CME 193 NOTES

ARUN DEBRAY
JANUARY 24, 2013

Contents

1. Introduction to Computing: 1/10/13

First it will be helpful to learn comments and variables. A variable is used to store information, and its value can change as `x = 2`, then `x = [3,4,5]`, `x = 'hi'`, etc. Python is strongly typed but dynamically typed, so one variable can hold data of multiple types throughout a program.

Comments are denoted with a #, and with syntax highlighting look like this: `# This is a comment.`

Arithmetic is unsurprising, though you can also use some arithmetic operators on strings (e.g. `'Hello' + ' world'`), which is not the case in Matlab. Python supports the standard comparisons, but also chained inequalities of the form `0 < a <= 1`. Boolean operators are just words, which is nice (e.g. `x == 1 or x == 3`, along with `and` and `not`). Note that there is no builtin xor function.

The `if` statement is the simplest for control flow:

```
if x:
    print 'hello1'
if x and y:
    print 'hello2'
```

There is also an `else` statement:

```
if x:
    print 'x is true'
else:
    print 'x is false'
```

In Python, as in many languages, `0` is interpreted as false and `1` as true.

There are two kinds of loops: `while` and `for`. The former loops as long as a statement evaluates to true. A `for` loop iterates over a structure such as an array, rather than the standard `for(int i = 0;i < n;i++)` syntax seen in C-like languages.

Functions are used to organize programs into coherent pieces, to generalize or abstractify code. For example, a square root function could be more generalized into the following example:

```
def root(x,tol):  # The square root of x with tolerance tol
    lower = 0.0
    upper = x
    guess = (upper + lower) / 2
    while (abs(x - guess * guess) > tol):
        if guess * guess > x:
            upper = guess
        else:
            lower = guess
    guess = (upper + lower) / 2
    return guess
```

it is also possible to provide default arguments to a function. In this example, setting `tol` to a default value looks like `def root(x,tol = 0.1)`. Note that functions cannot be overloaded: in Java, different functions may share the same name, but not so in Python unless they have a different number of arguments (which may interact badly with default arguments), since there isn't static typing. Nonetheless, this is usually bad style.

Exercise. Looking back at the function `root`:

- Are there cases where it will have an error? What if $x < 1$?
- How else can it be generalized?

Here is another example of a function:

```
def polyval(p,x):
    val = 0
    i = 0
    for coeff in p:
        val += coeff * (x ** i) # ** is exponentiation
        i = i + 1
    return val

print polyval([1,2,0,1],4) # prints '73'
```

This function can also be written more compactly as:

```
def polyval(p,x):
    val = 0
    for i, coeff in enumerate(p):
        val += coeff * (x ** i)
        i = i + 1
    return val
```

This function reads in a polynomial (though unlike the conventional notation, the highest-degree terms are to the right in the array) and a number to evaluate it at, and evaluates it there.

2. Data Structures: 1/15/13

Data structures are really the meat of Python. Specifically, lists, tuples, strings, and dictionaries will be covered today.

Lists (sometimes called arrays) are a sequence of data that supports indexing (i.e. you can index into a list).

```
arr = [2, 4, 6, 8]
print arr[3] # prints '8'
print arr[-1] # prints '8'
print arr[4] # error
arr.append(10)
print arr[4] # prints '10'
```

This list can serve as a linked-list, queue, array, as anything from your basic data structures class. The + operator concatenates lists, and `list.pop()` removes and returns the head of the list.

Here is an example of list slicing:

```
arr = [2, 4, 6, 8]
print arr[0:2] # prints '[2,4]'
```

One can also compute the length of a list, its maximum, and its minimum (`len()`, `max()`, and `min()`, respectively). More advanced operations such as `filter()` also exist.

Lists can be nested or heterogeneous, such as `arr = [2, [3, 5, 6], 'orange', 6, 'blue']`.

