# Strings, Lists, Sets, Dictionaries and Files

## *4.1 Strings*

*Use of String Variables*

We have already seen strings, but since we've been introduced to loops and index variables, we can learn a bit more about manipulating strings. The most natural way to initialize a string variable is through the input statement:

```
name = input("Please enter your name.\n")
```

In this example, whatever the user enters will be stored in name.

Once we have a string stored in a variable, there are a number of ways to access parts of the string, check the string for letters or manipulate the string.

*Checking to see if a letter is in a string*

Python allows for a very simple method to check to see if an letter or any other character for that matter is in the string, using the in operator:

```
name = input("Please enter your name.\n")

if 't' in name:
    print("You have an t in your name.")
```

This operation works as follows:

```
<character> in <string>
```

This expression is a Boolean expression that will evaluate to True if the specified character is in the string, and false otherwise. Not only will this work to check for a character, but it will also work to check for a substring, or a consecutive sequence of characters in a larger string. For example, the expression `'put' in 'computer'` evaluates to true since the fourth, fifth and sixth letters of `'computer'` are `'put'`.

*Indexing into a String – Non-negative Indexes*

A common operation with a string is to access one character in a string. This can be done with square brackets. If name is a string, then `name[0]` represents the first character in the string, `name[1]` represents the second character in the string name, and so on. In addition, to find the length of a string, we use the `len` function, which will return the total number of characters in the string. Thus, a method to determine if a character is in a string from first principles is as follows:

```
name = input("Please enter your name.\n")

flag = False
for i in range(len(name)):
    if name[i] == 't':
        flag = True

if flag:
    print("You have an t in your name.")
```

We can easily use indexing to count the number of times a particular character appears in a string. Here is the previous segment of code edited to print out the number of times ‘t’ appears in the string entered by the user:

```
name = input("Please enter your name.\n")

count = 0
for i in range(len(name)):
    if name[i] == 't':
        count = count + 1

print("You have an t in your name", count, "times.")
```

*Indexing into a String – Negative Indexes*

Python also allows negative indexes into a string, which is a feature many other languages do not support. If you give a negative integer as an index to a string, Python will start counting from the end of the string. For example, here are the corresponding indexes for the string hello:

| index | -5 | -4 | -3 | -2 | -1 |
|---|---|---|---|---|---|
| string | ‘h’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ |

Though you are not required to use negative indexes, they can come in handy sometimes, if you want to look for a character a specified number of positions from the end of a string. Without negative indexing, you would have to do the math on your own, using the len function and subtracting.

Here is a simple example where negative indexing simplifies a program. In the following program we will ask the user to enter a string with only uppercase letters and we will determine whether or not the string is a palindrome. A palindrome is a string that reads the same forwards and backwards.

The key strategy here will be to maintain two indexes: one from the front, counting from 0 and one from the back counting backwards from -1. We want to check to see if corresponding characters from the front and back match. If we find a mismatch, we immediately know our string is not a palindrome. We can stop halfway through the string. Remember we must use integer division to determine the point since the range function only takes in integers.

```
def main():

    word = input("Please enter a string, uppercase letters only.\n")

    back = -1
    isPal = True
    for i in range(len(word)//2):
        if word[i] != word[back]:
            isPal = False
            break
        back = back - 1

    if isPal:
        print(word,"is a palindrome.")
    else:
        print(word,"is not a palindrome.")

main()
```

*Slicing a String*

Slicing refers to obtaining a substring of a given string. An explicit way to denote a substring is to give both its starting index and ending index. In Python, just as we saw with the range function, the ending value is not included in the set of values described in a slice. Thus, the starting index is inclusive, while the ending index is exclusive. Given that the string word was set to “hello”, the slice word[2:4] would be “ll” and the slice word[1:2] would simply be “e”.

We can slice strings with negative indexes as well. This IDLE transcript should clarify the rules for slicing where both indexes are specified:

```
>>> word = "PYTHONISCOOL"
>>> print(word[2:7])
THONI
>>> print(word[6:6])

>>> print(word[4:2])

>>> print(word[5:11])
NISCOO
>>> print(word[8:12])
COOL
>>>
>>> print(word[-7:-2])
NISCO
>>> print(word[-9:8])
HONIS
>>> print(word[-3:-7])

>>>
```

Notice if you attempt to slice a string where the starting index refers to a position that occurs at or after the ending index, the slice is the empty string, containing no characters.

