LISTS

After int, float, boolean and str, we are now ready to learn a fifth data type: the list.

Lists are great! The list is probably the most versatile data type in Python. The major benefit of the list is that it allows us to store many values and associate them with the same variable. For instance, suppose your program needed to keep track of 30 numbers. Does it make sense to store them in 30 different variables? We would have to come up with distinct names for these 30 variables. Instead, we can put all these numbers into a single list. The magic of the list is that the basic representation and syntax is like a string.

For example, we can create a variable called data, and store the numbers 5, 4, 7 and 3 inside it like this:

```
data = [ 5, 4, 7, 3]
```

There are two things to note here. First, individual elements of a list are separated from the others by commas. Second, the brackets are used to indicate the beginning and end of the list. So, the bracket symbols are doing double duty, because as you saw with strings earlier we also use brackets to reference individual pieces of information.

After having initialized our list, we can refer to the individual numbers just like we did for strings.

data[0] is 5
data[1] is 4
data[2] is 7
data[3] is 3

And just like strings, we can use the [ ] notation to grab various parts of a list. Suppose L is a list.

- Negative indices allow us to count from the right end. L[–1] refers to the last item in a list.
- Slices to refer to part of a list. L[2:3] refers to the third and fourth elements, and this expression can itself be treated as a list.
- Slices with a third argument indicate a stride. L[5:50:10] means the part of the list containing just the items at indices 5, 15, 25, 35, and 45.
- Slices with arguments omitted. L[–5:] means we want just the last 5 elements from L.

There are two ways to create a list in a Python program. The first way is to “hard code” it into the source code, as you just saw above. This is very simple, but it suffers the disadvantage that if you wanted to initialize your list to contain different data instead, you would have to re-edit the program before running it again.

The second method for initializing the list is to start with an empty list and append data into it. I much prefer this method, because I can test my program on any data I want without having to change my program. Here is what you need to do to initialize a list this way:

- Begin with an empty list. For example, with the statement L = [ ]
- Use the built-in append( ) function to add one element at a time to the list.
- Use a loop to append multiple values, and obtain the list values from input.

Here is an example of what this looks like.

```
# We begin by asking the user for how many values.
size = int(input("How many values for list?  "))

L = [ ]
for i in range(0, size):
    L.append(int(input("Enter a value:  ")))

# Let's output the list
for item in L:
    print(item)
```

Here is another possible approach. Sometimes, the user does not know the size of the list in advance. In this case, the user can signal the end of the input with a special "sentinel" value such as –1, assuming that –1 is not a legitimate input value for the list.

```
L = [ ]

while True:
    value = int(input("Enter a value, -1 to quit:  "))
    if value == -1:
        break
    L.append(value)

for item in L:
    print(item)
```

Discussion:

There are two ways to print out the elements of a list. Experiment with both, shown below, and describe the difference in the output.

```
for item in L:
    print(item)
```

```
print(L)
```

Important list functions

Suppose that L is a list. Here are some things we may want to do with the list. Usually, our list contains numbers. There are functions to perform basic statistics. And then there are functions that make useful modifications to the list. You have seen the append( ) function already.

Function	Meaning
len(L)	Returns the number of values in the list
max(L)	Returns the highest number in the list
min(L)	Returns the lowest number in the list
sum(L)	Returns the sum of the numbers in the list
L.append(value)	Inserts the new value at the right end of the list
L.insert(index, value)	Places the value at a certain index of the list. The existing element at that index and everyone to its right get shifted over to the right to make room for the new value.
L.remove(x)	Removes the first object having the value x. Items to its right shift to the left.
L.reverse( )	Changes L to its reversal
L.sort( )	Changes L to its ascending order
L.index(value)	Returns the first location of a value in the list. For example, if L = [5, 4, 7, 4], then L.index(4) is 1. A run-time error results if the value doesn't exist.

Note that most of the functions listed above only make sense if the elements of L are all of the same type. Numbers can be compared with other numbers, strings can be compared with other strings, but numbers and strings cannot be compared. However, in practice our lists will generally contain values all of the same type, as in a list of only numbers or only of strings.