The `for` loop is used to iterate over a list, and the `enumerate` function is convenient for tracking the index:

```
arr = [0, 1, 3, 'blue']
for a in arr:
    print a

if 'blue' in arr:
    print 'Blue is in the string.'  # This gets printed.

squares = [0, 1, 4, 9, 16, 25]
```

```
for i, val in enumerate(squares):
    print i, val
```

This is cleaner and more concise than tracking indices manually.

List comprehensions also exist, and are one of the great advantages of Python: using `squares` as above, one can create a new list as `double_squares = [2 * v for v in squares]`.

Since + represents concatenation, adding two lists componentwise is a little more complicated; one option is `sum = [x + b[i] for i,x in enumerate(a)]` or `sum = [x + y for x, y in zip(a,b)]`. `zip` takes two lists (e.g. `a = [1, 3, 5]` and `b = [2, 4, 6]` and returns a list of tuples of these elements: `zip(a,b) = [(1,2),(3,4),(5,6)]`).

The `range(a,b)` function returns a list of integers between `a` and `b`. If $a = 0$, then the syntax `range(b)` is equivalent. This seems inefficient, and in Python 3, this is rectified by using a generator function. In Python 2.7, though, this is done with the `xrange` function, which is otherwise identical.

Tuples are similar to lists, but they are immutable, Thus, they can be used to enforce structure in code, and also have some performance advantages (though that's a bit beyond the scope of the class). Here are some things that can be done with tuples:

```
p1 = ('start',1.2,-3.0,17.222)
p2 = ('end',-7.3,0.0,-0.0001)

p1[3] = 17.2 # assignment causes an error
print p2[2] # prints '0.0'

# Unpacking is supported.
type1, x1, y1, z1 = p1
type2, x2, y2, z2 = p2
print x1 - x2 # prints '8.5'
```

Strings are also useful, which seems obvious but might be less so in scientific computation. Python has a lot of native string functions. For example, one might parse a vector in the following manner:

```
vec = '[12.4,3,4,7.22]'

# string away the brackets
vec = vec.lstrip('[')
vec = vec.rstrip(']')

nums = vec.split(',')
nums = [float(n) for n in nums]
```

This can actually be done in one line as `nums = [float(n) for n in vec.strip('[]').split(',')]`.

Dictionaries are maps from keys to values. They support assigment: `dict[key] = value` (and access and reassignment in the same way), and the `in` operator can be used to check if a key is present.

One important aspect of Python syntax is the indentation, or white space. Each code block (body of a loop, function definition, etc.) must be indented the same amount or Python will complain. Common standard are 2 or 4 spaces; pick either, but don't mix there. Here is an example of good indentation:

```
vals1 = [1,3,5,7,9]
vals2 = [2,4,6,8]

for v1 in vals1:
    for v2 in vals2:
        if not v1 % 3 and not v2 % 3:
            j = 0
            while j < v1 + v2:
                print j
                j += 1
    print v1
```

Tabs and spaces are another interesting story, but that is architecture- or even editor-dependent. This is a very good reason to not use the Python interpreter for large pieces of code, since any mistake in whitespace undoes all of your work in the event of an error.

If a statement is too long to be easily readable on a line, one can use a backslash at the end of the first line to indicate the lines are joined. In some cases (e.g. in the middle of an array), the interpreter knows what's going on.

A function is implemented with `def`; here are some examples:

```
import cmath # complex math library
def fftk(x,k):  # two arguments, x and k
def fftk(x, k = 0):  # k = 0 if not specified
def fftk(x, k = 0, all = True):  # another default.  Can be called as fftk(1,1,all=False)
```

Functions can be passed as arguments: after defining a function in the manner above, it can be passed to another function (e.g. `operate(x,k,fftk)`, which is defined in some way for these variables). An anonymous function can be defined with `lambda`, such as `operate(x,k,lambda x, k:  x + k)`.

A library can be imported, as in `import math`, after which its functions can be called as `math.sqrt`, etc. A library can be renamed, so `import math as m` means the same function would be called as `m.sqrt` (this is common for `NumPy`, which tends to be imported as `np`).

