Last time

We finished the Python introduction we started the previous lecture, talking briefly about BeautifulSoup and about a networking module that lets us fetch data from various web services

Then, we embarked on a more formal discussion of the basic data types in Python (numbers, strings, lists and dictionaries) -- We went over things pretty completely, but left it to your own study if it starts to feel tedious

We ended with the briefest glimpse at control structures like if-statements, for- and while-loops and so on -- At very least we saw a small Python program that broke our recipes into words
Today

We are going to finish our introduction to Python, looking at dictionaries (a collection of key-value pairs) and some basic control-of-flow programming structures (for- and while- loops as well as if-else-elif statements)

Next, we’ll talk a bit about data representation, about how you might format your recipe data once you have the components extracted...
Trip report

1. Lunch with Leonard Richardson (author of Beautiful Soup) -- His project started because he wanted to create RSS feeds from arbitrary web sites and needed robust code for reading HTML and writing XML

2. Data Science Bunch! A group of us (some from industry, some from academia and even a representative from O'Reilly) met to talk about what is meant by the term “data science” and, if there is such a science, what it's principles might consist of

3. We finished a preliminary version of our NY Times/bit.ly/Twitter visualization -- This is a project with data artist Jer Thorp who has made some amazingly beautiful graphics around what we’re calling “cascades of activity”
OP-ED COLUMNIST; Another Pill That Could Cause a Revolution

By NICHOLAS D. KRISTOF
Python re-cap: Objects and expressions

Everything in Python is an object and with objects we structure both data and computations -- We say that Python is strongly typed in the sense that you can perform on objects only operations that are valid for its type.

Every object in Python has an identity, a type, and a value and we have seen various ways to explore these objects, from snooping around with dir() to directly asking for help() -- We have introduced several built-in types (numbers, strings, lists, files, dictionaries) and have shown them to be a powerful toolkit for working with data (soon we'll even build our own types!)

Next, we have seen numerous examples of expressions that create and process objects -- We've seen how to define strings, for example, how to create lists, how to create an object that represents a file, and how to add two numbers or concatenate two lists.

Finally, we've noted several examples in which the operators in expressions (the + or *, for example) behave differently if the operands are floats or integers or strings -- This is often called polymorphism, meaning the mechanics of an operation depends on the type of objects being operated on.
Python re-cap: Variables

When we **assign an object to a variable** (i.e., give it a name), with statements like, `x=3` or `y=[4, "no"]`, we are really **creating references** from the names “x” and “y” to the appropriate integer and list objects, respectively.

Because they’re just references, variables are generic in nature, lacking any type information or having any constraints on them -- **Variables simply refer to a particular object at a particular point in time**, meaning we can happily assign `x=3` and later in a program let `x="some text"`.

Put another way, **variables don’t have types, objects do** -- When we actually use a variable, that is, when a variable appears in an expression, it is immediately replaced with the object to which it currently refers.

(Formally, we say that **Python is a dynamically typed language**, meaning we don’t have to “declare variables” as we would in languages like Java, say.)
Python re-cap: Variables

As a program runs, we create many objects and you might start to wonder about the space, the amount of memory, they are taking up -- Python employs a process of garbage collection that cleans up unused objects as your program runs.

It does this by keeping track of all the references to an object and deleting them as soon as the last reference to an object is removed.

```python
>>> import sys

>>> x = "some text"

>>> sys.getrefcount(x)    # 1 larger than you think -- ref in func call
2

>>> y = x

>>> sys.getrefcount(x)
3

>>> z = y

>>> sys.getrefcount(x)
4

>>> x = 5                   # x now refers to the integer 5

>>> sys.getrefcount(y)
3
```
Python re-cap: Code structure

We’ve also started to see some of the basic organizational structure behind a Python program

1. Programs are composed of modules
2. Modules contain statements
3. Statements contain expressions
4. Expressions create and process objects
Python re-cap: Code structure

Running backward through this list...

4. We’ve seen expressions like math.sin(3.0) or “Hi there”

3. We’ve also seen various kinds of statements, including print statements and assignment statements

2. Assembling these statements into a file (ending with the suffix “.py”) will provide us with a module

1. And we’ve seen how programs can make use of other modules, how they expose their data and functions
Python re-cap: Code structure

In terms of layout of code, whereas R and many other languages use { }’s to bracket off blocks of code, Python relies on indentation -- In general, the first block of code must begin in the first column

Consecutive blocks have to be indented; the amount of indentation is up to you, but it has to be consistent

Aside from indentation, blank spaces are usually ignored in Python statements
Python re-cap: Mutability

A couple of concepts came up when we started to make assignments with lists; in particular we needed to distinguish between what you can do with a list (a mutable sequence) and what you can do with a string (an immutable sequence)

We can select pieces of both strings and lists (subsetting and slicing) but we can only change the value of the elements of a list; we cannot change the elements of a string “in place”
Mutable objects are those that can be changed once they are created (these include lists and dictionaries); and immutable objects cannot be changed after they’ve been created (these include strings and numbers) -- Ultimately, mutability is a function of an object’s type.

Python offers several “container” classes that hold “references” to other objects; some of these containers are mutable (lists and dictionaries) and some are not (tuples) -- Note: The concept of mutability extends to the “identities” of the contained objects.
Python re-cap: Mutability

In Python, the mutability of an object determines how things like assignments take place -- *Immutable objects are copied* for you when you start to work with them, making their behavior similar to what you will see in R

When dealing with mutable objects, on the other hand, we are really working with a *reference to the underlying object* -- This means that when we change the underlying object, *all the variables that point to the same object have had their values changed* as well

Let's see what this means for the simple case of a list, a mutable object...
# recall the issues with assignment...

```python
>>> somewords1 = ['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
>>> somewords2 = somewords1
>>> somewords2
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
```

# somewords1 and somewords2 now “reference” or point to the same object; therefore # making a change affects both...

```python
>>> id(somewords1)
140095094341856
>>> id(somewords2)
140095094341856
```

```python
>>> somewords1[1] = "disdain"
```

```python
>>> somewords1
['unbridled', 'disdain', 'for', 'globe', 'artichokes']
```

```python
>>> somewords2
['unbridled', 'disdain', 'for', 'globe', 'artichokes']
```
# we can force (nearly) R-like copying using an “empty” slice

```python
>>> somewords1 = ['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
>>> somewords2 = somewords1[:]
>>> somewords1
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
>>> somewords2
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']

>>> id(somewords1)
4598680
>>> id(somewords2)
4599480

# now when you make a change, it applies to just one...

>>> somewords1[1] = "not"
```

```python
>>> somewords1
['unbridled', 'disdain', 'for', 'globe', 'artichokes']
```

```python
>>> somewords2
['unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']
```
# now, make a list that contains a list

```python
>>> somewords = ['unbridled', 'enthusiasm']

>>> words1 = ['globe', 'artichokes', somewords]
```

