I’m a big proponent of the use of multiple programming languages: use different languages for different types of tasks.

Statisticians, in particular, should be proficient in some “scripting language” (e.g., Perl, Python, or Ruby). These types of languages give you far more flexibility for manipulating data files.

I’ve long used Perl, but I’ve switched to Ruby, and I’m trying to also be proficient in Python. I prefer Ruby to Python, but Python is much more widely used, and so if you’re going to just learn one such language, learn Python.
Why python?

- Manipulating data files
- Simulations using others’ programs
- Web-related stuff
- Alternative to R for data analysis and graphics
- Jupyter notebooks

For statisticians, the most important use of Python is for the manipulation of data files. This sort of script language is great for manipulating text, and data files are mostly plain text files.

In addition, I find a scripting language critical for performing simulations to evaluate others’ command-line-based programs. They’re also good for web-related stuff.

Python can also serve as an alternative to R for data analysis and graphics. And Jupyter notebooks are a big deal for reproducible research (and they can be used more broadly than Python).
Python 2 vs Python 3

- Many people are using Python version 2.7
- Python 3 was introduced in 2008
  - A number of large changes
  - Some important Python programs haven’t been ported
  - Few people seem to be using it day-to-day
- Ideally, go straight for Python 3
  - But be aware of differences

The biggest annoyance about Python is the two competing versions, Python 2 and Python 3. For now, you should probably stick with Python 2.

Python 3 is much better than Python 2, but it hasn’t penetrated the Python community sufficiently. But it’s definitely getting better.

I’m still a bit torn about whether to recommend Python 2 or Python 3 when you’re starting out, but at this point it probably is better to go straight for Python 3, but be aware of the differences as they may bite you now and then.
Installing Python

- On Mac or Unix, Python 2 should be pre-installed
  ```bash
  python --version
  ```
- For Windows (or to be current, or to alternate between Python 2 and 3), install Anaconda
  Includes NumPy, SciPy, Pandas, iPython, Matplotlib, …
  [continuum.io/downloads](http://continuum.io/downloads)

When you’re just starting to learn, you can just stick with the pre-installed version of Python, if you are on some flavor of Unix.

Long term, I recommend Anaconda, which is an easy-to-install Python with basically all of the scientific packages you’d want. Installing these by hand seems really painful; installing Anaconda is easy.

Also, with Anaconda, it’s easy to switch between Python 2 and Python 3.
Learning a new language

- Find a good book
- Have good example tasks/problems
- Play around
- Force yourself to use the new language
- Develop a script illustrating different language features

It takes time to learn a new programming language. The only way you’ll learn it is by forcing yourself to use it regularly. You need good, realistic problems to tackle. And it might take you just 30 minutes with the language you know but all afternoon in the new language. But if you don’t force yourself, you’ll never learn.

If you go away from it for a week, you’ll be quite rusty when you come back. I’ve found it useful to develop a script that illustrates the various language features. (“How do I write a loop again? How do I define a function?”) Looking through that, you’ll pick it all up again quickly. It’s harder to look back through a book in the same way.
I can’t really hope to teach you Python in 50 minutes, but I’ll try. In this crash course, I’ll go through a medium-sized script to combine a few data files and convert them into a different form.

*markers.txt* contains a list of ordered genetic markers. *families.txt* contains information about subjects’ familial relationships. *genotypes.txt* contains subjects’ genotypes.

