Sorting

Sorting is an operation on a list that orders the list elements in a specific order.

For example:

1. **A list of names** may be sorted in “lexical” =dictionary=alphabetical order.

Here is a formal definition of lexical order from Wikipedia:

The name of the lexicographic order comes from its generalizing the order given to words in a dictionary: a sequence of letters (that is, a word)

   a1a2 ... ak

appears in a dictionary before a sequence

   b1b2 ... bk

if and only if at the first i where ai and bi differ, ai comes before bi in the alphabet.

That comparison assumes both sequences are the same length. To ensure they are the same length, the shorter sequence is usually padded at the end with enough "blanks" (a special symbol that is treated as coming before any other symbol).

Note that we can order the names in reverse order, from latest to earliest. In this case the words still are in lexicographic order, but from the last element to the first.

2. **A list of integers** may be listed in either from smallest to largest or vice versa.

**Why sort?**

It turns out that sorting is one of the most important operations that programs perform. Two examples.

1. **Searching a list.** In order to find a specific element in a list we often sort it first. A sorted list can be searched much more quickly than one that is unsorted. Imagine looking up a phone number in a phone book (list) with a million entries. If the list is unsorted, we might need to look at 1,000,000 entries. If it is sorted, we don’t need more than 20.

2. **A Scrabble dictionary.** We might want to bring all the words with the same letters next to each other in the list. So if we got the letters ‘ **opts**’ we would like to have **stop, pots, and tops all next to one another**. Imagine that we had a function called “signature()” that transforms each of stop, tops and post => opts. Then if D is the list of dictionary words then

   D.sort(key=signature)

would do this for us. We will actually do this later on.
Problem:

Given a list of integers, sort it so that its elements will be in ascending order.

Selection Sort


The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Here is an example of this sort algorithm sorting five elements:

| 64 25 12 22 11 |
| 11 25 12 22 64 |
| 11 12 25 22 64 |
| 11 12 22 25 64 |
| 11 12 22 25 64 |

And here is a simple (but not very efficient) implementation.

It uses two new list functions.

a=[4, 2, 7, 1, 45, 23]

def select_sort(x):
    for i in range(len(x)-1):
        y=x[i:]  # each time through the loop look for the minimum from position i to the end.
        m=min(y)
        pos=x.index(m,i,len(x))  # find the index of the first element with value m in the range [i,len(x) )
        x[i],x[pos]=x[pos],x[i]  # swap the element at position i with the element at position pos

select_sort(a)
print(a)

Notice that this function uses two list functions min() and index(). In the following, s is a list.

min(s) which returns the smallest item of s
s.index(x, i, j]) which return smallest k such that s[k] == x and i <= k < j

In the index function i and j are optional. If omitted index searches the whole list. If item x is not found in list s, Python returns an error. In general, we should first as Python “x in s” before using the index function. In function select_sort() we don’t have to do this since we know that m exists.
**Question:** Can you detect two inefficiencies in the implementation above?

**Answer:**

The following is a more efficient implementation of the same algorithm.

```python
def select_sort(x):
    for i in range(len(x)-1):
        m=x[i]
        pos=i
        for j in range(i,len(x)):
            if x[j]<m:
                m=x[j]
                pos=j
        x[i],x[pos]=x[pos],x[i]
```

**Questions:**

Why is it more efficient?

Why does the outer for loop have range \( \text{len(x)}-1 \) but the inner loop has range \((i,\text{len(x)})\)?
Here is another elementary sort called

**Bubble Sort**

Here is the beginning of its Wikipedia entry.

**Bubble sort**, sometimes incorrectly referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

The full article and an animation is here: [http://en.wikipedia.org/wiki/Bubble_sort](http://en.wikipedia.org/wiki/Bubble_sort)

Here is the code.

```
# Bubble Sort

def bubble_sort(a):
    for i in range(len(a)):
        sorted=True
        for j in range(len(a)-i-1):
            if a[j]>a[j+1]:
                a[j],a[j+1]=a[j+1],a[j]
                sorted=False
        if sorted==True:
            return

# Lets test it.
a=[33,6,3,21,1]
bubble_sort(a)
print(a)
```
Sorting … a third way, using Python’s built-in sort function.