We also need to be careful concerning the proper way to copy lists. This is one way in which lists do not quite behave like other variables. For example, if a is an integer variable, and you wanted to copy this value into a new variable b, this is easy: you would enter the statement b = a.

However, this technique does not work for lists. If A and B are lists, then B = A has a special meaning. It means that we want B to refer to the same list as A. The statement B = A does not copy anything. Instead the same list now has two different names (aliases). This is not what we want!

Here is the correct way to copy a list A into a new list B. This technique should remind you of how we initialized lists from input. First, create a new empty list B. Then, traverse the list A: for each value in A, we append this value onto the end of B. Here is the code:

```
B = [ ]
for value in A:
    B.append(value)
```

To review the meaning of =, let's look at a simple example involving integers.

```
a = 1
b = a
a = 2
```

In this example, the second statement places the value 1 (the current value of a) into b. The final value of a is 2, and the final value of b is 1.

Let's repeat this example using lists. Comments beside each statement explain the effect.

```
A = [1]     # Create a new list with the number 1 in it; call it A
B = A       # Make B refer to the same list.
A = [2]     # Create a new list with just 2 in it; call it A
```

In this example, the final value of A is [2] and the final value of B is [1]. The third statement creates an entirely new list and gives it to the variable A. The original list [1] is still referred to by B.

Finally, consider this example:

```
A = [1]     # Create a new list with 1 in it; call it A
B = A       # Make B refer to the same list.
A.append(2)   # Append 2 to the list referred to by A and B.
```

In this case, there is only one list! Both A and B refer to the same list, [1, 2].

Now, let's practice with lists. Assume that L is a list of numbers. Explain or show how to find …

1. How many values are positive
2. The sum of just the positive values
3. The last value
4. The largest value
5. The second largest value … Can you do it without changing the list?
6. The median

7. The location of the first zero

8. The location of the second zero

It's important to note that the individual elements of a list do not have to be numbers. They can be anything, including strings. For example, the following code creates a list of strings and prints each string out, one per line.

```
food = ["hamburger", "hot dog", "pizza"]
for s in food:
    print(s)
```

Discussion: How can we modify the above loop so that it counts how many times the letter 'g' appears in the list of strings?

An interesting problem that we can solve later has to do with a heterogeneous list, for example a list that has both numbers and strings. We can skip the strings when doing arithmetic on the list.

<u>The enumerate function</u>

For a list, the purpose of the enumerate function is to allow us to visit all the elements of a list, and be able to see both the value *as well as its index*.

Usually when we traverse a list, we are ignorant of the index. Here is an example.

```
L = [ 5, 7, 4, 2, 3, 8, 1 ]
for value in L:
    if value % 2 == 1:
        print("I found the odd number ", value)
```

The above code will find all of the odd numbers in the list. But where do these numbers live in the list? In other words, what are their indices or apartment numbers? This is why we need the enumerate function. Our code then becomes:

```
L = [ 5, 7, 4, 2, 3, 8, 1 ]
for index, value in enumerate(L):
    if value % 2 == 1:
        print("I found the odd number", value, "at index", index)
```

By using enumerate( ), we simultaneously know both the location and the contents of each number in the list. Had we not used enumerate( ), we would not know the locations.

Exercise: Modify the above code so that it also:

1. Finds the sum of all the odd numbers in the list.

2. Places all the odd numbers into a new list.

3. Places the indices of the odd numbers into a third list.

4. If the original list is L = [5, 7, 4, 2, 3, 8, 1], then what values are contained in the two new lists?

5. In general, how would you find the identity and location of the largest number in a list?

"Fixed" size lists and multidimensional lists

In Python, lists may grow and shrink as you want. But what if you know exactly how big you want the list to be in advance? For example, many games have a fixed board size. Python allows you to use the + and * operators to quickly create a list. If you want to use a list of a fixed size, this is easy: create the list to immediately have this size, and then be sure not to change its size when it is later used.

Python also allows us to create a multidimensional list. For example, a 2-D list would be created as a list of lists. And a 3-D list would be a list of lists of lists. Multidimensional lists with more than 2 dimensions are not commonly used. Let's focus on 2-D lists.

