CS303E: Elements of Computers and Programming

More on Lists

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: May 24, 2021 at 15:28

List Example: Counting Occurrences of Letters

Suppose we want to count the occurrences of letters in a given text. Here's an algorithm.

1. Break the text into words
2. Create a list called "`counts`" of 26 zeros.
3. For each letter in each word:
   - Convert it to lowercase
   - If it's the ith letter, increment `counts[i]` by 1
4. Print the counts list is a nice format.

There's a version of this program in Listing 10.8. We're solving a slightly different problem.

List Example: Counting Occurrences of Letters

In file `CountOccurrencesInText.py`:

```
def isLowerLetter( ch ):
    """ Test whether a character is a lowercase letter. """
    return ’a’ <= ch <= ’z’

def countOccurrences( text ):
    """ Count occurrences of each of the 26 letters
    (upper or lower case) in text.  Return a list of
    counts in order."""

    # Create a list of 26 0’s.
    counts = [ 0 for i in range( 26 ) ]
    # Not strictly necessary; could just count
    # chars in text.
    wordList = text.split()
    for word in wordList:
        word = word.lower()
        for ch in word:
            if isLowerLetter( ch ):
                index = ord( ch ) - ord( ’a’ )
                counts[ index ] += 1
    return counts
```


List Example: Counting Occurrences of Letters

Now we want to print the counts in a nice format, 10 per line.

```
def printCounts( counts ):
    """ Print the letter counts 10 per line. """
    onLine = 0
    for i in range( 26 ):
        # Convert the index into the array into the
        # corresponding lower case letter.
        letterOrd = i + ord(’a’)
        print( chr(letterOrd) + ":", counts[i], end = " ")
        onLine += 1
        # If we’ve printed 10 on the line, go to the next
           line.
        if ( onLine == 10 ):
            print()
            onLine = 0
    print()
```

List Example: Counting Occurrences of Letters

```
def main():
    text = "Fourscore and seven years ago our fathers \
       brought forth, on this continent, a new nation, \
       conceived in liberty, and dedicated to the \
       proposition that all men are created equal."
    counts = countOccurrences( text )
    printCounts( counts )
```

```
>>> from CountOccurrencesInText import *
>>> main()
a: 13 b: 2 c: 6 d: 7 e: 18 f: 2 g: 2 h: 6 i: 9 j: 0
k: 0 l: 4 m: 1 n: 14 o: 14 p: 2 q: 1 r: 11 s: 6 t: 15
u: 4 v: 2 w: 1 x: 0 y: 2 z: 0
```


Aside: Using Docstring

A list constant at the top of your function/class/module is stored by Python as the *docstring*, and accessible to your program. using the method `FunctionName.__doc__`.

```
>>> from CountOccurrencesInText import *
>>> printCounts.__doc__
' Print the letter counts 10 per line. '
>>> countOccurrences.__doc__
' Count occurrences of each of the 26 letters\n
    (upper or lower case) in text.  Return a list of\n
    counts in order.'
```

This also works for system defined functions.

```
>>> import math
>>> math.sqrt.__doc__
'sqrt(x)\n\nReturn the square root of x.'
```


Searching a List

A common operation on lists is **searching**. To search a list means to see if a value is in the list.

If all you care about is *whether or not* `lst` contains value `x`, you can use: `x in lst`.

Often you want to know the *index* of the occurrence, if any.

Linear Searching

If the list is not *sorted*, often the best you can do is look at each element in turn. This is called a **linear search**.

From file `LinearSearch.py`:

```
def linearSearch( lst, key ):
    for i in range( len(lst) ):
        if key == lst[i]:
            return i
    return -1
```

If the item is present, you stop as soon as you find it. On average, how many comparisons would you expect to make if the item is there? How many if it's not there?

Linear Searching

```
>>> from LinearSearch import *
>>> lst = [1, 3, 5, 7, 9]
>>> linearSearch( lst, 7 )
3
>>> linearSearch( lst, 1 )
0
>>> linearSearch( lst, 8 )
-1
>>> linearSearch( [1, 2, 1, 2, 1, 2], 2 )
1
```

We use -1 to indicate that the item is not in the list, since -1 is not a legal index.

Find Multiple Occurrences

Notice that `linearSearch` only finds the *first* occurrence of the key. To find all, you might do:

```
def findAllOccurrences( lst, key ):
    # Return a list of indexes of occurrences
    # of key in lst.
    found = []
    for i in range( len(lst) ):
        if key == lst[i]:
            found.append( i )
    return found
```

```
>>> from LinearSearch import *
>>> findAllOccurrences( [1, 2, 1, 2, 1, 2], 2)
[1, 3, 5]
```

Here you do have to search the whole list.