# and copy words1 into words2 using an empty slice

```python
>>> words2 = words1[:]
```

# now, make some changes...

```python
>>> words1[0] = 'small'

>>> words1
['small', 'artichokes', ['unbridled', 'enthusiasm']]

>>> words2
['globe', 'artichokes', ['unbridled', 'enthusiasm']]
```

# no difference. but if we change the list contained in words1...

```python
>>> words1[2][1] = 'disdain'

>>> words1
['small', 'artichokes', ['unbridled', 'disdain']]

>>> words2
['globe', 'artichokes', ['unbridled', 'disdain']]

>>> somewords
['unbridled', 'disdain']
```

# the moral: an empty slice only makes a top-level copy; references to mutable
# objects are maintained
Aside: Makin’ copies

It’s important to understand what’s going on when you make assignments and copies; these are subtle issues that will come up (somewhat inevitably) in the form of hard-to-track-down bugs

If you want to make something more than just a top-level copy of an object, you can use the \texttt{copy} module...
Some dangling threads: The Python interpreter

It’s worth mentioning that there is at least one fantastic project that has provided enhancements to the Python interpreter -- IPython has some features that are more R-like (name completion, the ability to examine objects and source code, and session restoration, for example)

It also provides support for so-called interactive parallel computing -- This lets you take advantage of multicore systems, but do so in a way that is lightweight, not requiring a lot of extra knowledge or language extensions

Any takers?
IPython: an interactive computing environment

The goal of IPython is to create a comprehensive environment for interactive and exploratory computing. To support this goal, IPython has two main components:

- An enhanced interactive Python shell.
- An architecture for interactive parallel computing.

All of IPython is open source (released under the revised BSD license). You can see what projects are using IPython here, or check out the talks and presentations we have given about IPython.

IPython supports Python 2.5 and 2.6 officially; we do not yet have a Python 3 port. If you need to use Python 2.4, the 0.10 series probably works OK but has not been extensively tested with 2.4.

Citing IPython

Several of the authors of IPython are connected with academic and scientific research, so it is important to us to be able to show the impact of our work in other projects and fields.

If IPython contributes to a project that leads to a scientific publication, please acknowledge this fact by citing the project, you can see CitingIPython for a ready-made citation entry. We have a listing of projects using IPython, for which updates are always also welcome.

Announcements

- **Development Version**: what will be 0.11 is taking shape, but lots of changes are coming (including some that break compatibility). We would like your feedback, from either the nightly documentation or source tarball (you can also follow the trunk from GitHub).
- **Current Version**: 0.10.1, released on October 11, 2010.
- **Previous Version**: 0.10, released on August 4, 2009.
Some dangling threads: Sorting

First, the sorting code from last time -- I tossed up some code rather frivolously and didn’t explain it in sufficient detail

In Python, there are two ways to sort a list -- We can sort “in place” using the sort method of the list class (the actual list is changed -- remember it is a mutable object after all) or we can call a built-in function sorted that returns a sorted version of the list

Python has default choices for sorting depending on the kind of object stored in the list -- Numbers are sorted numerically, strings alphabetically (or lexicographically) and mixed types...
Aside: Mixed types

Essentially all sorting is done via a comparison function -- The default is `cmp()` and you can give it two objects, `cmp(x, y)` and see whether `x<y` (-1) or `x==y` (0) or `x>y` (1).

By default, `cmp()` will 1. Check if `x` and `y` are of the same type and if so compare and return the result, 2. If they are different types, they check to see if they are numbers and if so, perform numeric coercion if necessary to compare, 3. If either element is a number, then the other element is said to be larger, and 4. Otherwise the two are sorted alphabetically.

You can compare lists as well, working elementwise comparing matched elements using lexicographic ordering -- The same comparison rules are applied to each pair, except that if you reach the end of one of the lists the longer is said to be larger.
comment = "Perhaps it is nothing more than unbridled enthusiasm for globe artichokes"

words = comment.split(" ")

words
['Perhaps', 'it', 'is', 'nothing', 'more', 'than', 'unbridled', 'enthusiasm', 'for', 'globe', 'artichokes']

# default sort is alphabetically...

words.sort()

words
['Perhaps', 'artichokes', 'enthusiasm', 'for', 'globe', 'is', 'it', 'more', 'nothing', 'than', 'unbridled']

# make a new function that can be used to sort a list -- here
# we give an example in which we sort by the length of the string

def len_compare(x,y):
    if len(x)>len(y): return 1
    elif len(x)==len(y): return 0
    else: return -1  # this means x < y

# there's a lot here -- we defined a function called len_compare
# that takes two arguments; it's body is the indented code; and it
# returns 1 if string x is longer than string y, 0 if they have the
# same length, and -1 if y is longer than x

words.sort(len_compare)

words
['is', 'it', 'for', 'more', 'than', 'globe', 'Perhaps', 'nothing', 'unbridled', 'artichokes', 'enthusiasm']
Something new: Dictionaries

Dictionaries are like lists except that you index them by names instead of by numbers; in more technical parlance, **you access data with keys**

In the example that follows, we will be using strings as keys, but you can also use integers or any other immutable object.
# create a new dictionary

```python
>>> x = {"Whitman":"It will be the same old same old.","Brown":"I've taken the heat."}
```

```python
>>> type(x)
<type 'dict'>
```

# see its attributes...

```python
>>> dir(x)
[ '__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__',
  '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
  '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
  '__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems',
  'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

```python
>>> help(x)
```

# extract the "keys" from the dictionary

```python
>>> x.keys()
['Whitman', 'Brown']
```

# or ask if the dictionary contains a key using a dictionary method...

```python
>>> x.has_key("Brown")
True
```

```python
>>> x.has_key("Browkaw")
False
```

# ... or an operator

```python
>>> "Brown" in x
True
```
# create a new dictionary

```python
>>> x = {"Whitman":"It will be the same old same old.", "Brown":"I've taken the heat."}
>>> type(x)
<type 'dict'>
```

# see its attributes...

