CSC108H Lecture 4

Dan Zingaro
OISE/UT

June 4, 2009

Clickers: While and String Review (while_clicker.py)

```
def func():
  correct = False
  while not correct:
    s = raw_input ("Enter a password: ")
    correct = len(s) == 5 and s[:2] == 'xy'
```

Which of the following passwords gets us out of the loop?

- A. xyz
- B. abxyz
- C. abcxy
- D. xyabc
- E. xyxy

While: Guessing Numbers

- Let's write a program that asks the user to guess a random number from 1 to 10
- The program will keep asking until the user gets it right
- When we do not know how many times to loop in advance, and we do not have a sequence to loop over, `while` is probably more appropriate than `for`
- If we import the `random` module, we can use its `randint` function to generate a random integer between two bounds

While: Guessing Numbers (while_guess.py)

```
import random

def guess():
  answer = random.randint(1, 10)
  guess = raw_input("Guess a number in the range 1 .. 10: ")
  while int(guess) != answer:
    guess = raw_input("Sorry, guess again: ")
```

Python Lists

- Remember: the reason we can loop over a string's characters is because a string is a sequence
- Python's more general type of sequence is the list; we can store any object in a list, not just characters
- Imagine we have taken 20 measurements, represented as floats, which will be used in calculations
- We could try to store them as 20 separate variables, which is awkward in itself; but then what if we had 50 measurements? 100?
- The solution is to store them in a list; e.g. for three measurements:
- `measurements = [45.27, 45.26, 45.24]`

Python Lists...

- Python lists can be heterogeneous
- i.e. we can store integers, floats, strings, pictures, and even other lists inside of lists
- Another important difference between strings and lists is that lists **are** mutable
- We can use the same string index notation when dealing with lists
- With lists, though, this allows us not only to obtain one of its objects, but also to change it

Python Lists...

Let's mess around in the shell.

```
a = ["Dan", "student", 1982]
a
len(a)
a[0] #Zero-based, just like strings
a[2]
a[1] = 'instructor'
a
...
```

User-input Loops

- We often use while-loops when obtaining user-input, since we don't know how many times the user will mess up before giving a valid response
- Let's write a function that takes a list of "valid command" strings, and keeps asking the user to type a command until they type one from the list
- One idea is to ask for a command outside of the loop; if invalid, keep looping and asking for a valid command
- But this repeats code! We might try to initialize `command` to the empty string before entering the loop; why might this be a bad idea?
- Instead, we can use the `None` placeholder object

While: User Input (get_valid.py)

```
def get_valid_command(valid):
  '''Prompt for and return a string that is a valid command
  i.e., a string in the list "valid".'''

  command = raw_input("Please enter a command: ")
  while command not in valid:
    print "Invalid command"
    command = raw_input("Please enter a command: ")

  return command # command is valid here
```

List Functions

- Just as for strings, we can use `len` to get the length of a list (i.e. the number of objects in the list)
- We can use `min` to get the minimum element of a list
- We can use `max` to get the maximum element of a list
- `min` and `max` work on lists of strings in addition to lists of integers!
- We can use `sum` to get the sum of the elements in a list
- . . . let's try

List Methods

- As with strings, there are lots of methods; use `dir (list)` or `help (list.method)` for help
- `append` is used to add an object to the end of a list
- `insert (index, object)` inserts `object` before `index`
- `sort` sorts a list
- `remove (value)` removes the first occurrence of `value` from the list
- ... let's try; we'll also experiment with `reverse`, `pop`, `extend` ...

Clickers: List Methods (list_methods_clicker.py)

```
L = [4, 8, 12, 16]
L.append (len(L))
L.remove (4)
```

What does L refer to after this code runs?

- A. [8, 12, 16, 4]
- B. [8, 12, 16]
- C. [4, 8, 12, 16]
- D. [8, 12, 16, 3]

Looping over Lists

- We loop over lists just as we loop over strings
- The pattern for looping through a list is as follows

```
for obj in lst:
  <do whatever with obj>
```

- The loop variable in the example above is `obj`; it "steps through" each object in the list `lst`

Exercise: Length of Strings

- Write a function that takes a list of strings, and prints out the length of each string in the list
- e.g. if the list is `['abc', 'q', '']`, the output would be as follows

```
3
1
0
```

- Remember: design-code-verify! Let's start with a design ...

