Lists

CSC 1200 - Principles of Computing

Overview

- Lists
- Accessing List Elements
- The in Operator
- List Operations
- List Slices
- List Methods
- Deleting Elements
- Lists and Strings
- Objects and Values
- The is Operator
- Aliasing
- Using Lists as Arguments

List

- Like a string, a **list** is a sequence of values.
  - In a string, everything in the sequence is a character.
  - In a list, the values in the sequence can be of any type (list of integers, floats, strings, other lists, or mixture of all of these)
- The values in a list are called **elements** or **items**.
- The simplest way to create and initialize a list is to enclose the elements in square brackets:

```
>>> instruments = ['violin', 'cello', 'piano', 'guitar', 'mandolin', 'saxophone', 'drum']
>>> years = [2000, 2002, 2005, 2007, 2009]
>>> empty = []
>>> print( instruments)
['violin', 'cello', 'piano', 'guitar', 'mandolin', 'saxophone', 'drum']
>>> print( empty )
[]
```

List Items Don't Have to Be the Same Type!

```
>>> tia_info = [2009, ['piano', 'violin', 'cello', 'ukulele']]
>>> print( tia_info )
[2009, ['piano', 'violin', 'cello', 'ukulele']]
```

```
>>> trey_info = [2005, 'piano', 'drum']
>>> print( trey_info )
[2005, 'piano', 'drum']
```

```
>>> len( trey_info )
3
>>> len( tia_info )
2
```

Accessing List Elements

- The bracket operator is used to access list elements (just like strings)
- Unlike strings, lists are mutable – meaning the contents of a list can be changed.

```
>>> print( instruments, '\n', years, '\n', trey_info, '\n', tia_info)
['violin', 'cello', 'piano', 'guitar', 'mandolin', 'saxophone', 'drum']
 [2000, 2002, 2005, 2007, 2009]
 [2005, 'piano', 'drum']
 [2009, ['piano', 'violin', 'cello', 'ukulele']]
>>> instruments[2]
'piano'
>>> years[4]
2009
>>> trey_info[1]
'piano'
>>> tia_info[1]
['piano', 'violin', 'cello', 'ukulele']
>>> tia_info[2]
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    tia_info[2]
IndexError: list index out of range
>>> tia_info[1][2]
'cello'
```

```
>>> instruments[3] = 'acoustic guitar'
>>> instruments
['violin', 'cello', 'piano', 'acoustic guitar', 'mandolin', 'saxophone', 'drum']
```

The in Operator

The in operator can be used with lists.

```
>>> 2002 in years
True
>>> '2005' in years
False
>>> trey_info[1] in instruments
True
>>> trey_info in instruments
False
```

```
>>> count = 0
>>> for item in trey_info:
        if item in instruments:
            count += 1


>>> print( count )
2
```

```
>>> for instrument in tia_info[1]:
        print( instrument )


piano
violin
cello
ukulele
```

```
>>> for i in range(len(instruments)):
        print( instruments[len(instruments)-i-1] )


drum
saxophone
mandolin
acoustic guitar
piano
cello
violin
```

List Operations

- The + operator concatenates lists (just like it does for strings)
- The * operator repeats a list a given number of times (just like it does for strings)

```
>>> fruit = ['apple', 'watermelon']
>>> veggie = ['pepper', 'peas', 'spinach']
>>> green = fruit + veggie
>>> print( green )
['apple', 'watermelon', 'pepper', 'peas', 'spinach']
>>> fruit * 3
['apple', 'watermelon', 'apple', 'watermelon', 'apple', 'watermelon']
>>> data = [0]*3 + [2]*2 + [4]*2 + [6]
>>> print( data )
[0, 0, 0, 2, 2, 4, 4, 6]
```

```
>>> veggie += ['beans']
>>> veggie
['pepper', 'peas', 'spinach', 'beans']
>>> veggie += 'kale'
>>> veggie
['pepper', 'peas', 'spinach', 'beans', 'k', 'a', 'l', 'e']
```

```
>>> data += [12]
>>> data
[0, 0, 0, 2, 2, 4, 4, 6, 12]
>>> data += 13
Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    data += 13
TypeError: 'int' object is not iterable
```

List Slices

- The slice operator works on lists the same way that it does for strings.

```
>>> g_scale = ['G', 'A', 'B', 'C', 'D', 'E', 'F#', 'G']
>>> g_scale[3:5]
['C', 'D']
>>> g_scale[:6]
['G', 'A', 'B', 'C', 'D', 'E']
>>> g_scale[3:]
['C', 'D', 'E', 'F#', 'G']
```

- Since lists are mutable, we can use assignment with slices.