```python
>>> dir(x)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__nonzero__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

```python
>>> help(x)
```

# extract the "keys" from the dictionary

```python
>>> x.keys()
['Whitman', 'Brown']
```

# or ask if the dictionary contains a key using a dictionary method...

```python
>>> x.has_key("Brown")
True
>>> x.has_key("Browkaw")
False
```

# ... or an operator

```python
>>> "Brown" in x
True
```
# create a new dictionary
>>> x = {"Whitman":"It will be the same old same old.","Brown":"I’ve taken the heat."}

>>> type(x)
<type 'dict'>

# see its attributes...
>>> dir(x)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__setitem__', '__str__', 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']

>>> help(x)

# extract the "keys" from the dictionary
>>> x.keys()
['Whitman', 'Brown']

# or ask if the dictionary contains a key using a dictionary method...
>>> x.has_key("Brown")
True

>>> x.has_key("Browkaw")
False

# ... or an operator

>>> "Brown" in x
True
# dictionaries are mutable so they can be changed “in place”

```python
>>> x = {
    "Brown": [1, 2, 3],
    "Whitman": 'It will be the same old same old.'
}
```

# you can delete items...

```python
>>> del x["Obama"]
>>> x
{'Whitman': 'It will be the same old same old.'}
```

# ... or add them

```python
>>> x["Brokaw"] = "NBC"
>>> x
{'Brokaw': 'NBC', 'Whitman': 'It will be the same old same old.'}
```

# we can also extract just the values (returned in a list)

```python
>>> x.values()
['NBC', 'It will be the same old same old.']
```

# or the separate key-value pairs (again in a list... this time a list
# of tuples...

```python
>>> x.items()
[('Brokaw', 'NBC'), ('Whitman', 'It will be the same old same old.')]}
```

# a convenience function for extracting values

```python
>>> x.get("Brokaw")
'NBC'
>>> x.get("Couric")
# returns None type if there is no such key
>>> x.get("Couric","not here")
# or some value you prescribe
'not here'
```
# finally, building up a dictionary in pieces...

```python
>>> x = {}
>>> x[1] = {}
>>> x
{1: {}}
>>> x[1][2] = "Brown"
>>> x[1][5] = "Whitman"
>>> x
{1: {2: 'Brown', 5: 'Whitman'}}
>>> x[10] = {}
>>> x
{1: {2: 'Brown', 5: 'Whitman'}, 10: {}}
>>> x[10]["quote"]='It will be the same old same old.'
>>> x
{1: {2: 'Brown', 5: 'Whitman'}, 10: {'quote': 'It will be the same old same old.'}}
>>> del x[10]
>>> x
{1: {2: 'Brown', 5: 'Whitman'}}
>>> del x[1][5]
>>> x
{1: {2: 'Brown'}}
```
Now we’re in business...

We’re now in good shape to at least read through the code we introduced at the beginning of the session -- The `word_count.py` program is a little crude but it exercises almost all of the data types we’ve seen so far

It also introduces some “control of flow’ constructions that we’ll go over in the next couple of lectures -- You have enough at this point, however, to at least read what’s going on
#!/usr/local/bin/python

# counts words from a list of (sgml) recipe files
# counts only words in <p> tags

import sys, re
from BeautifulSoup import BeautifulStoneSoup

recipe_files = sys.argv[1:]
word_totals = {}

for recipe_file in recipe_files:
    bs = BeautifulStoneSoup(open(recipe_file))

    for p in bs.findAll("p"):  
        line = p.getText()
        line = re.sub("\s+"," ",line)
        line = line.strip()

        words = line.split(" ")
        words = [w.lower() for w in words]  # sexy! a list comprehension
        words = [re.sub("\W","",w) for w in words]  
        words = [w for w in words if w]

        for w in words:
            if not w in word_totals: word_totals[w] = 0

            word_totals[w] += 1

for w in word_totals: print word_totals[w], w
A little more structure: Control-of-flow

To be seriously dangerous, we need a couple basic constructions in Python; essentially these **direct the flow of operation of our programs**

So far (except for the for-loops from last lecture) we present Python a series of expressions and it executes them all, in order; **sometimes we may want to act conditionally** or we may want to repeat steps some number of times...
if-elif-else statements

Before we can talk meaningfully about if statements, we need to introduce one last built-in type a Boolean value; as you might expect this takes on the values True and False

You will encounter them when you consider the sizes of numbers or containment in strings and lists; you combine them using so-called Boolean operators

For convenience, any object can be used to create a Boolean value; nonzero numbers or a non-empty objects are True (why would this be helpful?)
x = True  # Boolean objects specifying, well, true and false
x = False

x = bool(1)

x = bool([])
# empty sequences '', [], () are all False, as is an empty dict {
True or False?

Python provides boolean operators (and, or, not) to combine logical operands; the and and or return objects that evaluate to true and false, not just the values 0 or 1 (or True or False)

The order in which you write down these operators may matter!

1. For an or, Python employs short-circuit evaluation; it stops after the first operand that evaluates to true (and returns the associated object)

2. For and, operations also stop once we know the result; it will either return the object associated with the first false operand, or it will return the rightmost object

(These operators function like && and || in R)
```python
>>> x = 5
>>> y = 7
>>> z = x > 3 and y==7

>>> z
True

>>> type(z)
<type 'bool'>

>>> x<10 or y>10 and not x+y<3
True

>>> x and []  # Boolean operators return objects that "evaluate" that evaluate to True or False
[]

>>> {} and y
{}

>>> x and y
7
```
>>> x = [1, 2, 3, 4]  # for lists and strings, comparisons are "lexicographic"
>>> y = [2, 3, 4, 5]
>>> x < y
True

>>> x[0] = 100
>>> x
[100, 2, 3, 4]
>>> y
[2, 3, 4, 5]
>>> x < y
False

>>> x[0] = 2
>>> x
[2, 2, 3, 4]
>>> x < y
True

>>> x = range(10)
>>> id(x)
140610700852112
>>> y = range(10)
>>> id(y)
140610700946752
>>> x == y
True
if–elif–else statements

Back to this basic construction; remember from our work with the for-loop that we had to be mindful of indentation and the same is true here...
>>> x = 'unbridled enthusiasm for globe artichokes'
>>> if "artichokes" in x:
...     print "yum"
... elif "trotting" in x:
...     print "away we go"
... else:
...     print "no comment"
... yum
>>> x=0
>>> if x: print "found!"  # if your code block is just one line, you can write it like this
...]
>>> x = 2.3
>>> if x: print "found!"
... found!
>>> x = {}
>>> if x: print "found!"
...]
>>> x = {"a":12,"b":"hi"}
>>> if x: print "found!"
... found!
for loops

Unfortunately, we have a few of these in our previous lectures and it’s time to spell things out.