We’re going to convert these data into the form used by the CRI-MAP program (an old program for constructing genetic maps).
Input: markers.txt

<table>
<thead>
<tr>
<th>Marker Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>D20S103</td>
</tr>
<tr>
<td>D20S482</td>
</tr>
<tr>
<td>D20S851</td>
</tr>
<tr>
<td>D20S604</td>
</tr>
<tr>
<td>D20S1143</td>
</tr>
<tr>
<td>D20S470</td>
</tr>
<tr>
<td>D20S477</td>
</tr>
<tr>
<td>D20S478</td>
</tr>
<tr>
<td>D20S481</td>
</tr>
<tr>
<td>D20S159</td>
</tr>
<tr>
<td>D20S480</td>
</tr>
<tr>
<td>D20S451</td>
</tr>
<tr>
<td>D20S171</td>
</tr>
<tr>
<td>D20S164</td>
</tr>
</tbody>
</table>

This is the `markers.txt` file. It just has one marker name per line.
This is the `families.txt` file; each line is one subject. In the **Father** and **Mother** columns, 0 indicates missing: a founding individual in that family. In the **Sex** column, 2 = female and 1 = male.
The `genotypes.txt` file is a bit ugly. Rows are markers and individuals are in fixed-width columns, with the genotypes being two numeric alleles separated by a slash. Blank fields correspond to missing data.
The file we’re converting to, `data.gen` in the format used by CRI-MAP, is a bit weird: Number of families, number of markers, the marker names in order, and then for each family, the family ID, the number of subjects in that family, and then the subjects. For each subject, there’s a line with individual, mom, dad, and sex (0 = female, 1 = male), and then a line with genotype data, with two numbers for each marker, with 0’s for missing values.
The top of the Python script

```python
#!/usr/bin/env python
# Combine the data in "genotypes.txt", "markers.txt" and "families.txt" and convert them into a CRI-MAP .gen file
#
# This is the python 2 version

def read_markers (filename):
    "Read an ordered list of marker names from a file."
    with open(filename, 'r') as f:
        lines = f.readlines()
    return [line.strip() for line in lines]

class Person:
    "Person class, to contain the data on a subject."
    def __init__ (self, family, id, dad, mom, sex):
        self.family = family
        self.id = id
        self.dad = dad
        self.mom = mom
        self.sex = "0" if sex == "2" else sex # convert 1/2 -> 1/0
        self.famid = family + '-' + id
        self.gen = {}
```

The first line (#!/usr/bin/env python) makes it so you can run this script from the command line by just typing its name. Using /usr/bin/env allows that Python might be located in a different place on different systems.

To make the script executable (on unix), type `chmod +x convert2.py`

In python, comments begin with # (as in R).

Instead of using braces to delineate blocks of code, Python uses indentation. I was initially turned off by this, but I’ve been converted to the idea (mostly from having written a lot of CoffeeScript code). You’re going to indent anyway; why not have that indentation be meaningful?

You define functions with `def name (param):`

Unlike R, functions must have a `return` statement if you want to return a value.
The bottom of the Python script

```python
if __name__ == '__main__':
    # file names
    gfile = "genotypes.txt" # genotype data
    mfile = "markers.txt" # list of markers, in order
    ffile = "families.txt" # family information
    ofile = "data.gen" # output file

    # read the data
    markers = read_markers(mfile)
    people = read_families(ffile)
    read_genotypes(gfile, people)

    # write the data
    write_genfile(ofile, people, markers)
```

The `convert2.py` script is just a bunch of function definitions (and one class).

This bit at the bottom is executed only if the script is run from the command line. It does all of the real work: read in the data and then write it back out as a `.gen` file.
Write functions & modules not scripts

- Write a set of reusable functions
- Your code will be easier to read
- You may actually reuse the code, this way

With Python (and R), there’s a tendency to write a long mess of a script. It’s better to focus on writing a set of reusable functions.

With the given example script, you can use `import convert2` to load the functions into python. This is similar to `library()` in R.
Try it out

$ convert2.py
$ diff data.gen data_save.gen

$ python      # (or ipython)
>>> import convert2
>>> help(convert2)  
>>> help(convert2.read_markers)

>>> markers = convert2.read_markers("markers.txt")
>>> markers[0]
>>> len(markers)
>>> markers[-1]
>>> markers[0:2]
>>> markers[0:-1]
>>> markers[5:]
>>> markers[:5]
>>> markers[0:7:2]

>>> quit() 14

If you type convert2.py from the command line, it will run the script and create the data.gen file, which you’ll see (with diff) matches the target data_save.gen file.

