CSC110 Supplemental Reading: Week9

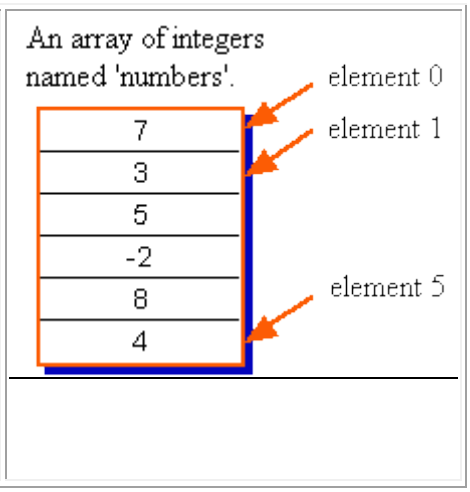
Contents

1 Introduction to Lists

The largest program you have written so far has probably used 6 - 12 different variables. Separate variables are good for these kinds of programs, but we may want to be able to store and access hundreds or thousands of values in larger programs. For this we need a data structure. A data structure is an organized collection of values.

A **list** is a powerful data structure that allows ***any number of related values to be stored together at the same time using one name***. (Note: I'll be using the term list and array interchangeably). Furthermore, a list is easy to use. The following figure shows a simple code example and a diagram illustrating how the information is stored:

```
# declare an array and initialize it with data
numbers = [ 7, 3, 5, -2, 8, 4]

// display
print ( numbers[ 0] )
print ( len(numbers) )
print ( numbers[ 1 ] )

# Test results:
7
6
3
```



Notice several things about this code:

- **A list can be pre-loaded with values** by including a list of values, in square brackets.
- Each value stored in a list is called an **element**.
- Like a string, each value stored in a list is identified by an **index number**. The first index number is zero (0), and they count up from there. This is also sometimes referred to as a **subscript**.
- Like with a string, an individual **element** of a list is accessed by using the name of the array followed by square brackets [ ] that contain the index number of the element. This expression: **numbers[2]** is the syntax to access the element at index 2. It can be read "numbers **sub** 2", meaning "the element in the array numbers at index 2".
- An individual list element (like **numbers[2]**) is just like any other variable that holds 1 value. It has a data type; you can use it in any expression; you can pass it as an argument to a function; and you can store a value into it.
- One of the more confusing points about lists is this: **don't confuse the index number with what is stored in the list**. For example, **numbers[5]** contains the value 4. **5** is the index number, **4** is the contents of the element.
- Remember: list indexes start at 0: the element at index 5 is actually the 6th element.
- You can use the len() function to find the length of a list, just like with strings.

1.1 Declaring a List

As you saw in the example above, you can declare a list and store information in it all in 1 statement. Here is another example:

```
places = ["London", "Rio De Janeiro", "New York"]
```

Quick review: what is the value of len(places)? (answer: 3) What is the index of "Rio De Janeiro"? (answer: 1)
Notice that the current length of the list is automatically determined by how many values are stored.

You can also declare an empty list and not initialize it with any particular value. For example:

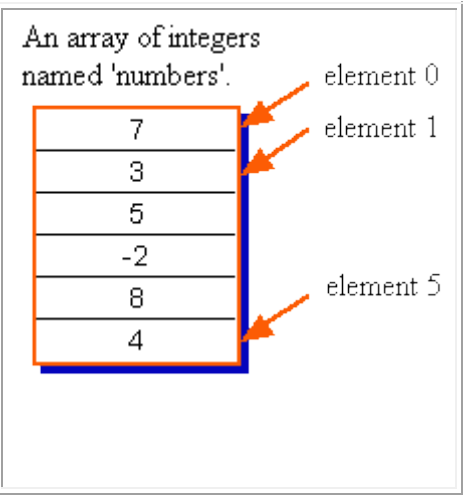
```
grades = [ ]
```

Your program would be able to add data in there at some later point. As the program adds data to the array, it will automatically expand in size.

