CSC 171 - Introduction to Computer Programming

Lecture #9 – Lists

Data Structures And Algorithms

- Part of the "science" in computer science is the design and use of data structures and algorithms
- As you go on in CS, you will learn more and more about these two areas

Data Structures

- Data structures are particular ways of storing data to make some operation easier or more efficient. That is, they are tuned for certain tasks
- Data structures are suited to solving certain problems, and they are often associated with algorithms.

Kinds Of Data Structures

- Roughly two kinds of data structures:
  - Built-in data structures, data structures that are so common as to be provided by default
  - User-defined data structures (classes in object oriented programming) that are designed for a particular task

Python Built In Data Structures

- Python comes with a general set of built in data structures:
  - lists
  - tuples
  - string
  - dictionaries
  - sets
  - others...

The Python List Data Structure

- A list is an ordered sequence of items.
- You have seen such a sequence before in a string. A string is just a particular kind of list (what kind)?

Make a List

- Like all data structures, lists have a **constructor**, named the same as the data structure. It takes an iterable data structure and **adds each item** to the list
- It also has a shortcut, the use of square brackets [ ] to indicate explicit items.

Constructing Lists

```
>>> a_list = [1, 2, 'a', 314159]
>>> weekdays_list = ['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday']
>>> list_of_lists = [[1, 2, 3], ['a', 'b', 'c']]
>>> list_from_collection = list("hello")
>>> a_list
[1, 2, 'a', 314159]
>>> weekdays_list
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists
[[1, 2, 3], ['a', 'b', 'c']]
>>> list_from_collection
['h', 'e', 'l', 'l', 'o']
>>> []
[]
>>>
```

Similarities With Strings

- concatenate/`+` (but only of lists)
- repeat/`*`
- indexing (the `[ ]` operator)
- slicing (`[:]`)
- membership (the `in` operator)
- `len` (the length operator)

Operators

```
[1, 2, 3] + [4] ⇒ [1, 2, 3, 4]
[1, 2, 3] * 2 ⇒ [1, 2, 3, 1, 2, 3]
1 in [1, 2, 3] ⇒ True
[1, 2, 3] < [1, 2, 4] ⇒ True
```

- Compare index to index, first difference determines the result

Differences Between Lists and Strings

- Lists can contain a mixture of any Python object; strings can only hold characters

  `1, "bill",1.2345, True`

- Lists are ***mutable*** (their values can be changed), while strings are ***immutable***
- Lists are designated with **`[ ]`**, with elements separated by commas, strings use " "or ' '

The Structure Of A List

```
myList = [1, 'a', 3.14159, True]
```

myList

1	'a'	3.14159	True	
0	1	2	3	Index forward
−4	−3	−2	−1	Index backward

```
myList[1] ⟶ 'a'
myList[:3] ⟶ [1, 'a', 3.14159]
```

Indexing

- can be a little confusing, what does the [ ] mean, a list or an index?

```
[1, 2, 3] [1]  ⇒ 2
```

- Context solves the problem.
  - Index always comes at the end of an expression.
  - Index is always preceded by something (a variable, a sequence).

List of Lists

```
my_list = ['a', [1, 2, 3], 'z']
```

- What is the second element (index 1) of that list? Another list.

```
my_list[1][0]  # apply left to right
mylist[1] ⇒ [1, 2, 3]
[1, 2, 3][0] ⇒ 1
```

List Function

- **`len(my_list)`** – number of elements in the list
  `len([1, [1, 2], 3]) ⇒ 3`
- **`min(my_list)`** – smallest element.
  – Must all be the same type
- **`max(my_list)`** – largest element
  – Must all be the same type
- **`sum(my_list)`** – sumof the elements – numeric only

Iteration

- You can iterate through the elements of a list like you did with a string:

```
>>> my_list = [1, 3, 4, 8]
>>> for element in my_list:
    print(element, end = ' ')

1 3 4 8
>>>
```

Strings Are Immutable

- Strings are immutable.
  - Once created, the object's contents cannot be changed.
  - New objects can be created to reflect a change, but the object itself cannot be changed

Strings Are Immutable

```
my_str = 'abc'
#my_str[0] = 'z' # Doesn't work

#Instead, make new str
new_str = my_str.replace('a', 'z')
print(new_str)
```

Output

```
zbc
```

But Lists are Mutable

- Unlike strings, lists are mutable. You **can** change the object's contents!

