Math and Python review and CTR data download

This notebook reviews vector and matrix math, the NumPy (http://www.numpy.org/) Python package, and Python lambda expressions. It also covers downloading the data required for Lab 4, where you will analyze website click-through rates. Part 1 covers vector and matrix math, and you'll do a few exercises by hand. In Part 2, you'll learn about NumPy and use `ndarray` objects to solve the math exercises. Part 3 provides additional information about NumPy and how it relates to array usage in Spark's MLlib (https://spark.apache.org/mllib/). Part 4 provides an overview of lambda expressions, and you'll wrap up by downloading the dataset for Lab 4.

To move through the notebook just run each of the cells. You can run a cell by pressing "shift-enter", which will compute the current cell and advance to the next cell, or by clicking in a cell and pressing "control-enter", which will compute the current cell and remain in that cell. You should move through the notebook from top to bottom and run all of the cells. If you skip some cells, later cells might not work as expected.

Note that there are several exercises within this notebook. You will need to provide solutions for cells that start with: `# TODO: Replace <FILL IN> with appropriate code`.

This notebook covers:

Part 1: Math review

Part 2: NumPy

Part 3: Additional NumPy and Spark linear algebra

Part 4: Python lambda expressions

Part 5: CTR data download

In [ ]:

```
labVersion = 'cs190_week1_v_1_2'
```

Part 1: Math review

(1a) Scalar multiplication: vectors

In this exercise, you will calculate the product of a scalar and a vector by hand and enter the result in the code cell below. Scalar multiplication is straightforward. The resulting vector equals the product of the scalar, which is a single value, and each item in the original vector. In the example below, a is the scalar (constant) and $\mathbf{v}$ is the vector.

$$a\mathbf{v} = \begin{bmatrix} av_1 \\ av_2 \\ \vdots \\ av_n \end{bmatrix}$$

Calculate the value of $\mathbf{x}$:

$$\mathbf{x} = 3 \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$$

Calculate the value of $\mathbf{y}$:

$$\mathbf{y} = 2 \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}$$

In [1]:

```
# TODO: Replace <FILL IN> with appropriate code
# Manually calculate your answer and represent the vector as a list of integers value
# For example, [2, 4, 8].
x = [3, -6, 0]
y = [4, 8, 16]
```

In [2]:

```
# TEST Scalar multiplication: vectors (1a)
# Import test library
from test_helper import Test
Test.assertEqualsHashed(x, 'e460f5b87531a2b60e0f55c31b2e49914f779981',
                        'incorrect value for vector x')
Test.assertEqualsHashed(y, 'e2d37ff11427dbac7f833a5a7039c0de5a740b1e',
                        'incorrect value for vector y')
```

```
1 test passed.
1 test passed.
```

(1b) Element-wise multiplication: vectors

In this exercise, you will calculate the element-wise multiplication of two vectors by hand and enter the result in the code cell below. You'll later see that element-wise multiplication is the default method when two NumPy arrays are multiplied together. Note we won't be performing element-wise multiplication in future labs, but we are introducing it here to distinguish it from other vector operators, and to because it is a common operations in NumPy, as we will discuss in Part (2b).

The element-wise calculation is as follows:

$$\mathbf{x} \odot \mathbf{y} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ \vdots \\ x_n y_n \end{bmatrix}$$

Calculate the value of $\mathbf{z}$:

$$\mathbf{z} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \odot \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}$$

In [9]:

```
# TODO: Replace <FILL IN> with appropriate code
# Manually calculate your answer and represent the vector as a list of integers value
z = [4, 10, 18]
```

In [10]:

```
# TEST Element-wise multiplication: vectors (1b)
Test.assertEqualsHashed(z, '4b5fe28ee2d274d7e0378bf993e28400f66205c2',
                        'incorrect value for vector z')
```

```
1 test passed.
```

(1c) Dot product

In this exercise, you will calculate the dot product of two vectors by hand and enter the result in the code cell below. Note that the dot product is equivalent to performing element-wise multiplication and then summing the result.