L is a list.

```python
>>> a.sort()
L.sort(key=None, reverse=False) -- stable sort *IN PLACE*
```

Notice: function sort() takes 2 optional key word arguments:

- key
- reverse

**key** specifies a function of one argument that is applied to the list elements before the comparison is made.

**reverse** specifies that the list should be in reverse order. That means that “>” is used for comparison rather than “<”.

**Why is it called a “keyword argument”?**

Because if you want to use it, you need to use the “keyword=value” syntax. We will see this below.

Say, for example, we want to sort a list of strings. String comparison depends on capitalization as in the following example. If we wanted to discount the capitalization in the comparison we could use the lower() function.

```python
>>> 'abc'<'ABC'
False
>>> 'abc'>'ABC'
True
>>> 'abc'.lower()<'ABC'.lower()
False
>>> 'ABC'.lower()
'abc'
```
What about the keyword argument “reverse”?

Here is an example.

```python
>>> a=[34, 4, 21, 77, 5, 45, 8]
>>> a.sort()
>>> a
[4, 5, 8, 21, 34, 45, 77]
>>> a.sort(reverse=True)
>>> a
[77, 45, 34, 21, 8, 5, 4]
```
Problem:

Given a list, print the elements of that list in reverse order. Do this in two ways.

Problem:

Given a list reverse the elements of the list. For example if

$x = [1,2,3]$, then after it is reversed $x$ would be $[3,2,1]$.

Do this in two ways.
Two dimensional lists

Many important applications use data that is represented in a 2-dimensional table.

```
   10  20  30
   40  50  60
   70  80  90
```

How do we represent this in Python?

We simply use a list of lists.

```
a=[[10,20,30],[40,50,60],[70,80,90]]
```

Notice that the length of list a is 3 (len(a)==3), but it's made up of three lists, each one of length 3.

```
>>> a=[[10,20,30],[40,50,60],[70,80,90]]
>>> len(a)
3
>>> len(a[0])
3
```

Problem:

Change element with a 50 to 500.

Solution:

The 50 is the second element of the second list. Recalling that lists are indexed starting with 0, we write:

```
>>> a[1][1]=500
>>> a
[[10, 20, 30], [40, 500, 60], [70, 80, 90]]
```
Nested loops and two dimensional lists

Even though a list is “really” is a list of lists, when we program its useful to think of it as a two dimensional table.

So, for the list a above, we can consider it a table with three rows and three columns. The rows and columns are each indexed starting at 0. We will say that the position with entry 500 is at row 1 and column 1.

We saw how lists and loops are “made for each other.” The same is true with two dimensional lists (we will sometimes refer to the as two dimensional “arrays”. This is what they are called in many other programming languages (though they are implemented differently).

Problem:

Print list a above so that each “row” of it prints a separate row.

```python
>>> for i in range(3):
    for j in range(3):
        print(a[i][j], end=' ')
    print()
10 20 30
40 500 60
70 80 90
>>> >>>
And formatted …

```python
>>> for i in range(3):
    for j in range(3):
        print(format(a[i][j],">6d"), end=' ')
    print()
10 20 30
40 500 60
70 80 90
```
Problem:

Create a 4X4 array and initialize each of the elements to 0.

Solution:

```python
a=[]
for i in range(4):
    a.append(4*[0])
```

What does $4*\[8\]$ mean?

Python lets us use $+$ and $*$ with lists.

```python
>>> a=[1,2,3]
>>> a+4
[1, 2, 3]
>>> a+4
Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    a+4
TypeError: can only concatenate list (not "int") to list
>>> a+[4]
[1, 2, 3, 4]
>>> a+[4]
```

So from here you see $+$ is **concatenate** i.e. it acts like the list function **extend**. But you can only $+$ a list to a list, not a string to a list like you can with extend.

What about **"*"**?