Or you can type python (or ipython) at the command line and then import the module and run some of the functions by hand. You’ll need to refer to the functions with the names preceded by convert2., or you can use from convert2 import * and then skip the convert2. part.

The read.markers function reads in the ordered list of markers as a vector. (In Python, they call it a list.) Vectors in Python are indexed starting at 0. You can use the len function (like length() in R) to get the length.

You can grab slices with :, but note that they don’t include the last element in the range.

And negative values are from the end, with -1 being the last value.

Also, you can use start:end:by. Remember that end is not included.
Read the marker names

```python
def read_markers (filename):
    "Read an ordered list of marker names from a file."
    with open(filename, 'r') as f:
        lines = f.readlines()
    return [line.strip() for line in lines]
```

This is the function to read the ordered list of markers. It takes a single argument: the name of the file.

The first line (between the double-quotes) is a description that will be shown if you import the module and type `help(convert2.read_markers)`. Strings in Python can be defined using single- or double-quotes, just like in R.

The `with` business looks a bit odd, but it ensures that the file will be closed if anything goes wrong. I could just as well have written `lines=open(filename).readlines()` (The `r`, for reading, is the default.) After that bit of code, `lines` contains a vector with one marker name per line.

The last line contains a “list comprehension.” It’s a sort of one-line `for` loop, which applies the `strip()` function to each element of the `lines` vector (removing any end-of-line character).

`read_markers` and `open` are ordinary functions, much like those in R. `readlines` and `strip` are object-oriented “methods.” Think of them as functions where the first argument precedes the function name.
I first define a class to contain the data for a single subject. It contains a function `__init__` for initializing an instance of the class (the data object for a single subject).

Within that function, `self` refers to the newly defined instance of the class, and `self.family`, etc., are the way to refer to the elements of the class object.

We create a new `Person` object by calling

```
Person(family, id, dad, mom, sex)
```
def read_families (filename):
    "Read family info and return a hash of people."
    with open(filename, 'r') as file:
        file.readline() # header row
        people = {}
        for line in file:
            vals = line.strip().split()
            person = Person(vals[0],vals[1],vals[2],vals[3],vals[4])
            people[person.famid] = person
    return people

This is the function to read the family information. I again use \texttt{with open() as file:} to open the file. If anything goes wrong, the file will be automatically “closed.”

I use \texttt{readline} to read (but ignore) the header line.

\texttt{people = {}} initializes a “hash.” This is like an unordered vector that is indexed by strings rather than numeric indices. (In Python, it’s called a “dictionary.”) I’m going to create a hash of \texttt{Person} objects, indicated by strings like "1-2" for individual 2 in family 1.

I use a \texttt{for} loop over lines in the file. For each line, I \texttt{strip} off any end-of-line character and then \texttt{split} it at the white space, into a vector.

I first call \texttt{Person} to define the \texttt{person} object, as then \texttt{person.famid} is defined, and I want to use that as the “hash key.”
The first function here cleans up a genotype string a bit. It strips off any white space, substitutes 0/0 in the case of a blank, and replaces the slash with a space. So a string like "78/125 " will be converted to "78 125". The replace function for strings is for doing text substitutions: it replaces every instance of its first argument with its second argument.

In the read_genotypes function, I grab all but the first element of the header line, which match what I’m using as the keys for my people hash. I then go through the rest of the file, one line at a time: I grab the marker name (getting rid of any spaces) and then go through the rest of the line, 7 characters at a time. range(n) returns the vector [0, 1, ..., n-1].

If you look back at the Person class, you’ll see that I’d initialized gen as a hash (with self.gen = {}). I’m filling this in, indexed by marker names. The read_genotypes function doesn’t return anything, because it modifies the input person object (as a “side effect”).

