

Sequence Types



FEB 23-25 2015

What we have not learned so far...



- How to store, organize, and access large amounts of data?
- **Examples:**
 - Read a sequence of million numbers and output these in sorted order.
 - Read a text, correct all spelling errors in the text, and output the corrected text.
- Programming languages typically provide tools and techniques to store and organize data.
- In Python we can use *sequence types* to do this.

Strings and *Lists* are examples of Sequence Types



- A *string* is a sequence of characters enclosed in quotes.

Examples: "hello", "8.397", "7", '34'

(The quotes can be single or double quotes)

- A *list* is a sequence of objects enclosed in square brackets.

Examples: [0, 1, 2, 3],
["Alice", "Bob", "Catherine"],
["hello", 4.567, True, 'bye']

(Objects of different types can be part of the same list)

- Lists are more “general” than strings; strings can be viewed as special instances of lists.

Accessing items in lists and strings



```
L = ["hi", 10, "bye", 100, -20, 123, 176, 3.45, 1, "it"]
```

"hi"	10	"bye"	100	-20	123	176	3.45	1	"it"
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
0	1	2	3	4	5	6	7	8	9

- One of the most useful features of sequence types is that elements in a sequence can be accessed efficiently and conveniently using their *position* in the sequence.
- This type of access is called *random access*. It refers to the fact that the amount of time to access an element via its index is independent of the value of the index or the size of the sequence.
- **Example:**

L[0] is "hi", L[1] is 10, L[2] is "bye",..., L[9] is "it"

An example with lists and strings



- **Example:**

$L = [\text{"hello, Pavan"}, [22, 0], 15]$

$L[0][4] = \text{"o"}$

$L[1][0] = 22$

$L[0][1] = \text{"e"}$

$L[2] = 15$

The len function



- Python has a built-in function `len(L)` that returns the length, i.e., the number of elements, in list `L`.
- `len(s)` returns the length of a string `s`.

Examples: `len([])` is 0, `len([34, 12, 45])` is 3,
`len("hello")` = 5.

- Thus the elements of a list `L` are indexed from 0 through `len(L)-1`.
- This simple observation is quite useful in iterating through a list.

Example 1: Iterating through a list



- This program walks through the list, printing each element.
- The program uses the positions of the elements to index into the list.

```
L = ["hi", 109, "go", 111, 1.16, [122,30], "hello"]  
i = 0  
while i < len(L):  
    print(L[i])  
    i = i + 1
```

Example 2: Testing membership in a list



```
# tests if a given element is a member of a given list.  
# Returns True if element is a member; False otherwise.  
def isMember(L, elem):  
    i = 0 # i serves as the index into list L  
  
    # Iterate through the elements of the list  
    # comparing each of them with elem  
    while i < len(L):  
        if elem == L[i]:  
            return True  
        i = i + 1  
    return False
```


The in operator



- The `isMember` function is rendered useless – by the Python `in` operator.
- The `in` operator is used as `x in L`, where `x` is an object and `L` is a list. This expression evaluates to `True` if `x` is an *element* in `L`; evaluates to `False` otherwise.

Examples: `67 in [34, 12, 45]` evaluates to `False`
`"hi" in []` evaluates to `False`, etc.

- This works on strings as well.

Examples:

`"hi" in "history"` evaluates to `True`

`"ei" in "piece"` evaluates to `False`

`"ace" in "Wallace"` evaluates to `True`

Example 3: Finding location of an element



```
# searches for a given element in a given list and  
# returns the index of the first occurrence of the  
# element, if it is present in the list. Otherwise,  
# returns -1.
```

```
def search(L, elem):
```

```
    i = 0 # i serves as the index into list L
```

```
    # Iterate through the elements of the list
```

```
    # comparing each of them with elem
```

```
    while i < len(L):
```

```
        if elem == L[i]:
```

```
            return i
```

```
        i = i + 1
```

```
    return -1
```

The “+” operator on lists



- Just like we use the “+” operator for string concatenation, we can use the “+” operator for “concatenating” lists.
- **Examples:**
 - `[10, 20, 30] + [10, 13]` evaluates to `[10, 20, 30, 10, 13]`
 - `[10, 20, 30] + ["hello"]` evaluates to `[10, 20, 30, "hello"]`
 - `[10, 20, 30] + []` evaluates to `[10, 20, 30]`
- In the following code, L is enlarged by the addition of three elements 10, 20, and 30 at the end of the list:
 - `L = L + [10, 20, 30]`

The “*” operator on lists



- The “*” operator can be used on lists (or strings) to make larger lists (or strings) obtained by concatenating many copies of a list (or string).
- **Example:** If $L = [2, 3, 7]$, then $L*2$ (or $2*L$) evaluates to $[2, 3, 7, 2, 3, 7]$.
- **Example:** If $s = \text{"cashew"}$, then $2*s$ (or $s*2$) evaluates to "cashewcashew" .

Programming Problem 4



- Read a file containing some number of nonnegative integers and output the number of *distinct* integers in the file.
- There is no specific format to the file – there could be several integers in a line or none, consecutive integers are separated by one or more white spaces (blanks, tabs, returns).

Example Input File (*test.txt*)



23 78

4567 123 789
230

1236765

78798 6768 678 678 78

Algorithm



1. `masterList = []`
2. Read a line of the file as a string.
3. “Parse” the line to extract a list `numbersInLine` of integers from the line.
4. Walk through list `numbersInLine` and for each element in `numbersInLine`, not in `masterList`, add it to `masterList`.
5. Go back to Line (2), if there are more lines to process.
6. Output the length of `masterList`.

Main Program



```
# Open a file called test.txt for read only and read the first line
f = open("test.txt", "r")
line = f.readline()
masterList = [] # keeps track of the list of distinct integers in the file

# Process each line, if line is non-empty
while line != "":
    # Parse the line to extract a list of numbers in the line
    numbersInLine = parse(line)

    # Extend the masterList by appending to it all the new
    # numbers in the line.
    masterList = uniqueExtend(masterList, numbersInLine)

    # Read the next line
    line = f.readline()

f.close()

print(len(masterList))
```


The function uniqueExtend



```
# Takes two lists L1 and L2 and returns the list obtained
# by appending to L1, all elements in L2 that are not in L1
def uniqueExtend(L1, L2):
    index = 0 # serves as index into list L2

    # Loop to walk through elements of L2
    while index < len(L2):
        # If current element of L2 is not in L1, then append it
        if not(L2[index] in L1):
            L1 = L1 + [L2[index]]
            index = index + 1

    return L1
```

The function parse



```
# Takes a string consisting of non-negative integers and
# returns a list containing all the integers in the line.
# The integers in the line are separated by 1 or more blanks.
```

```
def parse(s):
```

```
    listOfNumbers = [] # maintains the list of numbers in strings s
    currentNumber = ""
```

```
    # The function oscillates between two states: in one state
    # it is processing the digits of an integer and the other state
    # it is processing the white spaces between consecutive integers.
    # The boolean variable numberBeingProcessed is used to keep track
    # of this state.
```

```
    numberBeingProcessed = False
```

```
    i = 0 # serves as an index into the string s
    while i < len(s):
```

```
        # if the current character is a digit
```

```
        if s[i] >= "0" and s[i] <= "9":
            numberBeingProcessed = True
            currentNumber = currentNumber + s[i]
```

```
        # else if the current character is a non-digit
```

```
        # immediately following a number
        elif numberBeingProcessed:
            listOfNumbers = listOfNumbers + [int(currentNumber)]
            numberBeingProcessed = False
            currentNumber = ""
        i = i + 1
```

```
    return listOfNumbers
```