e62 Introduction to Optimization — Autumn 2016
Professor Benjamin Van Roy — September 15, 2016

Python Tutorial

In this course, we will use the python programming language and tools that support optimization in python. Many of you may be familiar with python from other classes or extracurricular programming experience. For those who are not, this presents a valuable opportunity to learn about what may be the most widely used language for data and decision sciences. This document aims to introduce a small subset of the python programming language and associated tools that suffices for the needs of computational assignments in this class. Because python is so widely used, there are numerous online resources and discussion forums, so it is easy to extend your knowledge of the language through web searches on specific topics or questions.

1 Virtual Machine

In order for all students to share a uniform environment for computation, we have created a virtual machine with all the software required for e62. A virtual machine is software that simulates a computer. By working through the e62 virtual machine, each student will essentially be working on an identical computer. This avoids idiosyncrasies that may allow one person but not another to get something to work on their computer. It also obviates the need to install multiple pieces of software – the virtual machine comes with the necessary software pre-installed. To obtain the virtual machine, students should download and install VirtualBox and then download and unzip the e62 virtual machine file. Links to these resources are on the e62 software web page. Once you have VirtualBox and the e62 virtual machine on your computer, you can turn on the virtual machine by launching the VirtualBox application and selecting from the menu bar Machine → Add, and then selecting and opening e62.vbox. Then select e62 in the left pane of the VirtualBox Manager and press Start (the green arrow).

You can maximize the virtual machine window so that it occupies your entire screen. When you first maximize the window, VirtualBox will provide you with instructions on keystrokes that you can later apply to reduce the window size in order to return to your host operating system. On a Macintosh computer its `command+F`. The e62 virtual machine runs Ubuntu, which is a linux-based operating system of choice for many professional software developers. It looks and feels somewhat like Windows and OS X. One of the reasons we use this operating system is that it is available free of charge and running it on any computer does not violate license agreements. Ubuntu also presents some differences. One that is useful to keep in mind is that to copy and paste highlighted text, one uses `ctrl-C` and `ctrl-V`, except in the terminal window, where one instead uses `ctrl-shift-C` and `ctrl-shift-V`. Copying and pasting can also be accomplished by using the pulldown menu that appears upon pressing the right mouse button while pointing at highlighted text.

For e62 assignments you will primarily be interacting with the command line terminal and a text editor called notepadqq. Each can be opened by clicking on the corresponding icon on the launcher, which runs along the left edge of the screen.

2 ipython

An easy way to get started with python is to use ipython, which is an interactive shell. To start ipython, open a command line terminal. At the command line, type `ipython`:

```
e62@e62-VM:~$ ipython
Python 2.7.12 |Anaconda 4.1.1 (64-bit)| (default, Jul  2 2016, 17:42:40)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print 'hello world'
hello world

In [2]:
```

The above transcript includes an instruction at the first ipython command prompt to print “hello world.”

It is straightforward to carry out basic arithmetic and variable assignments:

```
In [2]: 2 + 5
Out[2]: 7

In [3]: 2.1 * 10
Out[3]: 21.0

In [4]: a = 2

In [5]: b = 5

In [6]: c = a + b

In [7]: print c
7
```

One thing to be careful for, though, is that integers and floats are distinct types in python, and arithmetic operations are defined differently for them. The following example illustrates problems that can arise:

```
In [8]: 2.0/3.0
Out[8]: 0.6666666666666666

In [9]: 2/3
Out[9]: 0
```

Note that integer two divided by integer three is zero because integer division truncates fractions. When integers and floats are mixed in a calculation, python converts all the numbers to float:

```
In [10]: 2/3.0
Out[10]: 0.6666666666666666
```

The `type` command can be used to check the type of a variable. Further, python accommodates type conversions, as the following example illustrates:

```
In [11]: type(c)
Out[11]: int

In [12]: d = float(c)
In [13]: d
Out[13]: 7.0

In [14]: type(d)
Out[14]: float
```