Using Index

By the way, in Slideset 9 we described the `index` function on lists. You can use this to do linear search *if you know that the item is present*.

```
>>> lst = [ 9, 3, 5, 7, 1, 2, 4, 8 ]
>>> lst.index( 7 )
3
>>> lst.index( 10 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 10 is not in list
>>>
```

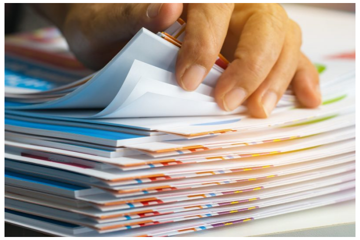
The `index` method is almost certainly implemented using linear search.

Searching a Sorted List

Suppose you were looking for your test in a pile containing all tests for the 600+ students in this class.

If they weren't sorted, you'd have to look through every one (linear search).

If they are sorted alphabetically by names:



- Divide the pile into two halves, pile1 and pile2.
- If your test is on top of pile2, you're done.
- If your name is alphabetically lower than the name on the test on top of pile2, then search pile1 using the same approach.
- Otherwise, search pile2 using the same approach.

Binary Search

In file `BinarySearch.py`:

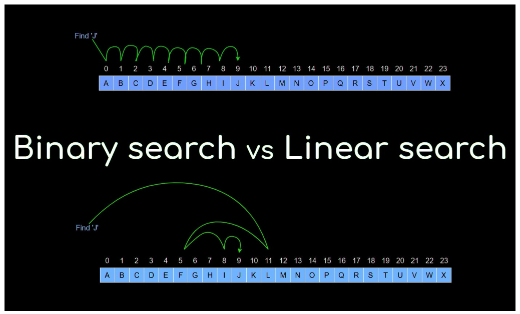
```
def BinarySearch( lst, key ):
    """ Search for key in sorted list lst. """
    low = 0
    high = len(lst) - 1
    while (high >= low):
        mid = (low + high) // 2
        if key < lst[mid]:
            high = mid - 1
        elif key == lst[mid]:
            return mid
        else:
            low = mid + 1
    # What's true here?  Why this value?
    return (-low - 1)
```


Using Binary Search

```
>>> from BinarySearch import BinarySearch
>>> lst = [ 2, 4, 7, 9, 10, 12, 14, 17, 20 ]
>>> BinarySearch( lst, 9 )
3
>>> BinarySearch( lst, 13 )
-7
>>> low = -( -7 + 1 )        # failed search
>>> low                      # returns -7 == (-low - 1)
6
>>> list.insert.__doc__
'L.insert(index, object) -- insert object before index'
>>> lst.insert( low, 13 )
>>> lst
[2, 4, 7, 9, 10, 12, 13, 14, 17, 20]
```

Is this guaranteed to find the *first* occurrence of the key in the list?

Complexity of Both Search Methods



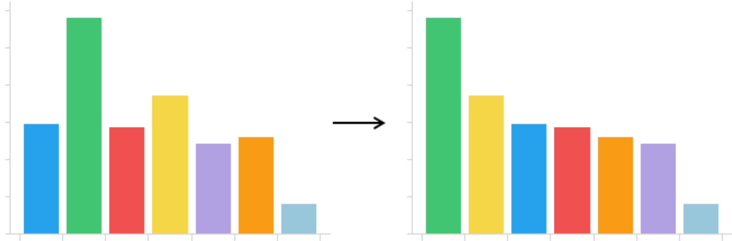
Linear Search: each comparison removes one item from the search space; number of comparisons proportional to the length of the list searched.

Binary Search: each step cuts the search space in half. With n items, you can only cut n in half $\log_2(n)$ times.

How many comparisons would you expect for a list of 1000 items?

Let's Take a Break

Sorting



Another very important function is **sorting** a list. This assumes that the list items are *comparable*.

