Scientific Computing for Biologists

Biology 313

Fall 2011

Tue 2:50-5:20

Instructor: Paul M. Magwene

Email: paul.magwene@duke.edu

Phone: 613-8159

30 August 2011

Overview of Lecture

- Course Mechanics
  - Goals of course
  - Structure of lectures
  - Grading
  - Survey of previous training
- Introduction to R and Python
  - Advantages of R and Python
  - R and Python Resources
  - Important programming concepts
  - Introduction to data types and data structures in R and Python
  - Literate programming
- Hands-On Session

Class Structure

- Lectures
  - Typically 60-75 minutes
  - Emphasize the mathematical basis of the methods/approaches from both a geometric and algebraic basis
  - Discuss algorithms underlying the methods
  - Highlight available R/Python libraries
- Hands-on
  - Walk through some examples
  - Apply the techniques and concepts to real data

Syllabus

Date	*Topic*
August 30	Introduction; Getting Acquainted with R and Python, Literate Programming
September 6	Data as Vectors and Exploratory Data Analysis; vector operations, dot product, correlation, regression as projection, univariate visualizations
September 13	Linear Algebra Review I; Descriptive statistics as matrix operations, multivariate visualizations
September 20	Linear Algebra Review II; Regression models
September 27	Eigenvectors and Eigenvalues; Principal Components Analysis
October 4	Singular Value Decomposition, Biplots, and Correspondence Analysis
October 11	Fall Break
October 18	Discriminant analysis and Canonical Variate Analysis
November 25	Analyses based on Similarity/Distance I; Hierarchical and K-means clustering
November 1	Analyses based on Similarity/Distance II; Multidimensional scaling
November 8	Randomization and Monte Carlo Methods; Jackknife, Bootstrap
November 15	Building Bioinformatics Pipelines I; Pipes, redirection, subprocesses
November 22	Building Bioinformatics Pipelines II; Putting the concepts to work
November 29	Building Bioinformatics Pipelines III; Polishing the interface and generating publication quality graphics

Texts for Course

- Janert, P. K. 2010. Data Analysis with Open Source Tools. O'Reilly, Cambridge.
- Downey, A. B., J. Elkner, and C. Meyers. How to think like a computer scientist: learning with Python.
  - Available at `http://www.ibiblio.org/obp/thinkCSpy/`
- Wickens, T. D. 1995. The geometry of multivariate statistics. Lawrence Earlbaum Associates, New Jersey.

Supplementary Texts

- R and Python
  - Jones, O. et al. 2009. Scientific programming and simulation using R. CRC Press.
  - Martelli, A. (2006). Python in a nutshell (2nd ed.). O'Reilly.
- Statistics
  - Krzanowski, W. J. 2003. Principles of multivariate analysis. Oxford University Press.
  - Sokal, R. R. and F. J. Rohlf. 1995. Biometry. W. H. Freeman.
- Math
  - Hamilton, A. G. 1989. Linear algebra: an introduction with concurrent examples. Cambridge University Press.

Grading

- Problem sets/programming assignments
  - 8-10 over the course of the semester
  - Prepared as ‘literate programming’ documents

Survey on Previous Background

- Programming experience?
- Mathematical preparation?
  - Linear algebra
    - Matrix arithmetic - addition, subtraction, multiplication
    - Dot product - projection, angle between vectors
    - Matrix inverse
    - Determinant, rank, subspace
    - Eigenvectors/eigenvalues
- Statistical preparation?
  - Previous R/S-plus experience
  - Class in or self-study of multivariate techniques

Introduction to R and Python

Advantages of R and Python

- Both R and Python can be used in an interactive mode
  - enter commands/instructions at an interactive prompt for immediate execution
  - facilitates exploratory analyses
- Large collection of libraries/packages are available for statistical and numerical analysis and visualization
- Relatively easy to learn

Why Both R and Python?