The exit command terminates the ipython session:

```
In [15]: exit
e62@e62-VM:~$
```

3 Scripts

To save sequences of instructions and to facilitate debugging, it is often helpful to write instructions in a script, which is a text file consisting of python instructions. To create a script, open notepadqq and create a new text file. As an example, consider a the following instructions:

```
print 'hello world'
a = 2
b = 5
print(a + b)
```

After saving this file as `/home/e62/python/script_example.py`, we can execute the script from the linux command line:

```
e62@e62-VM:~$ python script_example.py
hello world
7
e62@e62-VM:~$
```

Scripts can also be executed from ipython:

```
In [1]: %run script_example.py
hello world
7
```

One can insert text comments in a script on any line after the `#` symbol. Any text appearing to the right of the `#` is ignored by the python interpreter.

4 Data structures

Let us now discuss a few data structures that we will use regularly in computational assignments.

4.1 Lists

A list consists of a sequence of elements of any type. Let us illustrate construction, inspection, and manipulation of lists:

```
In [1]: alist = ['elephant', 532, 0.001, 'zebra', 50.05]

In [2]: alist
Out[2]: ['elephant', 532, 0.001, 'zebra', 50.05]

In [3]: alist[0]
Out[3]: 'elephant'
```

```
In [4]: alist[1]
Out[4]: 532

In [5]: alist[4]
Out[5]: 50.05

In [6]: alist[-1]
Out[6]: 50.05

In [7]: alist[-2]
Out[7]: 'zebra'

In [8]: blist = [1,2,3,4]

In [9]: alist + blist
Out[9]: ['elephant', 532, 0.001, 'zebra', 50.05, 1, 2, 3, 4]
```

Note that we can create lists of lists:

```
In [10]: clist = [[1,2], [3,4,5], 6]

In [11]: clist[0]
Out[11]: [1, 2]

In [12]: clist[1]
Out[12]: [3, 4, 5]

In [13]: clist[2]
Out[13]: 6
```

List comprehensions offer a useful tool for manipulating lists, as the following example illustrates:

```
In [1]: alist = range(10)

In [2]: alist
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: blist = [2*a for a in alist]

In [4]: blist
Out[4]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [5]: [2*a for a in range(10)]
Out[5]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The `range` function produces a list of integers; in this case, a list ranging from 0 to 9. The content within the square brackets used to define `blist` is interpreted as a loop that iterates over elements of `alist`, generating a new list by multiplying each element by 2.

4.2 Tuples

A tuple is similar to a list except that it is immutable. In other words, it cannot be changed in any way once it is created. While lists are constructed using square brackets, tuples are constructed using parentheses. The following example illustrates how lists can be modified while tuples can not:

```
In [14]: clist
Out[14]: [[1, 2], [3, 4, 5], 6]

In [15]: clist[0] = 101

In [16]: clist
Out[16]: [101, [3, 4, 5], 6]

In [17]: ctuple = ((1,2), (3,4,5), 6)

In [18]: ctuple[0] = 101
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-d5dcef241166> in <module>()
----> 1 ctuple[0] = 101

TypeError: 'tuple' object does not support item assignment

In [19]: ctuple
Out[19]: ((1, 2), (3, 4, 5), 6)
```

4.3 Numpy matrices

Numpy is a python software package that facilitates numerical computation. Among other things, numpy offers tools to work with matrices. The following example illustrates the construction of a matrix:

```
In [1]: A = np.matrix([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [2]: A
Out[2]:
matrix([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
```

Note that `np` is an abbreviation for numpy, and we produced the matrix using numpy's `matrix` function, providing it with a list of lists of numbers (a tuple of tuples would also work). This list can be thought of as a list of rows, where each row is a list of matrix entries. We can access rows, columns, or other subsets of elements:

```
In [1]: A = np.matrix([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [2]: A
Out[2]:
matrix([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])

In [3]: A[0,:]
Out[3]: matrix([[1, 2, 3, 4]])

