Bio 723

Scientific Computing for Biologists

Paul M. Magwene

Fall 2014

Contents

1 Getting your feet wet with Python

1.1 Starting the Python interpretter

The Python interpretter can be started in a number of ways. The simplest way is to open a shell (terminal) and type `python`. Go ahead and do this to make sure you have a working version of the default Python interpretter available on your system. From within the default interpretter you can type `Ctrl-d` (Unix, OS X) or `Ctrl-z` (Windows) to stop the interpretter and return to the command line.

For interactive use, the default interpretter isn't very feature rich, so the Python community has developed a number of GUIs or shell interfaces that provide more functionality. For this class we will be using an interface called IPython. Recent versions of IPython provides terminal and GUI-based shells as well as a web-browser interface called the IPython Notebook.

1.2 Accessing the Documentation in Python

Python comes with extensive HTML documentation and the Python interpreter has a help function that works similar to R's `help()`.

```
>>> help(sum)
Help on built-in function sum in module __builtin__:

sum(...)
    sum(sequence, start=0) -> value

    Returns the sum of a sequence of numbers (NOT strings) plus the value
    of parameter 'start'.  When the sequence is empty, returns start.
```

IPython also lets you use proceed the function name with a question mark:

```
In [1]: ?sum
Type:       builtin_function_or_method
Base Class: <type 'builtin_function_or_method'>
String Form:<built-in function sum>
Namespace:  Python builtin
Docstring:
sum(sequence[, start]) -> value

Returns the sum of a sequence of numbers (NOT strings) plus the value
of parameter 'start' (which defaults to 0).  When the sequence is
empty, returns start.
```

1.3 Using Python as a Calculator

The simplest way to use Python is as a fancy calculator. Let's explore some simple arithmetic operations:

```
>>> 2 + 10   # this is a comment
12
>>> 2 + 10.3
 12.300000000000001  # 0.3 can't be represented exactly in floating point
     precision
>>> 2 - 10
-8
>>> 1/2  # integer division
0
>>> 1/2.0  # floating point division
0.5
>>> 2 * 10.0
20.0
>>> 10**2  # raised to the power 2
100
>>> 10**0.5  # raised to a fractional power
3.1622776601683795
>>> (10+2)/(4-5)
-12
>>> (10+2)/4-5  # compare this answer to the one above
-2
```

In addition to integers and reals (represented as floating points numbers), Python knows about complex numbers:

```
>>> 1+2j  # Engineers use 'j' to represent imaginary numbers
(1+2j)
>>> (1 + 2j) + (0 + 3j)
(1+5j)
```

Some things to remember about mathematical operations in Python:

- Integer and floating point division are not the same in Python. Generally you'll want to use floating point numbers.
- The exponentiation operator in Python is **
- Be aware that certain operators have precedence over others. For example multiplication and division have higher precedence than addition and subtraction. Use parentheses to disambiguate potentially confusing statements.
- The standard math functions like `cos()` and `log()` are not available to the Python interpeter by default. To use these functions you'll need to `import` the math library as shown below.

For example:

```
>>> 1/2
0
>>> 1/2.0
0.5
>>> cos(0.5)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    cos(0.5)
NameError: name 'cos' is not defined
>>> import math  # make the math module available
>>> math.cos(0.5) # cos() function in the math module
0.87758256189037276
>>> pi   # pi isn't defined in the default namespace
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in -toplevel-
    pi
NameError: name 'pi' is not defined
>>> math.pi # however pi is defined in math
3.1415926535897931
>>> from math import * # bring everything in the math module into the
    current namespace
>>> pi
3.1415926535897931
>>> cos(pi)
-1.0
```

1.4 More Data Types in Python

You've already seen the three basic numeric data types in Python - integers, floating point numbers, and complex numbers. There are two other basic data types - Booleans and strings.