Lists are Mutable

- Unlike strings, lists are mutable. You **can** change the object's contents!
- Example

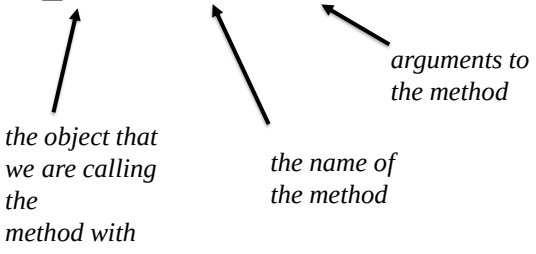
```
my_str = [1,2, 3]
my_str[0] = 127
print(my_str)
127, 2, 3
>>
```

List Method

- Remember, a function is a small program (such as **`len`**) that takes some arguments, the stuff in the parenthesis, and returns some value
- a method is a function called in a special way, the ***dot call***. It is called in the context of an object (or a variable associated with an object)

Lists Have Methods

```
my_list = ['a', 1, True]
my_list.append('z')
```



More List Methods

- **`my_list = ' a'  # index assignment`**
- **`my_list.append()`** – Adds an item to the end of the list
- **`my_list.extend()`** - Adds an item to the end of the list
- **`my_list.pop()`** – removes an item from a list and returns it
- **`my_list.insert()`** – inserts an item into a list
- **`my_list.remove()`** – removes an item from a list
- **`my_list.sort()`** – sorts items on a list
- **`my_list.reverse()`** – reverses the order of items on a list

More About List Methods

- Most of these methods ***do not return a value***.
- This is because lists are mutable, so the methods modify the list directly. No need to return anything.
- It can be confusing

Unusual Results

```
my_list = [4, 7, 1, 2]
my_list = my_list.sort()
my_list ⇒None
```

- What happened was the sort operation changed the order of the list in place (right side of assignment). Then the sort method returned **None**, which was assigned to the variable. The list was lost and **None** is now the value of the variable.

my_list.split()

- The string method split generates a sequence of characters by splitting the string at certain split-characters.
- It returns a ***list***.

```
>>> split_list = 'this is a test'.split()
>>> split_list
['this', 'is', 'a', 'test']
>>> print(split_list)
['this', 'is', 'a', 'test']
>>>
```

List Indices

- Just as [] can be used to indicate part of a character string, it can be used to indicate some elements in a list.
- The indices can be used to indicate more than one element and by using a negative sign, it can be used to count from the end of the list.

List Indices Examples

```
>>> x = [14, 18, 23, 28, 34, 42, 50, 59]
>>> print(x[5])
42
>>> print (x[3:7])
[28, 34, 42, 50]
>>> print(x[:-2])
[14, 18, 23, 28, 34, 42]
>>> print(x[2:-2])
[23, 28, 34, 42]
>>>
```

my_list.append()

- **append()** can be used to insert additional elements at the end of a list.

```
>>> x = [14, 18, 23, 28, 34, 42, 50, 59]
>>> print(x[5])
>>> x.append(66)
>>> print(x)
[14, 18, 23, 28, 34, 42, 50, 59, 66]
>>>
```

my_list.extend()

- **extend()** can be used to insert additional elements at the end of a list.
- **extend()** can be used with other data structures, such as tuples and sets.

`my_list.extend()` Examples

```
>>> x = [14,  18, 23, 28, 34, 43, 50, 59]
>>> y = [66, 72, 79, 86, 96]
>>> x.append(y)
>>> print(x)
[14, 18, 23, 28, 34, 43, 50, 59, [66, 72, 79, 86, 96]]
>>> x = [14, 18, 23, 28, 34, 42, 50, 59]
>>> x.extend(y)
>>> print(x)
[14, 18, 23, 28, 34, 42, 50, 59, 66, 72, 79, 86, 96]
>>>
```

`my_list.pop()`

- **`pop()`** removes an item from a list and returns the item.
- Example

```
>>> x = [14, 18, 23, 28, 34, 42, 50, 59]
>>> y = x.pop()
>>> print(x)
[14, 18, 23, 28, 34, 42, 50]
>>> print(y)
59
>>>
```

my_list.insert()

- **insert()** inserts an item into a list at a specified position on the list.

```
>>> x = [14, 23, 28, 34, 42, 50, 59]
>>> x.insert(1, 18)
>>> print(x)
[14, 18, 23, 28, 34, 42, 50, 59]
>>>
```

my_list.remove()

- remove – deletes a specified item from the list

```
>>> print(x)
[14, 18, 23, 28, 34, 42, 50, 59]
>>> x.remove(34)
>>> print(x)
[14, 18, 23, 28, 42, 50, 59]
>>>
```

`my_list.sort()`

- **`sort()`** arranges its elements in order.
- Example

```
>>> x = [34, 50, 14, 23, 18, 42, 59, 28]
>>> x.sort()
>>> print(x)
[14, 18, 23, 28, 34, 42, 50, 59]
>>>
```

Sorting

- Only lists have a built-in sorting method. Thus you often convert your data to a list if it needs sorting.

```
>>> my_list = list('xyzabc')
>>> print(my_list)
['x', 'y', 'z', 'a', 'b', 'c']
>>> my_list.sort()
>>> print(my_list)
['a', 'b', 'c', 'x', 'y', 'z']
>>>
```

sorted()