Consider this list: L = [[1, 2] , [3, 4, 5]]. How would we access the various numbers in this list? It may help to redraw this list in 2-dimensions:

```
L = [ [ 1, 2 ],
      [ 3, 4, 5 ] ]
```

Here, L[0] refers to the first "row" of the list, and L[1] is the second row. Thus, L[0] = [1, 2] and L[1] is [3, 4, 5]. We use two sets of brackets in order to access a single number. Since L[0] is the list [1, 2], it is logical to expect that L[0][0] means the first element of L[0], which is 1. Similarly, L[0][1] = 2. And we can continue: L[1][0] = 3, L[1][1] = 4 and L[1][2] = 5.

In practice, 2-dimensional arrays are usually rectangular. That is, each "row" has the same length. And speaking of length, we can use the len() function of lists to inquire as to the number of rows and columns in a rectangular 2-dimensional array. In this case, len(L) is the number of rows. And len(L[0]) is the number of columns, because it literally means how many elements are in the first row.

The following table shows how 1-d and 2-d lists can be created quickly from scratch in Python:

Expression	Result
[0] + [0]	[0, 0]
[0] * 8	[0, 0, 0, 0, 0, 0, 0, 0]
[1, 2] * 5	[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
[[0]] * 8	[[0], [0], [0], [0], [0], [0], [0], [0]]
[[0] * 8] * 8	[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]

In all honesty, Python is not the best language to use if you have an application that demands multidimensional data. For this purpose I would recommend another language such as Java.

Once a list is created it is not hard to initialize the elements to something other than zero. We can simply assign a value to the cells we want to change.

Here is an example of using a fixed-size list. Suppose we wanted to calculate the frequency distribution of digits among the first several digits of pi. In this case, we want to know how many times each digit occurs in a string. There are 10 digits, so we know that we want to maintain a list of 10 values, all initially zero. For each digit of pi, we increment the appropriate cell in the list. In the following program, dist is the name of the list holding the frequency distribution of digits, and pi is the string of digits.

```
# pi.py – Distribution of digits among some digits of pi.
# Note that the pi string has one character that is not a digit.

dist = [ 0 ] * 10
pi = "3.14159265358979323846264338327950288419716939937510582097494459"
numDigits = len(pi) – 1
for digit in pi:
    if digit != '.':
        dist[int(digit)] += 1
print("Distribution among the first {0:d} digits:".format(numDigits))
print(dist)
```

When we run this program, here is the output. How would we interpret these results?

```
Distribution among the first 62 digits:
[3, 5, 6, 9, 7, 7, 4, 5, 6, 10]
```

Discussion: How would we modify this program to calculate the frequency distribution of letters of a string of text (words)?

Tokenizing

Often in programming we have a string that contains several of pieces of data inside. We would like to have a convenient way to extract this information. For example, earlier we saw an exercise where we had to grab the number of minutes and seconds from a string that looked like this: "6:17". The two numbers were delimited by the colon. We found the location of the colon, and then created string slices around this colon to find the minutes and seconds. This technique works, but it becomes tedious in longer strings that have more data.

Consider this example. Suppose we have a string containing numbers separated by commas, such as the string "5,12,8,3,50". We could adapt the approach we did before to obtain all the numbers as follows:

- Create a list called commaLocation that contains the indices of all of its commas. The algorithm should work no matter how many numbers there are.
- Next, create a list L that consists of the numbers inside s that are separated by the commas. Use the values in commaLocation to help you create string slices.
- Finally, put each number in a list L.

This procedure seems needlessly complex. Fortunately, Python has a simpler method that we can use. It makes use of a construct called a list comprehension. We will not cover list comprehensions in general in this course, but they are useful for the sake of tokenizing. Continuing with our example, here is the pleasant way to solve the problem. In this case, let `line` be the name of the string variable containing "5,12,8,3,50", where the delimiter symbol is just the comma:

At the beginning of the program include this statement `import re`. The "re" stands for "regular expression."