- R is geared toward statistical computing
  - Great set of built-in facilities for statistically oriented tasks
  - Somewhat cumbersome syntax for non-statistical tasks
- Python is a general programming language
  - Clearer syntax
  - Wider range of modules
    - web programming, databases, numerical analysis, etc.
  - More natural language for simulation
  - More suitable as a ‘glue’ language
    - building bioinformatics pipelines

R Overview

What is R?

- ‘A language and environment for statistical computing and graphics’
- First developed in the mid-90s
- Derives from the S language
    - S was developed at Bell Labs in the mid-80s
- Advantages
    - Free and open-source
    - Much of the academic statistical community has adopted it
    - Active developer and user community
    - Wealth of built-in and user contributed libraries available for all types of analyses
- Disadvantages
    - GUI not as well developed as commercial statistical packages
        - S-Plus; site licensed by Duke - see OIT website
    - Has higher learning curve than some other simpler statistical software
    - Command-line can be intimidating

R Resources on the Web

- Home Page
  - `http://www.r-project.org`
- Comprehensive R Archive Network (CRAN)
  - `http://cran.r-project.org/mirrors.html`
  - See especially the ‘Task Views’
    - Statistical and population genetics
    - Environmental and ecological analysis
    - Spatial statistics
- Introductions and Tutorials
  - see `http://cran.r-project.org/other-docs.html`

Some R Packages of Interest

- Bioconductor – software package geared towards analysis of genomic data, especially microarray data, `http://www.bioconductor.org/`
- ape – 'Analysis of Phylogenetics and Evolution', `http://ape.mpl.ird.fr/`
- ade4 – Analysis of Ecological Data : Exploratory and Euclidean methods in Environmental sciences, `http://pbil.univ-lyon1.fr/ADE-4/home.php?lang=eng`

Python Overview

What is Python?

- High-level scripting/programming language
  - simple syntax, easy to learn
- Supports a variety of programming paradigms
  - Procedural, object-oriented, some functional programming idioms
- Invented by a computer scientist named Guido van Rossum at the Dutch National Research Institute for Mathematics and Computer Science
- First publicly released in 1991
- Named after Monty Python's Flying Circus!

Advantages of Python

- Active development
  - stable core
  - new language features being added
- Extensive standard library
  - wide range of programming tasks
- Large user community
  - good support
  - extensive set of 3rd party libraries
- Highly portable
  - available on pretty much any computing platform you're likely to run into
- Open-source and Free!

Python Resources on the Web

- Homepage
  - `http://www.python.org`
- Third party modules and packages
  - The Python Package Index - `http://www.python.org/pypi`
- Programming recipes/examples
  - ActiveState Python Cookbook - `http://aspn.activestate.com/ASPN/Cookbook/Python`

Python Resources of Particular Interest

- Numpy and SciPy
  - `http://www.scipy.org/`
  - linear algebra, statistical routines, numerical optimization
- Matplotlib
  - `http://matplotlib.sourceforge.net/`
  - 2D plotting library for Python (and now 3D too!)
- BioPython
  - `http://biopython.org/`
  - Libraries for computational molecular biology
- SimPy
  - `http://simpy.sourceforge.net/`
  - Discrete event simulation
- Python Enthought Edition
  - `http://www.enthought.com/`
  - A distribution of Python and related packages (many geared toward scientific/numerical computing) for Windows, OS X, and linux

Python Tutorials

- Pilgrim, “Dive Into Python"
  - http://www.diveintopython.org/
  - for experienced programmers
- Schuerer et al. ‘Introduction to Programming Using Python’
  - http://www.pasteur.fr/formation/infobio/python/
  - From the Pasteur Inst., aimed at biologists

Some Important Programming Concepts

- Data Types
  - refer to the types of values that can be represented in a computer program
  - determine the representation of values in memory
  - determine the operations you can perform on those values
  - Examples: integers, strings, floating point values
- Data Structures
  - a way of storing collections of data
  - different structures are more efficient for particular types of operations
  - Examples: lists, hash tables, stacks, queues, trees
- Variables
  - Variables are references to objects/values in memory
  - Think of them as labels that point to particular places in a computer's memory