```python
>>> a=4*[8]
>>> a
[8, 8, 8, 8]
```

**Important:** $n*s$ or $n*\[s\]$ create $n$ shallow copies of list $s$ concatenated.

What is a **shallow copy**?

**Answer:** Say we have $a=n*s$. If any of the elements of $a$ is a list, call it $x$, then list $a$ will **not** have $n$ independent copies of sublist $x$, but rather $n$ **references to one copy of list $x$**. See the example on the next page. The opposite of a shallow copy is called a **deep copy**. We will make deep copies when we need to create independent copies of some list.

**How?** We will see examples below.
Make sure that you can explain each of the following:

```python
>>> a=3*3*[[0]]
>>> a
[[0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```python
>>> a=3*3*[0]
>>> a
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```python
>>> a=3*[3*[0]]
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

The first example:

The second example:

The third example:

The fourth example (below). Does this produce the same result as third example above?

```python
>>> a=[]
>>> for i in range(3):
    a.append(3*[0])

>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```
Notice when we print list a, both seem to produce the same list:

```python
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

However, internally they are represented very differently.

The first one produces the following when it runs:

![Diagram of frames and objects for the first list creation]

What are the implications of this?

The second one, however, produces this:

![Diagram of frames and objects for the second list creation]

What are the implications of this?
Problem:

Ask the user for two integer $n$ and $m$ where $m > n$.

Create a list $x$ of size $n$. Populate the list with $n$ random integers in the range 1 through $m$.

Problem:

Modify the answer to the problem above so that all of the integers in list $x$ are unique.
Sets

Like a list, a set is a “container” that gives us a place to store a collection of elements. Unlike a list, a set can contain only one copy of any element.

```python
>>> s=set()
>>> s.add(1)
>>> s.add(2)
>>> s
{1, 2}
>>> s.add(1)
>>> s
{1, 2}
```  
We can use Python’s built-in set container to check for duplicates.

We can create a set S with the following syntax:

```python
s=set(iterable)
```

S=set() creates an initially empty set.

If IT is some iterable (like a set) the s=set(IT) creates a set that is initialized with the elements of IT (for example a list)

The set provides the following operations (among others):

- `add(elem)`  
  Add element elem to the set.
- `remove(elem)`  
  Remove element elem from the set. Raises KeyError if elem is not contained in the set.
- `discard(elem)`  
  Remove element elem from the set if it is present.
- `pop()`  
  Remove and return an arbitrary element from the set. Raises KeyError if the set is empty.
- `clear()`  
  Remove all elements from the set.
- `len(s)`  
  Return the cardinality (number of elements) of set s.
- `x in s`  
  Test x for membership in s.
- `x not in s`  
  Test x for non-membership in s.
Problem:

Ask the user for three integer \( n \) and \( m \) and \( k \), where \( k > m \cdot n \).

Create a two dimensional table (list of lists) \( x \) of size \( n \cdot m \). Populate the list with \( n \cdot m \) unique random integers in the range 1 through \( k \).

Problem:

Write a function \( \text{get}_\text{row}(x, i) \) which takes a square matrix (=table=list of lists) and returns a list containing the \( i \)th row of matrix \( X \).
Problem:

Write a function get\_col(x,i) which **takes a square matrix** (=table=list of lists) and **returns a list** containing the ith column of matrix X.

Problem:

Write a function to calculate and return the sum of all the elements on the “main diagonal” of the square two dimensional list x passed to it as an argument. This is the diagonal going from the **upper left to the bottom right**.
Problem:

Like the problem above, calculate and return the diagonal sum, but for the elements along the diagonal going from the top right to the lower left.

List Comprehensions

A list comprehension is a very compact and useful notation that Python provides for creating lists.

It is very similar to the notation that is uses in math for specifying sets. For example it is pretty clear what this means:

\[ \{x^2 : 1 \leq x \leq 100 , \text{if } x \text{ is even}\} \]

It is the set of the squares of the even numbers between 1 and 100.