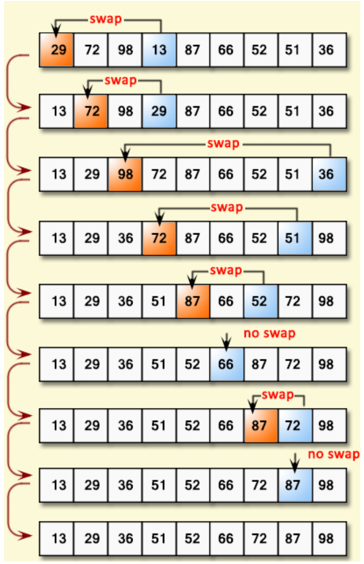
There are many different sorting algorithms; you will study several in CS313E. Two of the simplest are:

- selection sort
- insertion sort

Selection Sort

Algorithm:

1. Find the smallest item in lst.
2. Swap it with the first element.
3. Repeat for lst.slice[1:]
4. Stop when there's only one element left.



SelectionSort Code

In file `SelectionSort.py`:

```
def selectionSort( lst ):
    """ Sort lst in ascending order. """
    # For each element lst[i] in lst[0...len-1]:
    for i in range( len(lst) - 1 ):
        # prints the list with swap point marked
        printSorting( lst, i )
        currentMin = lst[i]
        currentMinIndex = i
        # find the smallest element in the remainder
        # the list and swap with lst[i].
        for j in range( i + 1, len( lst )):
            if currentMin > lst[j]:
                currentMin = lst[j]
                currentMinIndex = j
        # Swap lst[i] with lst[currentMinIndex]
        # if necessary.
        if currentMinIndex != i:
            lst[currentMinIndex] = lst[i]
            lst[i] = currentMin
    printSorting( lst, i+1 )
```


SelectionSort Executing

I printed the list at each step with a "|" showing the swap point.

```
>>> import random
>>> from SelectionSort import selectionSort
>>> lst = [random.randint(0, 99) for x in range( 15 )]
>>> lst
[54, 79, 20, 9, 74, 21, 78, 70, 54, 18, 96, 57, 28, 27, 67]
>>> selectionSort( lst )
[ | 54 79 20 9 74 21 78 70 54 18 96 57 28 27 67 ]
[ 9 | 79 20 54 74 21 78 70 54 18 96 57 28 27 67 ]
[ 9 18 | 20 54 74 21 78 70 54 79 96 57 28 27 67 ]
[ 9 18 20 | 54 74 21 78 70 54 79 96 57 28 27 67 ]
[ 9 18 20 21 | 74 54 78 70 54 79 96 57 28 27 67 ]
[ 9 18 20 21 27 | 54 78 70 54 79 96 57 28 74 67 ]
[ 9 18 20 21 27 28 | 78 70 54 79 96 57 54 74 67 ]
[ 9 18 20 21 27 28 54 | 70 78 79 96 57 54 74 67 ]
[ 9 18 20 21 27 28 54 54 | 78 79 96 57 70 74 67 ]
[ 9 18 20 21 27 28 54 54 57 | 79 96 78 70 74 67 ]
[ 9 18 20 21 27 28 54 54 57 67 | 96 78 70 74 79 ]
[ 9 18 20 21 27 28 54 54 57 67 70 | 78 96 74 79 ]
[ 9 18 20 21 27 28 54 54 57 67 70 74 | 96 78 79 ]
[ 9 18 20 21 27 28 54 54 57 67 70 74 78 | 96 79 ]
[ 9 18 20 21 27 28 54 54 57 67 70 74 78 79 | 96 ]
```

printSorting Code

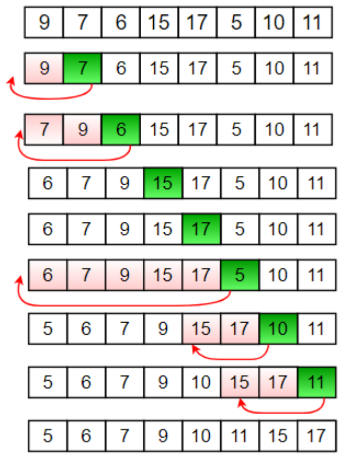
This is the code to print the list as the selectionSort proceeds:

```
def printSorting( lst, point ):
    print( "[ ", end="" )
    for i in range( point ):
        print( lst[i], end = " " )
    print( "|", end = " " )
    for i in range( point, len(lst) ):
        print( lst[i], end = " " )
    print( "]")
```


Insertion Sort

Another simple (but pretty inefficient) sorting algorithm is **insertion Sort**.

Algorithm: For each index in the list, take the element at that position and insert it into the sorted elements before it in the list.



