Data Structures

UW CSE 190p

Summer 2012

```
>>> xs = range(3)

>>> xs = [1,2,3]

>>> xs = ['a','b','c']

>>> xs = [1, 'a', 3]

>>> xs = [[1,2,3], ['a','b','c']]  #  list of lists?

>>> xs = [x * 2 for x in range(3)]

>>> xs = [cels_to_faren(temp) for temp in measurements]

>>> warmdays = [temp for temp in msrmts if temp > 20]
```

List “Comprehensions”

```
ctemps = [17.1, 22.3, 18.4, 19.1]
```

Compare these two snippets for converting a list of values from celsius to farenheit:

```
ftemps = []
for c in ctemps:
  f = celsius_to_farenheit(c)
  ftemps.append(f)
```

```
ftemps = [celsius_to_farenehit(c) for c in ctemps]
```

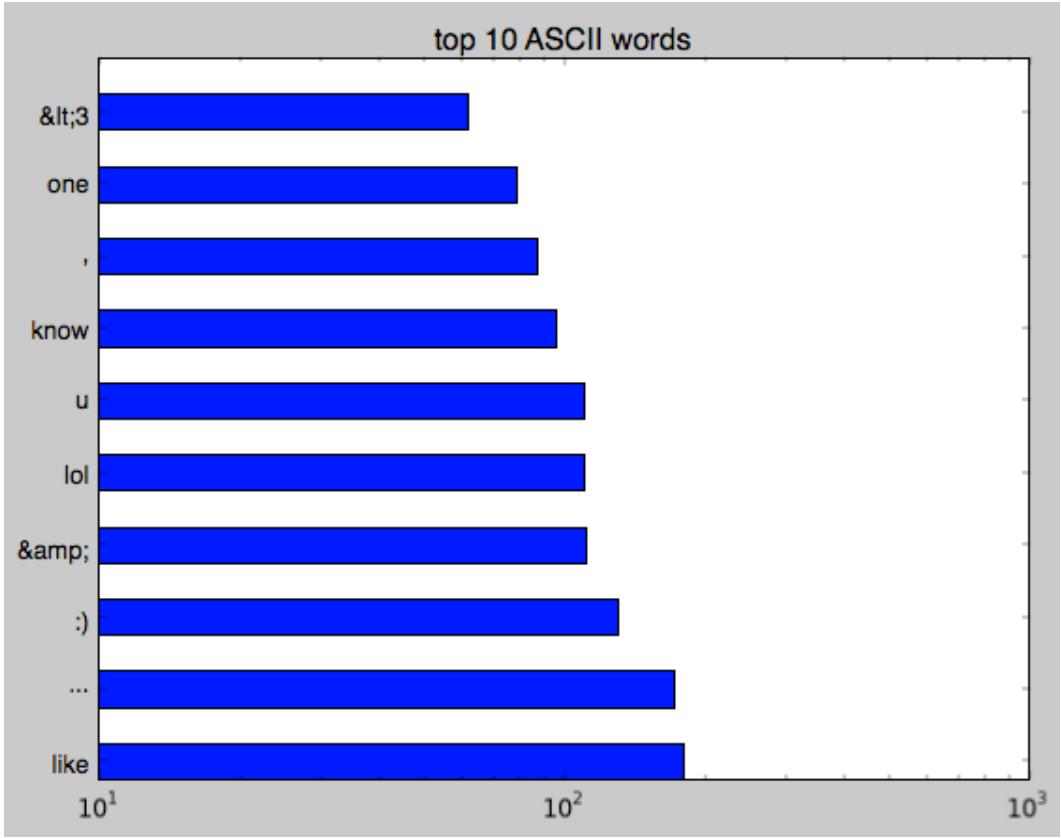
- **This syntax is VERY useful: It is usually shorter, more readable, and more efficient.**
- **Let it become second nature!**

From Last Class

We wrote this function:

```
def count_unique(doc):
  """return the number of unique words in  a list of strings"""
  unique_count = 0
  scratchpad = []
  for word in doc:
    if not word in scratchpad:
      unique_count = unique_count + 1
  return unique_count
```

New Problem: Top 10 Most Common Words



Exercise (5 min)

Sketch a function to return the top 10 most common words in a document

What data structure(s) will you use?

```
def top10(doc):
  """return a list of the top 10 most frequent words"""
  # initialize a histogram
  for word in doc:
    # if word in histogram
    #   increment its count
    # else
    #   add it to the histogram
  # search histogram for top 10
  # most frequent words
  # return result
```

First Attempt: Two lists

```
uniquewords = [‘lol’, ‘omg’, ‘know’]
counts = [45, 23, 12]
```

```
def top10(doc):
  """return a list of the top 10 most frequent words"""
  uniquewords = []
  counts = []
  for word in doc:
    if word in uniquewords:
      position = uniquewords.index(word)
      counts[position] = counts[position] + 1
    else:
      uniquewords.append(word)
  # now search for top 10 most frequent words…
```

A list of (count, word) pairs

```
>>> uniquewords = [ [45, 'lol'], [23, 'omg'], [12, 'know'] ]
>>> uniquewords[1][0]
```

```
def top10(doc):
  """return a list of the top 10 most frequent words"""
  uniquewords_with_counts = []
  for word in doc:
    match = [pair for pair in uniquewords if pair[1] == word]
    if match != []:
      pair = match[0]  # why is this line here?
      pair[0] = pair[0] + 1
    else:
      uniquewords.append([1, word])
  # now search for top 10 most frequent words
  uniquewords.sort()
  return uniquewords[0:10]
```

Digression: Lexicographic Order

Aaron	[1, 9, 9]
Andrew	[2, 1]
Angie	[3]

a	[1]
aaa	[1,1]
aaaaa	[1,1,1]

Mapping Values to Values

A list can be thought of as a (partial) function from integers to list elements.

```
>>> somelist = ['a','b','c']
>>> somelist[1]
'b'
```

We can say this list "maps integers to strings"

What if we want to map strings to integers? Or strings to strings?