Here is a list comprehension that creates a list with the same numbers:

```python
>>> a=[x**2 for x in range(1,101) if x%2==0]
>>> a
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484, 576, 676, 784, 900, 1024, 1156, 1296, 1444, 1600, 1764, 1936, 2116, 2304, 2500, 2704, 2916, 3136, 3364, 3600, 3844, 4096, 4356, 4624, 4900, 5184, 5476, 5776, 6084, 6400, 6724, 7056, 7396, 7744, 8100, 8464, 8836, 9216, 9604, 10000]
```

Notice that this is equivalent to the following:

```python
a=[]
for x in range(1,101):
    if x%2==0:
        a.append(x)
```

The list comprehension notation is more compact and much clearer (once you get used to it) than the equivalent code above.
The general form has three components:

[ expression using a value from an iterable  for iterable  if conditions ]

(1) (2) (3)

Some more examples:

**Example:**

Assume that we have a function prime(i) which returns True if i is a prime number and False otherwise.

Create a list of all the primes between 2 and 120.

\[ y = [ k \text{ for } k \text{ in range(2, 121) if prime(k)} ] \]

**Example:**

Let \( x \) and \( y \) be two lists of numbers, both of length \( n \). To get the dot product of \( x \) and \( y \) we do the following:

- form the pairs \( x[i] \ast y[i] \) for \( 0 \leq i < n \)
- sum up all the pairs.

For example:

\( x = [1, 2, 3] \) and \( y = [10, 11, 12] \) then \( x \cdot y = 1 \ast 10 + 2 \ast 11 + 3 \ast 12 \).

We can write this as list comprehension like this:

```python
>>> x = [1, 2, 3]
>>> y = [10, 11, 12]
>>> z = sum([x[i] * y[i] for i in range(len(x))])
>>> z
68
```

**Example:**

Let \( z \) be a list of 20 integers, and we want a list of tuples giving the value and position of each even number in the list. The following will do this:

```python
>>> z
[13, 14, 7, 12, 11, 3, 18, 20, 5, 7, 19, 17, 10, 16, 14, 12, 5, 19, 15, 1]
>>> k = [(i, z[i]) for i in range(20) if z[i] % 2 == 0]
>>> k
[(1, 14), (3, 12), (6, 18), (7, 20), (12, 10), (13, 16), (14, 14), (15, 12)]
```
Example:

Write a function get_row(x,i) which takes a square matrix (=table=list of lists) and returns a list containing the element in the ith row of matrix X.

Using a list comprehension we could write:

```python
def get_row(x,i):
    return [ x[i][j] for j in range(len(x)) ]
```

Or ... we could also write

```python
def get_row(x,i):
    return x[i]
```

since the ith row of x is just the ith sublist of x.

However there is an important difference between these two implementations of get_row().

The first version creates a new list made up of the elements of row i of matrix X.

The second one, which is much faster, just returns a reference to the ith row of matrix X.

In the following example, we call both versions of get_row(m,0) on the matrix m=[[1,2],[3,4]].

Notice that both versions return what seems to be the same result: [1, 2]

```python
>>> def get_row(x,i):
    return [ x[i][j] for j in range(len(x)) ]

>>> m=[[1,2],[3,4]]
>>> m
[[1, 2], [3, 4]]
>>> b=get_row(m,0)
>>> b
[1, 2]
>>> def get_row(x,i):
    return x[i]

>>> b=get_row(m,0)
>>> b
[1, 2]
``` 

Question: What is the important practical difference between the two versions?

Hint: What happens when we change list b after it is returned from get_row(). Why does this happen?
Example:

Write a function get_col(x, i) which takes a square matrix (=table=list of lists) and returns a list containing the items in the ith column of matrix X.

def get_col(x, i):
    return [ x[j][i] for j in range(len(x)) ]

Problem:

Write a function sum_col(x, i) which takes a square matrix (=table=list of lists) and the sum of the items in the ith column of matrix X. Use a list comprehension.

Problem:

Write a function diag_diff(x, i) which takes a square matrix (=table=list of lists) and the difference of two main diagonals of matrix X. In other words, let d1 be the diagonal of X from upper left to bottom right, and d2 to be the diagonal from upper right to lower left. The function returns d1-d2. Use list comprehensions.
Problem:

Now, here is a definition: A **saddle point** in a 2 dimensional square table is an entry in the table whose value is the minimum in its row and maximum in its column. In the table below, 0 is a saddle point.

```
2  0
0 -2
```

**Write a function** to find a saddle point in a 2 dimensional table in any two dimensional square of integers, if one exists. If a saddle point was found return a triple (value, x pos, y pos). If a saddle point was not found return “not found”
Problem:

Consider the 5X5 square below:

![5x5 grid diagram]

We would like to have a robot travel from the start square at the upper left to the end square at the lower right. At each square the robot has only one of two moves:

- It may go one square to the right or
- It may go one square down.

