CMPT 120
Intro to CS & Programming I

WEEK 10 (Mar. 17-21)

— *Jérémie O. Lumbroso* —

Lecture 24:
Modifying Lists

http://www.sfu.ca/~jlumbros/Courses/CMPT120/

First: Good News

(at least for me!!)

Last Friday CBS renewed my most favorite show ever, for a sixth season!!

And last night's episode was **AWESOME**!!!

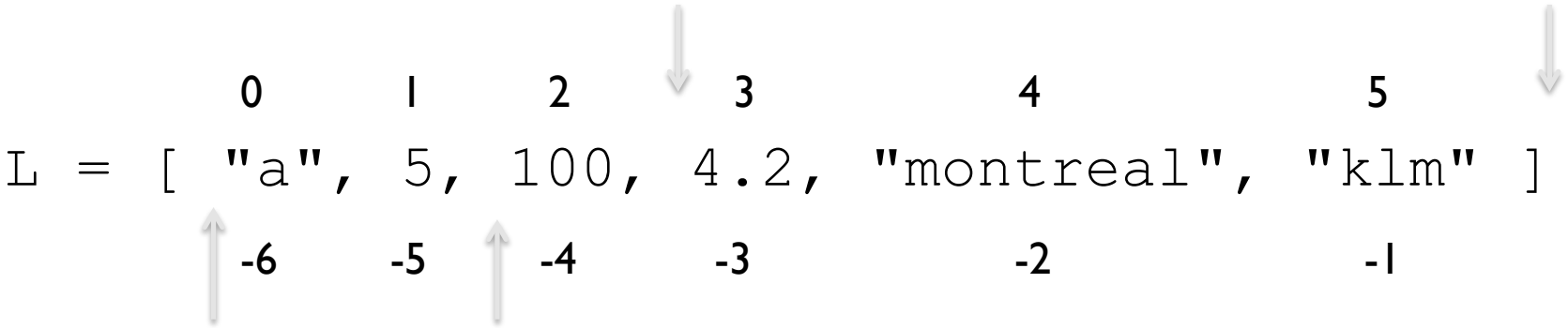
And now on to more serious (and less interesting?) stuff

MODIFYING LISTS & LISTS BY REFERENCES

Recap of Lectures on Lists

- A list L can store sequences of elements
- The empty list is `[ ]`
- They can mostly be manipulated like strings
  - `L[0]` accesses the first element and `L[0:3]` provides the sub-list containing the 1st to 3rd element `[ L[0], L[1], L[2] ]`
  - Concatenation: `[1, 2] + [4]` gives `[1, 2, 4]`
- The elements of lists can be **modified**
- Python provides functions to modify lists
- We have to be careful about **references**

A Reminder on Indexes/Slices

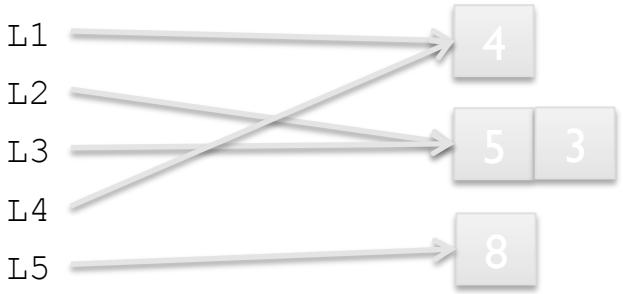


- `L[0]` accesses the element… `"a"`
- Indexes can be given in reverse (but index "starts" in -1)
  - `L[-1]` accesses `"klm"`
- `L[a:b]` put an arrow at the beginning of position `a` and the beginning of position `b` and take everything in between
  - `L[0:2]` gives `[ "a", 5 ]`
  - `L[-1:]` gives `[ "klm" ]` (last element)
  - `L[-3:]` gives `[ 4.2, "montreal", "klm" ]`

References?

```
L1 = []
L2 = []
L3 = L2
L4 = L1
L5 = L1 + L3

L1.append(4)
L3.append(5)
L5.append(8)
L2.append(3)
```



- L3 and L4 contain **references** to L2 and L1
- L5 contains a new list (which is the concatenation of a copy of L1 and L3 at that points)

A **reference** (an arrow) is created only when there is an **assignment** (of lists).

Why References?