Here's some examples of using the Boolean data type:

```
>>> x = True
>>> type(x)
<type 'bool'>
>>> y = False
>>> x == y
False
>>> if x is True:
...     print 'Oh yeah!'
...
Oh yeah!
>>> if y is True:
...     print 'You betcha!'
... else:
...     print 'Sorry, Charlie'
...
```

```
Sorry, Charlie
>>>
```

1.4.1 Comparison Operators Return Booleans

The standard comparison operators for numerical values are available in Python. Comparison operators return Boolean values:

```
>>> 4 < 5 # less than
True
>>> 4 > 3.99 # greater than
True
>>> 4 <= 4.0 # less than or equal to
True
>>> 4 >= 5 # greater than or equal to
False
>>> 4 == 4.0 # test equality
True
```

1.4.2 Strings

The string data type is used to represent ordered sets of characters, such as text:

```
>>> s1 = 'It was the best of times'
>>> type(s1)
<type 'str'>
>>> s2 = 'it was the worst of times'
>>> s1 + s2  # string concatenation
'It was the best of timesit was the worst of times'
>>> s1 + ', ' + s2
'It was the best of times, it was the worst of times'
>>> 'times' in s1
True
>>> s3 = "You can nest 'single quotes' in double quotes"
>>> s4 = 'or "double quotes" in single quotes'
>>> s5 = "but you can't nest "double quotes" in double quotes"
  File "<stdin>", line 1
    s5 = "but you can't nest "double quotes" in double quotes"
                                   ^
SyntaxError: invalid syntax
```

Note that you can use either single or double quotes to specify strings.

1.5 Simple data structures in Python: Lists

Lists are the simplest 'built-in' data structure in Python. List represent ordered collections of arbitrary objects.

```
>>> l = [2, 4, 6, 8, 'fred']
>>> l
[2, 4, 6, 8, 'fred']
>>> len(l)
5
```

Python lists are zero-indexed. This means you can access lists elements `0` to `len(x)-1`.

```
>>> l[0]
2
>>> l[3]
8
>>> l[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

You can use negative indexing to get elements from the end of the list:

```
>>> l[-1] # the last element
'fred'
>>> l[-2] # the 2nd to last element
8
>>> l[-3] # ... etc ...
6
```

Python lists support the notion of 'slices' - a continuous sublist of a larger list. The following code illustrates this concept:

```
>>> y = range(10)  # our first use of a function!
>>> y
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y[2:8]
[2, 3, 4, 5, 6, 7]
>>> y[2:-1] # the slice
[2, 3, 4, 5, 6, 7, 8]
>>> y[-1:0] # how come this didn't work?
[]
# slice from last to first, stepping backwards by 2
>>> y[-1:0:-2]
[9, 7, 5, 3, 1]
```

1.6 Using NumPy arrays

The Python user community has developed a module called NumPy for efficient numerical computing in Python. The basic data structure in the NumPy library is an array. Arrays can be used to reprsent both vectors and matrices, common mathematical structures we'll use throughout the course. Below are some examples illustrating the use of NumPy arrays:

```
>>> from numpy import array # a third form of import
>>> x = array([2,4,6,8,10])
>>> -x
array([ -2,  -4,  -6,  -8, -10])
>>> x ** 2
array([  4,  16,  36,  64, 100])
>>> pi * x # assumes pi is in the current namespace
array([  6.28318531,  12.56637061,  18.84955592,  25.13274123,
    31.41592654])
>>> y = array([0, 1, 3, 5, 9])
>>> x + y
array([ 2,  5,  9, 13, 19])
>>> x * y
array([ 0,  4, 18, 40, 90])
>>> z = array([1, 4, 7, 11])
>>> x+z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

The error above illustrates that basic arithmetic operations involving pairs of NumPy arrays in Python require that the arrays be of equal length.

Remember that lists and arrays in Python are zero-indexed rather than one-indexed.

```
>>> x
array([ 2,  4,  6,  8, 10])
>>> len(x)
5
>>> x[0]
2
>>> x[1]
4
>>> x[4]
10
>>> x[5]

Traceback (most recent call last):
  File "<pyshell#52>", line 1, in -toplevel-
    x[5]
IndexError: index out of bounds
```

NumPy arrays support the comparison operators and return arrays of booleans.

```
    >>> x < 5
    array([ True, True, False, False, False], dtype=bool)
    >>> x >= 6
    array([0, 0, 1, 1, 1])
```

NumPy also supports the combination of comparison and indexing

```
>>> x[x < 5]  # get the elements of x where that are less than 5
array([2, 4])
```

```
>>> x[x >= 6]  # elements of x that are greater than or equal to 6
array([ 6,  8, 10])
>>> x[(x<4)+(x>6)]  # 'or'
array([ 2,  8, 10])
```

Note that Boolean addition is equivalent to ‘or’ and Boolean multiplication is equivalent to ‘and’. There are also a variety of more complicated indexing functions available for NumPy; see the Indexing Routines in the Numpy docs.

