CS 115 Lecture 15

Lists part 1

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

5 November 2015

Lists

In Python a string is a sequence of characters, but there are other kinds of sequences, too. The most important is the **list**.

- A list is, like a string, a sequence of things.
- But unlike a string, the things can be any type:
  - List of numbers: `[7, 1, 3]`
  - List of strings: `[ "hello", "world" ]`
  - Even lists of lists.

List syntax

- To write a literal list in your program, use square brackets:
  `poets = [ "Coleridge", "Neruda", "Hughes" ]`
- The things in a list are called its **elements**.
  - This list has three elements, each a string.
  - Its **length** is 3 (the number of elements).
  - Use `len` to get the length:
    `print(len(poets)) # 3`
- Elements are numbered starting from zero.
  - As with strings, we call this the **index** or **position** of the element.
  - The last element has index `length - 1`.
- Lists can be concatenated with +:
  `print([3, 1, 4] + [1, 5, 9])`
  `→ [ 3, 1, 4, 1, 5, 9 ]`
  - But only with other lists!
    `[3, 1, 4] + 1 → TypeError`

Accessing elements of a list

You can get elements of a list using the subscript syntax:

```
scores = [ 75.0, 68.5, 83.0 ]
third = scores[2] # 83.0
```

- Negative indices count from the end:

```
last = scores[-1] # 83.0
```

- Notice the difference with strings:
  - Subscripting a string gives you another (one-character) string.
  - Subscripting a list gives you the element...
    - ★ ...which could have any type.
- Lists also support slicing. Slicing a list gives *another list*:

```
exams = scores[1:3] # [ 68.5, 83.0 ]
```

- Again:
  - Subscripting gives a single element.
  - Slicing gives a list of elements.

Searching in lists

There are two ways to search for an element in a list:

- You can use `in` to check whether it's there:
  ```
  if "Eliot" in poets:
  ```
- Check for that exact element; doesn't look "inside" the elements:
  ```
  if "ridge" in poets:  #False!
  ```
- To find an element's position in the list, use the `index` method.
  ```
  rank = poets.index("Coleridge") # 0
  ```
  - It works mostly like the string `find` method.
  - Can give another argument to specify where to start.
  - One important difference: if it was not found:
    - `mystring.find(...)` returns -1
    - `mylist.index(...)` gives a run-time error!
    - To be safe you can use `in` first:
      ```
      pos = -1
      if thing in list:
          pos = list.index(thing)
      ```

Traversing lists

You can traverse a list with a for loop:

```
scores = [ 85, 72, 56, 98, 84, 72 ]
sum = 0
for grade in scores:
    sum += grade
```

- This works the same as with a string.
- In each iteration the loop variable will be one element of the list.
- To get both indices and elements, you can use the same techniques as with strings:
  - Use a counter to track the index.
  - Use a range loop with subscripting.
  - Use `enumerate(mylist)`.

Strings versus lists

You've probably noticed a lot of similarities between lists and strings.

- Sometimes the very same code works with both!
  - `len`, subscripts, and slicing.
  - Traversal.
  - Concatenation.
- But there are also many differences:
  - Each element of a string is a character.
    - ★ In Python, characters are strings (of length 1).
  - But the elements of a list can be anything.
    - ★ And are usually *not* lists.
  - With strings, `in` searches for a substring.
  - With lists, `in` searches for single elements only!
  - Strings use `find` to locate a substring (-1 if not found)
  - Lists use `index` to locate an element (error if not found)
- Another big difference: lists are **mutable**.

Immutability

- Strings, ints, and floats are **immutable**.
- Which means: these objects doesn't change once they are created.
  - You can't change the number 4.
  - Instead, operations on these types create and return new objects.
  - . . . which you may then assign back into the same variable
- Ordinary assignment doesn't change the object!
  - It changes a *variable* to point at a different object.
  - More on this later. . .

Mutability

Some kinds of objects can be changed after they are created.

- Remember graphics shapes. You can:
  - Draw and undraw them.
  - Change the fill and outline colors.
  - Move them around.
  - Set the text (`Text` and `Entry` only).
- How does this differ from assignment?

The meaning of assignment

- A variable in Python is like a finger pointing at an object.
- Assigning to the variable makes the finger point somewhere else.
- The variable itself stays in the same location in memory.
  - Same finger!
- . . . but now points at ("refers to") a different value.

Mutability and functions

- Function parameters are separate variables from the arguments.
  - So assigning to the parameter doesn't change the argument.

```
def squareplus(x):
    x = x ** 2 # changes x, not num
    return x + 1
def main():
    num = 5
    sp1 = squareplus(num)
    print("sq+ of", num, "is", sp1) # num is still 5
```

  - The function gets what the finger points at, not the finger itself.
- However, they refer to the values of the arguments.
  - The parameter is another "finger" pointing at the same object.
  - And if that object is mutable, the function can mutate it:

```
def addseven(lst):
    lst.append(7) # mutates the list
def main():
    scores = [ 5, 9, 6 ]
    addseven(scores)
    print(scores) # [ 5, 9, 6, 7 ]
```

Mutability and functions

- **Call by reference**: functions can modify their arguments.
  - In Python, only by mutation, not assignment!
    - ★ Assignment changes a **variable** (re-point the finger)
    - ★ Mutation changes an **object** (the thing pointed to)
- **Side effects**: things that are changed by the function.
  - Printing output, creating a file, etc.
  - Mutating parameters.
  - Postconditions describe the return value and side effects.