In general, for loops circle over the associated block of statements; two statements, break and continue, can control flow through the loop.

Unique to Python is an optional else statement that is executed if it doesn’t exit by break-ing out of the loop.

```python
>>> x = [1, 2, 10, "Brown", "Whitman"]
>>> for i in x:
...     if isinstance(i, int):
...         y = i + 10
...         print y
...     ...
11
12
20
>>> for i in x:
...     if i == "Brown": break
...     print i
...     else:
...         print "not found"
1
2
10
>>> for i in x:
...     if i == "Fiorina": break
...     print i
...     else:
...         print "not found"  # the else is executed if
...         print "not found"  # the loop exits normally
1
2
10
Brown
Whitman
not found
```
List comprehensions

In many cases, we can dispense with looping and use an alternate construction that is a fair bit more readable -- in addition to being concise, they can also run much faster than their for-loop competitors (the iterations are performed at C language speed inside the interpreter, rather than with the manual Python code)

Below we give an example of a simple for-loop for incrementing the elements of a list along with the list comprehension counterpart

```python
>>> test = [3, 5, 7, 9]
>>> for i in range(len(test)):  # regular for-loop
...     test[i] += 100
... >>> test
[103, 105, 107, 109]
```

```python
>>> test = [3, 5, 7, 9]
>>> test = [t+100 for t in test]  # list comprehension
>>> test
[103, 105, 107, 109]
```
List comprehensions

The full list comprehension specification allows for a trailing test which will let you keep only certain elements of the list -- Again, this construction is both more readable and more efficient than a for-loop implementation (use it!)

```python
>>> test = range(100)

>>> test = [t for t in test if t%2]  # keep the odds
>>> test
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99]
```
while loops

The other kind of loop you will likely use repeatedly is a while loop.

This loop will execute a block of statements repeatedly until a test turns up false.