Most of the standard mathematical functions can be applied to NumPy arrays however you must use the functions defined in the NumPy module.

```
>>> x
array([ 2,  4,  6,  8, 10])
>>> import math
>>> math.cos(x)

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in -toplevel-
    math.cos(x)
TypeError: only length-1 arrays can be converted to Python scalars.
>>> import numpy
>>> numpy.cos(x)
array([-0.41614684, -0.65364362,  0.96017029, -0.14550003, -0.83907153])
```

2 Bivariate Data

2.1 Plotting Bivariate Data in Python

Let's use a dataset called Anderson's Iris data to explore bivariate relationships between variables. This data set was published by a a plant biologist named Edgar Anderson in the 1930's (Annals of the Missouri Botanical Garden 23 (3): 457–509) but was made famous by R. A. Fisher who used it to illustrate many of the fundamental statistical methods he developed. The data set consists of four morphometric measurements on specimens of three different iris species. We'll be using this data set repeatedly in future weeks so familiarize yourself with it.

First, let's use the command line utility `wget` to download the file 'iris.cvs' from the course repository to our virtual machine:

```
wget https://github.com/pmagwene/Bio723/raw/master/datasets/iris.csv
```

Once the 'iris.csv' file is available we can import into Python. We'll introduce another Python library, called the Pandas, which is an increasingly popular library for statistical analyes in Python. Pandas share a number of features with R, in particular the use of a data structure called a 'data frame' to hold and manipulate multivariate data.

```
>>> import pandas as pd
>>> iris = pd.read_csv('iris.csv')
>>> iris.shape  # get number and rows and columns in data
(150, 5)
>>> iris.columns  # get the names of the data columns
Index([u'Sepal Length', u'Sepal Width', u'Petal Length', u'Petal Width', u'
    Species'], dtype='object')

>>> iris.head()  # show first five rows of data set
   Sepal Length  Sepal Width  Petal Length  Petal Width Species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa

>>> iris.tail() # show last five rows of data set
     Sepal Length  Sepal Width  Petal Length  Petal Width    Species
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
```

```
148          6.2          3.4          5.4          2.3  virginica
149          5.9          3.0          5.1          1.8  virginica



>>> iris['Species']  # get the Species column
... lots of output ...
>>> pd.unique(iris['Species'])  # get the unique names in the Species
    column
```

2.1.1 Bivariate scatter plots

We'll start with the conventional 'variable space' representation of bivariate relationships – the scatter plot.

```
>>> %matplotlib inline  # required in IPython notebook
>>> from pylab import *  # import commonly used plotting functions
>>> scatter(iris['Sepal Length'], iris['Sepal Width'])
>>> xlabel("Sepal Length")
>>> ylabel("Sepal Width")
```

This plots Sepal Length on the x-axis and Petal Length on the y-axis. In many cases it's useful to ensure that the axes of the plot have identifical unit values, so that a one unit change in the x-axis is that as on the y-axis. Here's how you can ensure that.

```
>>> gca().set_aspect('equal')
```

In the function above, the `gca()` function called is short for "get current axis", and then we use the `set_aspect` function to make the axes cale equally.

From these plot we can see that these two variables are positively associated (i.e. when one increases the other tends to increase). You will also notice there seem to be distinct clusters of points in the plot. Recall that the iris data set consists of three different species. Let's regenerate the plot, this time coloring the points according to the species names. First, let's note that the Species column is a categorical variable, which in R we refer to as a 'factor'.

```
>>> setosa = iris[iris["Species"] == 'setosa']
>>> setosa.shape
(50, 5)
>>> versicolor = iris[iris["Species"] == 'versicolor']
>>> virginica = iris[iris["Species"] == 'virginica']
>>> scatter(setosa['Sepal Length'], setosa['Sepal Width'], color='crimson',
     alpha=0.5)
>>> scatter(versicolor['Sepal Length'], versicolor['Sepal Width'], color='
    royalblue', alpha=0.5)
>>> scatter(virginica['Sepal Length'], virginica['Sepal Width'], color='
    forestgreen',alpha=0.5)
```

Let's also label our axes, add a title, and a legend:

```
>>> xlabel("Sepal Length")
>>> ylabel("Sepal Width")
>>> title("Petal Length vs. Sepal Length")
>>> legend(["I. setosa", "I. versicolor", "I. setosa"], loc="upper right")
```