3. File I/O and Classes: 1/17/13

Suppose there is a text file of chemical compounds with lines of the form `salt:  NaCl`, stored as `compounds.txt`. Ideally, one might want to read this file into a dictionary.

The basic syntax is `f = open('compounds.txt','r')` (the latter argument indicates the permission; here, we only want to read it). Then, calling `f.read()` stores the entire file into a string. Each line can be printed individually:

```
f = open('compounds.txt','r')
for i, line in enumerate(f):
    print 'Line #' + str(i) + ':  '  + line
```

This can also be used to make a dictionary:

```
compounds = {}
with open('compounds.txt','r') as f:
    for line in f:
        compounds[line.split(':')[0]] = line.split(':')[1].strip()
```

The `with-as` formalism makes everything more concice, but in particular means one doesn't have to worry about closing the file.

Of course, this can be done in one line, which is harder to read:

```
compounds = dict([(line.split(':')[0],line.split(':')[1].strip()) \
            for line in open('compounds','r' )])
```

Note how much harder this is to read, and because of the list comprehension, it's both necessary and impossible to close `f`. Additionally, `line.split()` is called twice, which is more work than necessary.

To write a file, replace the `'r'` with a `'w'` in `open()`. Then, appending data to a file is as easy as calling `f.write()`. Often, there's a tradeoff between human readability of file data versus computer readability.

Python also supports object-oriented programming. Specifically, a Python class has:

- instance variables as data (though since Python is interpreted, new instance variables can be added at runtime)
- Functions (called methods; also changeable at runtime).

Some languages (Java and C++) provide data protection with `public` or `private` methods. Python does not do that, but methods or variables prefixed with a double underscore are understood to be internals.

Here is an example:

```
class Stock():
    def __init__(self,name,symbol,prices=[]):
        self.name = name
        self.symbol = symbol
        self.prices = prices

google = Stock('Google','GOOG')
apple = Stock('Apple','AAPL',[500.43,570.60])

print google.symbol
print max(apple.prices)
```

The `__init__` function is a class constructor, called when a new object is created. The double underscores indicate that it is a special Python function.

Functions can also have methods: suppose the following were part of the Stock class.

```
def high_price(self)
    if len(self.prices) is 0:
        return 'MISSING PRICES'
    else:
        return max(self.prices)
```

Then, calling `print apple.high_price()` returns 570.6.

Classes have name-method associations that suggest a resemblance to dicts; this is no coincidence, and interpreted languages tend to implement them as dicts. However, dictionaries are messier in some instances.

There is also inheritance, which leads to subclasses:

```
class StockOption(Stock):
    def __init__(self,name.symbol,opt_price,date,prices=[])
        Stock.__init__(self, name, symbol, prices)
        self.opt_price = opt_price
        self.date = date
```

It is not necessary to inherit everything; some functions can be overwritten if necessary.

Another example of an internal function is `__contains__`, which allows use of the `in` statement (and therefore dictionaries, etc.). `__contains__(self,key)` returns `True` if the key is in the object.

The NumPy library is called `NumPy`, sometimes imported as `np`. This makes math fairly pretty: there are many examples that were mentioned in class but whose contents aren't that useful. The takeaway is that NumPy is very good at math, including linear algebra and statistics.

4. Learning Scientific Tools: NumPy and SciPy: 1/22/13

One of the main features of NumPy is the *n*-dimensional array, the `ndarray` type. The goal is to be able to do lots of interesting things with these arrays, including high-level mathematical functions and functions that call compiled code.

There are many ways to create arrays in NumPy. For example, lists can be assembled into an array: `arr = np.array([[1,2,3],[4,5,6]])` returns a two-dimensional array (the syntax is reminiscent of the TI-83 matrix notation). One can also create sequences, such as `np.arange(0,10,0,1)` and `np.linspace((0,2 * np.pi, 100))`. Arrays can be filled with zeroes and ones, as in `np.zeros((5,5))` (and similarly with `np.ones`), which makes a 5×5 ero matrix. Random matrices also exist, such as `np.random.random(size=(3,4))`.