Note the backslash in the second-to-last line; this allows me to split a long line into two. If I left it off, Python would give an error.
Here are a few helper functions that I use in the last write_genfile function.

get_families returns a vector of distinct family IDs. I use a list comprehension again, which gives a vector with all of the family names. Then set turns this into a “set” of distinct values. It acts here sort of like unique() in R.

get_family_members returns a vector of the famid codes for the family members in a given family. I’m using a list comprehension again, but with an additional if qualification.

writeln is just a little wrapper for the write function, to write a string to a file. Note that in this function, the end argument has a default value, much like in R functions.
This is the function to write the CRI-MAP file. There are two interesting bits here.

First, `sorted()` returns a sorted version of a vector. I use it twice, the second time with
key=lambda famid: int(people[famid].id)
which is an anonymous function for sorting Person objects by their individual IDs (numerically).

The second interesting bit is
"%s %s %s %s" % (person.id, person.mom, person.dad, person.sex)
which is like `sprintf`, in that I’m formatting a bunch of stuff as a string.
The bottom of the Python script

```python
if __name__ == '__main__':
    # file names
    gfile = "genotypes.txt" # genotype data
    mfile = "markers.txt"   # list of markers, in order
    ffile = "families.txt"  # family information
    ofile = "data.gen"      # output file

    # read the data
    markers = read_markers(mfile)
    people = read_families(ffile)
    read_genotypes(gfile, people)

    # write the data
    write_genfile(ofile, people, markers)
```

We made it through the whole file; here’s the bit at the bottom again. This bit is run only if you’re executing the script from the command line.

I define the file names, read in the data, and then write it back out in a different form.
Basic types

- float
  \[ x = 0.3 \]
- int
  \[ m = 5 \]
- string
  \[ s = "blah" \]
- bool
  \[ x = True \]
  \[ y = False \]
- None
  \[ x = None \]
- complex
  \[ x = 5+0j \]

These are the basic types. Python 2 also distinguishes between `int` and `long`, while Python 3 has just `int`.

`None` is a null object; you can use it as `NA`. 
It’s important to be able to convert between types, particularly for converting between strings and floats.

The `%` operator is particularly handy for formatted output, or just to convert things into strings. With `%s`, numbers will be converted to strings, but with `%d` and `%f`, strings will not automatically be converted to numbers – you’ll get an error.

The `type` function returns the type of an object. The `dir` function returns a list of the methods

You can use `len` on strings, and you can subset them like a vector (aka list), and you can even loop over the characters in a string.
Lists are like lists in R: they’re ordered vectors whose elements can be basically anything.

Dictionaries are what I call hashes: an un-ordered list indexed by strings (called “keys”).

Tuples are like lists, but the contents can’t be changed. They’re useful as return values from a function.

Sets are lists with only unique values.
matrices as lists of lists

\[
x = \begin{bmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
\end{bmatrix}
\]

\[x[1][3] = 25\]

The simplest way to handle a matrix is as a list of lists. They’d typically be stored by rows, as you’d then index the thing as \(\text{mat}[\text{row}][\text{col}]\).

Also see numpy (numpy.org), for formal matrices and matrix methods.
for loops

```python
vec = range(4)
for x in vec:
    print (x+1)**2

import math
for i in xrange(len(vec)):
    print math.log( vec[i] + 1 )

h = {'x':3, 'y':4, 'z':2}
for k in h:
    print k, h[k]

for k in sorted(h.keys()):
    print k, h[k]

for k,v in h.iteritems():
    print k, v

for v in h.itervalues():
    print v
```

for loops over lists successively take each possible value in the list. If you want the indices, you need to create a vector of indices with `range`. Or use `xrange`, which is avoids actually creating the vector, but rather creates the elements when they’re needed.

`xrange` is in Python2 only; in Python 3, just use `range`. The Python3 `range` is really the Python2 `xrange`. So actually in Python3, the first line needs to be `vec = list(range(4)).`

for loops over dictionaries (aka hashes) successively take each possible key. You can use `iteritems()` to iterate over key-value pairs or `itervalues()` to iterate over just the values. Like `xrange`, these generate the vector of iteracted values as needed rather than in advance. There’s also a `iterkeys()` method, which is what is used as the default for `for` loops with dictionaries.