Exercise: Uppercase List (bad_uc_list.py)

Given a list of strings, return a list that contains all of these strings in uppercase. One (bad) attempt is below. Let's make it work!

```
def uc_list (lst):
  '''Return a list that contains the words from list lst
  with uppercase letters.'''
  for s in lst:
    s = s.upper()
```

Using range

- The `range` function generates lists of integers
- If we call `range(n)`, integers from 0 to `n - 1` are generated
- If we call `range(m, n)`, integers from `m` to `n - 1` are generated
- If we call `range(m, n, s)`, integers from `m` to `n - 1`, in increments of `s`, are generated
- Whenever we want to create a list that follows one of these patterns, `range` is a good candidate
- e.g. let's write a function to generate a list of the squares of the first `n` positive integers ...

Lists and While

- Sometimes, it's more natural to use a `while` loop instead of a `for` loop when looping through a list
- Warmup: a while-loop that prints each element of a list
- Let's write a function, `our_index`, similar to the `index` list method
- Our function will take a list and a value, and return the first index of the value in the list, or return the length of the list if the value is not present
- For example, `our_index ([1,2,3], 2)` returns 1

Lists and While... (bad_index.py)

- A for-loop gives us the elements of the list, but not their indices — and we want to return an index!
- We could increment an index variable on each iteration of the for-loop, but then how could we terminate the loop once our value is found?
- Let's start with the bad code below ...

```
def our_index (lst, value):
  i = 0
  num = lst[i]
  while num != value:
    i = i + 1
    num = lst[i]
  return i
```

Our Version of Index (our_index.py)

What happens if `value` is not found? Let's carefully consider the loop guard!

```
def our_index (lst, value):
  i = 0
  while i < len(lst) and lst[i] != value:
    i = i + 1
  return i
```

Exercise: Uppercase List Again

- Again: given a list of strings, return a list that contains all of these strings in uppercase
- This time, let's write it "in-place" by modifying the original list instead of creating a new one
- Since we're modifying the original list, we don't have to `return` anything (why?)
- We'll do this using `for` and then again with `while`.