It also comes complete with break, continue and else statements.

```python
>>> from random import uniform

>>> # create a random walk using increments from [-1,-0.5] and [0.5,1] and look for first excursion beyond 10 (lame, I know)

>>> while x < 10:
...    inc = uniform(-1,1)
...    # take out (-0.5,0.5)
...    if inc < 0.5 and inc > -0.5: continue
...    x += inc  # shorthand
...    k += 1
...    if x < -10:
...        print "negative excursion"
...    else:
...        print x,k
...```

10.2058336958 206
#!/usr/local/bin/python

# counts words from a list of (sgml) recipe files
# counts only words in <p> tags

import sys,re
from BeautifulStoneSoup import BeautifulStoneSoup

recipe_files = sys.argv[1:]
word_totals = {}

for recipe_file in recipe_files:
    bs = BeautifulStoneSoup(open(recipe_file))
    for p in bs.findAll("p"):
        line = p.getText()
        line = re.sub("\s+"," ",line)
        line = line.strip()
        words = line.split(" ")
        words = [w.lower() for w in words]  # sexy! a list comprehension
        words = [re.sub("\W","",w) for w in words]
        words = [w for w in words if w]
        for w in words:
            if not w in word_totals: word_totals[w] = 0
            word_totals[w] += 1

for w in word_totals: print word_totals[w],w
Data representation

The text “Introduction to Data Technologies” by Paul Murrell is an excellent resource for introductory material on data formats (and a number of other concepts we’ll cover in this class!)

http://www.stat.auckland.ac.nz/~paul/ItDT/

The book is freely available from the web site above -- The URL also describes how you can get a 20% discount from the publisher!
Plain text formats

In some sense, the most basic form of data storage is as rows in a text file; **each row contains information on a separate observational unit**

The “data as rectangle” or tabular model that’s prevalent in most statistical models adds another twist -- Each row in the file contains the same number of measurements on the observational unit

A **fixed-width format** assigns the same number of characters to a given field in each row -- A **delimited format** uses a special character to indicate divisions between separate fields

In addition to the data, a plain text format will often contain **metadata** at the top of the file (the first line, say) that describes the fields -- this is often called a header
Turnstile Data

Key/Resources

Field Description
Remote Unit/Control Area/Station Name Key

Data Files

Saturday, October 09, 2010
Saturday, October 02, 2010
Saturday, September 25, 2010
Saturday, September 18, 2010
Saturday, September 11, 2010
Saturday, September 04, 2010
Saturday, August 28, 2010
Saturday, August 21, 2010
Saturday, August 14, 2010
Saturday, August 07, 2010
Saturday, July 31, 2010
Saturday, July 24, 2010
Saturday, July 17, 2010
Saturday, July 10, 2010
Saturday, July 03, 2010
Saturday, June 26, 2010
Saturday, June 19, 2010
Saturday, June 12, 2010
Saturday, June 05, 2010
Saturday, May 22, 2010
Field Description

C/A, UNIT, SCP, DATE1, TIME1, DESC1, ENTRIES1, EXITS1, DATE2, TIME2, DESC2, ENTRIES2, EXITS2, DATE3, TIME3, DESC3, ENTRIES3, EXITS3, DATE4, TIME4, DESC4, ENTRIES4, EXITS4, DATE5, TIME5, DESC5, ENTRIES5, EXITS5, DATE6, TIME6, DESC6, ENTRIES6, EXITS6, DATE7, TIME7, DESC7, ENTRIES7, EXITS7, DATE8, TIME8

C/A = Control Area (A002)
UNIT = Remote Unit for a station (R051)
SCP = Subunit Channel Position represents an specific address for a device (02-00-00)
DATEn = Represents the date (MM-DD-YY)
TIMEn = Represents the time (hh:mm:ss) for a scheduled audit event
DESCn = Represent the "REGULAR" scheduled audit event (occurs every 4 hours)
ENTRIESn = The comulative entry register value for a device
EXISTn = The cumulative exit register value for a device

Example:
The data below shows the entry/exit register values for one turnstile at control area (A002) from 03/21/10 at 00:00 hours to 03/28/10 at 20:00 hours

A002, R051, 02-00-00, 03-21-10, 00:00:00, REGULAR, 002670738, 000917107, 03-21-10, 04:00:00, REGULAR, 002670738, 000917107, 03-21-10, 08:00:00, REGULAR, 002670746, 000917117, 03-21-10, 12:00:00, REGULAR, 002670790, 000917166, 03-21-10, 16:00:00, REGULAR, 002670932, 000917204, 03-21-10, 20:00:00, REGULAR, 002671164, 000917230, 03-22-10, 00:00:00, REGULAR, 002671181, 000917231, 03-22-10, 04:00:00, REGULAR, 002671181, 000917231, 03-22-10, 08:00:00, REGULAR, 002671220, 000917324, 03-22-10, 12:00:00, REGULAR, 002671364, 000917640, 03-22-10, 16:00:00, REGULAR, 002671651, 000917719, 03-22-10, 20:00:00, REGULAR, 002671164, 000917230, 03-22-10, 00:00:00, REGULAR, 002671181, 000917231, 03-22-10, 04:00:00, REGULAR, 002671181, 000917231, 03-22-10, 08:00:00, REGULAR, 002671220, 000917324, 03-22-10, 12:00:00, REGULAR, 002671364, 000917640, 03-22-10, 16:00:00, REGULAR, 002671651, 000917719, 03-22-10, 20:00:00, REGULAR, 0026712430, 000917789, 03-23-10, 00:00:00, REGULAR, 002672473, 000917795, 03-23-10, 04:00:00, REGULAR, 002672474, 000917795, 03-23-10, 08:00:00, REGULAR, 002672516, 000917876, 03-23-10, 12:00:00, REGULAR, 002672652, 000917934, 03-23-10, 16:00:00, REGULAR, 002672879, 000917996, 03-23-10, 20:00:00, REGULAR, 002673636, 000918079, 03-24-10, 00:00:00, REGULAR, 002673683, 000918079, 03-24-10, 04:00:00, REGULAR, 002673683, 000918079, 03-24-10, 08:00:00, REGULAR, 002673722, 000918171, 03-24-10, 12:00:00, REGULAR,
CSV, comma-separated values

An extremely common form of plain text, delimited format is **CSV**; it’s not strictly speaking a standard, and yet it has sufficient history to make it a well-defined quantity

- As its name suggests, the delimiter is a comma and so commas are special characters: **If a field contains a comma, it must be separated by double quotes**

- This, of course, means that the double quote character (") is also special: **If a field contains a double quote, it must also be contained in ““s and the internal double quotes must be escaped** (replacing “ with “”)

- The header information is contained in a single line, the first line of the file

We often refer to this kind of data storage as a “flat file” meaning that it is a single table -- we will see shortly another model in which the data are arranged in a series of tables
Alternatives

CSV files are extremely popular and you will often get data from collaborators in various departments in this format -- Both R and Python have special tools for reading these data.

It's not hard to find the limit of these formats for describing complex data types -- Also we often want to process data automatically, implying that we need to be able to understand its structures without a lot of external meta-data.

Take for example...
See who’s here

Friends and industry peers you know. Celebrities you watch. Businesses you frequent. Find them all on Twitter.

Top Tweets

OGOchoCinco Having a boyfriend or girlfriend is simply practice on being able to discipline yourself on being with 1 individual, am i correct?
about 1 hour ago

GreatestQuotes "If I have the belief that I can do it, I shall surely acquire the capacity to do it even if I may not have it at the beginning." – Gandhi
about 1 hour ago

AngelaSimmons First examine your own life before looking into what's wrong with mine. ❤
about 1 hour ago
As you move around Twitter, your browser is requesting, receiving and displaying web pages -- pages that are described using **HTML** (the HyperText Markup Language)

HTML is a plain text format, but it includes special annotation or markup that helps web browsers (primarily) present text and graphics -- the markup includes “tags” like `<p>` to indicate the start of a paragraph and `</p>`, the end of a paragraph

You can see how any page is coded by selecting the “View Source” option in your favorite browser; what you will realize quickly is that the markup underneath most pages on the web is not written (directly) by humans

Here is a simple example of the tweets posted by my pal Nick Bilton; first how the browser represents the page and then what (a portion of) the HTML underneath it looks like...
Site allows you to give it an article URL & it creates a summarized version of the content: http://tldr.it

A UV light cellphone sanitizer. Kills "Strep, E.Coli, Salmonella & Listeria" – http://amzn.to/d3mDsJ

@richardfeindel Thanks Richard! Appreciate the kind Tweet! Don't forget to rate it on Amazon if you have a chance. Glad you enjoyed it.

Video: A traffic stop in Russia & a scared traffic cop: http://j.mp/clIKGoW

Off to Boston this morning to speak at the Boston Book Fair with Nicholas Negroponte and Kevin Kelly, two of my tech idols. Yippee!

With no video cameras allowed, Twitter reports trial for Connecticut Murders: http://j.mp/97wzor
Site allows you to give it an article URL & it creates a summarized version of the content:

http://tldr.it

3 minutes ago

via Echofon