A string can also be sliced using only one index. If only one index is given, Python must know if it's the start or end of the slice. It assumes that the omitted index must refer to the beginning or end of the string, accordingly. These examples should clarify slicing using only one index:

```
>>> print(word[8:])
COOL
>>> print(word[-6:])
ISCOOL
>>> print(word[:6])
PYTHON
>>> print(word[:-4])
PYTHONIS
>>>
```

*String Concatenation*

We have briefly seen string concatenation before. It is accomplished via the plus sign (+). If two strings are "added", the result is sticking the first string followed by the second string. This is helpful in printing and especially in the input statement, which only takes a single string as a parameter. Here is a short example that utilizes string concatenation:

```
first = input("Please enter your first name.\n")
last = input("Please enter your last name.\n")
full = first+" "+last
print(full)
```

In order for python to recognize that you want string concatenation, BOTH operands must be strings. For example, if you try to do "hello" + 7, you will get a syntax error since strings and integers are not allowed to be added or concatenated.

*Pig Latin Example*

Pig Latin is a common children's code used to (sort of) hide the meaning of what is being said. There are many variations, but the variation implemented here will use the following rules[1]:

1) For words that start with a consonant and have a vowel, take the first consonant cluster, cut and paste it to the end of the word and add "ay" after it.

2) For words that start with a vowel, add "way" after it.

3) For words with no vowels, keep them as is, since they are probably difficult enough to understand![2]

Our first task will be to find the index of the first vowel, if it exists. We must be careful to not commit an index out of bounds error, where we attempt to index the string at an invalid index. We do this by first checking to see if the index is within the appropriate bounds before indexing the string with it. Short-circuiting allows us to do this in one check. Namely, if the first part of a Boolean expression with an and is False, then Python will NOT evaluate the second portion of the expression at all.

Once we identify this index, we split our work into the three cases outlined above, using slicing and string concatenation appropriately.

[1] The first two rules are a simplification of what is posted on Wikipedia.
[2] This rule is my own incarnation so that our program distinguishes between words with and without vowels.

Here is the program in its entirety:

```
def main():

    VOWELS = "AEIOU"
    ans = "yes"

    # Allow multiple cases.
    while ans == "yes":

        mystr = input("Enter a word, all uppercase letters.\n")

        # Pig Latin Rule - Find first vowel
        index = 0
        while index < len(mystr) and (not mystr[index] in VOWELS):
            index = index+1

        # Just add "way" to words that start with a vowel.
        if index == 0:
            print(mystr+"WAY")

        # Move first consonant cluster to end and add "ay"
        elif index < len(mystr):
            print(mystr[index:]+mystr[:index]+"AY")

        # If there are no vowels, just keep it as is!!!
        else:
            print(mystr)

        ans = input("Would you like to translate another word?\n")

main()
```

The outer loop allows the user to test multiple words. The inner loop finds the first index storing a vowel. Note that we could have created a string with consonants to see if mystr[index] was in it, but since it was shorter to type out the vowels, this route was chosen and the not operator was used. In each of the three cases, the appropriate slicing and concatenating is done to produce the output string. Notice that in the second case, the slicing works out nicely, since we are able to use the same index in specifying both slices to “reorganize” the word, so to speak.

## ***4.2 Lists***

*Creating an Empty List*

A list is a sequence of items. In python, a list is an ordered sequence of items, not necessarily of the same type, but typically, most lists contain items all of the same type. Here is how we create an empty list in python:

```
food = []
```

*Adding an Item to the End of a List*

To add something to the end of a list, we can use the append function:

```
food.append("ham")
print(food)
```

The outcome of these two lines is as follows:

```
['ham']
```

Now,  let's add a couple more items:

```
food.append("cheese")
food.append("ice cream")
print(food)
```

Which results in the following list being printed:

```
['ham', 'cheese', 'ice cream']
```

*Removing an Item from a List*

To remove an item from a list, use the remove method:

```
food.remove("ham")
print(food)
```

Specifically, this removes the FIRST instance of the item listed in the parentheses, resulting in the following output:

```
['cheese', 'ice cream']
```

To more specifically illustrate this issue, consider adding the following segment of code:

```
food.append("ice cream")
food.append("cheese")
food.append("ice cream")
food.append("cheese")
food.remove("ice cream")
print(food)
```

This results in the following output:

```
['cheese', 'ice cream', 'cheese', 'ice cream', 'cheese']
```

Note that it's now clear that the first occurrence of "ice cream" was removed.