2.2 Vector Mathematics in Python

NumPy arrays support basic arithmetic operations that correspond to the same operations on geometric vectors. The following examples illustrate this:

```
>>> import numpy as npy
>>> x = np.array([1, 2, 3, 4, 5])
>>> y = np.array([6, 7, 8, 9, 10])
>>> x + y
array([ 7,  9, 11, 13, 15])
>>> x - y
array([-5, -5, -5, -5, -5])
>>> x * 3    # multiplication by a scalar
array([ 3,  6,  9, 12, 15])
>>> x * y    # elementwise multiplication of arrays, NOT dot product!
array([ 6, 14, 24, 36, 50])
```

Above notice that the multiplication operator carries out element-wise multiplication. The get the dot product we need to use the `dot()` function defined in NumPy.

```
>>> np.dot(x, y)   # check your answer with pen and paper
130
```

In lecture we saw that many useful geometric properties of vectors could be expressed in the form of dot products. Let's start with some two-dimensional vectors where the geometry is easy to visualize:

```
>>> a = np.array((1, 0))
>>> b = np.array((0, 1))
```

Now let's draw our vectors:

```
# create empty plot w/specified x- and y- limits
# set aspect ratio equal
>>> ax = axes(aspect='equal')
>>> xlim(-1.5, 1.5)
>>> ylim(-1.5,1.5)
>>> ax.arrow(0, 0, a[0], a[1])
<matplotlib.patches.FancyArrow object at 0x108997450>
>>> ax.arrow(0, 0, b[0], b[1])
<matplotlib.patches.FancyArrow object at 0x10894bf10>
>>> show()
```

You should now have a figure that looks like the one below: Let's see what the dot product can tell us about these vectors. First recall that we can calculate the length of a vector as the square-root of the dot product of the vector with itself ($|\vec{a}|^2 = \vec{a} \cdot \vec{a}$)

```
>>> len_a = np.sqrt(np.dot(a, a))
```

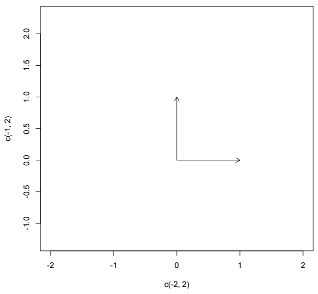

Figure 2.1: A simple vector figure.

```
>>> len_a
1.0
>>> len_b = np.sqrt(np.dot(b, b))
```

How about the angle between a and b?

```
>>> dot_ab = np.dot(a,b)
>>> dot_ab
0.0
>>> cos_ab = dot_ab / (len_a * len_b)
>>> cos_ab
0.0
```

A key point to remember dot product of two vectors is zero if, and only if, they are orthogonal to each other (regardless of their dimension).

2.3 Writing Functions in Python

So far we've been mostly using Python's built in functions. However the power of a true programming language is the ability to write your own functions.

The general form of an Python function is as follows:

```
def funcname(arg1, arg2):
    # body of function
    return result
```

To make this concrete, here's an example where we define a function in the interpreter and then put it to use:

```
>>> def veclength(x):
...     return np.sqrt(np.dot(x,x))
...
>>> x
array([1, 2, 3, 4, 5])
>>> veclength(x)
```

```
7.416198487095663
>>> y
array([ 6,  7,  8,  9, 10])
>>> veclength(y)
18.165902124584949
```

2.3.1 Putting Python functions in module

When you define a function at the interactive prompt and then close the interpreter your function definition will be lost. The simple way around this is to define your Python functions in a module that you can than access at any time.

In your programmer's editor of choice, create a new blank document. Enter your function into the editor and upload the source file to your virtual machine with a name like `vecgeom.py`.

```
# functions defined in vecgeom.py

# this import statement is necessary for the module to use
# functions defined in numpy

import numpy as np

def veclength(x):
    """ Given a numeric vector, returns length of that vector. """
    return np.sqrt(np.dot(x,x))



def unitvector(x):
    """ Return a unit vector in the same direction as x. """
    return x/veclength(x)
```

There are two functions defined above, one of which calls the other. Both take single vector arguments. These functions have no error checking to insure that the arguments passed to the functions are reasonable but Python's built in error handling will do just fine for most cases.