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters[2:4] = ['X', 'Y', 'Z']
>>> letters
['a', 'b', 'X', 'Y', 'Z', 'e', 'f', 'g']
>>> letters[4:] = ['h', 'i', 'j', 'k']
>>> letters
['a', 'b', 'X', 'Y', 'h', 'i', 'j', 'k']
```

List Methods

- Python provides methods that operate on lists. We must use dot notation to use these.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Examples of Using List Methods

```
>>> snacks = ['popcorn', 'chips']
>>> snacks.append('nuts')
>>> snacks
['popcorn', 'chips', 'nuts']
>>> snacks.extend(['crackers', 'pretzels'])
>>> snacks
['popcorn', 'chips', 'nuts', 'crackers', 'pretzels']
>>> snacks.count()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    snacks.count()
TypeError: list.count() takes exactly one argument (0 given)
>>> snacks.count('chips')
1
>>> snacks.count('carrots')
0
>>> snacks.index('chips')
1
>>> snacks.insert(1, 'carrots')
>>> snacks
['popcorn', 'carrots', 'chips', 'nuts', 'crackers', 'pretzels']
```

```
>>> snacks.pop()
'pretzels'
>>> snacks
['popcorn', 'carrots', 'chips', 'nuts', 'crackers']
>>> snacks.pop(1)
'carrots'
>>> snacks
['popcorn', 'chips', 'nuts', 'crackers']
>>> snacks.reverse()
>>> snacks
['crackers', 'nuts', 'chips', 'popcorn']
>>> snacks.sort()
>>> snacks
['chips', 'crackers', 'nuts', 'popcorn']
>>> snacks.remove('nuts')
>>> snacks
['chips', 'crackers', 'popcorn']
>>> snacks.remove('carrots')
Traceback (most recent call last):
  File "<pyshell#86>", line 1, in <module>
    snacks.remove('carrots')
ValueError: list.remove(x): x not in list
```

Example Lists Uses

- A **reduce** operation combines a sequence of elements into a single value.
  - Suppose scores is a list of test scores. Write a function that returns the average of all the values in the list.
- A **filter** operation selects some of the elements in the sequence and filters out others.
  - Suppose names is a list of first names. Write a function that returns a list that contains names that begin with a specific letter.
- A **map** operation defines a correspondence between each element of the sequence and elements of another sequence.
  - Suppose months contains a list of names of months. Write a function that returns a list abbreviations that are the first 3 letters of the month name.

Deleting Elements

There are several ways to delete elements from a list.

- If you know the index of the element you want to remove, you can use pop.

```
>>> animals
['ant', 'bear', 'cat', 'dog', 'eel', 'frog']
>>> animals.pop(3)
'dog'
>>> animals
['ant', 'bear', 'cat', 'eel', 'frog']
```

- If you know the element you want to remove (but not the index), you can use remove.

```
>>> animals.remove('cat')
>>> animals
['ant', 'bear', 'eel', 'frog']
```

- To remove more than one element, you can use del with a slice

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del numbers[3:6]
>>> numbers
[0, 1, 2, 6, 7, 8, 9]
>>> del numbers[:2]
>>> numbers
[2, 6, 7, 8, 9]
>>> del numbers[2:]
>>> numbers
[2, 6]
```

Lists and Strings

- To convert a string into a list of characters, you can use the built-in type casting function list.

```
>>> word = 'Python'
>>> letters = list(word)
>>> letters
['P', 'y', 't', 'h', 'o', 'n']
```

- To break a string into words, you can use the split method.
  - By default it uses a space to break up the string.
  - You can specify an argument called a **delimiter** to specify a character to use to split the string.

```
>>> sentence = 'Some words with spaces between them.'
>>> words = sentence.split()
>>> words
['Some', 'words', 'with', 'spaces', 'between', 'them.']
```

```
>>> hyphenated = 'non-life-threatening injuries'
>>> delimeter = '-'
>>> words = hyphenated.split(delimeter)
>>> words
['non', 'life', 'threatening injuries']
```