1.2 List Processing

Since a list is a sequence, it is very easy to visit each element using a for loop.

```
numbers = [ 7, 3, 5, -2, 8, 4 ]

for value in numbers:

print ( value )

# Test results:
7
3
5
-2
8
4
```



However, we've seen that using the for loop can be limiting. What if we wanted to print the list in reverse order? Then, we must use the indexes, starting at the top index and making our way down to 0. For this we'll use a while loop:

```
numbers = [ 7, 3, 5, -2, 8, 4 ]

index = len(numbers) - 1 # why -1?
while index >= 0:

print ( numbers[index] )
index -= 1

# Test results:
4
8
-2
5
3
7
```

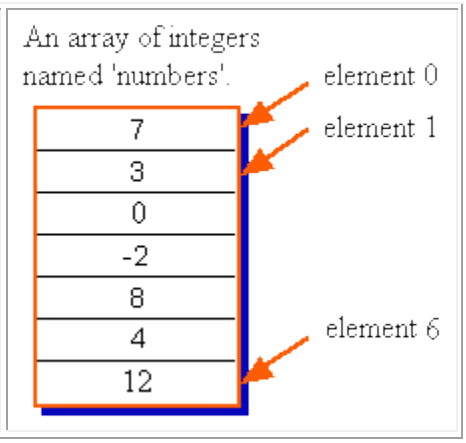
1.3 Adding/Changing the contents of a List

One big difference between a string and a list is that the data contained in a list may be changed in the following ways:

- The values of the individual list elements can be changed by using a list element on the left-hand side in an assignment statement; treat it just as if it were a regular variable.
- New elements may be added at the end of an array by using the **append() method**.

Here is an example:

```
numbers = [ 7, 3, 5, -2, 8, 4 ]
numbers[ 2 ] = 0
numbers.append( 12 )

# The final form of the array is shown at right...
```



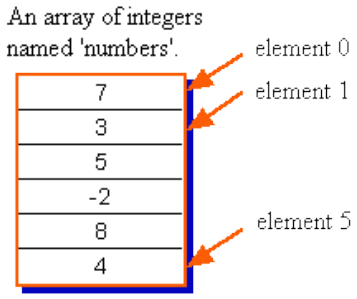
Notice the following things about this example:

- The second line in the program changes the element at index 2 to a value of 0 (again, be aware of index of an element vs. content of the element). This is just a normal assignment statement, except that the value on the right-hand side of the equal sign is being assigned to a particular element of the list, specified by providing a subscript. As we've seen, the subscript could be a variable instead of a literal number.
- The third line in the code adds a new element at the end of the array. The **append() method** is used, and the value of the new element is provided as an argument in the method call. So room is made in the list and the value 12 is stored. The new element is at index 6, and the length of the list is now 7.
- Any number of elements may be added to a list. The length of the list will expand automatically. **Careful**: this expandability feature is something specific to Python; you many not find this in other programming languages.

1.4 Sums and Averages

The ability to visit each element is very powerful. Consider the following example that calculates the sum and average of the elements in the array:

```
numbers = [ 7, 3, 5, -2, 8, 4]

# This code is to sum up numbers, so we need an accumulator variable
sum = 0

# the loop to visit every element in the array and add it to the accumulator
for num in numbers:
    sum += num # update accumulator

print ( 'The sum is " , sum  )
print  ("The average is " , float(sum) / len(numbers )  )
```



Test results:
The sum is 25
The average is 4.166666666666667

We say that with a loop, we "visit" every element in an array. The only thing that changes from one loop example to another is what to do in the body. In other words, what to do with an individual element. For example, what's in the body of the loop if we want to display an element? What if we want to sum up the elements? Or we want to test or change the element? All that code is in the body of the loop. The structure of the loop (initialization, test, update) are usually always the same. Look back over the loops on this page and you'll see they are all designed the same.

1.5 Summary

This has been just an introduction to the topic of lists. Much more could be added. The Python list has many other methods; some you'll read about in the book. Other programming algorithms involving lists are covered in the next notes. The key points of this exposure are:

- Lists are powerful data structures that allow many values to be stored and referenced by a single variable at the same time.
- Each value stored in a list is called an **element** of the list.
- A numerical **index** (or **subscript**) is used to identify each element. The indexing begins with the number zero.
- The number of elements in an array may be determined by calling the len() function.
- Loops are almost always used when working with lists. In the typical situation, **a loop is used to iterate through all elements** in the list.

Lists typically hold **a collection of related values, each of which represents the same kind of thing**. When this is true, many problem-solving techniques can be employed using the contents of the list, such as computing sums and averages. Other list algorithms , such as finding highest and lowest values, searching and sifting through the elements in the list, and sorting the data we will cover this week and next week.