Insertion Sort

```
def insertionSort( lst ):
    for i in range( 1, len(lst) ):
        printSorting( lst, i )
        # insert lst[i] into sorted sublist
        # lst[0:i] so that lst[0:i+1] is sorted
        currentElement = lst[i]
        k = i - 1
        while k >= 0 and lst[k] > currentElement:
            lst[k + 1] = lst[k]
            k -= 1
        # Insert the current element into lst[k+1]
        lst[k + 1] = currentElement
    printSorting( lst, i+1 )
```


Insertion Sort Execution

```
>>> from InsertionSort import insertionSort
>>> import random
>>> lst = [random.randint(0, 99) for x in range( 15 )]
>>> lst
[94, 38, 59, 36, 72, 89, 65, 76, 63, 90, 39, 49, 34, 27, 47]
>>> insertionSort( lst )
[ 94 | 38 59 36 72 89 65 76 63 90 39 49 34 27 47 ]
[ 38 94 | 59 36 72 89 65 76 63 90 39 49 34 27 47 ]
[ 38 59 94 | 36 72 89 65 76 63 90 39 49 34 27 47 ]
[ 36 38 59 94 | 72 89 65 76 63 90 39 49 34 27 47 ]
[ 36 38 59 72 94 | 89 65 76 63 90 39 49 34 27 47 ]
[ 36 38 59 72 89 94 | 65 76 63 90 39 49 34 27 47 ]
[ 36 38 59 65 72 89 94 | 76 63 90 39 49 34 27 47 ]
[ 36 38 59 65 72 76 89 94 | 63 90 39 49 34 27 47 ]
[ 36 38 59 63 65 72 76 89 94 | 90 39 49 34 27 47 ]
[ 36 38 59 63 65 72 76 89 90 94 | 39 49 34 27 47 ]
[ 36 38 39 59 63 65 72 76 89 90 94 | 49 34 27 47 ]
[ 36 38 39 49 59 63 65 72 76 89 90 94 | 34 27 47 ]
[ 34 36 38 39 49 59 63 65 72 76 89 90 94 | 27 47 ]
[ 27 34 36 38 39 49 59 63 65 72 76 89 90 94 | 47 ]
[ 27 34 36 38 39 47 49 59 63 65 72 76 89 90 94 | ]
```

Two-Dimensional Lists

Recall that lists in Python are *heterogeneous*, meaning that you can have items of various types. Lists items can themselves be lists, lists of lists, etc.

```
>>> grades = [ ['Susie',   80, 59, 90, 75, 100], \
               ['Frank',   67, 87, 49, 24, 90], \
               ['Albert',  86, 59, 74, 82, 99], \
               ['Charles', 79, 69, 70, 80, 94] ]
>>> grades[0]        # a list
['Susie', 80, 59, 90, 75, 100]
>>> grades[0][0]     # an element of a list
'Susie'
>>> grades[2][3]
74
```

Note that if the item at `lst[i]` is itself a list, you can index into that list. You can think of them as row and column indexes.

Processing 2D Lists

In slidesets 3 and 9 we tackled the problem of processing student grades and printing a nice table of results. Let's try it again with a 2D representation of grades:

```
grades = [ ['Susie',   80, 59, 90, 75, 100], \
           ['Frank',   67, 87, 49, 24, 90], \
           ['Albert',  86, 59, 74, 82, 99], \
           ['Charles', 79, 69, 70, 80, 94] ]
```

Here each item in grades is a list containing a name, and 5 exam grades.

Processing 2D Lists

In file `ProcessStudentGrades.py`:

```
# The number of exams is a global constant.
EXAM_COUNT = 5

# As usual, we need to print the header lines.

def printHeader():
    """ Print the header line for our table of grades. """
    print( "Name       | ", end = "" )
    for i in range(1, EXAM_COUNT + 1):
        print( " T" + str(i) + " ", end = "" )
    print(" Avg")
    print( "-----------|-", "----" * (EXAM_COUNT + 1), \
          "----", sep = "" )
```

Note that the header depends on `EXAM_COUNT`/

```
>>> printHeader()
Name       |  T1  T2  T3  T4  T5  Avg
-----------|------------------------------
```


Processing 2D Lists

```
def printGrades( grades ):
    """ Given a set of names and grades in a 2D list, print
        them out in a nice tabular format. """
    printHeader()

    # There is one line/record for each student.
    numStudents = len(grades)

    for student in range( numStudents ):
        # Print the student name.
        print( format( grades[student][0], "10s"), \
               "|", end = "" )

        # Compute the sum of exam grades for this student.
        gradesSum = 0
        for j in range( 1, EXAM_COUNT+1 ):
            print( format( grades[student][j], "4d" ), \
                   end = "" )
            gradesSum += grades[student][j]

        # Print average for this student's exams.
        print( format( gradesSum / EXAM_COUNT, "6.2f" ) )
```