More Important Programming Concepts

- Statement
  - an instruction that a computer program can execute
  - Example: `print "Hello, World!"`
- Operators
  - Symbols representing specific computations
  - Example: `+`, `-`, `*` (addition, subtraction, multiplication)
- Expression
  - a combination of values, variables, and operators
  - Example: `1 + 1`
- Functions (subroutines, procedures, methods)
  - A piece of code that carries out a specific task, set of instructions, calculations, etc.
  - Typically used to encapsulate algorithms

Basic Data Types, Data Structures and Operators in R

Numeric Data Types in R

- Floating point values (‘doubles’)

```
> x <- 10.0
> typeof(x)
[1] "double"
```

- Complex numbers

```
> x <- 1+1i
> typeof(x)
[1] "complex"
```

- Integers
  - Default numeric type is double, must explicitly ask for integers if single values

```
> x <- as.integer(10)
> typeof(x)
[1] "integer"
```

Additional Data Types in R

- Boolean('logical')

```
> x <- TRUE # or x <- T
> x <- F # or x <- FALSE
> typeof(x)
[1] "logical"
```

- Character strings

```
> x <- 'Hello' # or x <- "Hello"
> typeof(x)
[1] "character"
```

Arithmetic Operators and Mathematical Functions in R

```
> 10 + 2 # addition
[1] 12
> 10 - 2 # subtraction
[1] 8
> 10 * 2 # multiplication
[1] 20
> 10 / 2 # division
[1] 5
> 10 ^ 2 # exponentiation
[1] 100
> 10 ** 2 # alternate exponentiation
[1] 100
> sqrt(10) # square root
[1] 3.162278
> 10 ^ 0.5 # same as square root
[1] 3.162278
> pi*(3)**2  # R knows some useful constants
[1] 28.27433
> exp(1) # exponential function
[1] 2.718282
```

Simple Data Structures in R: Vectors

Vectors are the simplest data structure in R

- vectors represent an ordered list of items

```
> x <- c(2,4,6,8)
> y <- c('joe','bob','fred')
```

- vectors have length (possibly zero) and type

```
> typeof(x)
[1] "double"
> length(x)
[1] 4
> typeof(y)
[1] "character"
```

Simple Data Structures in R: Vectors

Accesing the objects in a vector is accomplished by ‘indexing’:

- The elements of the vector are assigned indices $1 \dots n$ where n is the length of the vector

```
> x <- c(2,4,6,8)
> length(x)
[1] 4
> x[1]
[1] 2
> x[2]
[1] 4
> x[3]
[1] 6
> x[4]
[1] 8
```

Simple Data Structures in R: Vectors

- Single objects are usually represented by vectors as well

```
> x <- 10.0
> length(x)
[1] 1
> x[1]
[1] 10
```

- Every element in a vector is of the same type
  - If this is not the case the the values are coerced to enforce this rule

```
> x <- c(1+1i, 2+1i, 'Fred', 10)
> x
[1] "1+1i" "2+1i" "Fred" "10"
```

Arithmetic Operators Work on Vectors in R

Most arithmetic operators work element-by-element on vectors in R

```
> x <- c(2, 4, 6, 8)
> y <- c(0, 1, 2, 3)
> x + y
[1]  2  5  8 11
> x - y
[1] 2 3 4 5
> x * y
[1]  0  4 12 24
> x^2
[1]  4 16 36 64
> sqrt(x)
[1] 1.414214 2.000000 2.449490 2.828427
```

Simple Data Structures in R: Lists

Lists

- Lists in R are like vectors but the elements of a list are arbitrary objects (even other lists)

```
> x <- list('Bob',27, 10, c(720,710))
> x
[[1]]
[1] "Bob"

[[2]]
[1] 27

[[3]]
[1] 10

[[4]]
[1] 720 710
```

Simple Data Structures in R: Lists

Accessing objects in Lists:

- Items in lists are accessed in a different manner than vectors.
  - Typically you use double brackets (`[[]]`)to return the element at index `i`
  - Single brackets always return a list containing the element at index `i`

```
> x <- list('Bob', 27, 10, c(720,710))
> typeof(x[1])
[1] "list"
> typeof(x[[1]])
[1] "character"
```

Simple Data Structures in R: Lists

- Objects in R lists can be named

```
> x <- list(name='Bob',age=27, years.in.school=10)
> x
$name
[1] "Bob"

$age
[1] 27

$years.in.school
[1] 10
```

- Named list objects can be accessed via the $ operator

```
> x$years.in.school
[1] 10
> x$name
[1] "Bob"
```

- The names of list objects can be accessed with the `names()` function

```
> names(x)
[1] "name"   "age"   "years.in.school"
```

Basic Data Types, Structures and Operators in Python

Numeric Data Types in Python

- Floating point values

```
>>> x = 10.0
>>> type(x)
<type 'float'>
```

- Complex numbers

```
>>> x = 1 + 1j
>>> type(x)
<type 'complex'>
```

- Integers

```
>>> x = 10
>>> type(x)
<type 'int'>
```

Additional Data Types in Python

- Boolean('bool')

```
>>> x = True
>>> type(x)
<type 'bool'>
>>> y = False
>>> type(y)
<type 'bool'>
>>> 1 == 2
False
>>> type(1 == 2)
<type 'bool'>
```

- Character strings

```
>>> x = 'Hello, world'
>>> y = "Hello, world"
>>> type(x), type(y)
(<type 'str'>, <type 'str'>)
```

Arithmetic Operators in Python

```
>>> 10 + 2 # addition
12
>>> 10 - 2 # subtraction
8
>>> 10 * 2 # muliplication
20
>>> 10 / 2 # division
5
>>> 11 / 2 # division (surprising answer!)
5
>>> 11.0 / 2 # division (expected answer)
5.5
>>> 10 **2 # exponentiation, ^ doesn't work in Python
100
>>> from math import * # import all the standard math
                       # functions like sqrt, sin
>>> sqrt(10)
3.1622776601683795
>>> 10 ** 0.5
3.1622776601683795
```

Simple Data Structures in Python: Lists

Lists are the simplest 'built-in' data structure in Python, and like R lists they are ordered collections of arbitrary objects.

- Creating a Python list

```
>>> x = [2,4,6,8,'fred']
```

- Python lists have length (possibly zero)

```
>>> len(x)
5
```

- Python lists are zero-indexed, this means you can access list elements 0 . . . `len(x)-1`

```
>>> x[0]
2
>>> x[3]
8
>>> x[5]
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in ?
    x[5]
IndexError: list index out of range
```

Simple Data Structures in Python: Tuples

Python 'tuples' are like lists, but they are immutable, meaning that they can't be changed once you create them.

- Creating a Python tuple

```
>>> y = (2,4,6,8,'fred') # rounded parentheses
```

- Tuples have length (possibly zero) and are zero indexe

```
>>> len(y)
5
>>> y[0]
```

- Tuples can't be changed after creation.

```
>>> x = [2,4,6,8,'fred'] # create list
>>> y = (2,4,6,8,'fred') # create tuple
>>> x[1] = 'WOW'
>>> x
[2, 'WOW', 6, 8, 'fred']
>>> y[1] = 'WOW'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Simple Data Structures in Python: NumPy arrays

There is no 'built-in' Python data structure that behaves the same as an R vector.

- To get similar behavior in Python we use a data structure called an `array` from the package called NumPy. Like R vectors, NumPy arrays are homogenous collections of objects (typically numbers, but they can also hold references to other types of objects).
- Creating Numeric arrays

```
>>> from numpy import array
>>> x = array([2,4,6,8]) # note the inner list
>>> z = array(2,4,6,8) # this won't work (output omitted)
>>> y = array(["bob","fred","joe"])
```

- arrays have length and are indexed in a manner similar to lists.