2 Lists and Functions

2.1 Lists Elements as Arguments

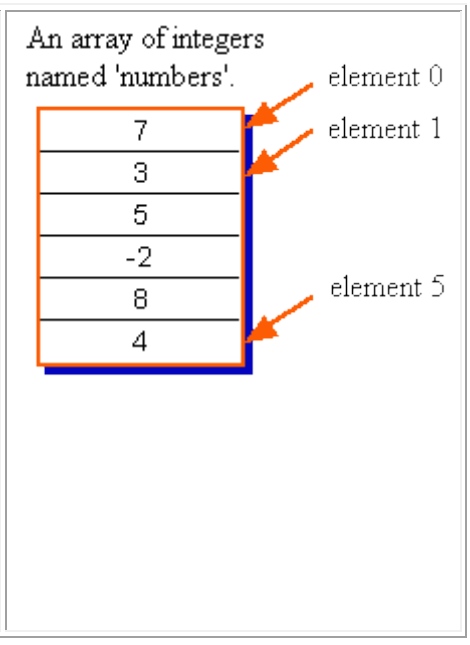
As mentioned previously, the elements of a list can be thought of as variables, just like the ones we are used to. We can store data into them, use them in expressions, and pass them as arguments to functions. Here are some examples:

```
import math
scores = [100, 85, 88, 99, 100]

scores[3] = scores[2] / 4 # this would store 22 at index 3, replacing the 99

x = math.sqrt(scores[0]) # this would store 10 into x

if scores[1] >= scores[4] - 30: # true or false?
    print  ( "here " , scores[1] )
else :
    print ( "no here" , scores[4]  )

y = math.ceil( scores[3] / 2)
```

2.2 Whole Lists as Arguments

An entire list variable may be passed as an argument to a function. Recall the print algorithm you saw in the array introduction notes. Let's modify it to make use of a function:

```
def showList( arr ):
for val in arr:
print ( val )
def main():
numbers = [7, 3, 5, -2, 8, 4]
showList( numbers )
main()


# Test results:
7
3
5
-2
8
4
```



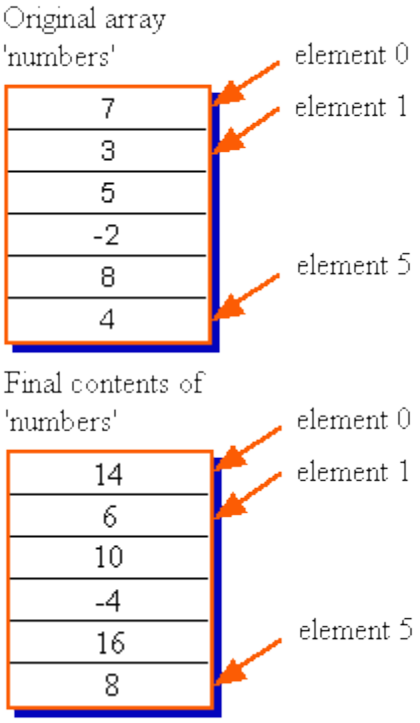
Notice the following things about this example:

- The function definition for showList() has one **parameter** named **arr**. This parameter will refer to a list, and is used inside the function like any other list variable. Remember that **arr** is a **local variable** -- it has no meaning outside the function.
- The last line of main is the **function call** to showList(), and includes one **argument**: **numbers**. This ***list is passed into the function*** so that the values of its elements may be printed.
- The beauty of this function is that it can be called with any list of any length holding any data type.

When a list variable is used as an argument in a function call, the parameter in the function definition becomes a **reference**, or **alias**, for the list argument. It is just another name for the exact same list object. This is called **passing by reference**. So, while the function showList() is executing, both **numbers** and **arr** are names for the **same list in memory**. When the function terminates, the name **arr** is gone, but the list still exists, only with the name **numbers**.