NumPy has support for saving and loading arrays, as in `np.savetxt(fname='array_out.txt',X=arr)` (where `arr` is some saved `ndarray`) and `loaded_arr = np.loadtxt(fname='array_out.txt')`. The latter function takes a delimiter argument to handle comma-separated values, etc. Then, `np.all(arr == loaded_arr)` returns `True`.[1]

Arrays have lots of attributes and methods, since an array is an object: suppose `arr = np.arange(10).reshape((2,5))` (which takes a list and reshapes it as an array).

- `arr.ndim` returns 2, since it is a 2-dimensional array.
- `arr.shape` returns `(2,5)`.
- There are many, many others.

Most operations on arrays are ufuncts (i.e. they are vectorized). The operations $+$, $-$, $*$, $/$, $**$, `np.log`, $<$, $\leq$, $>$, $\geq$, and `==` are all supported, but perform elementwise operations (and thus possibly returning arrays of a different type, such as bools). This is in contrast to other languages, where `arr1 * arr2` might perform matrix multiplication. Some of these operations can be done in place, such as `arr1 += arr2`, but this preserves the type of the first array, so be careful. Some other supported functions include the dot product, `np.dot(arr1, arr2)` (which is much more general than what one normally defines it). Other NumPy operations, such as `np.sin()` and `np.sqrt()`, also operate on arrays.

Array slicing is somewhat more complicated. In a 4×5 array, there are several ways to return all rows and the last two columns:

- `arr[0:4, 3:5]`
- `arr[:4, 3:5]`
- `arr[:, 3:]`
- `arr[slice(None), slice(3, None)]`. `slice` takes three arguments, a starting index, a stopping index, and a step (so one could just get even-numbered rows, etc.).

[1] One subtlety of this is that loaded arrays are stored as floats unless otherwise specified, so if an array of ints is stored and then retrieved in this manner, a technically different object is created.

There is suport for integer indices, one can use `arr[[1,2], :]` to return the first two rows and all columns, but also `arr[np.array([1,2]), :]`. It is even possible to use boolean indices, as in `arr[[False, True, True, False], :]` or `arr[np.array([False, True, True, False]), :]`; both of these are equivalent to the previous integer examples.

Broadcasting is a method of operating on arrays of different sizes or shapes by copying arrays whenever possible. Vectorization is the use of braodcasting to make more efficient or readable code, with fewer `for` loops, which tends to be faster. For example,

```
import numpy as np
arr = np.random.random((4,5))
arr * 5 # componentwise multipication by 5
arr * np.arange(5) # Scale the nth column by n.
```

An important point is that avoiding `for`loops makes things more readable and faster.

SciPy is another useful library, including statistics (`scipy.stats`), optimization (`scipy.optimize`), sparse matrices (`scipy.sparse`), signal processing (`scipy.signal`), and so on. The goal here is to have lots of methods for scientific computing, so as to prevent reinventing of the wheel. SciPy also works well with NumPy.

Here is an example with statistics:

```
import scipy.stats as stats

# generate two data samples from differently centered distributions
samp1 = stats.norm.rvs(loc=0., scale=1., size=100)
samp2 = stats.norm.rvs(loc=2., scale=1., size=100)
# Perform a ks test:  null hypothesis is that they are the same
D, pval = stats.ks_2samp(samp1,samp2) # Returns D = .58, pval=1.34e-15
```

One can also use SciPy with NumPy for (statistical) bootstrapping, to obtain error bars on an estimate of the average of the data if the actual distribution of the data isn't fully known.

```
import numpy as np
import scipy.stats as stats

B = 1000
N = 100

arr = stats.norm.rvs(loc=np.pi,size=100)
# computes distribution of mean estimates
mean_distn = np.array([np.mean(arr[np.random.randint(N,size=N)]) for i in xrange(B)])
# 95% confidence interval [2.99, 3.39]
confidence_bounds = stats.mstats.mqualtiles(mean_distn,prob=[.025,.975])
```

