CIS 231

Ch. 10 – Python Lists

10.1 – A list is a sequence

- It has *elements*, similar to a string
- Can create by listing items between [ ]
  - Just [] for initially empty list
- You can have different types of values, including other lists (*nested* lists)

10.2 Lists are mutable

- You can access an item based on its position by specifying its index after the name and between [ ], e.g. `cheeses[0]`
  - Lists are *zero-indexed*
- Since they're mutable, you can change the contents at any index or assign a new list
- Each item is *mapped* to a particular index
- The `in` operator works on lists (p. 90)

10.3 Traversing a list

- A/k/a iterating across a list
- Using for – in, with or without indices:

```
for cheese in cheeses:
    print cheese
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

- Nested lists are single elements

10.4 – List operations

- The `+` operator creates a new list that *concatenates*, or joins, two lists together
- The `*` operator creates a new list containing the values of a given list repeated the specified number of times

10.5 – List slices

- The slice operator used with strings is also used with lists:

```
t = [“a”, “b”, “c”, “d”, “e”, “f”]
t[1:3]   # displays [“b”, “c”]
```

- Omitting the first index takes values starting at the beginning. Omitting the second prints the last values.
- A slice operator *can* be on the left side of an assignment, changing multiple items

10.6 – List methods

- Don't return anything; change the list directly
- `append()` adds an element to the end of a list
  - e.g. `t.append("d")`
- `extend()` appends the elements of a list to the end of another list
  - e.g. `t1.extend(t2)`
- `sort()` arranges the elements from low to high
  - e.g. `t.sort()`

10.7 – Map, filter, and reduce

- *Augmented* (a/k/a *compound*) assignment:
  - Syntactic sugar for when destination variable is also one of the operands, e.g. `total += x`
- `sum()` sums the values in the provided list
  - Combining a sequence into a single value is known as a *reduction*, or *reduce* method
- A *mapping* is taking a function and applying it to a list of values, generating a new list by applying the mapping function to each value in the list
- (continued)

10.7 continued

- Selecting certain items from a list and returning a sublist is commonly known as a *filter* operation

10.8 – Deleting elements

- `pop()` deletes an item at a particular index in the list and returns it
  - Remaining items are moved over as needed
- The `del` operator deletes an item from the list but doesn't return it
  - Can also use a slice to delete multiple items
- You can delete a particular item (rather than by index) with `remove()`

10.9 – Lists and strings

- To convert a string to a list of its component characters use the `list()` function
- `split()` creates a list of strings by separating an existing string based on *whitespace delimiting*
  - You can pass `split()` a specific delimiter if you need to separate strings by a different character
- `join()` uses a delimiter to combine strings into a single string separated by that delimiter

10.10 – Objects and values

- Objects are their own entities – when we talk about an object we often refer to it as an *instance* of a particular type
- We have to differentiate between whether we are talking about the same instance (`is`) "identical" or two instances with the same *state*, or contents (==) "equivalent"
  - Example on next slide

10.10 continued

```
# same instance vs. same value (state)
a = [1,2,3]
b = [1,2,3]

# False, distinct instances of lists
print("separate lists, same instance: ", a is b)
# True, both have the same contents (state)
print("separate lists, same state: ", a == b)

a = b
# True, both now reference the same list instance
print ("lists after assignment: ", a is b)
```

10.11 - Aliasing

- A *reference* is an association with a particular instance of an object
- We can copy references just like values – a copy of a reference is called an *alias*
- Since both references are to the same object instance, changes made via one reference are reflected in the other one (since it's two references to the same object) (Fig.10.4, p. 96)
- Therefore be careful w/mutable object

10.12 – List arguments

- Lists are passed to functions *by reference* – a copy of the reference to the list (an alias) is passed rather than a copy of the list itself
  - As we saw in the previous section, the alias refers to the original list sent, so any changes to the list in the function change that original list
- Distinguish between operations that modify existing lists and those that create a new list
- (continued)

10.12 continued

```
# example on p. 96
# doesn't delete the head of list
def bad_delete_head(t):
    t = t[1:]
```

- The slice created a new list, so *t* no longer references the same list that was sent
- You can return an altered list, or assign a mutated list to a different variable (also p. 97)

Next Time

- Ch. 11 – Dictionaries
  - Dictionary as a set of counters
  - Looping and dictionaries
  - Reverse lookup
  - Dictionaries and lists
  - Memos
  - Global variables
  - Long integers