Note that these loops with dictionaries will be in arbitrary order. If you want a particular order, you first need to create a sorted vector of keys.

In Python3, use `.items()` and `.values()` in place of `.iteritems()` and `.itervalues()`, respectively.
List comprehensions are really useful for transformations or subsetting.

The `dict` function will convert a list of key-value pairs into a dictionary.

You can get by without them. But they can provide compact but readable code.

Note, again, that in Python3 you should use `.items()` in place of `.iteritems()`.

```python
vec = range(10)
[v**2 for v in vec if v > 5]

h = {'x':3, 'y':4, 'zz':2}
[h[k]**2 for k in h]
[h[k]**2 for k in h if len(k) == 1]
[[k, v**3] for k,v in h.items()]
dict( [[k, v**3] for k,v in h.items()] )

x = [k+1 for k in range(6)]
y = [True, False, True, False, False, False]
[x[i] for i in range(len(x)) if y[i]]
```
More with strings

```python
x = "bread and jam"
y = x.split(" ")
z = " ".join(y)
dir(x)
help(x.index)
x.endswith("jam")
x.startswith("bre")
x.count("a")
x.find("and")
x.find("jelly")
x.index("and")
x.index("jelly")
x.replace("jam", "jelly")
x.capitalize()
x.title()
x.upper()
x.upper().lower()
```

Python has also sorts of methods for doing things with strings.

Use `dir` to get a list, eg `dir(str)`, `dir("")`, or `dir(x)` where `x` is a string.

Use `help` to get a description of one of the methods, eg `help(str.find)`, `help("".find)`, or `help(x.find)`. 
A big reason to use scripting languages is for regular expression facilities. But I find regular expressions cumbersome in Python relative to Perl or Ruby.

There is some messiness about backslashes, so it’s best with regular expressions to use “raw strings” by preceding the string with an r, as so: r'blah'.
Unit tests: Nose

# This is nosetest_convert2.py
#
# At command line, type "nosetests nosetest_convert2.py"

from nose.tools import assert_equal
from convert2 import *

def test_parse_genotype():
    assert_equal(parse_genotype(" "), "0 0")
    assert_equal(parse_genotype("100/98 "), "100 98")
    assert_equal(parse_genotype("90/96 "), "90 96")
    assert_equal(parse_genotype("90/ 96 "), "90 96")
    assert_equal(parse_genotype(" 3 / 8 "), "3 8")

Unit tests are important for ensuring the correctness of Python code, as with any other programming effort.

Nose is simple tool for making Python unit tests. The above example is a minimal use of the tool.

At the command line, with tests in the file nosetest_convert2.py, you’d type nosetests nosetest_convert2.py

The nose.tools module contains a bunch of assertion functions. Here, I’m just using assert_equal.
Unit tests: unittest

```python
#!/usr/bin/env python
# Test one of the functions in convert2.py
#
# on the command line, type "test_convert2.py"

import unittest
from convert2 import *

class check_parse_genotype(unittest.TestCase):
    def test_parse_genotype(self):
        self.assertEqual(parse_genotype(" "), "0 0")
        self.assertEqual(parse_genotype("100/98 "), "100 98")
        self.assertEqual(parse_genotype("90/96 "), "90 96")
        self.assertEqual(parse_genotype(" 3 / 8 "), "3 8")

if __name__ == '__main__':
    unittest.main()
```

Python also has a built-in unittest module, but its use requires a bit more gunk. I don't totally understand all of this.

Like Nose, there are a variety of different assertion functions that you can use.
Summary

- Learn a scripting language, like Python
  - Not just for manipulating data files, but worth the effort just for that.

- Force yourself to use it

Applied statisticians need to be savvy with data file manipulation.

Don’t let your scientific collaborators do any copy-paste to move data around; any data file manipulation should be done with a computer program.

In the long run, knowing Python, you’ll be more self-sufficient and versatile.