Write a program that determines in how many ways may this be done? In other words, how many paths are there from start to end with each move restricted as above?

Solution:

```python
#robot paths
a=5*[5*[0]]
#set up the left and top boundaries of the table (5X5 square array)
for i in range(5):
    a[i][0]=1
    a[0][i]=1

for i in range(1,5):
    for j in range(1,5):
        a[i][j]=a[i-1][j]+a[i][j-1]

print(a[4][4])
```

Here is another approach. Can you figure out how this works? What happened to the table??

```python
n=int(input("Please enter the 'size' of the array, n= "))

def num_paths(n,i,j):
    if i==0 or j ==0:
        return 1
    return num_paths(n,i-1,j)+num_paths(n,i,j-1)

print(num_paths(n, n-1, n-1))
```
Problem:
Write a program to generate all the eight-digit base 8 integers from 00000000 ➞ 77777777

What is a base eight integer?

The following code will do it.

for i0 in range(8):
    for i1 in range(8):
        for i2 in range(8):
            for i3 in range(8):
                for i4 in range(8):
                    for i5 in range(8):
                        for i6 in range(8):
                            for i7 in range(8):
                                q=[i0,i1,i2,i3,i4,i5,i6,i7]
                                print(q)

Problem:
Write a function get_num(n) which returns a list of length 8 representing the base eight representation of the decimal number n.
More on Strings … and Files

We have been using strings all along. Some of the things we have seen include:

- Strings are immutable.
- Strings are iterables so we can use “for”.
- Since strings are sequences, we can access elements and substrings them [].
- We can concatenate strings: s1+s2.

But Python provides many many functions for working with strings. Let’s check out some of the functions as described in the online documentation. The full list is in the Python Library Reference documentation in section 4.6.1.

Here are some of the most useful with definitions and examples from the Library Reference.

```
str.count(sub[, start[, end]])
```

Return the number of non-overlapping occurrences of substring `sub` in the range `[start, end]`. Optional arguments `start` and `end` are interpreted as in slice notation.

```
str.find(sub[, start[, end]])
```

Return the lowest index in the string where substring `sub` is found, such that `sub` is contained in the slice `s[start:end]`. Optional arguments `start` and `end` are interpreted as in slice notation. Return -1 if `sub` is not found.

```
str.join(iterable)
```

Return a string which is the concatenation of the strings in the `iterable`. A `TypeError` will be raised if there are any non-string values in `seq`, including `bytes` objects. The separator between elements is the string providing this method.

```
str.replace(old, new[, count])
```

Return a copy of the string with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.

```
str.split([sep[, maxsplit]])
```

Return a list of the words in the string, using `sep` as the delimiter string. If `maxsplit` is given, at most `maxsplit` splits are done (thus, the list will have at most `maxsplit+1` elements). If `maxsplit` is not specified, then there is no limit on the number of splits (all possible splits are made).

If `sep` is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `',,''.split('')` returns `['1', '', '2']`). The `sep` argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

121
If `sep` is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example, `' 1 2 3 '.split()` returns `['1', '2', '3']`, and `' 1 2 3 '.split(None, 1)` returns `['1', '2 3']`.

`str.splitlines([keepends])`

Return a list of the lines in the string, breaking at line boundaries. **Line breaks are not included in the resulting list** unless `keepends` is given and true.

`str.rstrip([chars])`

Return a copy of the string with trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. **The `chars` argument is not a suffix; rather, all combinations of its values are stripped:**

There are also functions `lstrip()` and `rstrip()` that act in the expected way.

`str.upper()`

Return a copy of the string converted to uppercase.

`str.lower()`

Return a copy of the string converted to lowercase.
Problem:

Write a function that reverses a string. Remember, a string is not mutable! Do it two ways.

Answer:

Problem:

Write a function is_len(s,n) which returns True if string s is at least n characters long, and False otherwise.