Python Dictionaries

```
>>> phonebook = dict()
>>> phonebook["Alice"] = "212-555-4455"
>>> phonebook["Bob"] = "212-555-2211"

>>> atomicnumber = {}
>>> atomicnumber["H"] = 1
>>> atomicnumber["Fe"] = 26
>>> atomicnumber["Au"] = 79

>>> state = {"Atlanta" : "GA", "Seattle" : "WA"}
```

Python Dictionaries (2)

```
>>> atomicnumber = {}
>>> atomicnumber[“H”] = 1
>>> atomicnumber[“Fe”] = 26
>>> atomicnumber[“Au”] = 79
>>> atomicnumber.keys()
['H', 'Au', 'Fe']
>>> atomicnumber.values()
[1, 79, 26]
>>> atomicnumber.items()
[('H', 1), ('Au', 79), ('Fe', 26)]
>>> atomicnumber["Au"]
79
>>> atomicnumber["B”]
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    atomicnumber["B"]
KeyError: 'B'
>>> atomicnumber.has_key("B")
False
```

Top k with a dictionary

```
>>> uniquewords = { 'lol':45, 'omg':23, 'know':12 }
>>> uniquewords['omg']
```

```
def top10(doc):
  """return a list of the top 10 most frequent words"""
  uniquewords = {}
  for word in doc:
    if uniquewords.has_key('omg'):
      uniquewords['omg'] = uniquewords['omg'] + 1
    else:
      uniquewords['omg'] = 1
  # now search for top 10 most frequent words
  bycount = [(pair[1], pair[0]) for pair in uniquewords.items()]
  bycount.sort()
  return bycount[0:10]
```

This "default" pattern is so common, there is a special method for it.

Top k with a dictionary

```
>>> uniquewords = { 'lol':45, 'omg':23, 'know':12 }
>>> uniquewords['omg']
```

```
def top10(doc):
  """return a list of the top 10 most frequent words"""
  uniquewords = {}
  for word in doc:
    uniquewords['omg'] = uniquewords.set_default('omg', 0) + 1
  # now search for top 10 most frequent words
  bycount = [(pair[1], pair[0]) for pair in uniquewords.items()]
  bycount.sort()
  return bycount[0:10]
```

Types: some definitions and context

- Some historical languages were *untyped*
  - You could, say, divide a string by a number, and the program would continue.
  - The result was still nonsense, of course, and program behavior was completely undefined.
  - This was considered unacceptable
- Modern languages may be *staticly typed* or *dynamically typed*
  - "staticly typed" means that types are assigned before the program is executed
  - "dynamically typed" means that types are assigned (and type errors caught) at runtime
- Modern languages may be *strongly typed* or *weakly typed*
  - For our purposes, "weakly typed" means the language supports a significant number of implicit type conversions.
    - For example, (5 + "3") could trigger a conversion from "3" to 3
- For our purposes, Python can be considered
  - strongly typed
  - dynamically typed

Guess the Types

```
def mbar_to_mmHg(pressure):
    return pressure * 0.75006
```

Guess the Types

```
def abs(x):
  if val < 0:
    return -1 * val
  else:
    return 1 * val
```

Guess the Types

```
def debug(x):
  print x
```

Guess the Type

```
def index(value, some_list):
  i = 0
  for c in somelist:
    if c == value:
      return i
    i = i + 1
```

Duck Typing

"If it walks like a duck and it talks like a duck, then it must be a duck."

(Note: this analogy can be misleading!)

At runtime, the operands are checked to make sure they support the requested operation.

```
>>> 3 + "3"
>>> for i in 5:
...     print i
```

Takeaway

- Think about types when designing functions, when debugging, when reading code, when writing code….all the time.
- Ask yourself “What is this variable allowed to be?”
  - A list, or anything compatible with a for loop?
  - An integer, or anything that can be multiplied by an integer?

Mutable and Immutable Types

```
>>> def increment(uniquewords, word):
...     """increment the count for word"""
...     uniquewords[word] = uniquewords.setdefault(word, 1) + 1

>>> mywords = dict()
>>> increment(mywords, "school")
>>> print mywords
{'school': 2}

>>> def increment(value):
...     """increment the value???"""
...     value = value + 1
>>> myval = 5
>>> increment(myval)
>>> print myval
5
```

Tuples and Lists

```
def updaterecord(record, position, value):
    """change the value at the given position"""
    record[position] = value

mylist = [1,2,3]
mytuple = (1,2,3)
updaterecord(mylist, 1, 10)
updaterecord(mytuple, 1, 10)
print mylist
print mytuple
```

Why did they do this?

```
>>> citytuple = (“Atlanta”, “GA”)
>>> type(citytuple)
<type 'tuple’>
>>> citylist = [“Atlanta”, “GA”]
<type ’list'>
>>> weather[citytuple] = “super hot”
>>> weather[citylist] = “super hot”
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Answer: Performance. If the system knows for sure that a value can never be changed, it can cheat.

No really, why?

```
>>> citylist = [“Atlanta”, “GA”]
>>> weather[citylist] = “super hot”
>>> citylist[1] = “Georgia”
>>> weather[[“Atlanta”, “GA”]]
???
```

Mutable and Immutable Types

- Immutable
  - numbers, strings, tuples
- Mutable
  - lists and dictionaries