Once your functions are in a script file you can make them accesible by using the `import()` function:

```
>>> import vecgeom
>>> x = np.array((1, 0.4))
>>> vecgeom.veclength(x)
1.077032961426901
>>> ux = vecgeom.unitvector(x)
>>> ux
array([ 0.92847669,  0.37139068])
>>> a = np.array([1,2,3,4])
>>> from vecgeom import veclength, unitvector
>>> ua = unitvector(a)
>>> ua
```

```
array([ 0.18257419,  0.36514837,  0.54772256,  0.73029674])
>>> veclength(ua)  # how come this value isn't exactly 1.0 as expected?
0.99999999999999989
```

Note that our functions work with vectors of arbitrary dimension.

Assignment 2.1

Write a function that uses the dot product and the `acos()` function to calculate the angle (in radians) between two vectors of arbitrary dimension. By default, your function should return the angle in radians. Also include a logical (Boolean) argument that will return the answer in degrees. Test your function with the following two vectors: `x = [-3, -3, -1, -1, 0, 0, 1, 2, 2, 3]` and `y = [-8, -5, -3, 0, -1, 0, 5, 1, 6, 5]`. The expected angle for these test vectors is 0.441 radians (25.3 degrees).

Let's also add the following function to `vecgeom.py` to aid in visualizaing 2D vectors:

```

from matplotlib import pyplot as plt

def draw_vectors(a, b, colors=('red', 'blue'), clear = True):
    """ Given vectors a and b, draw their geometric representation. """"
    # figure out the limits such that the origin and the vector
    # end points are all included in the plot
    xhi = max(0, a[0], b[0])
    xlo = min(0, a[0], b[0])
    yhi = max(0, a[1], b[1])
    ylo = min(0, a[1], b[1])

    xlims = np.array((xlo, xhi))*1.10 # give a little breathing space
        around vectors
    ylims = np.array((ylo, yhi))*1.10

    if clear:
        ax = plt.axes()
        plt.xlim(*xlims)
        plt.ylim(*ylims)
        plt.xlabel("x-coord")
        plt.ylabel("y-coord")

    plt.arrow(0, 0, a[0], a[1], color=colors[0])
    plt.arrow(0, 0, b[0], b[1], color=colors[1])
```

You can use this new function as follows:

```
# if a module has been loaded once, and then you change it
# you must reload the module for the changes to take effect
>>> reload(vecgeom)
<module 'vecgeom' from 'vecgeom.pyc'>
>>> x = np.array((1, 0.4))
>>> y = np.array((0.2,0.8))
>>> vecgeom.draw_vectors(x,y)
```

The resulting figure should resemble the one below.

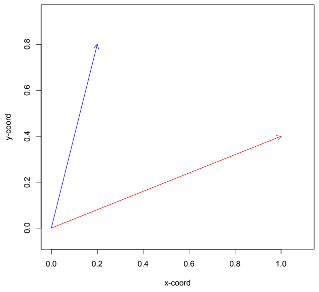

Figure 2.2: Another vector figure.

Notice that we included a `clear` argument in our `draw_vectors` function. I included this so we could add additional vectors to our plot, without overwriting the old vectors, as demonstrated below:

```
>>> ux = vecgeom.unitvector(x)
>>> uy = vecgeom.unitvector(y)
>>> vecgeom.draw_vectors(ux, ux, colors=['black','green'], clear=False)
>>> ## you might have to change xlim and ylim
```

Unlike the other functions we wrote, `draw_vectors` only works properly with 2D vectors. Since any pair of vectors defines a plane, it is possible to generalize this function to work with arbitrary pairs of vectors.

Assignment 2.2

Write a function, `vproj()`, that takes two vectors, $\vec{x}$ and $\vec{y}$, and returns a list containing the projection of $\vec{y}$ on $\vec{x}$ and the component of $\vec{y}$ in $\vec{x}$:

$$P_{\vec{x}}(\vec{y}) = \left(\frac{\vec{x}\cdot\vec{y}}{|\vec{x}|}\right)\frac{\vec{x}}{|\vec{x}|}$$

and

$$C_{\vec{x}}(\vec{y}) = \frac{\vec{x}\cdot\vec{y}}{|\vec{x}|}$$

Use the test vectors from Assignment 2.1 to test your function. The list returned by your function for these test vectors should resemble that shown below:

```
>>>> vproj(x, y)
[array([-6, -6, -2, -2, 0, 0, 2, 4, 4, 6]), 12.32883]
```