Type these three statements:

```
tok = re.compile(",")
tokenlist = tok.split(line)
tokenlist = [ s for s in tokenlist if len(s) > 0 ]
```

You should not memorize these 3 statements. Let's see what they do.

The first statement creates a tokenizing object that will be on the lookout for delimiter characters, in this case just commas.

The second statement says to split the string called `line` into a list of token strings, using the comma as the delimiter. Everything that appears between commas will become a token.

The third statement eliminates blank strings from the list of tokens. In this example, it turns out there would not be any blank strings because we don't have two consecutive commas. But I highly recommend you include this statement anyway.

Now, the variable `tokenlist` is this list of strings: ["5", "12", "8", "3", "50"]. We can use the int() function on each element of the list in order to perform arithmetic if we wish.

If you have more than one possible delimiter, such as both commas and spaces, then you would include all such symbols inside the call to the function `re.compile()`.

Let's look at another example.

```
# tokenize1.py - Let's look at another simple case of
# splitting a string into tokens.

import re

line = "please...give..us some......cookies!!!! thank...you"

# These two statements are the magic.  They could be
# combined into a single statement if you wish.  First,
# we create a tokenizer object, then use it to split the string.
tok = re.compile("[.! ]")
tokenlist = tok.split(line)

print ("\nTokenizing the string gives us:")
print (tokenlist)

# Now, the problem we have is that our list of tokens
# includes a lot of empty strings.  Let's remove them
# from the list:  use a list comprehension with an 'if'.

tokenlist = [s for s in tokenlist if len(s) > 0]

print ("\nAfter removing empty tokens:")
print (tokenlist)
```

Here is the output, so that you can see the removal of the empty tokens by the list comprehension:

```
Tokenizing the string gives us:
['please', '', '', 'give', '', 'us', 'some', '', '', '', '', '',
'cookies', '', '', '', '', 'thank', '', '', 'you']

After removing empty tokens:
['please', 'give', 'us', 'some', 'cookies', 'thank', 'you']
```

Discussion: Suppose that the variable line contains a line of text (words). How would you determine:

1. The number of words?

2. The longest word? Note that there could be a tie for the longest word. In this case, print all such words of maximum length.

3. If two consecutive words are the same?

4. How many words begin with a capital letter?

<u>Practice with lists</u>

In the following problems, it is more important to know the algorithm than to type in all the necessary code.

1. Given a list of numbers, print the locations (i.e. indices) of the numbers that are multiples of 5. Also tell the user how many numbers it found.

2. The Fibonacci sequence is this list of numbers: 1, 1, 2, 3, 5, 8, 13, 21, etc. The first two numbers are 1 and 1, and each subsequent number in the list is the sum of the previous two. Create a list containing the first 20 Fibonacci numbers.

3. You are given a list of 10 integers. Go through the list and see if it contains any values divisible by three. If so, print out all such values, along with their locations in the list.

4. You are given a list of 5 numbers. Determine the second largest value in the list. Don't change the list.

5. You are given a list of 5 numbers. Determine if this list is sorted in ascending order.

6. You are given a list of numbers. Find the smallest positive number in the list, as well as its location.

7. You are given a list of numbers. Determine how many values are unique. For example, the list of numbers [1, 7, 4, 1, 4] has three unique values.

8. Ask the user to enter a sequence of numbers. Store these numbers in a list. Stop reading input once you encounter the same number entered twice in a row.

9. Ask the user to enter a sequence of numbers. Store these numbers in a list. Stop reading input once you encounter a number that has already been entered.

10. From a deck of cards, deal yourself 7 cards. Determine how many of these cards are face cards (Jack, Queen, King).

11. Given a poker hand, determine if it's a full house.

12. Suppose s is a string containing 3 numbers separated by colons. It represents an amount of time given in hours, minutes and seconds. Tokenize this string to compute the total number of seconds. For example, "13:53:20" represents 50,000 seconds.

13. The following code attempts to determine if a list contains an odd number. What is wrong with the following code, and how would you fix it?

```
list = [ 8, 5, 2, 8, 3, 6, 1, 9, 4 ]

for number in list:
    if number % 2 == 1:
        found = True
    else:
        found = False
```