- References are **arrows** pointing at values
- Sometimes, because an object is so large, it makes sense to be careful how many times it is stored
- Lists can be potentially
  - huge
  - manipulated very often
- It would be terrible if every modification implied the entire list was copied

References vs. Copies

- A reference to a list is created when **a variable containing a list is assigned (a single =) to another variable**
  - `L1 = L2`
  - `L3 = myList`
- A copy is created in all other situations
  - Concatenating two lists creates a copy of both lists, and joins them: `L4 = L1 + L3`
  - Taking a slice creates a copy of the sublist: `L5 = L3[:]`
  - Etc.
- There are some special "sticky" (the technical term) situations, but you will not encounter them in CMTP 120 (hopefully)

Delete an Element of a List

- Let `L` be a list, you can remove its item in position `i` by doing **`del L[i]`**

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

Remove All Even Elements

- `L` is a list containing even and odd integers
- Take for instance `L = [1,3,4,5,7,13]`
- Remove all even integers by using **`del`**

Your code:

- Iterates over the list
- If an element is even (use %) delete it
- Print the final list

A Done

B Help!

Our Expectations

It is important to **always decide in advance what code should do…**

… instead of running it and then saying, "*Yeah, huh, that looks about right? Right?*"

Then: "*Shiiiiit! that hidden CodeWrite test failed.*"

So here:

- We want to remove even elements
- Since we have `L = [1,3,4,6,7,12]`
- We expect to obtain `[1,3,7]` as a result

First Attempt

```
L1 = [1,3,4,6,7,12]

print "Before:", L1

for x in L1:
  if x % 2 == 0:
    del x

print "After:", L1
```

A Works

B Incorrect

- Prints: `After [1,3,4,6,7,12]`
- **Conclusion:** when deleting items (or modifying liststs) cannot iterate through lists using items — must use positions

Second Attempt

```
L2 = [1,3,4,6,7,12]

print "Before:", L2

for i in range(len(L2)):
  if L2[i] % 2 == 0:
    del L2[i]

print "After:", L2
```

A Works

B Incorrect

- Causes an "index out of bound" error during the loop, and never reaches the second print
- **Conclusion:** when deleting items we cannot use a for loop iterating over the positions, because the number of elements in the lists changes (and so the range of positions over which we can iterate changes as well)

Try Again

- Since we cannot use a for loop to iterate over the positions, we can use a while loop

```
L = [1,3,4,6,7,12]
i = 0
while i < len(L):
   if L[i] % 2 == 0:
   # ...
# ...
```

- Complete the rest of the program

A Done

B Help!

Third Attempt

```
L3 = [1,3,4,6,7,12]

print "Before:", L3

i = 0
while i < len(L3):
  if L3[i] % 2 == 0:
    del L3[i]
  i = i + 1

print "After:", L3
```

A Works

B Incorrect

- Prints: `After [1,3,6,7]` which means it forgot to delete `6`
- **Conclusion:** when we delete an element `L[i]`, the next element that was in position `i+1` now is in position `i`, so if we increment the position, we skip that element.

Ground Hog Day?

- You are wondering: "How many attempts??"

Fourth Attempt

```
L4 = [1,3,4,6,7,12]

print "Before:", L4

i = 0
while i < len(L4):
  if L4[i] % 2 == 0:
    del L4[i]
  else:
    i = i + 1

print "After:", L4
```

A Works

B Incorrect

- Prints: `After [1,3,7]` it works!!
- **Conclusion:** through hard work and perseverance, anything is possible!

list.insert

```
help(list)
...
 |  insert(...)
 |      L.insert(index, object) -- insert object
 |      before index
...
```

- inserts an element at position `index`

Mystery Code

- Assume L is an increasing (sorted) list

```
def mystery(L, x):
  for i in range(len(L)):
    if L[i] > x:
      L.insert(i,x)
      return
  L.insert(len(L), x)

mylist = [1,5,10]
mystery(mylist, 6)
print mylist
```

A. Compute the maximum

B. Insert the element at the end

C. Insert elmt in its sorted place

D. Search for `x` in the list

E. Insert the element after the largest one

- What does this function do?

Pacing and Understanding

How well did you understand today?

- **A** Too easy, this lecture is way below my abilities
- **B** Everything went at a good pace, and I am fine
- **C** Too fast, but I will catch up on my own
- **D** Too fast, and I need you to slow down
- **E** I really do not think I can handle this