Just like strings, we can index into a list in the exact same manner:

```
print(food[0])
print(food[1])
print(food[-1])
```

results in the following output:

```
cheese
ice cream
cheese
```

Namely, non-negative indexes count from the beginning of the string, so we first printed the first two items in the list, and negative indexes count from the end of the string, so the last line printed out the last element of the list.

*Searching for an item in a list*

In most programming languages, you are required to search for an item, one by one, in a list. In order to do this, we must know the length of the list. In python, we can use the len function to determine the length of a list, much like we used the len function to determine the length of a string. To implement this strategy in python, we might do the following:

```
item = input("What food do you want to search for?\n")
for i in range(len(food)):
    if food[i] == item:
        print("Found",item,"!",sep="")
```

In this particular code segment, we print out a statement each time we find a copy of the item in the list. We very easily could have set a flag to keep track of the item and only made a single print statement as follows:

```
item = input("What food do you want to search for?\n")
flag = False
for i in range(len(food)):
    if food[i] == item:
        flag = True

if flag:
    print("Found ", item, "!", sep="")
else:
    print("Sorry, we did not find ", item, ".", sep="")
```

One advantage here is that we always have a single print statement with the information we care about. The first technique may produce no output, or multiple outputs. Either can easily be adjusted to count HOW many times the item appears in the list.

Python, however, makes this task even easier for us. Rather than having to run our own for loop through all of the items in the list, we can use the in operator, just as we did for strings:

```
item = input("What food do you want to search for?\n")
if item in food:
    print("Found ", item, "!", sep="")
else:
    print("Sorry, we did not find ", item, ".", sep="")
```

The in operator allows us to check to see if an item is in a list. If an instance of an object is in a list given, then the expression is evaluated as true, otherwise it's false.

Also, just like strings, we can slice a list:

```
allfood = ["ham", "turkey", "chicken", "pasta", "vegetables"]
meat = allfood[:3]
print(meat)
```

The result of this code segment is as follows:

```
['ham', 'turkey', 'chicken']
```

Thus, essentially, what we see is that many of the operations we learned on strings apply to lists as well. Programming languages tend to be designed so that once you learn some general rules and principles, you can apply those rules and principles in new, but similar situations. This makes learning a programming language much easier than a regular language, which requires much more memorization.

If we want, we can assign a particular item in a list to a new item. Using our list meat, we can do the following:

```
meat[0] = "beef"
print(meat)
```

This produces the following output:

```
['beef', 'turkey', 'chicken']
```

In addition, we can assign a slice as follows:

```
meat[:2] = ['ham', 'beef', 'pork', 'lamb']
print(meat)
```

Here, we take the slice [:2] and replace it with the contents listed above. Since we've replaced 2 items with 4, the length of the list has grown by 2. Here is the result of this code segment:

```
['ham', 'beef', 'pork', 'lamb', 'chicken']
```

*del Statement*

We can also delete an item or a slice of a list using the del statement as illustrated below:

```
>>> del meat[3]
>>> print(meat)
['ham', 'beef', 'pork', 'chicken']
>>> meat.append('fish')
>>> meat.append('lamb')
>>> print(meat)
['ham', 'beef', 'pork', 'chicken', 'fish', 'lamb']
>>> del meat[2:5]
>>> print(meat)
['ham', 'beef', 'lamb']
```

*sort and reverse Methods*

Python allows us to sort a list using the sort method. The sort method can be called on a list, and the list will be sorted according to the natural ordering of the items in the list:

```
>>> meat.sort()
>>> print(meat)
['beef', 'ham', 'lamb']
```

We can then reverse this list as follows:

```
>>> meat.reverse()
>>> print(meat)
['lamb', 'ham', 'beef']
```

*Using lists to store frequencies of items*

A compact way to store some data is as a frequency chart. For example, if we asked people how many hours of TV they watch a day, it's natural to group our data and write down how many people watch 0 hours a day, how many people watch 1 hour a day, etc. Consider the problem of reading in this information and storing it in a list of size 24, which is indexed from 0 to 23. We will assume that no one watches 24 hours of TV a day!

We first have to initialize our list as follows:

```
freq = []
```

```
for i in range(24):
    freq.append(0)
```

At this point, we are indicating that we have not yet collected any data about TV watching.

Now, we will prompt the user to enter how many people were surveyed, and this will be followed by reading in the number of hours each of these people watched.