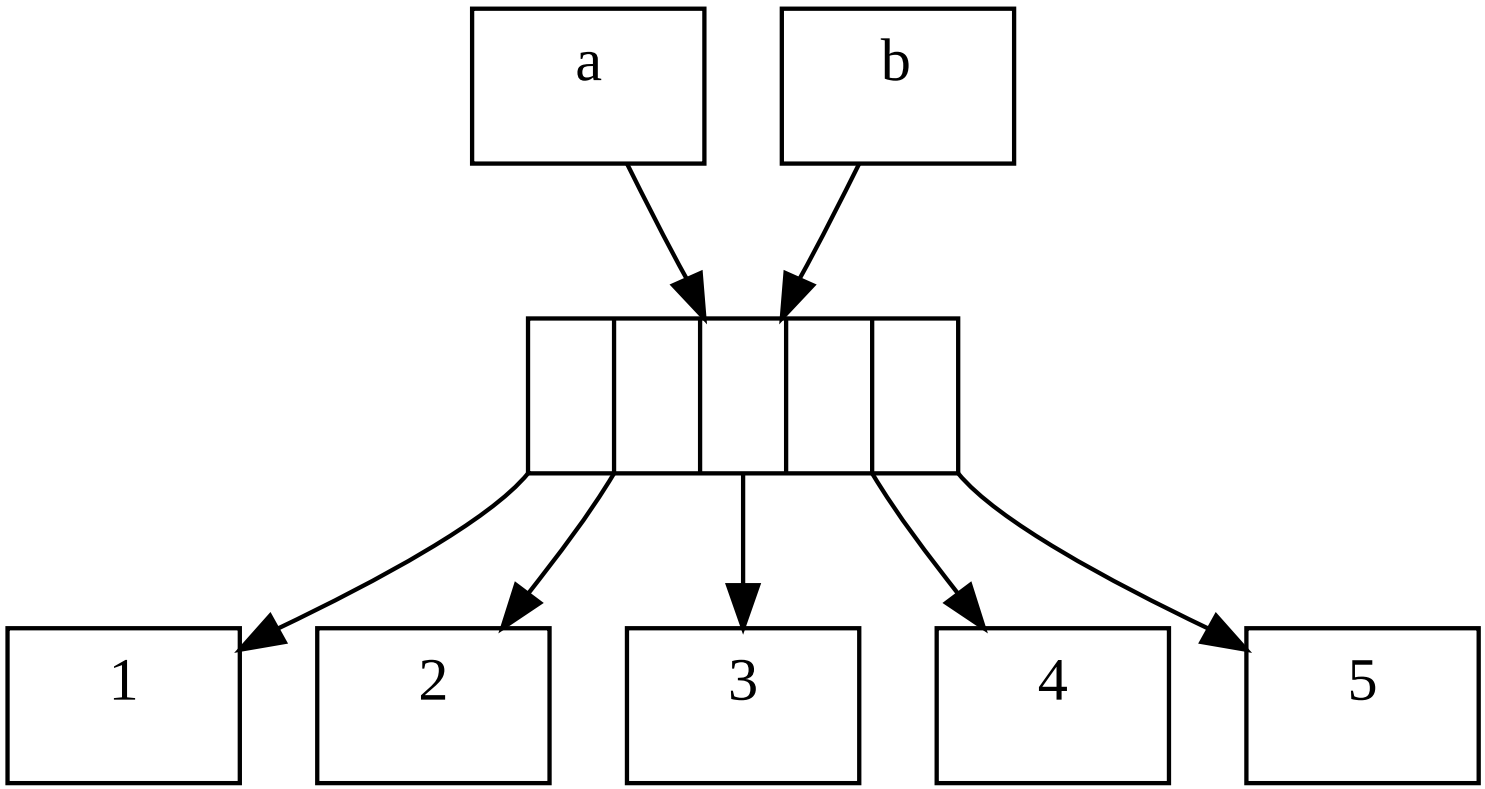
List Slice Syntax

- The slice syntax we used to extract characters of strings can also be used on lists
- The slice syntax always returns a list, even if it is one object (and even if it is the empty list)
- With lists, however, we can also assign to a segment of a list with slice syntax; we can even paste in a list of a different size than the slice!

```
a = [1, 2, 3, 4, 5]
a[1:3] # we know this
a[1:3] = [6, 7] # new!
a[0:1] = [9, 9, 9]
a[:] = [6]
...
```

Lists are Mutable!

```
a = [1, 2, 3, 4, 5]
b = a # Shared reference!
b[1] = 42 # check a!
```

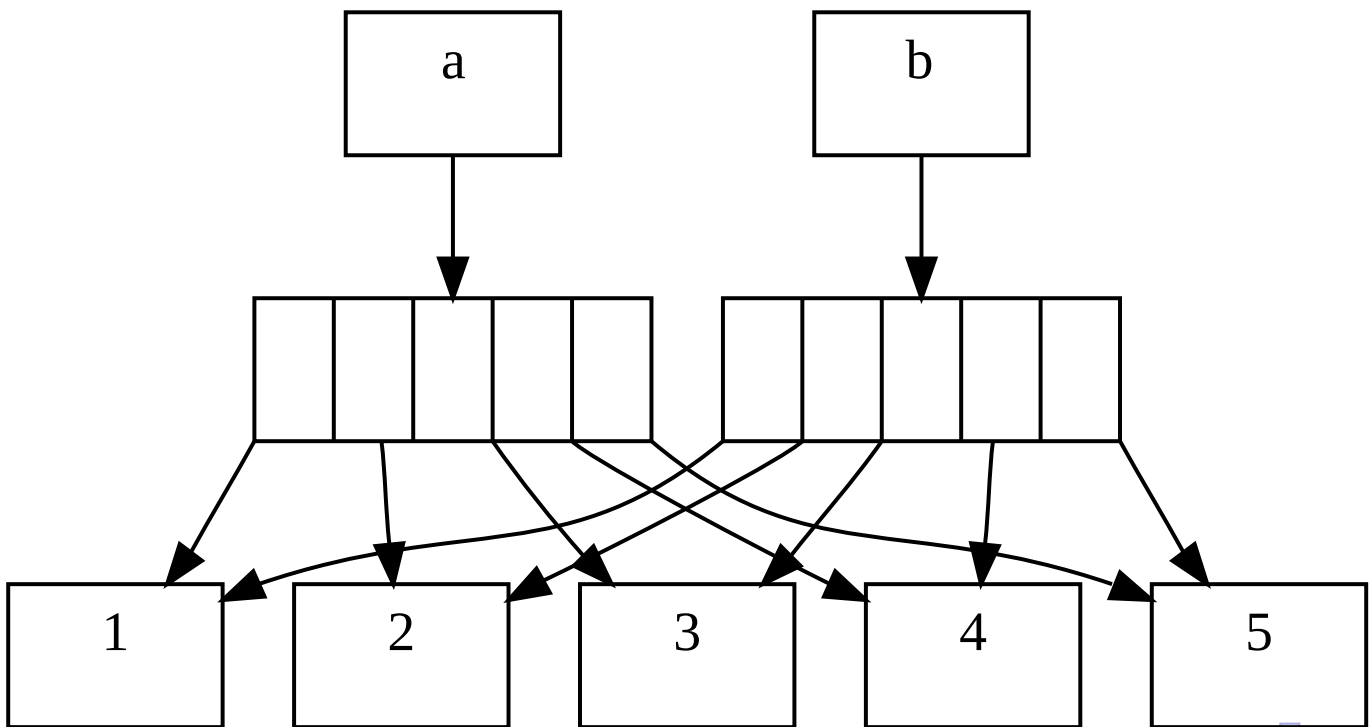


- Above, we assign two variable names to the same list
- We then change the list through only **one** of the variables — but what happens?

