CS 61A
DISCUSSION 3

September 22, 2016

ANNOUNCEMENTS

- Midterm 1 on Gradescope; regrade requests by Sunday night
- HW 4 released and due *today* (11:59pm)
- HW 5 released and due next Tuesday
- Maps released and due 9/29; extra credit point if submitted on or before 9/28. Project party next Wednesday (details on website).

ATTENDANCE

Link: http://tiny.cc/disc03

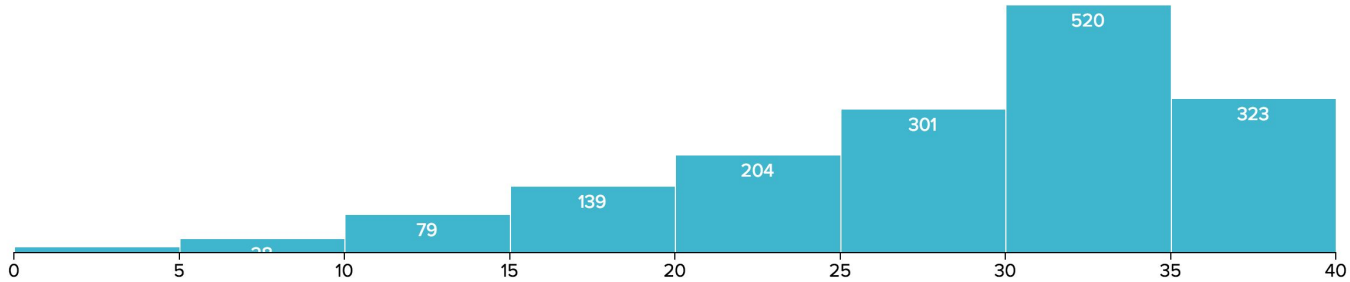
Secret phrase is on board

TOPICS FOR TODAY

- Midterm 1 Recap
- Sequences
- Trees

Midterm 1

Congratulations on making it through – and great job overall!
But don't let up…



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
0.5	30.5	40.0	28.17	7.79

(Feel free to talk to me if you have any concerns about your performance.)

SEQUENCES

Sequences

A **sequence** is an ordered collection of elements. Every sequence must have a **length** and also allow for **element selection (indexing)**.

Examples: lists [ ], tuples ( ), strings “ ”

```
>>> len(([4, 5], 6, ‘7’)[0])
2
```

Sequence Identification

Is a set (e.g. { 4 }) a sequence? Why or why not?

Sequence Identification

Is a set (e.g. { 4 }) a sequence? Why or why not?

No; sets aren’t ordered and as a result you can’t index into them.

Lists are pretty cool

Lists are perhaps the most versatile of our three main Python sequences.

You can populate a list with different types…

```
>>> whoa = [1, [1], “one”, {1: 1}, None, True, (1), 1.0]
>>> len(whoa)
8
```

List creation

To create a list, use either square brackets or the list constructor.

What happens below?

```
>>> lst1 = [4, 2]
>>> lst2 = list(4, 2)
>>> lst1 == lst2
```

List creation

You can create lists either with square brackets or with the list constructor.

What happens below?

```
>>> lst1 = [4, 2]
>>> lst2 = list(4, 2) # IT ERRORS HERE
>>> lst1 == lst2
```

List creation, continued

You can't call `list` on an int! The argument to the list constructor MUST be an iterable.

```
>>> list(4) # error!
>>> list((4,)) # this one's good
[4]
>>> list('345') # so is this one, funnily enough
['3', '4', '5']
```

List indexing

You index into the list with `lst[idx]`, where `idx` is any integer from 0 to `len(lst) - 1`.

(If the list is empty, you of course can't index into it with anything!)

If you use negative integers, it counts from the end of the list to the beginning. -1 is the last index, -2 is the second-to-last index, … and so on so forth.

List indexing: example

```
>>> nns = list(range(5))
>>> nns[1] = list(range(5))
>>> nns
[0, [0, 1, 2, 3, 4], 2, 3, 4]
>>> nns[-5] + nns[4]
>>> nns[5] + nns[-4]
>>> nns[1][1]
>>> nns[-6]
>>> nns[-3][3]
>>> nns[nns[-4][-4]][-4]
```

List indexing: example

```
>>> nns = list(range(5))
>>> nns[1] = list(range(5))
>>> nns
[0, [0, 1, 2, 3, 4], 2, 3, 4]
>>> nns[-5] + nns[4]
4
>>> nns[1][1]
1
>>> nns[-3][3]
Error
```

```
>>> nns[5] + nns[-4]
Error
>>> nns[-6]
Error
>>> nns[nns[-4][-4]][-4]
1
```

One final reminder

Indexing in Python starts at 0, not 1!