The key logic will be as follows:

```
hrs = input("How many hours does person X watch?\n")
freq[hrs] = freq[hrs] + 1
```

The key here is that we use the number of hours watched as an index to the array. In particular, when we read that one person has watched a certain number of hours of TV a day, we simply want to increment the appropriate counter by 1. Here is the program in its entirety:

```
def main():

    freq = []
    for i in range(24):
        freq.append(0)

    numPeople = int(input("How many people were surveyed?\n"))

    for i in range(numPeople):
        hrs = int(input("How many hours did person "+(str(i+1))+" watch?\n"))
        freq[hrs] = freq[hrs] + 1

    print("Hours\tNumber of People")
    for i in range(24):
        print(i,'\t',freq[i])

main()
```

*Using lists to store letter frequencies*

Imagine we wanted to store the number of times each letter appeared in a message. To simplify our task, let's assume all of the letters are alphabetic letters. Naturally, it seems that a frequency list of size 26 should be able to help us. We want 26 counters, each initially set to zero. Then, for each letter in the message, we can simply update the appropriate counter. It makes the most sense for index 0 to store the number of a's, index 1 to store the number of b's, and so on.

Internally, characters are stored as numbers, known as their Ascii values. The Ascii value of ‘a’ happens to be 97, but it’s not important to memorize this. It’s only important because solving this problem involves having to convert back and forth from letters and their associated integer values from 0 to 25, inclusive. There are two functions that will help in this task:

Given a letter, the ord function converts the letter to its corresponding Ascii value.
Given a number, the chr function converts an Ascii value to its corresponding character.

This short IDLE transcript should illustrate how both functions work:

```
>>> ord('a')
97
>>> ord('c')
99
>>> ord('j')
106
>>> chr(102)
'f'
>>> chr(97)
'a'
>>> chr(122)
'z'
```

You’ll notice that the Ascii values of each lowercase letter are in numerical order, starting at 97. The same is true of the uppercase letters, starting at 65. This, means that if we have a lowercase letter stored in a variable ch, then we can convert it to its corresponding number in between 0 and 25 as follows:

```
ord(ch) - ord('a')
```

Similarly, given a number, num, in between 0 and 25, we can convert it to the appropriate character as follows:

```
chr(num + ord('a'))
```

In essence, ord and chr are inverse functions.

In the following program we’ll ask the user to enter a sentence with lowercase letters only and we’ll print out a frequency chart of how many times each letter appears:

```
def main():

    # Set up frequency list.
    freq = []
    for i in range(26):
        freq.append(0)

    sentence = input("Please enter a sentence.\n")

    # Go through each letter.
    for i in range(len(sentence)):

        # Screen for lower case letters only.
        if sentence[i] >= 'a' and sentence[i] <= 'z':
            num = ord(sentence[i]) - ord('a')
            freq[num] = freq[num] + 1

    # Print out the frequency chart.
    print("Letter\tFrequency")
    for i in range(len(freq)):
        print(chr(i+ord('a')),'\t',freq[i])

main()
```

## ***4.3 Sets***

*Difference between a set and a list*

Whereas a list can store the same item multiple times, a set stores just one copy of each item. Sets are standard mathematical objects with associated operations (union, intersection, difference, and symmetric difference). Python implements each of these standard operations that would take quite a few lines of code to implement from first principles.

*Motivation for using Sets*

Consider the problem of making a list of each student in either Mr. Thomas's English class or Mrs. Hernandez's Math class. Some students might be in both, but it doesn't make sense to list these students twice. This idea of taking two lists and merging them into one with one copy of any item that appears in either list is identical to the mathematical notion of taking the union between two sets, where each class is one set. Similarly, creating a list of students in both classes is the same as taking the intersection of the two sets. Making a list of each student in Mr. Thomas's class who ISN'T in Mrs. Hernandez's class is the set difference between the first class and the second class. Notice that order matters here, just as it does in regular subtraction. Finally, symmetric difference is the set of students who are in exactly one of the two classes. In different contexts it may be desirable to find the outcome of any of these four set operations, between two sets.

*Initializing a set*

We can create an empty set as follows:

```
class1 = set()
```

This creates class1 to be an empty set.

We can initialize a set with elements as follows:

```
class2 = set(["alex", "bobby", "dave", "emily"])
```

*Adding an item to a set*

We can add an item as follows:

```
class1.add("bobby")
class1.add("cheryl")
```

Note: Sets can store anything, not just strings. It's just easiest to illustrate the set methods using sets of strings.

