Computer Science 134C
Introduction to Computer Science, in Python
Lecture #9 (Lists & Tuples)
September 26

Keywords
car & cdr, comprehensions, don't care (underscore), factors, heterogeneous, iterable, lists, l-values, mutability, parallel assignment, primes, r-values, shallow copy, tuples

Lists and Tuples.

1. Puzzle solution: in this week.
2. Exam three weeks from yesterday. Final is December 17, 9:30am; location TBA.
3. Questions?
4. List object. A lot can be learned from `pydoc3 lists`
   (or visit `https://docs.python.org/3/library` and search for `list`)
   (a) Lists keep their objects in order. Their objects can be accessed by index. They can be modified; they're *mutable*.
   (b) Like strings, they can be indexed and sliced. Unlike strings, their objects can be different types; they're *heterogeneous*.
   (c) They can be constructed from other *iterable* objects:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(set(sorted('hello, world')))
['o', 'h', 'w', 'd', ' ', 'l', 'e', ',', 'r']
```

   (d) They, themselves, are *iterable*: you can encounter their elements, in order:

```
>>> l = ['bon','do','go','iwa','pi','to'] # halfword words
>>> for x in l
...     print(x+x)
bonbon
dodo
gogo
iwaiwa
pipi
toto
```

   (e) You can determine their length (use `len`), and you can con+catenate them (use +).
   (f) Because they're mutable, you can change them:
      i. You can assign pre-existing elements (`l[i] = 'tar'`).
      ii. You can add new elements at the end of a list with `l.append(x)`. This is common when you're building up lists of results.

iii. You append all the elements from another iterable with `l.extend(container)`.

iv. You can remove elements. Remove and return the last with `l.pop()`, or the element at position `i` with `l.pop(i)`:

```
>>> d = [ 'Bashful', 'Doc', 'Dopey', 'Grumpy', 'Happy', 'Sleepy', 'Sneezy' ]
>>> l = list(d)
>>> while l:
...     i = randint(0,len(l)-1)
...     print(l.pop(i))
Sneezy
Sleepy
Grumpy
Happy
Doc
Dopey
Bashful
>>> print(l)
[]
>>> print(d)
['Bashful', 'Doc', 'Dopey', 'Grumpy', 'Happy', 'Sleepy', 'Sneezy']
```

Python has a similar operator, `del`, that does not return the value.

(g) Other methods of `lists` that are useful:

i. `l.clear()`. Destructively remove all elements of the list.

ii. `l.copy()`. Return a new, *shallow copy*: a new list containing shared references to contained objects. The following is true of non-empty lists, `l`:

```
l.copy()[0] is l[0]
```

iii. `l.count(v)`. Returns the number of times `v` appears in `l`.

iv. You can check to see if a value is in a list with `v in l`.

v. `l.index(v)`. Returns the index of the first occurrence of `v`, or raises an error. (Sadly, unlike strings, there is no `find(v)` method, so check first, if necessary.)

vi. `l.remove(v)`. Removes (without returning) a value from `l`.

vii. `l.insert(i,o)`. Inserts `o` in `l`, so that it will have index `i`.

viii. `l.reverse()`. *Destructively* reverses `l`.

ix. `l.sort(key=None,reverse=False)`. Sorts a list, in-place, destructively. The `key` argument allows you to specify a function that, given a value constructs a key to be used for sorting; if `reverse=True`, the sort is opposite. *All sorting in python is stable.*

(h) List comprehensions. You can construct new lists using iteration with a `for` expression:

```
# Typical, simple:
>>> words = [ line.strip() for line in open('/usr/share/dict/words') ]
# Guarded.  The \id{if} always follows.
>>> wds = [ word for word in words if len(word) == 3 ]
# Nested.
>>> split6 = [ x+y for x in wds for y in wds if x+y in words ]
```

(i) A more elaborate example:

```
>>> def factors(n):
...     return [ f for f in range(1,n+1) if n%f == 0 ]
>>> primes = [ n for n in range(100+1) if len(factors(n)) == 2 ]
>>> primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
    61, 67, 71, 73, 79, 83, 89, 97]
```

(Note how I use `n+1` and `100+1` at the tail of the `range` statements: this makes explicit *I want to include* `n` *(or* `100`*) as the final value.*)

5. Tuples, more formally. A tuple is an immutable, ordered list of values.

   (a) Tuples are represented using parentheses:
   `(4,2,3,1)`, `('north','north','east')`, `(3,)`, or `()`.
   (Notice that a singleton requires a comma to distinguish it, syntactically, from standard use of parentheses. Actually, you can always have an "extra" comma after a final list, tuple, or set element.)

   (b) When it's not ambiguous, you can drop the parentheses!

```
>>> a = 1,2
>>> a
(1, 2)
```

   Wowza.

   (c) The `tuple(i)` constructor will build a tuple from any iterable source of values.

   (d) As with lists, you can index and slice tuples.

   (e) You can concatenate them, with `+`; you can replicate them with `*`:

```
>>> (1, 2)+(3, 4)
(1, 2, 3, 4)
>>> (1,) * 3
(1, 1, 1)
```

   (f) You test for membership in a tuple with `in`, `t.find(item)` finds an item within a tuple, and `t.count(item)` counts occurrences of `item` within `t`.

6. Left- and right-values.

   (a) A *left value* or `l-value` is an assignable object. It is any expression that may occur on the left side of an assignment. Variables are obvious l-values, but so are items in lists.

   (b) A *right value* or `r-value` is any expression that has a value that may appear on the right of an assignment. In python, everything is an r-value.

   (c) Traditionally, the underscore (`_`) is used as a place-holder for an l-value when we don't care about the result of the assignment.

   (d) In assignment, a tuple of l-values is, itself an l-value:

```
>>> (a,b,c) = (1,2,3)
>>> a,b,c = 1,2,3
>>> (a,b),_,c = (1,2,),9,3
```

Each of these effectively assigns a=1, b=2, and c=3.

(e) These complex assignments happen *in parallel*, so we can exchange values with:

```
>>> a,b = b,a
```

Really: it's tuple assignment!

(f) Here's Euler's algorithm for finding greatest common divisors:

```
def gcd(a,b):
    while a > 0:
        a,b = (b,a) if a > b else (b%a,a)
    return b
```

(g) Wowza: when an asterisk precedes a variable name used as an l-value, it means *assign this variable the remaining r-values as a list*. This is very powerful:

```
>>> car,*cdr = (1,2,3)
>>> cdr
[2, 3]
```

(From an old language, Lisp, the *car* of a list is the first element, and the *cdr* ("could-r") is what remains.) This asterisk notation can be used to assign the last formal parameter all the actual parameters that remain:

```
>>> def min(first,*args):
...   m = first
...   for x in args:
...     if x < m: m = x
...   return m
>>> min(31,331,21)
21
```

7. Tuples have comprehensions, too. They're surrounded by parens and, in most ways, are similar to list comprehensions.

⋆