In [4]: A[:,1]
Out[4]:
matrix([[ 2],
        [ 6],
        [10]])
```

```
In [5]: A[1,1:2]
Out[5]: matrix([[6]])

In [6]: A[1,1:3]
Out[6]: matrix([[6, 7]])

In [7]: A[[0,2],[1,3]]
Out[7]: matrix([[ 2, 12]])
```

Note that a range of the form `m:n` includes the $n-m$ indices $m, m+1, n-1$, while excluding n. Transposition is applied via `.T`:

```
In [8]: A.T
Out[8]:
matrix([[ 1,  5,  9],
        [ 2,  6, 10],
        [ 3,  7, 11],
        [ 4,  8, 12]])
```

Row and column vectors are simply matrices with single rows or columns. Vector components can be specified by a list of numbers rather than a list of lists of numbers:

```
In [1]: np.matrix([1,2,3])
Out[1]: matrix([[1, 2, 3]])

In [2]: np.matrix([1,2,3]).T
Out[2]:
matrix([[1],
        [2],
        [3]])
```

Rows or columns of a matrix can also be converted to vectors:

```
In [1]: A = np.matrix([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [2]: A[1,:]
Out[2]: matrix([[5, 6, 7, 8]])

In [3]: A[:,2]
Out[3]:
matrix([[ 3],
        [ 7],
        [11]])
```

Matrix arithmetic is straightforward:

```
In [1]: A = np.matrix([[1,2],[3,4]])

In [2]: B = np.matrix([[5,6],[7,8]])

In [3]: A+B
Out[3]:
matrix([[ 6,  8],
        [10, 12]])

In [4]: A*B
Out[4]:
matrix([[19, 22],
        [43, 50]])

In [5]: np.multiply(A,B)
Out[5]:
matrix([[ 5, 12],
        [21, 32]])
```

Note that `np.multiply` and `np.divide` carry out element-wise multiplication and division for matrices of identical dimensions. On the topic of dimensions, the dimensions of a matrix can be obtained in the form of a tuple as follows:

```
In [6]: A.shape
Out[6]: (2, 2)
```

Matrices can be concatenated if their dimensions are compatible:

```
In [8]: np.concatenate((A,B),axis=0)
Out[8]:
matrix([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])

In [9]: np.concatenate((A,B),axis=1)
Out[9]:
matrix([[1, 2, 5, 6],
        [3, 4, 7, 8]])

In [10]: np.concatenate((A,B,B,A),axis=1)
Out[10]:
matrix([[1, 2, 5, 6, 5, 6, 1, 2],
        [3, 4, 7, 8, 7, 8, 3, 4]])
```

Note that the first argument to `np.concatenate` is a tuple of matrices, while the axis argument specifies whether the concatenation should be done vertically or horizontally.

There are dedicated functions for producing particular kinds of matrices:

```
In [1]: np.matlib.identity(3)
Out[1]:
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])

In [2]: np.matlib.zeros((2,3))
Out[2]:
matrix([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

In [3]: np.matlib.ones((2,3))
Out[3]:
matrix([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])

In [4]: np.matlib.rand((2,3))
Out[4]:
matrix([[ 0.8925401 ,  0.62319141,  0.23885902],
        [ 0.95715811,  0.27609046,  0.32808951]])
```

```
In [5]: np.matlib.randn((2,3))
Out[5]:
matrix([[ 0.22242987,  0.51574683,  0.59468841],
        [-1.78117514, -0.01304228, -1.16989767]])
```

The functions `np.matlib.rand` and `mp.matlib.randn` return random matrices. In the first case, each entry is drawn from a uniform distribution over the interval $[0, 1]$. In the second case, each entry is drawn from a Gaussian distribution with mean zero and variance one.

There is also a function that inverts matrices:

```
In [6]: A = np.matlib.randn((2,2))

In [7]: A_inv = np.linalg.inv(A)

In [8]: A
Out[8]:
matrix([[-0.45949096,  1.55846699],
        [-0.25718063,  3.08378464]])