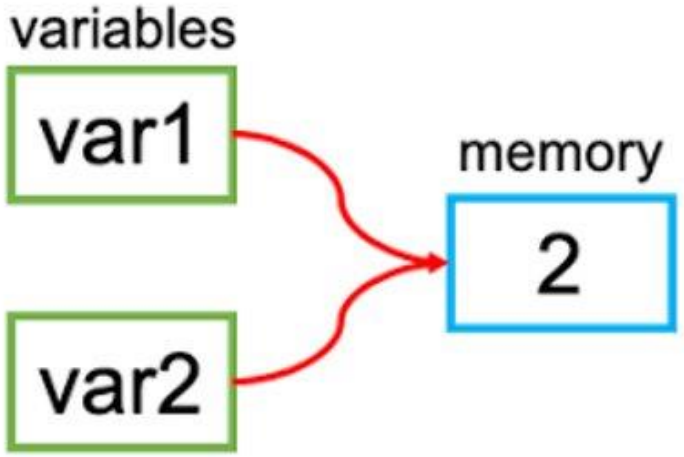
Objects and Values

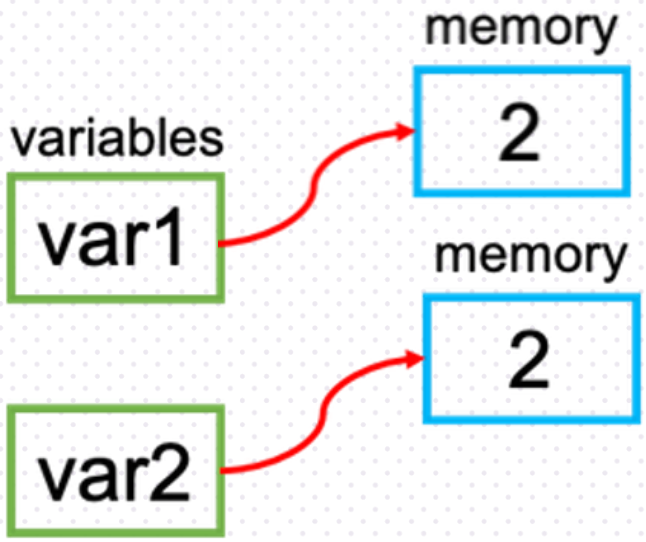
- Suppose we execute the statements:

  var1 = 2

  var2 = 2

- Which of the situations below does this represent?





The is Operator

- The is operator will tell you if two variables refer to the same object or different objects.

```
>>> a = 2
>>> b = 2
>>> a is b
True
>>> b += 2
>>> a is b
False
```

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

```
>>> a = '12'
>>> b = '1'
>>> b += '2'
>>> a is b
False
>>> a == b
True
```

not identical

equivalent

```
>>> a = 'apple'
>>> b = 'apple'
>>> b is a
True
>>> b = b.lower()
>>> b is a
False
```

Aliasing

- If two different variables refer to the same object, than we say that object is **aliased**.
- If the aliased object is mutable (e.g., a list) changes made with one alias affect the other.

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True          <-- aliased
>>> b.append(10)
>>> a
[1, 2, 3, 10]
>>> a.remove(2)
>>> b
[1, 3, 10]
```

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False         <-- not aliased
>>> b.append(10)
>>> a
[1, 2, 3]
>>> a.remove(2)
>>> b
[1, 2, 3, 10]
```

Using Lists as Arguments

- When you pass a list as an argument to a function, the list will be aliased. If the function modifies the alias (parameter) the original list will be altered.

```
def list_has( L, item ):
    found = False
    for i in range( len(L) ):
        L[i] = L[i].lower()
        item = item.lower()
        if L[i] == item:
            found = True
    return found

myList = ['Hello', 'Hi There', 'hi there', 'HOLA']
if list_has( myList, 'HI THERE' ):
    print( 'The list contains HI THERE' )
print( myList )
```

```
The list contains HI THERE
['hello', 'hi there', 'hi there', 'hola']
```

Debugging

- Most list methods modify the argument and return None (this is different from string methods)

  myList = myList.sort() ← this does NOT result in a sorted list being assigned to myList

- Know which functions and methods change a list and which ones create and return a new list.
- In general, make copies of lists to avoid aliasing.