```
>>> len(x)
4
>>> x[2]
6
```

Arithmetic Operators Work on NumPy arrays

NumPy arrays work element-by-element, similar to R vectors

```
>>> import numpy
>>> from numpy import array
>>> x = array([2,4,6,8])
>>> y = array([0,1,2,3])
>>> x + y
array([ 2,  5,  8, 11])
>>> x * y
array([ 0,  4, 12, 24])
>>> x ** 2
array([ 4, 16, 36, 64])
>>> from math import *
>>> sqrt(x)

Traceback (most recent call last):
  File "<pyshell#23>", line 1, in -toplevel-
    sqrt(x)
TypeError: only length-1 arrays can be converted to Python scalars.
>>> numpy.sqrt(x) # use sqrt function in the Numeric package
array([ 1.41421356,  2.,  2.44948974,  2.82842712])
>>> numpy.sqrt(10)
3.1622776601683795
```

Literate Programming

"Literate programming" is a concept coined by Donald Knuth, a preeminent computer scientist:

- Programs are useless with descriptions
- Descriptions should be literate, not comments in code or typical reference manuals.
- The code in the descriptions should work.

Literate Programming and Reproducible Research

How literate programming can help to ensure your research is reproducible:

- The steps of your analyses are explicitly described, both as written text and the code and function calls used.
- Analyses can easily checked for correctness and reproduced from your literate code.
- Your literate code can serve as a template for future analyses, saving you time and the trouble of remembering all the gory details.

Tools for literate programming in R and Python

How literate programming can help to ensure your research is reproducible:

- R – Sweave; works together with LaTeX to produce output.
- Python – Pweave; patterned after Sweave. Can produce LaTeX output but also produce a text-based format called 'reStructuredText' which can be converted to HTML or other formats

Tools for literate programming in R and Python

Both Sweave and Pweave using a simple markup syntax called 'noweb', where you weave your code into your description by putting it between <<>>= and @ blocks.

Example:

```
Here are some trivial R examples that will help to
illustrate how Sweave works:

<<>>=
z <- 1:10
mean(z)
summary(z)
z[z > 5]
@

The above text was a code block woven into my
description. It gets evaluated and integrated into
the output. Cool, eh?
```

Sweave output

Output produced by Sweave and $\LaTeX$ for the code on the previous slide:

Here are some trivial R examples that will help to illustrate how Sweave works:

```
> z <- 1:10
> mean(z)
```

```
[1] 5.5
```

```
> summary(z)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00
```

```
> z[z > 5]
```

```
[1]  6  7  8  9 10
```

The above text was a code block woven into my description. It gets evaluated and integrated into the output. Cool, eh?

Fancier pgfSweave output

There's a new package called pgfSweave that produce even nicer output (code highlighting, better figure formatting):

A Sweave example that incorporates graphics is always nice. First, let's generate the data by drawing 1000 observations from the standard normal ($\mu = 0, \sigma = 1$).

```
> data <- rnorm(1000)
```

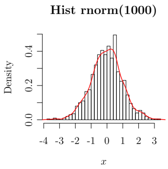
Next, we create a summary table:

```
> summary(data)

     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-3.688000 -0.651800  0.005649 -0.020380  0.587000  3.331000
```

Finally, we create a nice figure in which a density estimate is superimposed on a histogram:

```
> hist(data, breaks = 50, freq = F, main = "Hist rnorm(1000)",
+     xlab = "$x$")
> lines(density(data), col = "red", lwd = 2)
```



Things to Remember

- Try it out - programming involves experimentation
- Don't reinvent the wheel - it's usually worth spending some time finding out if someone has already written code that does what you need.
- Practice - learning to program, like learning a foreign language, requires lots of practice.
- Persist - many new tools/concepts can be hard to grasp at first. Keep plugging away until you get that 'Aha!' moment

You might be surprised to find that...

- Programming is fun! (at least sometimes)
- Math is fun! (at least sometimes)
- Statistics is fun! (at least sometimes)

- Gaining new insights into how your biological system of interest works is fun! (always)