In general, markup (or markup languages) are a kind of notational device that describes how text is structured or how it should be displayed; it’s fair to say that any document consists of content and markup.

You see hints of markup whenever the text in an electronic document (as viewed in your browser or word processing program, say) changes color or size or font.

HTML (or “web”) pages, TeX and Word 2007 documents are just a few examples where we find powerful “markup languages”
Broadly speaking, the information provided by a markup language falls into two categories: **structure and formatting**

As a reader, you are often most aware of the role that markup plays in formatting; however, an over-emphasis on the look of a document (read data) might, in fact, make it hard to share or reuse.
SGML, the Standard Generalized Markup Language, started with the idea that tags in a markup language should focus on structure and identify “the role of the data element”; SGML separates these roles from formatting instructions.

SGML is not a markup system by itself; instead, think of it as a programming language to build working programs rather than a program by itself.

SGML grew up in the late 1960s at a time when companies began using computers for document processing; it quickly became obvious that any markup should include descriptive, humanly readable information the nature and role of every element in the document.

The first working system that employed this notion was developed at IBM by Charles Goldfarb, Edward Mosher, and Raymond Lorie.

http://www.webreference.com/dlab/books/html/
HTML and SGML

Tim Berners-Lee created HTML as “a mechanism that would describe the processes going on within his browser in a form that could be safely transmitted over the Internet”; as such, we find simple, generic document elements like headings and paragraphs and lists.

As of Version 2.0, HTML was written as an SGML application; and yet the document descriptions was still somewhat loose, making it hard for people to extract higher-level information from web pages.

Therefore, documents written in HTML are really meant for a browser to process, with markup to indicate paragraphs and headings and lists; we’ve found that there’s nothing about HTML that would tell us much more about the structure of the data it contains.

For example, how would we extract the content of Nick’s tweets from the HTML source? What the times they were posted? If we’re lucky, the author of the HTML will give us clues.
Retrieving and Semantically Integrating Heterogeneous Data from the Web

Martin Michalowski, José Luis Ambite, Snehal Thakkar, and Rattapoom Tuchinda, University of Southern California

Craig A. Knoblock, University of Southern California and Fetch Technologies

Steve Minton, Fetch Technologies

The Semantic Web promises seamless integration of heterogeneous data from distributed sources, letting agents (human users or automated programs) perform sophisticated and detailed analyses of this data. An agent would send a query, expressed in terms of its preferred ontology (schema), to a system that would then find and integrate

Building Finder uses
Semantic Web technologies to integrate different data types from various online data sources.

The application’s use of the RDF and RDF Data Query Language makes it usable by computer agents as well as human users.

the relevant data from multiple sources and return it using the agent’s ontology.

Before achieving this vision, however, we must address several challenges. We need technologies to integrate data described in different ontologies, for example, as well as different types of data, such as images or structured data. In addition, a Semantic Web-based system must recognize when different objects at different sites denote the same real-world entity. Other challenges include efficiently querying distributed information and converting legacy data in traditional databases and Web sites (HTML) into more semantic representations such as RDF.

Building Finder is a running application that showcases our approach to meeting these challenges. The application integrates satellite imagery, geospatial data, and structured and semistructured data from various online data sources using Semantic Web technologies. Users can query an integrated view of these sources and request Building Finder to accurately superimpose buildings and streets obtained from various sources on satellite imagery. The data sources integrated by Building Finder are heterogeneous not only in terms of the data, but also in terms of how the application accesses the sources.

Building Finder overview

Building Finder helps users obtain satellite imagery, street information, and building information about an area. Users can request that Building Finder superimpose the building information on the satellite imagery, or they can click on a particular building or house to obtain detailed information from property tax records or a white page directory. Similarly, Building Finder can superimpose street information or provide detailed information about a house. Building Finder accesses satellite imagery from Microsoft’s Web service TerraService (http://terra.nasa-usa.com); streets from the US Census Bureau’s Tigerline files, which are available as a database hosted at the University of Southern California; property tax information from the Los Angeles County Assessor’s property tax information Web site; and residence information from the Yahoo White Pages Web site (http://people.yahoo.com).

What makes Building Finder even more attractive is that users can navigate through the Building Finder interface manually or have agents query the application using RDF Data Query Language (RDQL) queries and obtain results in RDF.

As Figure 1 shows, our GUI consists of an input form and an image. The input form lets users specify attributes the system will use to formulate RDQL queries and retrieve the URL as well as additional
Much information on the Web is formatted for human readers, not machines. Software wrappers let programs or agents retrieve and translate data from Web sources into a format the software can easily manipulate. Building Finder relies on wrappers to provide the needed data on a per-query basis because the breadth and depth of queries prevents us from storing or caching all data locally.
XML, the EXtensible Markup Language is a descendant of SGML which is billed as being easier to implement

Like SGML, it allows you to build descriptions of data types; it focuses on what data are, their basic structure and is intended for use in transportation, storage and processing

Which brings us back to Twitter; Twitter publishes a web API (Application Programming Interface) which describes how programmers can make use of their services programmatically, publishing data in XML format

To understand the implications of a better format for data exchange on the web, we have to take a small detour...
... into the clouds
The cloud

In the last couple of years, a concept has emerged (no matter how fuzzy) that the various elements of computing (data, software and “platforms”) can be provided as a kind of service.

“The cloud” is a metaphor for a model of computing which uses the Internet to deliver these services.
1. **SaaS**

   This type of cloud computing delivers a single application through the browser to thousands of customers using a multitenant architecture. On the customer side, it means no upfront investment in servers or software licensing; on the provider side, with just one app to maintain, costs are low compared to conventional hosting. Salesforce.com is by far the best-known example among enterprise applications, but SaaS is also common for HR apps and has even worked its way up the food chain to ERP, with players such as Workday. And who could have predicted the sudden rise of SaaS "desktop" applications, such as Google Apps and Zoho Office?

2. **Utility computing**

   The idea is not new, but this form of cloud computing is getting new life from Amazon.com, Sun, IBM, and others who now offer storage and virtual servers that IT can access on demand. Early enterprise adopters mainly use utility computing for supplemental, non-mission-critical needs, but one day, they may replace parts of the datacenter. Other providers offer solutions that help IT create virtual datacenters from commodity servers, such as 3Tera's AppLogic and Cohesive Flexible Technologies' Elastic Server on Demand. Liquid Computing's LiquidQ offers similar capabilities, enabling IT to stitch together memory, I/O, storage, and computational capacity as a virtualized resource pool available over the network.

3. **Web services in the cloud**