- The **sorted** function will break a sequence into elements and sort the sequence, placing the results in a list

```
>>> sort_list = sorted('hi mom')
>>> print(sort_list)
[' ', 'h', 'i', 'm', 'm', 'o']
>>>
```

my_list.reverse()

- **reverse()** reverses the order of elements in a list.

```
>>> print(x)
[14, 18, 23, 28, 34, 42, 50, 59]
>>> x.reverse()
>>> print(x)
[59, 50, 42, 34, 28, 23, 18, 14]
>>>
```

Reverse Words in a String

- **join** method of string places the calling string between every element of a list

```
>>> my_str = 'This is a test'
>>> string_elements = my_str.split()
>>> print(string_elements)
['This', 'is', 'a', 'test']
>>> reversed_elements = []
>>> for element in string_elements:
    reversed_elements.append(element[::-1])

>>> print(reversed_elements)
['sihT', 'si', 'a', 'tset']
>>>
```

Program To Find Class Average On A Test

```
#  Find the class average on a test

numGrades = 10
def get_grades() :
    grades = []
    #  Each time the loop, read another grade
    #  into the array
    for count in range(0, 10) :
        grade_str = input("Enter a grade  ")
        grades.append(int(grade_str))
    return(grades)
```

```
#  calc_average() - Add up the grades, divide
#                   by the number of grades to
#                   find the average
def calc_average(grades) :
    sum = 0
    for  this_grade in grades :
        sum = sum + this_grade

    return(sum/numGrades)
```

```
#  letter_grade() - Translate the score into
#                     a letter grade
def letter_grade(score) :
    if score >= 90 :
        return('A')
    elif score >= 80 :
        return('B')
    elif score >= 70 :
        return('C')
    elif score >= 60 :
        return('D')
    return('F')
```

```
#  printResults() - Print the average and
#                   the grades
def print_results(grades, mean) :
    print("The grades are:")

    for std_grade in grades :
        print(std_grade)

    print("The average is ", mean, \
          " corresponding to a grade of ", \
          letter_grade(mean))
```

```
# The main program
grades = get_grades()

#  Find the average
average = calc_average(grades)

#  Print the average and the grades
print_results(grades, average)
```

Example - Anagrams

- Anagrams are words that contain the same letters arranged in a different order. For example: 'iceman' and 'cinema'
- Strategy to identify anagrams is to take the letters of a word, sort those letters, than compare the sorted sequences. Anagrams should have the same sorted sequence

`are_anagrams()`

```
def are_anagrams(word1, word2) :
    """Return True if words are anagrams"""
    # Sort the characters in the words
    word1_sorted = sorted(word1)
    word2_sorted = sorted(word2)

    #  Check that the sorted words are identical
    #  Compare the sorted lists
    if word1_sorted == word2_sorted :
        return True
    else :
        return False
```

The Complete Anagram Program

```
def are_anagrams(word1, word2) :
    """Return True if words are anagrams"""
    # Sort the characters in the words
    word1_sorted = sorted(word1)
    word2_sorted = sorted(word2)

    #  Check that the sorted words are identical
    #  Compare the sorted lists
    if word1_sorted == word2_sorted :
        return True
    else :
        return False
```

```
print("Anagram Test")

#  Input two words
two_words = input("Enter two space-separated
words: ")

#  Split them into a list of words
word1, word2 = two_words.split()

# Return True or False
if are_anagrams(word1, word2) :
    print("The words are anagrams.")
else :
    print("The words are not anagrams.")
```

Repeating Input Prompt for Valid Input

```
valid_input_bool = False
while not valid_input_bool :
    try :
        two_words = input\
          ("Enter two space-separated words: ")

        #  Split them into a list of words
        word1, word2 = two_words.split()
        valid_input_bool = True
    except ValueError :
        print("Bad Input")
```

only runs when no error, otherwise go around again

Final, Complete Program

```
def are_anagrams(word1, word2) :
    """Return True if words are anagrams"""
    # Sort the characters in the words
    word1_sorted = sorted(word1)
    word2_sorted = sorted(word2)

    #  Check that the sorted words are identical
    #  Compare the sorted lists
    if word1_sorted == word2_sorted :
        return True
    else :
        return False
```

```
print("Anagram Test")

#  Input two words, checking for errors
valid_input_bool = False
while not valid_input_bool :
    try :
        two_words = input\
           ("Enter two space-separated words: ")

        #  Split them into a list of words
        word1, word2 = two_words.split()
        valid_input_bool = True
    except ValueError :
        print("Bad Input")

# Return True or False
if are_anagrams(word1, word2) :
    print("The words %s and %s are anagrams."
                %(word1, word2))
else :
    print("The words %s and %s are not anagrams."
                 %(word1, word2))
```