Answer:

Problem:

Write a function one_upper(s) which returns True if exactly one character in string s is capitalized, and False otherwise. You can assume that the string s contains only alphabetic characters and no blanks. You might want to consider other string functions from the documentation.

Answer:
Problem:

Write a function clean(x) where x is a list of “words”. Each word is a string that might have either a ‘.’ ‘,’ or ‘;’ tacked on at the end. clear() will return a list with the original words stripped of the punctuation.

```python
>>> a=['asd','er.','rt','fgh;']
>>> clear(a)
['asd', 'er', 'rt', 'fgh']
```

Answer:

Problem:

Write a function scrape(s) which take a string s representing the HTML of a web page and returns a list of all links found on page s. We recognize the beginning of a link by looking for ‘http://’.

Answer:
Files

What are “files” and how are they represented on the drives?
Basic file operations

output = open(r'C:\spam', 'w') Create output file ('w' means write)
input = open('data', 'r') Create input file ('r' means read)
input = open('data') Same as prior line ('r' is the default)
aString = input.read() Read entire file into a single string
aString = input.read(N) Read up to next N characters (or bytes) into a string
aString = input.readline() Read next line (including \n newline) into a string
aList = input.readlines() Read entire file into list of line strings (with \n)
output.write(aString) Write a string of characters (or bytes) into file
output.writelines(aList) Write all line strings in a list into file
print(value, …file='filename') Write to file “filename” instead of to the screen
output.close() Manual close (done for you when file is collected)
output.flush() Flush output buffer to disk without closing any File.
seek(N) Change file position to offset N for next operation
for line in open('data'): use line File iterators read line by line
open('f.txt', encoding='latin-1') Python 3.0 Unicode text files (str strings)
open('f.bin', 'rb') Python 3.0 binary bytes files (bytes strings)

Problem:

Create this file (bears.txt) in your python directory:

Once upon a time
there were
three bears,
a poppa, a momma,
and a little baby bear!

Read the file line by line and print it out..

Answer:

f=open('bears.txt')
for i in f:
    print(i)

produces: Why the spaces between the lines?

>>> 
once upon a time
there were
three bears
a poppa, a momma,
and a little baby bear!
>>>
**Problem:**
Read the file into one string and print the string.

**Answer:**

```python
f=open('bears.txt')
z=f.read()
print(z)
```

produces: What happened to the blank lines?

```python
>>> Once upon a time
there were
three bears
a poppa, a momma,
and a little baby bear!
```  

**Problem:**
Read the file into a string s, separate the words into a list (use ‘ ‘ to indicate the separator between words. Print out the list.

**Answer:**

```python
f=open('bears.txt')
z=f.read()
z=z.split(' ')
print(z)
```

produces:

```python
['Once', 'upon', 'a', 'time
there', 'were
three', 'bears\na', 'poppa', 'a', 'momma,\nand a little baby bear!']
```  

Note the ‘\n’, the newline character. This causes a line break.

but …

```python
f=open('bears.txt')
z=f.read()
z=z.splitlines()
print(z)
```
Problem:

Create a text file, 'grades.txt', with the following data.

Bob 78 98 67 77
Joan 78
Sally 90 97 77 56 88 98

Write a program to read this file and print out the students name followed by their average on all exams.

Answer:

```python
#Student averages
f=open('grades.txt')
for i in f:
    s=i.split()
    average=sum([int(s[i]) for i in range(1,len(s))])/(len(s)-1)
    print(format(s[0], '<7s'), 'Average=', format(average, '.2f'))
```

Problem:

Modify the above program so that it writes the result to a file called averages.

Answer:
Dictionaries

Think of an on-line dictionary. You type in a word and the dictionary returns one or more meanings of the word you entered.

We can think of the dictionary as a “list” that is indexed by the “word” whose definition you seek and the value that is returned is the set of meaning associated with that word.

Or …

Think of an on-line phone book. You type in the name of the person whose phone number you want and the phone book app returns the associated number.