One consequence of this is that **a function can modify the contents of a list argument** just by changing the elements of the list parameter. Because only one copy of the list exists, any changes made by the function will be reflected in the original variable. Here is an example:

```
def showList( arr ):
for var in arr:
    print ( var )

def doubleData( arr ):
for i in range(len(arr)):
    arr[ i ] *= 2

def main():
    numbers = [7, 3, 5, -2, 8, 4 ]
    doubleData( numbers )
    showList( numbers )

main()

# Test results:
14
6
10
-4
16
8
```

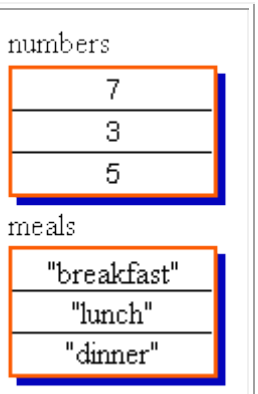


The function doubleData() multiplies every element of the array by 2. Understand that even though the syntax shows the list name **arr**, it is really the list **numbers** that is being changed: **numbers** is the only list there is; **arr** (the parameter) is an alias.

Notice that the loops in showList and doubleData are different. How come we couldn't use a loop like **for var in arr:** in doubleData? Because inside the loop, the variable that we would be changing is var. That's a different variable than the cell in the list. Since we want to actually change the list, we must use the index values and access the locations in the list using the list syntax.

So far, most of our examples have involved numbers. However, ***any kind of data may be stored in a list***. Let's modify a previous example to include a list of strings.

```
def showList( arr ):
for var in arr:
    print ( var )

def main():
    numbers = [ 7, 3, 5 ]
    meals = [ "breakfast", "lunch", "dinner"]
    showList( numbers )
    showList( meals )

main()


#Test results :
7
3
5
breakfast
lunch
dinner
```



Notice the following things about this example:

- The showList() function works equally well with lists of numbers or strings.
- The showList() function can be called with different lists. **Each time the function is called, the parameter** (**arr**) **becomes an alias for the argument** (**numbers** or **meals**), so that the function can operate on the particular list specified by the call.
- The showList() function works with lists of any length.

3 Common List Algorithms

A list holds a collection of data. Typically, a list holds a collection of related values, each of which represents the same kind of thing. For example, we might work with a set of test scores, or a group of names. If the data stored in the list are chosen carefully, then it is possible to perform many useful operations on the data. One example has been shown already -- finding the average of all the numbers in the list . Here are more examples.

3.1 Finding the largest value

Imagine a list contains a series of interest rates offered by banks, and you need to write the code that finds the highest interest rate. First, think about how you would solve this problem by hand (in other words, the algorithm). Imagine that instead of a list, you have a stack of papers, each showing one interest rate. How would you find out what the highest rate is?
I'm sure you could do it even if you were allowed to look at each paper only once. All you need to do is remember one number -- the highest rate you have seen so far. If you see a rate that is higher, then remember the new number (and forget the previous one). This one number you need to remember corresponds to a variable in your program.

The process of looking at each piece of paper one time corresponds to the loop you will use in the program to solve the problem. Remember, you use a loop to visit every element in a list, one at a time. Since we don't have to change the list, using a for loop makes sense. Here is a code that implements this solution:

```
def findHighest( nums ):
highest = nums[0]
for val in nums:
    if val > highest:
        highest = val
return highest

def main():
interestRates = [ 1.5, 0.8, 3.5, 2.0, 1.8, 1.9 ]
ans = findHighest( interestRates )
print ( "Highest rate is ", ans , "%" )

main()
```

The display would be:
Highest rate is 3.5%

Notice the following things about this example:

- The variable highest is used to "remember" the largest value seen. It is initialized to the value of the first element in the list before the loop begins.
- Each element in the array is checked inside the loop. We could also design the loop to skip the first value; then we would have to work with index numbers, starting with 1.
- A decision is made inside the loop -- the value of highest is changed only if the list element being checked is actually higher than the highest value remembered.
- Since this process has been implemented inside a function, a return statement sends the highest value back to the calling statement.

3.2 Searching through a List

Searching is an extremely common task with lists. Now, in Python, we have the operator in to test if something is in a list. Very nice ☺However, not all languages support that. So let's look at an implementation of a search:

```
# Returns true if the value target is
# in the list arr; false otherwise.

def findInList( arr, target ):
found = False
ix = 0
while ix < len(arr) and not(found):
    if arr[ ix ] == target:
        found = true
    ix += 1
return found

def main():
sizes = [ 6, 7, 8, 8.5, 9, 11 ]
ans9 = findInList( sizes, 9 )
ans10 = findInList( sizes, 10 )
print ( "Contains 9? ", ans9 , ". Contains 10? " , ans10 )

main()
```

The displayed results would be:
Contains 9? true. Contains 10? false

Notice the following things about this example:

- The code to perform the search is written as a function. Although this is not necessary (see next example), if the code is written this way, it can be re-used with other target values and/or other lists.
- The goal of the search is only to determine if the target is present or not. The function returns True if the target is present in the list, or False if it is not.
- Both the list and the value being sought (the 'target') are passed into the function as arguments.
- The loop iterates through all elements of the list. An if statement is used to examine the value of each element. If the target is found, the loop is terminated immediately and the function returns the value True. If the entire loop completes without ever encountering the target value, the function returns False.
- When the function is called, the order of the arguments must match the order of the parameters -- the list must be first, followed by the target value.