Printing Table

Here's the result printing the table:

```
>>> from ProcessStudentGrades import *
>>> EXAM_COUNT
5
>>> grades = [ ['Susie',   80, 59, 90, 75, 100], \
               ['Frank',   67, 87, 49, 24, 90], \
               ['Albert',  86, 59, 74, 82, 99], \
               ['Charles', 79, 69, 70, 80, 94] ]
>>> printGrades( grades )
Name       |  T1  T2  T3  T4  T5   Avg
-----------|----------------------------
Susie      |  80  59  90  75 100 80.80
Frank      |  67  87  49  24  90 63.40
Albert     |  86  59  74  82  99 80.00
Charles    |  79  69  70  80  94 78.40
```


Let's Take a Break

Computing Averages

Now let's compute and print the averages for each exam:

```
def computeTestAverages( grades ):
    """ Given a 2D list of student grades, compute the
        average of each test and print them. """

    # Create an array of EXAM_COUNT 0's.
    sums = [ 0 for x in range( EXAM_COUNT ) ]

    # There is one line/record for each student.
    numStudents = len(grades)
    for student in range( numStudents ):
        for exam in range(1, EXAM_COUNT + 1):
            # grades has a name at the start of each line,
            # but sums doesn't.
            sums[exam - 1] += grades[student][exam]
    # Compute and print the averages for each Exam.  Exams
    # are numbered from 1 to EXAM_COUNT.
    for i in range( EXAM_COUNT ):
        print( "Test" + str(i+1) + " average: ", \
               sums[i] / numStudents )
```


Putting It Together

```
>>> from ProcessStudentGrades import *
>>> printGrades( grades )
Name       |  T1  T2  T3  T4  T5   Avg
-----------|----------------------------
Susie      |  80  59  90  75 100 80.80
Frank      |  67  87  49  24  90 63.40
Albert     |  86  59  74  82  99 80.00
Charles    |  79  69  70  80  94 78.40
>>> computeTestAverages( grades )
Test1 average:  78.0
Test2 average:  68.5
Test3 average:  70.75
Test4 average:  65.25
Test5 average:  95.75
```

Think about how you'd add the class average.

One More 2D List Example

Let's think about how we might generate a 2D list and fill it with random ints. Here's the book's solution (section 11.2.2).

In file Lists2D.py:

```
def listOfListsRandomValues ():
    # This generates a 2D list of random numbers in [0..99].
    # Dimensions are input at run time by the user.
    numberOfRows = int( input ("How many rows?: ") )
    numberOfColumns = int( input ("How many columns?: ") )
    matrix = []          # create an empty list
    for row in range( numberOfRows ):
        # Add an empty new row
        matrix.append( [] )

        # Fill it with numberOfColumns random ints
        for column in range( numberOfColumns ):
            matrix[row].append( random.randint( 0, 99 ) )

    # Finally, print out the newly generated matrix in
    # tabular format.
    printMatrix( matrix, numberOfRows )
```


2D List Example

```
> python Lists2D.py
How many rows?: 8
How many columns?: 10
[ [60, 4, 80, 55, 3, 13, 32, 29, 95, 11]
  [58, 91, 4, 68, 73, 19, 68, 79, 65, 11]
  [44, 93, 54, 59, 46, 34, 56, 74, 9, 2]
  [41, 70, 9, 64, 63, 47, 2, 30, 18, 13]
  [10, 46, 83, 31, 70, 39, 79, 24, 41, 69]
  [82, 19, 65, 78, 65, 42, 9, 31, 40, 51]
  [94, 49, 49, 82, 75, 19, 95, 42, 72, 34]
  [58, 29, 59, 49, 70, 36, 31, 46, 99, 20] ]
```


printMatrix Code

BTW: Here's the function used to print out the matrix in a nice tabular format. *Review this code on your own.*

```
def printMatrix( matrix, numRows ):
    """ Print a 2D matrix in nice format.  Note
        that we're OK with the way a 1D matrix
        prints. """

    print("[ ", end = "" )
    for i in range( numRows ):
        if ( not i ):               # i.e., i == 0
            print( matrix[i], end = "" )
        else:
            print( "  ", matrix[i], end = "" )
        if (i == numRows - 1 ):
            print(" ]" )
        else:
            print()
```


2D List Example Alternative

An alternative approach to solving this problem is to notice that the 2D list contains `numberOfRows` lists, each containing `numberOfColumns` random integers.

```
def listOfRandomValues ( num ):
    # This generates a list of random numbers
    # in [0..99] of length num.
    return [random.randint(0, 99) for x in range( num )]
```

Notice the use of **list comprehension**.