*Standard Set Operations*

The following table shows the four key operators for sets. Assume that s and t are arbitrary sets.

| Operation | Expression |
|---|---|
| Union | s \| t |
| Intersection | s & t |
| Set Difference | s – t |
| Symmetric Difference | s ^ t |

Using our two set variables from above, we can evaluate each of these operations as follows:

```
>>> class3 = class1 | class2
>>> class4 = class1 & class2
>>> class5 = class1 - class2
>>> class6 = class2 - class1
>>> class7 = class1 ^ class2
>>> print(class3)
{'cheryl', 'dave', 'alex', 'bobby', 'emily'}
>>> print(class4)
{'bobby'}
>>> print(class5)
{'cheryl'}
>>> print(class6)
{'dave', 'alex', 'emily'}
>>> print(class7)
{'cheryl', 'dave', 'alex', 'emily'}
```

## ***4.4 Dictionaries***

*Look Up Ability*

A regular dictionary is one where the user inputs some value (a word), and receives some answer/translation (a definition). A pair of lists or a list of pairs could be used to maintain a dictionary, but in Python, a special dictionary type is included to simplify this sort of look-up ability. Whereas a list must be indexed by a 0-based integer, a dictionary is indexed with a word, or anything you want it to be indexed by! To retrieve an item, simply index it with its corresponding key.

*Initializing a Dictionary*

We create an empty dictionary as follows:

```
phonebook = {}
```

We can create an initial dictionary with entries as follows:

```
phonebook = {"Adam":5551234, "Carol":5559999}
```

Thus, in general, each item is separated with a comma, and within each item, the key (input value) comes first, followed by a colon, followed by the value (output value) to which it maps.

*Adding an Entry into a Dictionary*

We can add an entry as follows:

```
phonebook["Dave"] = 5553456
```

*Accessing a Value*

To access a value stored in a dictionary, simply index the dictionary with the appropriate key:

```
name = input("Whose phone number do you want to look up?\n")
print(name,"'s number is ",phonebook[name], sep="")
```

For example, if we enter Carol for the name, we get the following output:

```
Carol's number is 5559999
```

*Invalid Key Error*

If we attempt the following:

```
print(phonebook[“Bob”])
```

We get the following error:

```
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    phonebook["Bob"]
KeyError: 'Bob'
```

Thus, it’s important NOT to attempt to access a dictionary entry that isn’t there. Here is a segment of code that makes sure an invalid access does not occur:

```
def main():

    phonebook = {"Adam":5551234, "Carol":5559999}
    name = input("Whose phone number do you want to look up?\n")

    if name in phonebook:
        print(name,"'s number is ",phonebook[name], sep="")
    else:
        print("Sorry, I do not have a number for ",name,".",
sep="")

main()
```

This sort of error is similar to an index out of bounds error for strings and lists. We can avoid such errors by checking our indexes, or in this case, our key, before we ever use it to access our dictionary.

It’s important to note that although we just used a dictionary to link names to phone numbers in this extended example, we can use a dictionary to link any one type of object to another type of object.

*Changing a Dictionary Entry*

This can be done as expected, via the assignment operator:

```
phonebook["Adam"] = 5557654
```

When Python discovers that there's an entry for "Adam" already, it will simply replace the old corresponding value, 5551234, with this new one, 5557654.

*Deleting a Dictionary Entry*

While deletion in a regular dictionary doesn't make sense all that often because words don't usually cease to exist, in many contexts deleting items from a dictionary makes sense. For example, if we are no longer friends with someone, we may want to delete their entry from our phone book. We can accomplish this with the del command. We follow the keyword del with the name of our dictionary indexed at the entry we want to delete. This following trace through in IDLE illustrates the use of the del command:

```
>>> phonebook = {"Adam":5551234, "Carol":5559999}
>>> phonebook["Bob"] = 5556666
>>> phonebook
{'Bob': 5556666, 'Carol': 5559999, 'Adam': 5551234}
>>> phonebook["Adam"] = 5557654
>>> phonebook["Adam"]
5557654
>>> del phonebook["Adam"]
>>> phonebook
{'Bob': 5556666, 'Carol': 5559999}
```

## ***4.5 Reading Input from a File***

*Motivation*

Up until now, we've read in all of our input from the keyboard. But imagine that you wanted to read in 1000 food items and put them all in a list! It would be rather tedious to physically type in all of that information. Secondly, quite a bit of information already exists electronically in various types of files. In this section, we'll learn how to read in information from a standard text file. (Note: Many common files you may use, such as Word documents or Excel spreadsheets are NOT text documents. They have their own complicated file formats. Text files are ones you can create in simple text editors, by simply typing regular keys without any special features, such as bolding text, different fonts, etc.)