   Closely related to SaaS, Web service providers offer APIs that enable developers to exploit functionality over the Internet, rather than delivering full-blown applications. They range from providers offering discrete business services -- such as Strike Iron and Xignite -- to the full range of APIs offered by Google Maps, ADP payroll processing, the U.S. Postal Service, Bloomberg, and even conventional credit card processing services.

4. **Platform as a service**

   Another SaaS variation, this form of cloud computing delivers development environments as a service. You build your own applications that run on the provider's infrastructure and are delivered to your users via the Internet from the provider's servers. Like Legos, these services are constrained by the vendor's design and capabilities, so you don't get complete freedom, but you do get predictability and pre-integration. Prime examples include Salesforce.com's Force.com, Coghead and the new Google App Engine. For extremely lightweight development, cloud-based mashup platforms abound, such as Yahoo Pipes or Dapper.net.

5. **MSP (managed service providers)**

   One of the oldest forms of cloud computing, a managed service is basically an application exposed to IT rather than to end-users, such as a virus scanning service for e-mail or an application monitoring service (which Mercury, among others, provides). Managed security services delivered by SecureWorks, IBM, and Verizon fall into this category, as do such cloud-based anti-spam services as Postini, recently acquired by Google. Other offerings include desktop management services, such as those offered by CenterBeam or Everdream.

6. **Service commerce platforms**

   A hybrid of SaaS and MSP, this cloud computing service offers a service hub that users interact with. They're most common in trading environments, such as expense management systems that allow users to order travel or secretarial services from a common platform that then coordinates the service delivery and pricing within the specifications set by the user. Think of it as an automated service bureau. Well-known examples include Rearden Commerce and Ariba.

7. **Internet integration**

   The integration of cloud-based services is in its early days. OpSource, which mainly concerns itself with serving SaaS providers, recently introduced the OpSource Services Bus, which employs in-the-cloud integration technology from a little startup called Boomi. SaaS provider Workday recently acquired another player in this space, CapeClear, an ESB (enterprise service bus) provider that was edging toward b-to-b integration. Way ahead of its time, Grand Central -- which wanted to be a universal "bus in the cloud" to connect SaaS providers and provide integrated solutions to customers -- flamed out in 2005.
Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.

Amazon EC2’s simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon’s proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Amazon EC2 provides developers the tools to build failure resilient applications and isolate themselves from common failure scenarios.
Welcome to Google App Engine

Run your web applications on Google's infrastructure.

Google App Engine enables developers to build web applications on the same scalable systems that power our own applications.

No assembly required.
Google App Engine exposes a fully-integrated development environment.

It's easy to scale.
Google App Engine makes it easy to design scalable applications that grow from one to millions of users without infrastructure headaches.

It's free to get started.
Every Google App Engine application will have enough CPU, bandwidth, and storage to serve around 5 million monthly pageviews.

This is a preview release of Google App Engine. For now, applications are restricted to the free quota limits. To get started, sign in to Google App Engine with your Google Account, or explore our documentation to download our SDK and learn about what we're working on.
Web services

While the cloud is a big (lofty?) concept (and an important development for those interested in computing), we’ll focus on one part of the vision, so-called web services

At the heart of the enterprise are data formats -- You can think of a web service as a web page “meant for a computer to request and process; more precisely, it is a web page that is meant to be consumed by an autonomous program as opposed to a web browser or similar UI tool”

When you access Twitter, say, through your browser, you are requesting and viewing web pages, HTML files that include the data we’re interested in reading -- Technically, however, HTML pages can be hard to parse (identify the data hidden inside) and hard to deal with programmatically

With web services, better (or actual) data formats offer you the ability to build on and reuse services “mash-up” style*

*Although I’m not sure the hip kids are saying “mash-up” any longer
API Documentation

REST API · Streaming API · Search API

Twitter exposes its data via an Application Programming Interface (API). These documents are the official reference for that functionality.

Getting started with the Twitter API is easy. Jump right into the resource documentation or read some of the fine literature below.

As we prepare this new developer portal, you might find some pieces of documentation missing. Consider looking on the Twitter API Wiki if you can't find what you're looking for while we are in transition.

Guidelines and Terms

- Developer Guidelines, Rules of the Road, & Terms of Service
- Display Guidelines
- Geo Developer Guidelines

Authentication

- Which authorization path should I choose?
- Authentication
- Transitioning from Basic Auth to OAuth
- OAuth Libraries
- OAuth FAQ
- Overview of “Sign in with Twitter”

Timeline resources

- statuses/public_timeline
- statuses/home_timeline
- statuses/friends_timeline
- statuses/user_timeline
- statuses/mentions
- statuses/retweeted_by_me
- statuses/retweeted_to_me
- statuses/retweets_of_me

Tweets resources

User resources

Trends resources

Local Trends resources

List resources

List Members resources

List Subscribers resources

August 31, 2010: Basic Auth has been deprecated. All applications must now use OAuth. Read more »
<statuses type="array">
  <status>
    <created_at>Mon Oct 18 18:31:59 +0000 2010</created_at>
    <id>27758944463</id>
    <text>
      Site allows you to give it an article URL & it creates a summarized version of the content: http://tlgr.it
    </text>
    <source>
      <a href="http://www.echofon.com/" rel="nofollow">Echofon</a>
    </source>
    <truncated>false</truncated>
    <in_reply_to_status_id/>
    <in_reply_to_user_id/>
    <favorited>false</favorited>
    <in_reply_to_screen_name/>
    <retweet_count/>
    <retweeted>false</retweeted>
  </status>
  <user>
    <id>1586501</id>
    <name>Nick Bilton</name>
    <screen_name>nickbilton</screen_name>
    <location>Brooklyn, NY</location>
    <description>
    </description>
    <profile_image_url>
      http://a3.twimg.com/profile_images/1106643215/Screen-shot-2010-08-19-at-1.47.09-PM_normal.jpg
    </profile_image_url>
    <url>http://www.nickbilton.com</url>
    <protected>false</protected>
    <followers_count>25463</followers_count>
    <profile_background_color>000000</profile_background_color>
  </user>
</statuses>
Data formats

On the previous slide, we’ve requested and XML file; like HTML, the markup in XML consists of tags; in this case, the tags are a fair bit more descriptive than paragraph or heading, but instead refers to the structure of the data.

Consider, for example, the status tag; it contains a number of nested tags that describe the time the Tweet was submitted to the system (created_at), some details about the user (name, location) and the message itself (text).

As an aside, the FireFox browser is a reasonably good tool for looking at XML documents; it generates the nested format automatically (try typing the same address into a Safari browser).
APIs

You might (reasonably) ask how one would know that you could ask for these different kinds of pages; and whether this is the end of the story about web services.

Twitter spells out its how you are supposed to make use of its services through its Application Programming Interface.

* RESTful services are those which adhere to the principles of REST (Representational State Transfer) -- a topic we might return to later in the quarter.
An **API abstracts the services** (objects or classes or methods or functions or procedures or ... ) Twitter offers so that we can make use of them in a programmatic way.

Twitter’s API for accessing public data is very simple; as you can see on the right, you are just requesting a document from their web server.
Invoking the service

A few slides back, we simply typed the name of the service into a browser; it is, of course, possible to access this service via a command in Unix.

Curl is a simple tool for transferring files with URL syntax via protocols like HTTP, FTP, SCP and so on -- Let's give it a try...

http://curl.haxx.se/