Don’t forget this!

```
>>> lst = [‘first’, ‘second’]
>>> lst[1]
‘second’
>>> lst[0]
‘first’
```

List concatenation

You can glue multiple lists together with the + operator.

```
>>> nns = list(range(1, 4))
>>> nns[1] = list(range(2, 5)) # nns = [1, [2, 3, 4], 3]
>>> nns[-2] + nns[1]

>>> nns + [‘638’] + list(nns[1])
```

List concatenation

You can glue multiple lists together with the + operator.

```
>>> nns = list(range(1, 4))
>>> nns[1] = list(range(2, 5)) # nns = [1, [2, 3, 4], 3]
>>> nns[-2] + nns[1]
[2, 3, 4, 2, 3, 4]
>>> nns + ['638'] + list(nns[1])
[1, [2, 3, 4], 3, '638', 2, 3, 4]
```

To check whether an element is in a list

```
>>> your_grades = [‘a-’, ‘a+’, ‘a’, ‘a+’]
>>> ‘f’ in your_grades
False
>>> ‘a’ in your_grades
True
```

List slicing

A list slice gives you back a **list** that is some subset of the original list. It is also a *copy* of that original subset – which is to say that list slicing always *creates a new list in memory*.

```
>>> lst = [1, 2, 3]
>>> lst[1:3]
[2, 3]
```

(Difference between indexing and slicing: **indexing** gives you one of the elements of the list. **Slicing** provides you with a LIST of some of the elements in the list.)

Syntax

You can tell it's a list slice because there are colons in the square brackets.

List slicing accepts three arguments, all of which are technically optional:

>>> lst[*start index* : *end index* ± *1* : *step size*]

>>> lst[*start index* : *end index* ± *1*]

>>> lst[*start index* :]

>>> lst[: *end index* ± *1*]

>>> lst[:]

Slicing examples

```
>>> nns = [list(range(4)), 4, 5]
>>> original = nns[:]
>>> nns[0][2], nns[-1] = 50, 6
>>> nns
[[0, 1, 50, 3], 4, 6]
>>> nns[1::-1]

>>> nns[:2]

>>> nns[:-3:-1]
```

```
>>> original
[[0, 1, 50, 3], 4, 5]
>>> nns[:2:3]

>>> nns[-2:]

>>> nns[0][::-1][1:5:2]
```

Slicing examples

```
>>> nns = [list(range(4)), 4, 5]
>>> original = nns[:]
>>> nns[0][2], nns[-1] = 50, 6
>>> nns
[[0, 1, 50, 3], 4, 6]
>>> nns[1::-1]
[4, [0, 1, 50, 3]]
>>> nns[:2]
[[0, 1, 50, 3], 4]
>>> nns[:-3:-1]
[6, 4]
```

```
>>> original
[[0, 1, 50, 3], 4, 5]
>>> nns[:2:3]
[[0, 1, 50, 3]]
>>> nns[-2:]
[4, 6]
>>> nns[0][::-1][1:5:2]
[50, 0]
```

List comprehensions

An easy way to create a new list.

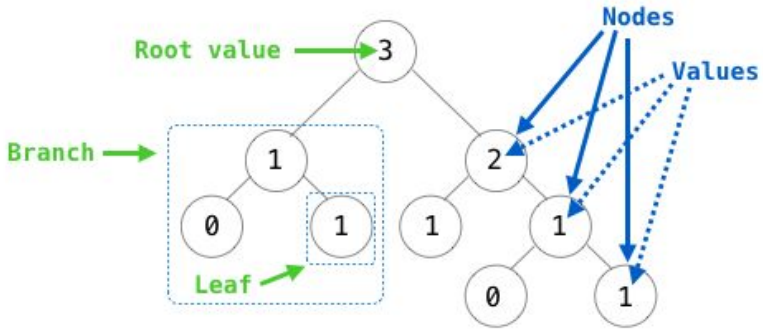
```
lst = [<expression> for x in <iterable> if <conditional expression>]
```

– is equivalent to –

```
lst = [ ]
for x in <iterable>:
      if <conditional expression probably involving x>:
            lst = lst + [<expression probably involving x>]
```

TREES

Tree vocabulary



Recursive description (wooden trees):

A **tree** has a **root** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **value**

One node can be the **parent/child** of another

People often refer to values by their locations: "each parent is the sum of its children"

Other terms to know

- **Root** (node at top of tree. Has no parent!)
- **Leaf** (node at bottom of tree. Has no children)
- **Subtree** (a tree within a tree)
- **Depth** (number of levels between node and root)
- **Height** (maximum depth throughout entire tree)

Tree ADT

Constructor:

- `def tree(root, branches=[ ])`

Selectors:

- `def root(tree)`
- `def branches(tree)`