2D List Example Alternative

```
def listOfListsRandomValues2 ():
    # This generates a 2D list of random numbers
    # in [0..99]. Dimensions are input at run time
    # by the user.

    numberOfRows = int( input ("How many rows?: ") )
    numberOfColumns = int( input ("How many columns?: ") )
    matrix = []          # create an empty list
    for row in range( numberOfRows ):
        # Add a new row, which is just a list of
        # numberOfColumns random ints.
        matrix.append(listOfRandomValues( numberOfColumns ))

    # Finally, print out the newly generated matrix
    printMatrix( matrix, numberOfRows )
```


Running Version 2

```
>>> from Lists2D import *
>>> listOfRandomValues( 10 )
[77, 86, 16, 9, 79, 32, 50, 7, 63, 85]
>>> listOfRandomValues( 10 )
[21, 27, 91, 15, 26, 83, 5, 0, 58, 87]
>>> listOfListsRandomValues2()
How many rows?: 9
How many columns?: 6
[ [64, 58, 77, 89, 93, 24]
  [56, 57, 73, 35, 29, 78]
  [69, 11, 74, 24, 3, 72]
  [85, 14, 91, 26, 41, 63]
  [0, 5, 23, 34, 59, 53]
  [48, 29, 75, 83, 88, 90]
  [63, 86, 43, 99, 38, 58]
  [53, 27, 21, 69, 2, 8]
  [27, 45, 86, 99, 39, 45] ]
```


2D List Example Alternative Alternative

This uses list comprehension in a pretty sophisticated way:

```
def listOfListsRandomValues3 ():
    # This generates a 2D list of random numbers in [0..99].
    # Dimensions are input at run time by the user.

    numberOfRows = eval( input ("How many rows?: ") )
    numberOfColumns = eval( input ("How many columns?: ") )

    matrix = [ [ random.randint( 0, 99 ) \
                 for col in range( numberOfColumns ) ] \
               for row in range( numberOfRows ) ]

    # Finally, print out the newly generated matrix
    printMatrix( matrix, numberOfRows )
```


Running Version 3

```
>>> from Lists2D import *
>>> listOfListsRandomValues3 ()
How many rows?: 7
How many columns?: 8
[ [58, 16, 60, 81, 42, 76, 83, 49]
  [18, 71, 10, 12, 65, 84, 86, 21]
  [57, 54, 30, 12, 65, 9, 70, 6]
  [70, 97, 3, 71, 77, 30, 3, 88]
  [28, 93, 12, 66, 38, 90, 94, 75]
  [38, 23, 7, 42, 50, 8, 38, 71]
  [15, 60, 74, 3, 17, 42, 9, 59] ]
```

Sorting 2D Lists

If you call the `sort` method on a 2D list, it sorts into *lexicographic order*—sort on the first column of each row. If two rows match in the first columns, sort those rows on the second column, etc.

```
>>> from ProcessStudentGrades import *
>>> printGrades( grades )
Name       |  T1  T2  T3  T4  T5  Avg
-----------|-----------------------------
Susie      |  80  59  90  75 100 80.80
Frank      |  67  87  49  24  90 63.40
Albert     |  86  59  74  82  99 80.00
Charles    |  79  69  70  80  94 78.40
>>> grades.sort()
>>> printGrades( grades )
Name       |  T1  T2  T3  T4  T5  Avg
-----------|-----------------------------
Albert     |  86  59  74  82  99 80.00
Charles    |  79  69  70  80  94 78.40
Frank      |  67  87  49  24  90 63.40
Susie      |  80  59  90  75 100 80.80
```

Had there been two records for Albert, they'd have been sorted by Test1. If those matched, by Test2, etc.

Ragged Lists

There's no reason why a 2D list be "rectangular." That is, the rows can be of different lengths.

```
listOfLists = [ [ 1, 2, 3, 4, 5 ],
                [ 6, 7, 8 ],
                [ ],
                [ 9, 10, 11, 12 ] ]
```

Writing code to process such a "ragged" list requires a bit more care.

Multidimensional Lists

It is sometimes useful to process 3D lists, 4D lists, or lists of even higher dimension. A 3D list is simply a 1D list where each element is a 2D list.