In [9]: A_inv
Out[9]:
matrix([[-3.0347323 ,  1.53367718],
        [-0.25308977,  0.45218205]])

In [10]: A * A_inv
Out[10]:
matrix([[  1.00000000e+00,   0.00000000e+00],
        [ -1.11022302e-16,   1.00000000e+00]])
```

Note that the product of the matrix and its inverse is virtually equal to the identity matrix; the small difference is due to errors stemming from limited numerical precision.

5 Conditional and Iterative Statements

As with other common programming languages, python supports syntax for conditional and iterative statements.

5.1 if...else

This is an example of a simple conditional statement:

```
if x == 3:
    x = x + 1
    print "new value of x is 4"
print "done"
```

If the value of the variable x is 3, the value of x will increase by one and then the message will print; otherwise, the two lines following the `if` statement will be skipped. A distinctive element of python is its use of white space to distinguish code blocks. In the above example, the two lines following the `if` statement make up a code block, while the line that prints “done” is not part of the code block, and as such is executed regardless of the value of x. The code block is distinguished by a common level of indentation relative to the `if` statement. For consistency, we recommend using a tab or four spaces for identation as in the example above.

Here is a more complicated conditional statement:

```
if x < 3:
    print "x is less than 3"
elif (x >= 3) and (x < 6):
    print "x is greater or equal to 3 and smaller than 6"
else:
    print "x is greater or equal to 6"
```

A conditional statement can have multiple `elif` statements but it can have at most one `else` statement.

5.2 for...in

Below is an example of looping in python:

```
In [11]: alist = ['elephant', 532, 0.001, 'zebra', 50.05]
In [12]: for i in alist:
    ...:     print i
elephant
532
0.001
zebra
50.05
```

Python allows iterating over lists. Similar to conditional statements, indentation is required to distinguish the code block that is repeatedly executed.

To iterate over a list of consecutive integers, one can use the `range` function:

```
In [13]: for i in range(4):
    ...:     print i
0
1
2
3
```

6 Modules

A module is a file containing python code. When a module is loaded in ipython, its code is executed, and variables and functions defined in by the module are available for use. To distinguish names used in the module from other work, when accessing a function or variable from a loaded module, the module name must be used as a prefix. For example, numpy is a module. To use a function from numpy in ipython, one would load it using the `import` command:

```
In [1]: import numpy

In [2]: numpy.sqrt(2)
Out[2]: 1.4142135623730951

In [3]: numpy.nan
Out[3]: nan

In [4]: numpy.inf
Out[4]: inf
```

The names `sqrt`, `nan`, and `inf` are defined in numpy. The first is a square-root function. The latter represent undefined or infinite outcomes to computations carried out using numpy tools. For example:

```
In [5]: numpy.divide(7.0,0.0)
Out[5]: inf

In [6]: numpy.divide(0.0,0.0)
Out[6]: nan
```

Note that there is no need for us to load numpy; the virtual machine is set up to do this automatically when ipython starts. Further, an alias of np is assigned. To assign an alias when loading a module, one can do this:

```
In [1]: import numpy as np
```

The code that is executed to load modules when ipython starts up is in the file `/home/e62/python/startup/startup.py`. A script that makes use of a module must load the module within the script before it is used. This is true even if the module has already been loaded in ipython and the script is being executed from ipython.

7 The e62 module

The e62 module was written specifically for the course and is among the modules automatically loaded when ipython starts. The module is in the file `/home/e62/python/startup/e62.py`. In ipython, documentation provided by comments in a module can be obtained by using the help function. If you type `help(62)` at the ipython prompt and press return, you will see documentation pages listing the functions and describing their behavior. You can scroll up or down when viewing this documentation by using up and down arrow keys for single-line moves or the space bar to move forward to the next page. Press `q` to exit the documentation viewing mode. You can also view documentation for a single function, for example, by entering `help(e62.print_matrix)`.

The e62 module includes functions for printing tables and saving and loading matrices:

```
In [1]: A = np.matrix([[4, 2.86], [3, 6], [4.44, 0], [0, 6.67]])
In [2]: rows = ['metal stamping', 'engine assembly', 'automobile assembly', 'truck assembly']

In [3]: columns = ['automobile', 'truck']

In [4]: e62.print_matrix(A, rows, columns)
labels                 automobile    truck
-------------------  ------------  -------
metal stamping               4        2.86
engine assembly              3        6
automobile assembly          4.44     0
truck assembly               0        6.67

In [5]: e62.save_matrix('resource_usage.csv', A, rows, columns)
In [6]: A_copy,rows_copy,columns_copy = e62.load_matrix('resource_usage.csv')

In [7]: e62.print_matrix(A_copy, rows_copy, columns_copy)

labels                 automobile    truck
-------------------  ------------  -------
metal stamping               4        2.86
engine assembly              3        6
automobile assembly          4.44     0
truck assembly               0        6.67
```

Matrices are saved as comma-separated-value files. Here is the content of `resource_usage.csv`:

```
e62@e62-VM:~$ cat resource_usage.csv
labels,automobile,truck
metal stamping,2.0,2.86
engine assembly,3.0,6.0
automobile assembly,4.44,0.0
truck assembly,0.0,6.67
```

Of particular pertinence is a function for solving linear programs:

```
In [1]: A = np.matrix([[4, 2.86], [3, 6], [4.44, 0], [0, 6.67]])

In [2]: b = np.matrix([100, 100, 100, 100]).T

In [3]: c = np.matrix([3, 2.5])

In [4]: x, objective_value = e62.linprog(A, b, c)
Optimization terminated successfully.

In [5]: e62.print_matrix(x, ['automobiles', 'trucks'], ['quantity (in thousands)'])

labels         quantity (in thousands)
-----------  -------------------------
automobiles                    20.3632
trucks                          6.48508
```

Note that when a function generates multiple outputs and the output is assigned to a single variable, the variable is a tuple containing all the outputs:

```
In [6]: sol = e62.linprog(A, b, c)
Optimization terminated successfully.

In [7]: sol[0]
Out[7]:
matrix([[ 20.36316472],
        [  6.48508431]])

In [8]: sol[1]
Out[8]: 77.302204928664082
```

Among other functions, the e62 module offers one that solves linear systems of equations:

```
In [1]: A = np.matrix([[4, 2.86], [3, 6]])

In [2]: b = np.matrix([100, 100]).T

In [3]: x, solved = e62.lineqsolve(A, b)
solved

In [4]: x
Out[4]:
matrix([[ 20.36316472],
        [  6.48508431]])

In [5]: solved
Out[5]: True
```

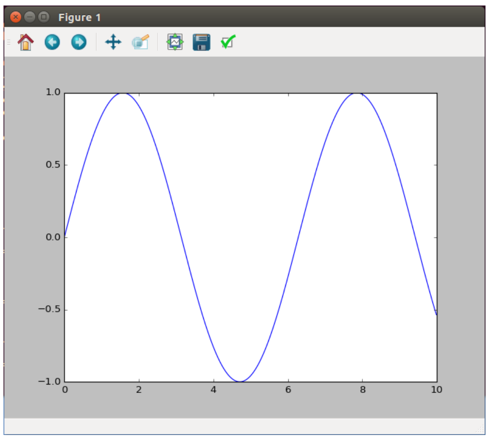

Figure 1: sinusoid

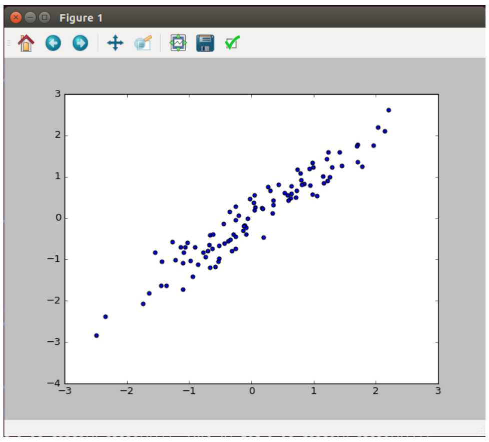

Figure 2: scatter plot