*Opening a File*

In order to open a file in Python, all you have to use is the open function. The following opens the file numbers.txt to read from:

```
myFile = open("numbers.txt", "r")
```

It's important to note that this ONLY works if the file from which you are reading is in the same directory as your python program. If it is not, then you have to provide the entire path to the file inside of the double quotes as well.

Once the file is open, Python provides for us two ways in which we can read in the contents of the file. The first method, which will probably be used a majority of the time, is to read in the file, one line at a time. The method that reads in a single line is the readline method.

Let's say that the file numbers.txt has a single integer, *n*, on the first line, indicating the number of values appear in the file, subsequently. Then, following this line, we'll have *n* lines, with one number each. For now, let's assume our goal is simply to add all of the numbers in the file (except for the first one, which is indicating the number of numbes.)

The readline function returns a string storing the contents of the whole line. Thus, in order to truly read in the value on the first line of the file, we have to call two functions as follows:

```
numValues = int(myFile.readline())
```

Python knows WHERE to read from because we called the readline method on the file object, myFile, that we had previously initialized. This instructs Python to read an entire line from the designated file. Since this is the very first readline after the file was opened, it will read in the

first line of the file. We then take that string and convert it to an integer using the int function. Then, this gets stored in the variable numValues.

A sample input file may contain the following contents:

```
5
27
16
22
25
14
```

Now that we know how to read in one integer on a line by itself, we can repeat this task appropriately to finish our program, so that it sums up all the numbers in the file and prints this number to the screen:

```
def main():

    myFile = open("numbers.txt", "r")
    numValues = int(myFile.readline())

    sum = 0
    for i in range(numValues):
        num = int(myFile.readline())
        sum = sum + num

    print("The total was ",sum,".",sep="")
    myFile.close()

main()
```

When running this program, using the sample file given above as numbers.txt, we get the following output:

```
The total was 104.
```

*Example Reading from a File with Multiple Items on One Line*

In the previous example, we were only able to read in one item per line, since we were forced to read in the whole line all at once. However, in many text files, more than one item is contained on a single line. Luckily, there's a method for strings that allows us to easily read in more than one piece of information on a line, called the split method. The split method will "split" a string into separate items (each of these items must be separated by spaces or tabs), and then return all of the items in a list.

Before we try a full program that utilizes this capability, let's look a a couple lines typed in the IDLE interpreter so we can understand the split method:

```
>>> sentence = "sally goes to the store."
>>> words = sentence.split()
>>> words
['sally', 'goes', 'to', 'the', 'store.']
>>> for i in range(len(words)):
     print("word",i+1,"is",words[i])

word 1 is sally
word 2 is goes
word 3 is to
word 4 is the
word 5 is store.
>>>
```

Thus, if we read a line from a file that has more than one item, we can simply split the string and store the contents into a list. From there, we can access each component one by one. One detail we need to worry about is that we need to know the type of each item, so that we can convert it from a string (which is how it's stored in the list) to whichever type is necessary.

Let's look at a program that reads in a file that has more than one piece of information on a line:

Consider the problem of calculating grades for a class. Typically, a teacher will have several assignments and each assignment will be worth a percentage of the course grade. For example, if the homework is worth 20%, two exams are worth 25% and the final exam is worth 30% and a student scored 90, 80, 100, and 85 on these assignments respectively, we can calculate her final grade as follows:

$$.20 \times 90\% + .25 \times 80\% + .25 \times 100\% + .30 \times 85\% = 88.5\%$$

Essentially, we multiply each grade (out of 100) by its relative weight and add the results together. The relative weights must add up to 1 (100% of the course grade).

In this problem, we'll assume a class has four grades we need to average. Our input file will tell us what percentage each grade is worth, the number of students in the class, and each student's grades. In particular, the input file format will be as follows:

The first line of the input file has a single positive integer, *n*, on it, representing the number of students in the class. The second line of the input file will contain 4 positive real numbers representing the proportion of the class grade of the four assignments. Each of these will be separated by spaces and each is guaranteed to add up to 1. The following *n* lines will each contain four positive integers in between 0 and 100, inclusive, representing the grades on the four assignments, respectively for that student.

Consider an input file with the following contents:

```
5
.6 .1 .1 .2
100 50 70 80
85 90 90 90
78 100 100 100
83 0 100 95
99 92 45 88
```

This file stores information about five students. The second student got an 85% on the first assignment, worth 60% of the course grade and 90% on the rest of her assignments. The third student, got a 78% on the assignment worth 60% of the class and got 100% on the rest of his assignments, and so on.