List mutability

Lists are mutable: they can be changed in several ways:

- Appending or inserting a new element.
- Removing an element.
- Sorting and reversing.
- Changing the values of existing elements.

Inserting into a list

- The `append` method adds a new element to the end of a list:
  `poets.append("Angelou")`
  - Mutates the list.
  - Increases the length by one.
  - Does not return a value!
- To add a whole list to the end, use the `extend` method:
  `scores.extend([55, 88, 79])`
  - This example increases the length by 3.
  - Also returns nothing.
  - What would happen if you used append instead?
    - That would add the list as a single element!
    - Not usually what you want.
- The `insert` method adds a new element in the middle.
  `poets.insert(2, "Homer")`
  - The new element will be at index 2.
  - The indices of the following elements shift up by one to make room.

Mutation versus making new objects

- Notice that `append`, `extend`, and `insert` return nothing!
  - Most mutating functions in Python work this way.
  - (With a few exceptions we'll point out).
  - So don't do this:
    ```
    colors = colors.append("yellow") # ERROR: colors = None
    ```
  - Instead:
    ```
    colors.append("yellow") # GOOD: mutates colors
    ```
- Conversely, concatenating with + doesn't mutate the list.
  - Instead, it returns a new list.
  - So don't do this:
    ```
    colors + primaries # ERROR: throws away new list
    ```
  - Instead:
    ```
    colors = colors + primaries # OR
    colors += primaries
    ```

Deleting from a list

You can delete from a list by index:

- Syntax: `del list[index]`
  - Removes the element at position `index`.
  - Shifts down the following elements to fill in the gap:
    ```
    list[index] = list[index + 1]
    list[index + 1] = list[index + 2]
    ...
    ```
  - Can also delete a range by using a slice:
    ```
    del list[2:5] # remove elements 2, 3, and 4
    ```

Or you can remove a specific value ("search-and-destroy"):

- Syntax: `colors.remove("blue")`
- Searches for the first occurrence of "blue" and deletes it.
- Gives a runtime error if it wasn't found!
- How could you do this using `del`?
  ```
  pos = colors.index("blue")
  del colors[pos]
  ```

Sorting and reversing

The `reverse` method reverses the order of a list.

```
mylist = [ "red", "green", "blue" ]
mylist.reverse()
print(mylist) → [ "blue", "green", "red" ]
```

- Reverse mutates the list!
  - So the original order is lost.
- And doesn't return a value.
  - So *don't* assign back into the list:
    `mylist = mylist.reverse() # ERROR: mylist = None`
  - If you need a *new* reversed copy, use:
    `backwards = list(reversed(mylist))`
    - ★ `mylist` is unchanged.
- Note the differences:
  - `reverse` is a method that mutates the list.
  - `reversed` is a function that returns a new sequence.
    - ★ Not actually a list—convert with `list(...)`

Sorting a list

You can also sort a list with the `sort` method.

- Defaults to ascending order.

```
scores = [ 75, 63, 92 ]
scores.sort()
print(scores) → [ 63, 75, 92 ]
```

- On strings, that means alphabetic order:

```
poets = [ "Coleridge", "Neruda", "Hughes", "Eliot" ]
poets.sort()
print(poets)
→ [ "Coleridge", "Eliot", "Hughes", "Neruda" ]
```

- Can do descending order instead:

```
scores.sort(reverse = True)
print(scores) → [ 92, 75, 63 ]
```

- To make a new sorted list and keep the original:

```
ascending = sorted(scores) # Doesn't mutate
```

  - `scores` is unchanged.
  - Similar to the difference between `reverse` and `reversed`.

Lists and assignments

The slots in a Python list work like variables.

- They refer to (point to) objects:
  - A list is like a box of fingers (eww)
- They can be assigned to, making them refer to new values.
  `colors[0] = "purple"`
  - This is mutation! Doesn't work with strings!
  - A function that takes a list parameter can change the list this way.
  - . . . mutating the original list argument.
  - When you get the box, you get all the fingers inside it.
    - ★ But not the finger that points at the box.
    - ★ Assigning to the *whole list* won't change the original.
- Can't assign into a slot that doesn't exist!
  - It is an error if the index is $\geq$ the length.
  - Need `append` instead.

Lists, mutability, aliasing

Remember aliasing from when we looked at the graphics package.

- Aliasing happens with all mutable objects.
- It is possible to have two variables referring to the very same list.
  - Arguments and parameters, for example.
  - Or by assignment.
- If so, mutations to one variable will be reflected in the alias.

```
testscores = [ 84, 100, 78 ]
myscores = testscores
myscores.append(96)
print(testscores) → [ 84, 100, 78, 96 ]
```

- Often you want the two variables to be independent.
  - You need to "break the alias"
    - ★ That was the purpose of the graphics shape `clone` method.
  - There are two ways to clone a list:
    - ★ Use a whole-list slice: `newcopy = orig[:]`
    - ★ Or the built-in copy method: `newcopy = orig.copy()`
    - ★ Now copy and orig point to two different lists...
    - ★ ...but those lists hold the same values.

How to create a list

We've seen several different ways we can make a list:

- Hard-code it: `lst = [ 1, 2, 3 ]`
- Start out empty and append:

```
lst = []
lst.append(1)
lst.append(2)
```

  - Useful as an accumulator.

- Start out empty and concatenate:

```
lst = []
lst += [1]
lst += [2]
```

- Split a string: `lst = "one two three".split()`
- Replication: `lst = [ 0 ] * 100`
  - Makes a list with 100 copies of 0.