This search function would work just as well with an array of strings.

Sometimes it is more useful to know the index where the target was found, not just if it was found. One of the practice activities is to rewrite this function, but instead of returning true and false, either return the index where the target was found, or return -1 if the target was not found (notice that -1 is not a legitimate index value. Here it is acting like a flag value)

3.3 Sifting (looking for multiple elements that match a certain criteria)

Sometimes, the goal of the search is to produce more than one result. A search of this kind is often called a sifting operation -- a name that calls to mind the process of sifting through a mixture to find elements of a particular size, weight, etc. For example, we may wish to find all the values in a numeric list greater than a particular threshold, or maybe all the values in a string list beginning with a particular character.

Here is an example to find all the elements that begin with "M":

```
members =["Homer", "Marge", "Lisa", "Bart", "Maggie" ]
for var in members:
    if var[ 0 ] == "M": # compare the first character of the string stored in var
        print ( var )
```

The displayed results would be:
Marge
Maggie

Notice the following things about this example:

- The loop must iterate through every element in the list, even if the target is found. This is because it is possible for more than one element to meet the search criteria.
- The search criteria are expressed as the condition of an if statement. Any condition that can be tested in this way can be used as the basis for a sift. Imagine searching for all sizes between 10 and 12, or for all interest rates greater than or equal to 2%.
- Some means must be provided to show multiple results. In this example, the results are printed, so a print statement is executed each time a matching element is found. Another common approach is to create a new list and fill it with all elements from the original array that match the search criteria.
- A search or sift operation should not normally disturb (change) the list being searched.

Many variations on searches and sifts are possible. In all cases, the basic principles are the same:

- Iterate through all elements of the list.
- Use an if statement to examine each element to see if it meets the search criteria.
- Take appropriate action if a match is found.

The sequential search described here works fine for small lists but can be inefficient for large lists. Other, more efficient techniques (like binary search) are available, but are beyond the scope of this course.

4 A More Complex List Algorithm

Sometimes when working with lists, you want to visit more than one element at a time. For example, what if we wanted to know if there were 2 of the same value in a row in a list, one immediately after the other.

Let's think about the pairs of elements we need to compare:

we need to compare the value at index [0] with the value at index [1]
we need to compare the value at index [1] with the value at index [2]
we need to compare the value at index [2] with the value at index [3]

Are you seeing a pattern here? Let's write this is an expression:

We need to compare the value at index [i] with the value at index [i + 1]

You see the repetition, so we'll need a loop. We see that i will start at 0 (initialization) and will be incremented by 1 (the update). The last piece to determine is the test.

Typically, when we visit every element in a loop, we want to go all the way to the last index, len(list) - 1. However, remember that with each iteration of this algorithm, we are accessing both list[i] *and* list[i + 1] If i is the last index, then where are we going with [i + 1]? Beyond the end of the list! Therefore, we have to stop the loop one index early.

Let's put this loop together (notice, this is still incomplete):

```
def containsPair(aList):
idx = 0
while idx < len(aList) - 1:
    # do something with aList[i] and aList[i + 1]
    idx += 1
# return True or False
```

Ok, now let's figure out the logic of determining if we have a the same value next to each other. That will require an if statement. If we find a pair that are the same, we can stop testing, because we found our answer. Therefore, I'm going to use a flag variable to signal when we've found a pair. I'll initialize it to False which makes sense, we haven't found a pair before the loop even starts. Also, notice that I've added a second expression to the loop's test. The loop doesn't have to keep repeating if I we found a pair.

Here is the completed function:

```
def containsPair(aList):
idx = 0
foundPair = False
while idx < len(aList) - 1 and not(foundPair):
    # did we find identical values next to each other?
    if aList[i] == aList[i + 1]:
        foundPair = True
    idx += 1
return foundPair
```