We can model the above in Python using the dictionary.

```
>>> pb=dict()
>>> pb['Bob']='212-444-5670'
>>> pb['Joan']='718-767-3223'
>>> pb['George']='212-998-6756'
SyntaxError: invalid syntax
>>> pb['George']='212-998-6756'
>>> pb['Bob']
'212-444-5670'
>>> pb['Bob']='617-788-3479'
>>> pb['Bob']
'617-788-3479'
>>> pb['Chuck']
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    pb['Chuck']
KeyError: 'Chuck'
>>> 'Bob' in pb
True
>>> 'Chuck' in pb
False
>>> 'Chuck' in pb
False
>>> for i in pb:
    print(i)
Bob
Joan
George
>>> for i in pb:
    print(i, pb[i])
Bob 617-788-3479
Joan 718-767-3223
George 212-998-6756
```
Here are some of the dictionary methods:

```python
dict()
```
Create a new dictionary.

```python
len(d)
```
Return the number of items in the dictionary `d`.

```python
d[key]
```
Return the item of `d` with key `key`. Raises a `KeyError` if `key` is not in the map.

```python
d[key] = value
```
Set `d[key]` to `value`.

```python
del d[key]
```
Remove `d[key]` from `d`. Raises a `KeyError` if `key` is not in the map.

```python
key in d
```
Return `True` if `d` has a key `key`, else `False`.

```python
key not in d
```
Equivalent to `not key in d`.

```python
clear()
```
Remove all items from the dictionary.

```python
copy()
```
Return a shallow copy of the dictionary.

Create a new dictionary with keys from `seq` and values set to `value`.

```python
items()
```
Return a new view of the dictionary’s items (`(key, value)` pairs). See below for documentation of view objects.

```python
keys()
```
Return a new view of the dictionary’s keys. See below for documentation of view objects.

```python
pop(key[, default])
```
If \textit{key} is in the dictionary, remove it and return its value, else return \textit{default}. If \textit{default} is not given and \textit{key} is not in the dictionary, a \texttt{KeyError} is raised.

\texttt{values()}

Return a new view of the dictionary’s values

There are additional methods. Check out the Python on-line documentation.

\textbf{Problem:}

What does the following program do?

\begin{verbatim}
def squish(x):
    result=[]
    count=0

    a=x[0]
    for i in range(len(x)):
        if x[i]==a:
            count+=1
        else:
            result.append((count,a))
            count=1
            a=x[i]
            result.append((count,a))
    return(result)

def clean(x):
    y=''
    s=['.','',' ','-']
    for i in x:
        if i in s:
            continue
        else:
            y+=i
    return y

d={}
ln=0
for line in open('GB.txt'):
    ln+=1
    line=clean(line)
    l=line.split()
    for word in l:
        if word not in d:
            d[word]=[]
            d[word].append(1)
            d[word].append([ln])
        else:
            d[word][0]+=1
            d[word][1].append(ln)

print('list of d')
ld=list(d)
ld.sort()
for k in ld:
    print(k,d[k])
    #print(k,d[k][0],squish(d[k][1]))
\end{verbatim}
Problem:

What does the following program do?

```
import pickle

def signature(w):
    w1=list(w)
    w1.sort()
    w1=''.join(w1)
    return w1

# create or load?
mode=input("Create or Load C or L: ")
print()
if mode.upper()=='C':
    # create a "Scrabble Dictionary"
    d={}
    print('Creating dictionary ... please wait.')
    f = open('C:/python32/six letter words.txt', 'r')
    print()
    sl = f.read()
    z=sl.split(' ')
    print()
    for w in z:
        sig=signature(w)
        if sig not in d:
            d[sig]=[]
            d[sig].append(w)
        else:
            d[sig].append(w)
    f.close()
else:
    print('Unpickling dictionary ... please wait.')
    f=open('slwords','rb')
    d=pickle.load(f)

word=input("Please enter word: ")
print()

while word!='done':
    if len(word)!=6:
        print("word not 6 chars")
    else:
        word=word.upper()
        word=signature(word)

        if word in d:
```
    print(d[word])
else:
    print(word, 'not found.')
word = input("Please enter word: ")
print()

f = open('slwords', 'wb')
print('Pickling ... please wait.')
pickle.dump(d, f)
f.close()