5. Data Visualization and Web Scraping: 1/24/13

`matplotlib` is a Python library for plotting 2D and 3D data in a way that should be reminiscent of Matlab. Here is a simple example:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,10,1000)
y = np.power(x,2)
plt.plot(x,y)
plt.xlim((1,5))
plt.ylim((0,30))
plt.xlabel('my x label')
ply.ylabel('my y label')
plt.title('plot title, including $\ Omega$')
plt.legend(('$x^2$','$x^3$'))
```

Interestingly, one can use LaTeX notation in the titles and such of plots.

Histograms are also possible, as `plt.hist`. Another useful option is `plt.savefig('filename.png')`, which does what one might expect it to. This is a common theme; box plots are created as `plt.boxplot`, which accepts any list or tuple and an awful lot of options that allow one to customize the picture.

A scatter plot matrix (which is used to compare correlations of several different attributes of data) is in fact not part of the `matplotlib` defaults. One has to use Matlab or a package called `pandas`.

Image plots exist as well, with syntax `plt.imshow(data)`. Here it helps to add `plt.colorbar()`, which provides a key.

3D plots exist as well, but the syntax is slightly different:

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
```

This additional library ships with `matplotlib`, so this isn't really that much more of a hurdle.

It will be helpful to see some more (not necessarily Python-specific) ideas or concepts that help one develop in Python. For example, Python has no standard IDE; Eclipse has a Python mode and is probably the easiest to set up, and emacs and vim also have some resources for using Python.

Another useful concept is version control. This is incredibly important for proper organization, so that many people can see the same piece of code, merge changes, and coordinate the whole process. The industry standard is a program called git, but an older standard called subversion also exists. Using a website called GitHub makes this all much easier, and was actually used by the instructors in designing this class.

Another incredibly important technique is debugging. Consider the following code:

```
import numpy as np
def buggy():
    A = np.arange(1,10)
    import ipdb; ipdb.set_trace() # breakpoint
    A /= 2.
    return np.sum (5/A)
```

Ignoring the breakpoint, this will cause a runtime division by zero. However, adding the line of `ipdb` code causes a breakpoint and enters a debugging environment similar to `gdb`. For example, one can use commands such as `n` to advance the program, and one can print variables and stuff as if one were in the interpreter. This also allows conditional setting of breakpoints, which is very helpful for some bugs.

Note that if you have many breakpoints, it's better to import `ipdb` at the top of the file; otherwise, it might not get imported for one of the breakpoints one wants.

Another useful application is Web scraping. This is a three-step process involving visual inspection (using, say, Chrome developer mode or Firebug), browser sessions and interacting with the HTML (filling out forms, navigating links), which is aided by the `navigate` module of Python, and then parsing HTML, which can be accomplished by a module called `BeautifulSoup`. Note that Javascript is sometimes confusing; a module called `selenium` might be helpful there.

For example, one could create a Browser object via `mechanize.addBrowser()`, and then adding the user-agent information with, for example, `br.addheaders = [('User-agent',...)]`. In order to parse the HTML, one creates a BeautifulSoup object:

```
# soup is the BeautifulSoup object
tbl = soup.find(lambda tag:  (tag.name == 'caption') and (tag.text == 'Anscombe\'s quartet'))
arr_list = []
for row in tbl.findAll('tr') # iterate over rows
    elements = row.findAll('td')
    if len(elements) != 0:# so that there is actual data
        try:  # This structure guards against potential errors.
            np.float(elements[0].string)
        except:  # This is executed if there is an error.
            continue
        arr_list.append(np.array([np.float(elem.string) for elem in elements]))
data = np.vstack(arr_list)
# Then create the plot...
grid = np.linspace(2, 20, 100)
for i in xrange(4):
    x = data[:, 2 * i]; y = data[:, 2 * i + 1]
    a,b = scipy.polyfit(x, y, 1)
    plt.subplot(2, 2, i + 1)
    # et cetera
```