Our task will be to write a program that prints out the course average for each student. The output should include one line per student with the following format:

```
Student #k: A
```

where k is the number of the student, starting with 1, and A is their average in the class.

Let's break our task down into smaller pieces:

1) Read in the number of students.
2) Read in the weights of each assignment and store into a list (of floats).
3) Go through each student
    a) Read in all of the grades and store into a list (of ints).
    b) Create a variable to store their grade and set it to zero.
    c) Go through each grade:
        i) Multiply this grade by the corresponding weight and add to the grade.
    d) Print out this student's average.

For the purposes of his program, we will read the input from the file grades.txt. Now that we've planned our program, let's take a look at it.

```
def main():

    myFile = open("grades.txt", "r")

    # Read in the number of students and weights.
    numStudents = int(myFile.readline())
    weights = myFile.readline().split()

    # Go through each student.
    for i in range(numStudents):

        # Store their grades.
        allgrades = myFile.readline().split()

        # Add up each grade's contribution.
        grade = 0
        for j in range(len(weights)):
            grade = grade + float(weights[j])*int(allgrades[j])

        print("Student #",i+1,": ","%.2f"%grade, sep="")

    myFile.close()

main()
```

Note that since the lists store strings, we had to convert each list item to a float or int, respectively to properly carry out the desired calculation.

## 4.6 ***Writing to a File***

*Motivation*

While it's often adequate to view information scrolling in the IDLE window, sometimes it would be nice to save that information in a file. Thus, in some cases it becomes desirable to store information that a program processes directly into a file. One simple example of the use of an output file would be to store high scores for a video game, for example.

*Opening and Closing a File*

This works very, very similar to opening and closing a file for reading. The only difference is in the opening step, where you must indicate that the file be opened for writing mode by making "w" the second parameter to the open function:

```
outFile = open("results.txt", "w")
```

When a file is opened for writing, if it previously existed, its contents are emptied out and anything the program writes to the file will be placed at the beginning of the file. In essence, opening a file for writing erases its previous contents.

We close an output file in the identical manner to which we close an input file:

```
outFile.close()
```

*Writing to a File*

Python keeps writing to a file simple by providing a single method that writes a string to a file:

```
file.write(<string>)
```

The trick in using this method is to only provide a single string to print to the file each time. Also realize that while newlines are automatically inserted between prints to the IDLE screen, they are NOT inserted into a file between each write method. Thus, the programmer has to manually indicate each newline that is printed in the file. The standard way to accomplish writing various items into a file is to use string concatenation (+) frequently. This is similar to how we can get varied prompts for the input method.

*Grades Example Extended*

We will now extend the grades example so that it produces output to the file "results.txt". Very few changes have been added between the old version and this one:

```
def main():

    myFile = open("grades.txt", "r")
    outFile = open("results.txt", "w")

    # Read in the number of students and weights.
    numStudents = int(myFile.readline())
    weights = myFile.readline().split()

    # Go through each student.
    for i in range(numStudents):

        # Store their grades.
        allgrades = myFile.readline().split()

        # Add up each grade's contribution.
        grade = 0
        for j in range(len(weights)):
            grade = grade + float(weights[j])*int(allgrades[j])

        # Output the result.
        outFile.write("Student #"+(str(i+1))+": "+"%.2f"%grade+"\n")

    myFile.close()
    outFile.close()

main()
```

In fact, except for opening and closing the file, the key change was creating a single string as input for the write method. The most difficult portion of this was remembering to convert the integer (i+1) to a string using the str function. Everything else is identical to the previous version of the program. When this program is executed, assuming that grades.txt is in the same directory as the python program itself, results.txt will appear in the same directory with the appropriate contents. The corresponding output for the input file provided previously is:

```
Student #1: 88.00
Student #2: 87.00
Student #3: 86.80
Student #4: 78.80
Student #5: 90.70
```

## *4.7 Problems*

1) Write a program that prompts the user to enter two strings and prints out the match score to indicate how similar the words are. If the two strings are different lengths, then you should print a score of 0. Otherwise, if the two strings are the same length, the score printed should be the number of characters in corresponding locations that are the same. For example, the match score between "home" and "host" should be 2 and the match score between "paper" and "caper" should be 4.

2) Write a program that asks the user to enter 10 words and prints out the word that comes first alphabetically.