Slice Copies

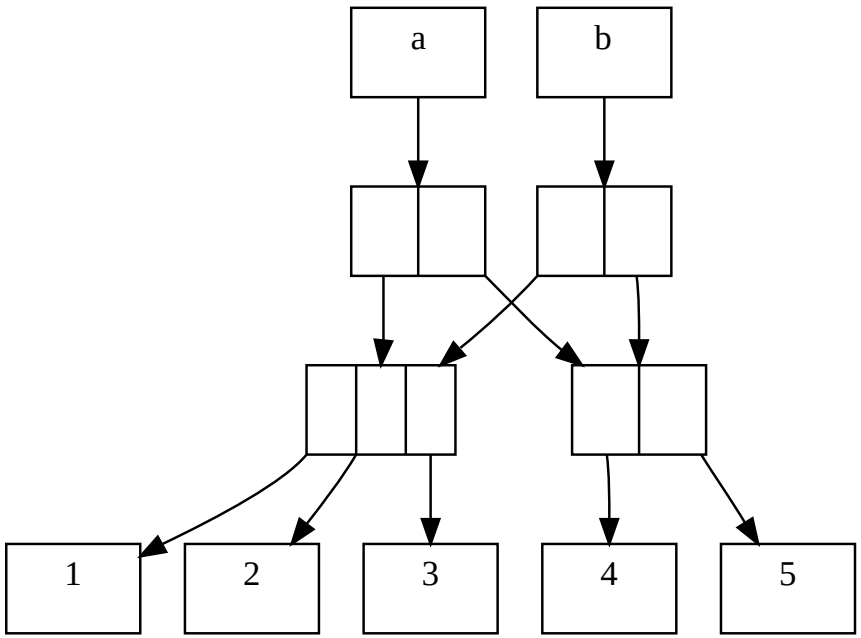
- If we do not want this list-sharing behavior, we have to make a new copy of the list
- One way is through slice syntax: `a[:]` is a new list that holds references to the same objects as a

```
a = [1, 2, 3, 4, 5]
b = a[:] # Not a shared reference!
b[1] = 42 # check a now
```



Nesting and Mutability

```
a = [[1, 2, 3], [4, 5]]
b = a[:] # New list
b.append (8) # a is not modified
b[1][1] = 42 # But now it is!
```



- Remember: a slice is a new list, but the object references it holds are to the same objects!
- As is the case here, this matters when lists hold references to other mutable objects

Concatenation and Repetition

- The + operator is overloaded (again) to operate on two lists
- It returns a new list resulting from concatenating the right-hand list to the end of the left-hand list
- It is similar to the `extend` method, but `extend` does not return a new list!
- The `*` operator takes a list `L` and an integer `n`, and creates a new list by stringing together `n` independent copies of `l` (like using + `n-1` times)

Clickers: List Mutability (star_clicker.py)

```
L = [4, 5, 6]
y = [L] * 3
L[1] = 42
```

What does `y` refer to after this code runs?

- A. `[4, 5, 6, 4, 5, 6, 4, 5, 6]`
- B. `[4, 42, 6, 4, 42, 6, 4, 42, 6]`
- C. `[[4, 5, 6], [4, 5, 6], [4, 5, 6]]`
- D. `[[4, 42, 6], [4, 42, 6], [4, 42, 6]]`
- E. `[4, 42, 6, 4, 5, 6, 4, 5, 6]`

More Operations with Slicing

- We can mimic a lot of the list methods with (sometimes obscure!) slice syntax
- To `extend` list `L`, we can assign a new list to the slice `L[len(L):]`
- To `append` `elt` to the end of list `L`, we can assign `[elt]` to the slice `L[len(L):]`
- To `insert` `elt` in list `L` at index `i`, we can use `L[i:i+1] = [elt, L[i]]`
- Exercise: how can we perform a `pop` using slice syntax?

Nested Lists

- We have already seen inner lists nested in an outer list; here is another example using student names and grades

```
student_grades = [['Dan', 80], ['Joe', 42],
                  ['Steph', 0]]
```

- Exercise: How can we access the first student's record? How about the first student's grade?
- Exercise: calculate the average grade of the students in the list
- Exercise: create a list of the names of all of the students in the list
- Exercise: use a nested loop to print out all names and grades on separate lines