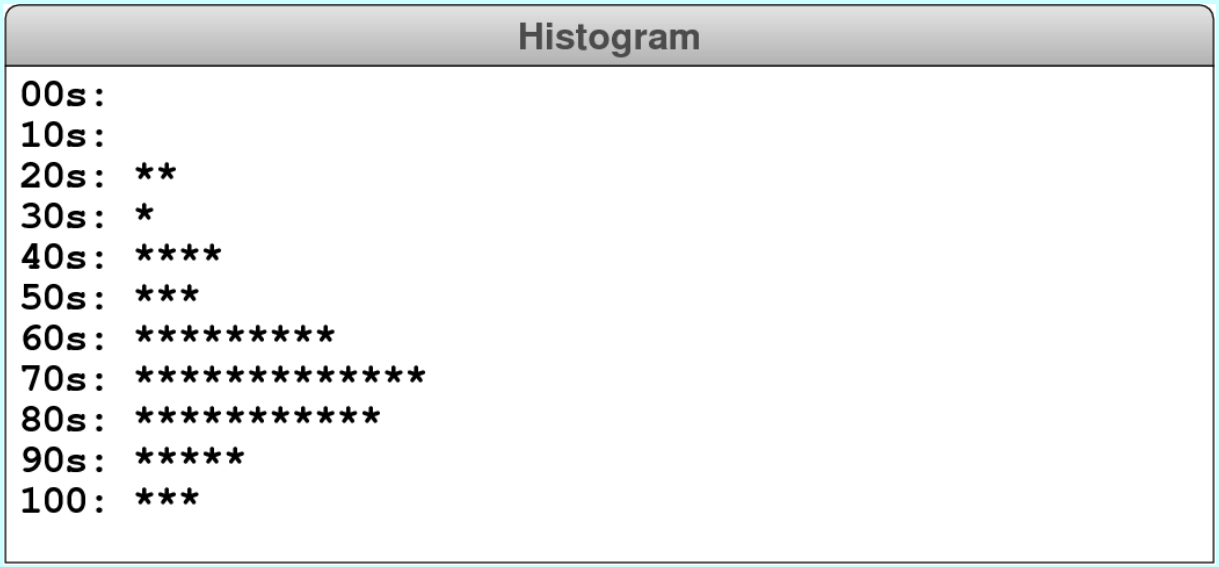
```
def updateFrequencyTable(counts, source):
    """
    Updates the frequency table by scanning the source string and
    recording the number of times each letter is encountered in
    the element of the counts array that corresponds to the index
    of the letter in the alphabet.
    """
    for 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()
```

Exercise: Display a Histogram

- Write a function **`showHistogram(scores)`** that takes a list of scores in the range 0 to 100 and then displays on the console a histogram of those scores divided into ranges 0-9, 10-19, and so on.



- Histograms are usually presented vertically. The one in this exercise is drawn horizontally because that program is so much easier to write.

Reading Data from Files

- Applications that work with lists often need to work with lists that are too large to enter by hand. In many cases, it is easier to read the list values 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 local hard disk, though occasionally files get stored on removable devices like USB sticks and networked filesystems like Dropbox.
- Files can contain information of many different types. The most common type of file—and the primary type we'll consider in CS 106AX—is a ***text file***, which contains readable 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 them:

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 function returns, unless that variable is referenced by a closure in JavaScript or Python, in which case the string typically vanishes when the program ends. 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 use in Assignments #5 and #6, ensures that 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 iterable.
  - 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:

O	n	e		f	i	s	h	\n	t	w	o		f	i	s	h	\n	r	e	d		f	i	s	h	\n	b	l	u	e		f	i	s	h	.	\n

- One downside to this approach is that reading an entire file into a single string can require a large amount of memory if the file itself is large.

Reading One Line at a Time

- Python offers several alternatives for reading 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 iterable.
- Of these, the last strategy is often the easiest and works well when used as follows:

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

- The primary drawback with these strategies is that the newline characters are retained as part of each line, which is rarely what you want.

Finding the Longest Line in a File

- So 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. Stripping the newlines from strings is sometimes referred to as ***chomping*** them.

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.

The End