3) Write a program that simulates using a Magic Eight Ball. Your program should have a list of pre-programmed responses to questions stored in a list of strings. Ask the user to enter a question and the present the user with a randomly chosen response from the list you have created.

4) Write a program that reads in a text file of test scores and prints out a histogram of the scores. The file format is as follows: the first line of the file contains a single positive integer, *n*, representing the number of test scores. The following *n* lines will contain one test score each. Each of these test scores will be a non-negative integer less than or equal to 100. The histogram should have one row on it for each distinct test score in the file, followed by one star for each test score of that value. For example, if the tests scores in the file were 55, 80, 80, 95, 95, 95 and 98, the output to the screen should look like:

```
55*
80**
95***
98*
```

5) Write a simulation where the user collects packs of baseball cards. Each pack contains a set of 10 cards, where each card is numbered from 1 to 100. In the simulation, have the user continue to buy packs of cards until she has collected all 100 distinct cards. Each set must contain distinct cards, but two different sets may contain the same card.

6) Generate a set containing each positive integer less than 1000 divisible by 15 and a second set containing each positive integer less than 1000 divisible by 21. Create a set of integers that is divisible by either value, both values and exactly one value. Print out the contents of each of these resultant sets.

7) Write a program that allows the user to add telephone book entries, delete people from a telephone book, allows the user to change the number of an entry, and allows the user to look up a person's phone number. Put in the appropriate error checking.

8) Write a program that asks the user to enter a list of censored words, along with their approved replacements. (For example, "jerk" might be replaced with "mean person".) Read in a sentence from the user and write a modified sentence to the screen where each censored word in the sentence was replaced with the approved replacement. In the given example, the sentence "he is such a jerk" would be converted to "he is such a mean person".

9) Rewrite program #8 so that it reads in the list of censored and replacement words from the file "censordictionary.txt", reads the sentence to convert from the input file "message.txt" and outputs the converted message to the file "safemessage.txt". Create appropriate file formats for all three files.

10) The following problem is taken from a programming contest. The input for the problem is to be read from the file "idnum.in" and the output is to be printed to the screen. The exact file format is given as well as some sample input and the corresponding output for those cases.

**The Problem**

There are many new summer camps starting up at UCF. As new departments try to start up their summer camps, word got around that there was a computer science summer camp at UCF that was already established. One of the tools that these other summer camps need is a tool to create identification numbers for all the campers. These summer camps have kindly asked Arup to create a computer program to automate the process of allocating identification numbers. Naturally, Arup has decided that this would be an excellent exercise for his BHCSIers. For each summer camp in question, identification numbers will be given in the order that students sign up for the camp. Each camp will have a minimum number for which to start their student identification numbers. Each subsequent number will be generated by adding 11 to the previously assigned number so that each number is sufficiently spaced from the others. After assigning all the identification numbers, your program will need to print an alphabetized list of each student paired with his/her identification number.

**The Input**

The first line of the input file will consist of a single integer *n* representing how summer camps for which you are assigning identification numbers. For each summer camp, the first line of input will contain one integer $k\ (0 < k \leq 200)$, representing the number of students in that summer camp. The second line of input for each summer camp will contain a single positive integer, *minID*, which represents the minimum identification number for all the students in the camp. (This is the number that will be given to the first student to sign up for the camp.) The following *k* lines will each contain a single name consisting of only 1 to 19 upper case letters. These names are the names of all the students in the class, in the order in which they signed up for the camp. The names within a single summer camp are guaranteed to be unique.

**The Output**

For every summer camp, the first line will be of the following format:

Summer camp #m:

where m is the number of the summer camp starting with 1.

The following *k* lines should list each student in the summer camp in alphabetical order and his/her identification number, separated by a space.

Put a blank line of output between the output for each summer camp.

**Sample Input**

```
2
8
2000
SARAH
LISA
ARUP
DAN
JOHN
ALEX
CONNER
BRIAN
10
100001
JACK
ISABELLA
HAROLD
GARY
FRAN
EMILY
DANIELLE
CAROL
BOB
ADAM
```

**Sample Output**

```
Summer camp #1:
ALEX 2055
ARUP 2022
BRIAN 2077
CONNER 2066
DAN 2033
JOHN 2044
LISA 2011
SARAH 2000

Summer camp #2:
ADAM 100100
BOB 100089
CAROL 100078
DANIELLE 100067
EMILY 100056
FRAN 100045
GARY 100034
HAROLD 100023
ISABELLA 100012
JACK 100001
```