Lists

In a way, you have been using lists already. A string is known as an array of ___char___acters. Think of a list as a smarter string, except you can put anything into it.

Creating a list

To create an empty list you do the following:

```
l0 = list() # l0 = []
l1 = []
```

To create a list with some known values you do the following:

```
l0 = [2,4,"cat",3.45]
```

Notice that I have floats, ints, and strings inside the same list.

Alias and cloning

Look at the following code:

```
l0 = [2,3,4]
l1 = l0
l2 = list(l0)
l0.remove(3)
l3 = l2 + l1
l1.append(1)
```

The function `remove(item)` looks through the list and finds the first occurrence of that `item` (known as an *element*) and removes it from the list. If the list was `[1,2,3,1]` and you did used `remove(1)`, you would have the list `[2,3,1]`.

The function `append(item)` takes the item and adds it to the end of the list. The opposite of this is a function called `pop()` that removes *and returns* the last element in the list.

After this code runs, what are the values of `l0`, `l1`, `l2`, and `l3`? Don't go to the next section until you've made a guess.

Answers

Here are the values:

```
l0 = [2,4,1]
l1 = [2,4,1]
l2 = [2,3,4]
```

```
l3 = [2,3,4,2,4]
```

Why are `l0` and `l1` the same? Because `l1` is an alias for `l0`. They are different variable names but they are connected (pointing) to the same data in the computer. Why are `l0` and `l2` different? Because `list()` has two meanings: `list()` and `list(list)`. One of them creates an empty list, and the other makes a copy of the list. `l2` and `l0` have the same values of data, but they are not the same data.

```
l3 = l2 + l1
```

This is a good talking put about *operators*. An operator is a symbol (like ,!@#$%^& and so on) that is linked to a special action for that data. For ints and floats, it is addition. For strings it is concatenation. Lists use the `+` for concatenation. I am appending the elements of the second list to the end of the first list.

(Member) functions of a list

Here are some important member functions of lists:

- [i]
- del
- append(a)
- insert(i,a)
- remove(a)
- pop(i= -1)
- index(a)
- count(a)
- sort()
- reverse()

All functions for data structures either access, modify, and create.

Create

We've talked about `list()` and `= []` earlier in the lecture. They are used to create new representation of data in your program.

Access

[i]

The function `[i]` takes an int value and will `return` the value at that place in the list. If I have `l0 = [5,4,1]`, and `print l0[1]`, the program will print the returned value of `4`.

count

The function `count(a)` goes through the entire list and counts every time the element, `a`, appears. It then returns this number to you.

index(a)

The function `index(a)` goes through the entire list and will return the index value, the location, of the first element that matches element `a`. If it is not found, it will return `-1`.

Modify

Append, remove, and pop

We already explained `append()`, `remove()`, and `pop()`, but I want to hit `pop()` one more time. The notation `i = -1` means that if you *do not* put a value into the function `pop()`, the function will assume the value is `-1`. The `pop()` function's action is comparable to the using `[i]` followed by `del l0[i]`.

del

The function `del` deletes an element at a certain place (an index) in the list.

Insert

The function `insert(i,a)` takes a index value, i, and an element, a, and adds that element to that position in the list.

```
l0 = [0,1,2,3]
l0.insert(2,"here")
print l0
```

This would print `[0,1,"here",2,3]`. Notice that it pushes elements at and after the `2` location back.

[i]

This operator is also a modifier.

```
numbers = [3,5,6,"dog"]
numbers[3] = 10
```

The list numbers would have the values `[3,5,6,10]` now.

reverse

The function `reverse()` reverse the order of the list from it's current order.

sort

The function `sort()` orders the elements from smallest value to largest value. Later on in the lectures will we talk about certain sorts and how they work. Python's sort algorithm is the Quicksort algorithm from C++.

range(3) == [0,1,2]

when you do a `for` loop in the fashion of `for i in range(x):` with x being some integer, the range function returns a list. The list contains `x` elements going from `0` to `x-1`. Then `i` takes every value of that list, one at a time, and uses them in the loop. It is the same process as `for i in ["blue","red","green"]:` for our list of colors in an earlier lecture but instead of strings we have numbers.

Lists of lists

At some point in programming, you might be asked to store data in a grid fashion. It might be used to hold locations of an certain items, values in a board game, battleship coordinates, or some other thing. How would you do this? You have rows and columns, x and y, to deal with. We know that we can access elements inside of a list by using the `[i]` operator at the end.

```
row = list()
table = list()
for i in range(10):
  row.append(0)
for i in range(10):
  temp = list(row)
  table.append(temp)
```

There. A table 10x10 with zeros. Why didn't I do the following?

```
row = list()
table = list()
for i in range(10):
  row.append(0)
for i in range(10):
  table.append(row)
```

Because of aliases! If I were do the second program. I would have a table that points to the list of zeros.

For the first program, if I did:

```
table[0][0] = "X"
```

I would get the following print out:

```
[[0,0,0,0,0,0,0,0,0,"X"],
[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],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0]]
```

While for the second program I would get this:

```
[[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X],
[0,0,0,0,0,0,0,0,0,X]]
```

3D lists and beyond

If I had a list of lists of lists, how would I access the first element of the first list inside the first list? It's not too difficult. Let's say the variable is called `cube`. Then it would be `cube[0][0][0]`.

Closing

Lists are powerful data structures and come in handy all the time. One of the things they are useful for is fileIO. FileIO is file input and output.