% curl http://twitter.com/statuses/public_timeline.xml > ptl.xml

  % Total    % Received  % Xferd  Average Speed   Time    Time     Time  Current
        Dload  Upload   Total   Spent    Left  Speed
100  45265  100  45265  0  0  82770  0  --:--:--  --:--:--  --:--:--   99k

% head ptl.xml

<?xml version="1.0" encoding="UTF-8"?>
<statuses type="array">
  <status>
    <created_at>Mon Oct 18 21:00:11 +0000 2010</created_at>
    <id>27768865000</id>
    <text>tipo, msn &#233; t&#227;o aberto n&#233;</text>
    <source>web</source>
    <truncated>false</truncated>
    <in_reply_to_status_id/>
    <in_reply_to_user_id/>
  </status>
</statuses>
Invoking the service, an aside

Over the course of the quarter, we will find several ways to access network resources -- Python and R both provide for similar functionality (RcURL being one we’ll exercise) -- Once we embed the ability to call services in a program, the data we have at our disposal and the kinds of problems we can tackle grows exponentially

We’ve also painted a fairly simple view of the Twitter API as Twitter now employs an authentication model that requires you submit, in essence, credentials so that you can pull data -- The authentication hurdle (required to make more than 20 requests in an hour, I believe) is not tall, just beyond what I want to get into today
One issue with the plain text formats we’ve seen so far is that they don’t contain information about how data values are arranged -- as we have started to see, XML employs its markup to label data

Murrell points out that one of the advantages to XML is that it is, essentially, computer code, and, as such, can be read or parsed more directly -- assuming, of course, you observe its syntax

XML documents are made up of elements or nodes, and an element is just a pairing of a tag and the data it contains; our Twitter data consists of a set of status elements, each one, in turn, containing elements that describe the user’s ID, the time they, um, tweeted and the content of their message (to name a few)

Each tag can also contain one or more attributes associated with the element; the statuses tag at the top of the file, for example, has an attribute type that takes on the value “array”
An XML document is said to be well-formed if it obeys certain syntax rules:

- Tag names are case-sensitive (unlike HTML) and so the start and end tags for a node must match exactly.
- Tag names can contain letters, numbers and other characters, but they cannot start with a number, with punctuation or with the letters xml (in any case) -- names also cannot contain spaces and you cannot leave space between the “<“ and the start of the tag name.
- A node needs both an opening and a closing tag, unless its empty -- meaning it is of the form `<tagname/>`.
- Tags must nest properly.
- All attributes must be of the form `name = “value”`.
- Since `< and > are special characters, they need to be escaped when they appear in the document using `&lt;` and `&gt;` -- the same goes for “(`&quot;`) and ‘(‘`&apos;`) and & (`&amp;`).
- The document must have a root node, the parent to all the other elements.
- Comments in XML are of the form `<!-- a comment -->`. 
XML

Finally, it’s good practice to start the XML document with an indication that the file is, in fact, an XML document, which version of XML is being used (yes, there can be flavors) and finally what encoding is used

The Twitter document you downloaded, for example, is using XML version 1.0...
As we indicated earlier, there is only one root or document node in the tree, and all other nodes are contained within it -- you can imagine the other nodes being descendants of the root.

This, together with the nesting properties of XML implies a tree structure to these documents -- the relationships between nodes being characterized as parents, ancestors, children and siblings.

We also refer to terminal nodes as leaf nodes -- by definition, content always falls in the leaf nodes.
XML summary

As a data format XML has much to recommend it -- it is said to be self-describing and human-readable; it is easily generated and parsed programmatically (unlike finding content in HTML); and its standard is widely adopted

It is a technology, however, not without its detractors -- many find XML documents overly verbose and, as a result, hard to read; the wordiness adds to file sizes and some label it as a heavy format
XML specifications for recipes

Not surprisingly, there are a number of XML schema describing recipes -- A simple web search will turn up many, each with differing complexities.

For example, Aaron Straup Cope (a speaker in our seminar series next quarter!) has one from 2005 -- We present pieces of it to give you a sense of how much detail you can provide.
“I would also remind readers that reliance on precise recipes alone can be a trap. “The dangerous person in the kitchen,” wrote Marcel Boulestin, “is one who goes rigidly by weights, measurements, thermometers and scales.”

“If all recipes were written on these lines there would be no end to them. Nobody would use cookery books. They would be too dull, too forbidding, and too bulky to handle. To specify therefore “enough oil to cover the bottom of your saucepan” or “about a teacup of olive oil” is a short cut. It is also an indication that a precise quantity is not of great moment. Except for sauces, one does not often measure oil by tablespoons. One pours it out of a bottle into the pan One uses one's eye and one's loaf.”

Elizabeth David
<requirements>
  <ingredients>
    <set>
      <name>
        <common>The topping</common>
      </name>
      <ing>
        <amount>
          <quantity>
            <n type="int" value="2"/>
          </quantity>
          <measure>
            <unit content="cup"/>
          </measure>
        </amount>
        <item>sour cream</item>
        <detail>room temperature</detail>
      </ing>
      <ing>
        <amount>
          <quantity>
            <n type="frac" value="1/3"/>
          </quantity>
          <measure>
            <unit content="cup"/>
          </measure>
        </amount>
        <item>sugar</item>
      </ing>
      ...
    </set>
  </ingredients>
</requirements>
<directions>
  ...
  
  <stage>
    <name>
      <common>The topping</common>
    </name>
    <step>
      <para>
        Whisk sour cream, sugar and vanilla to combine. Spread over warm cheesecake.
      </para>
    </step>
    <step>
      <para>
        Return to oven, and bake until just set, 5 minutes. Cool on wire rack.
      </para>
    </step>
    <step>
      <para>
        Remove side of pan, and chill for a few hours or overnight.
      </para>
    </step>
  </stage>
  
</directions>
Data formats

XML is not the only data format out there; and with the advent of client-side tools like JavaScript (a language that runs in your browser and was originally meant to let programmers work with “pages” displayed by the Netscape Navigator; what kinds of objects might this language “expose”? What methods?)

JSON (JavaScript Object Notation) is billed as a “light-weight data-interchange format that is easy for humans to read and write”; why might a program running in your browser need to send and receive data?

As a format, JSON uses conventions that are familiar to users of languages like C or, as luck would have it, Python; here’s what you get when you request the Twitter public timeline “page” in JSON*

* http://twitter.com/statuses/public_timeline.json
Look familiar?
Introducing JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In JSON, they take on these forms:

An object is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).
As you might expect, a JSON object has a (relatively) direct translation into Python built-in types (numbers, strings, dictionaries, lists) -- For this reason, it is exceedingly popular as a tool for storing data

As we will see in a later lecture, there are also very efficient databases for storing, indexing and retrieving JSON strings -- One such offering is MongoDB, something we’ll work with once our recipes are done

How might this help us?
% curl http://twitter.com/statuses/public_timeline.json > ptl.json

% python

Python 2.7 (r27:82500, Oct 10 2010, 16:27:47)
[GCC 4.4.3] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> import json

>>> f = open("ptl.json")

>>> tweets = json.loads(f.readline())

>>> type(tweets)
<type 'list'>

>>> type(tweets[0])
<type 'dict'>

>>> tweets[0].keys()
['favorited', 'contributors', 'truncated', 'text', 'created_at', 'retweeted', 'coordinates', 'source', 'in_reply_to_status_id', 'in_reply_to_screen_name', 'user', 'place', 'retweet_count', 'geo', 'id', 'in_reply_to_user_id']

>>> tweets[0]['text']
u'Agora sim! Tudo bem, alvinegra? Mudou de foto, n\xe9? Gostei! ;) / @_OMaisQuerido Algu\xe9m ai?'

>>> original = json.dumps(tweets)  # convert it back to a string (and write to a file, say)