```
In [6]: A = np.matrix([[4, 2.86], [3, 6], [4.44, 0], [0, 6.67]])

In [7]: b = np.matrix([100, 100, 100, 100]).T

In [8]: x, solved = e62.lineqsolve(A, b)
no solution

In [9]: x
Out[9]:
matrix([[ 17.98479899],
        [ 11.54330476]])

In [10]: solved
Out[10]: False
```

In the event that there is no solution to the system of linear equations, solved is set to `False` and `x` is the vector that minimizes the Euclidean norm of $Ax - b$. If there are multiple solutions, `x}`is taken to be the one with minimal Euclidean norm.

8 Plotting

The `matplotlib.pyplot` module provides useful tools for plotting data. This module is automatically loaded when you start ipython. Here is an example where we plot a sinusoid:

```
In [1]: x = [z/100.0 for z in range(1000)]

In [2]: y = [np.sin(z) for z in x]

In [3]: plt.plot(x, y)
Out[3]: [<matplotlib.lines.Line2D at 0x7f7d1411d7d0>]

In [4]: plt.show(block=False)
```

Figure 1 shows the popup window that appears with the execution of `plt.show(block=False)`. Without the `block=False` argument, ipython freezes until the plot window is closed. The `range` function here produces a list of integers from 0 to 999. Note that the variables `x` and `y` are defined by list comprehensions.

We can also produce a scatter plot, as depicted in Figure 2:

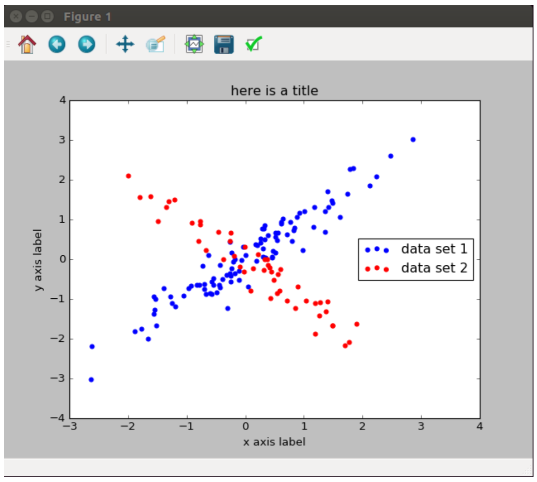

Figure 3: a more sophisticated scatter plot

```
In [1]: x = np.matlib.randn((100,1))

In [2]: y = x + 0.3 * np.matlib.randn((100,1))

In [3]: plt.scatter([x[i,0] for i in range(x.shape[0])], [y[i,0] for i in range(y.shape[0])])
Out[3]: <matplotlib.collections.PathCollection at 0x7ffb116cea10>

In [4]: plt.show(block=False)
```

More sophisticated things can also be done, like plotting multiple data sets and adding labels and legends. The following script generates Figure 3:

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

## generate data sets
x1 = np.matlib.randn((100,1))
y1 = x1 + 0.3 * np.matlib.randn((100,1))
x1 = [x1[i,0] for i in range(x1.shape[0])]
y1 = [y1[i,0] for i in range(y1.shape[0])]
x2 = np.matlib.randn((50,1))
y2 = -x2 + 0.3 * np.matlib.randn((50,1))
x2 = [x2[i,0] for i in range(x2.shape[0])]
y2 = [y2[i,0] for i in range(y2.shape[0])]

## produce figure and plots
myfig = plt.figure()
myplot = myfig.add_subplot(111)
scatter1 = myplot.scatter(x1, y1, color='blue')
scatter2 = myplot.scatter(x2, y2, color='red')
myplot.set_title('here is a title')
myplot.set_xlabel('x axis label')
myplot.set_ylabel('y axis label')
myplot.legend((scatter1, scatter2), ('data set 1', 'data set 2'), loc='right')
myfig.show()
```