Below, you'll find the calculation for the dot product of two vectors, where each vector has length n:

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{n} w_i x_i$$

Note that you may also see $\mathbf{w} \cdot \mathbf{x}$ represented as $\mathbf{w}^\top \mathbf{x}$

Calculate the value for c_1 based on the dot product of the following two vectors:

$$c_1 = \begin{bmatrix} 1 \\ -3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 5 \end{bmatrix}$$

Calculate the value for c_2 based on the dot product of the following two vectors:

$$c_2 = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

In [11]:

```
# TODO: Replace <FILL IN> with appropriate code
# Manually calculate your answer and set the variables to their appropriate integer v
c1 = -11
c2 = 26
```

In [12]:

```
# TEST Dot product (1c)
Test.assertEqualsHashed(c1, '8d7a9046b6a6e21d66409ad0849d6ab8aa51007c', 'incorrect va
Test.assertEqualsHashed(c2, '887309d048beef83ad3eabf2a79a64a389ab1c9f', 'incorrect va
```

```
1 test passed.
1 test passed.
```

(1d) Matrix multiplication

In this exercise, you will calculate the result of multiplying two matrices together by hand and enter the result in the code cell below.

Below, you'll find the calculation for multiplying two matrices together. Note that the number of columns for the first matrix and the number of rows for the second matrix have to be equal and are are represented by n:

$$[\mathbf{XY}]_{i,j} = \sum_{r=1}^{n} \mathbf{X}_{i,r} \mathbf{Y}_{r,j}$$

First, you'll calculate the value for $\mathbf{X}$.

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Next, you'll perform an outer product and calculate the value for $\mathbf{Y}$. Note that outer product is just a special case of general matrix multiplication and follows the same rules as normal matrix multiplication.

$$\mathbf{Y} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

In [13]:

```
# TODO: Replace <FILL IN> with appropriate code
# Represent matrices as lists within lists. For example, [[1,2,3], [4,5,6]] represent
# two rows and three columns. Use integer values.
X = [[22, 28], [49, 64]]
Y = [[1,2,3], [2,4,6], [3,6,9]]
```

In [14]:

```
# TEST Matrix multiplication (1d)
Test.assertEqualsHashed(X, 'c2ada2598d8a499e5dfb66f27a24f444483cba13',
                        'incorrect value for matrix X')
Test.assertEqualsHashed(Y, 'f985daf651531b7d776523836f3068d4c12e4519',
                        'incorrect value for matrix Y')
```

```
1 test passed.
1 test passed.
```

Part 2: NumPy

(2a) Scalar multiplication

NumPy (http://docs.scipy.org/doc/numpy/reference/) is a Python library for working with arrays. NumPy provides abstractions that make it easy to treat these underlying arrays as vectors and matrices. The library is optimized to be fast and memory efficient, and we'll be using it throughout the course. The building block for NumPy is the ndarray (http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html), which is a multidimensional array of fixed-size that contains elements of one type (e.g. array of floats).

For this exercise, you'll create a `ndarray` consisting of the elements [1, 2, 3] and multiply this array by 5. Use np.array() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html) to

create the array. Note that you can pass a Python list into `np.array()`. To perform scalar multiplication with an `ndarray` just use `*`.

Note that if you create an array from a Python list of integers you will obtain a one-dimensional array, *which is equivalent to a vector for our purposes*.

In [15]:

```
# It is convention to import NumPy with the alias np
import numpy as np
```

In [16]:

```
# TODO: Replace <FILL IN> with appropriate code
# Create a numpy array with the values 1, 2, 3
simpleArray = np.array([1,2,3])
# Perform the scalar product of 5 and the numpy array
timesFive = 5 * simpleArray
print simpleArray
print timesFive
```

```
[1 2 3]
[ 5 10 15]
```

In [17]:

```
# TEST Scalar multiplication (2a)
Test.assertTrue(np.all(timesFive == [5, 10, 15]), 'incorrect value for timesFive')
```

```
1 test passed.
```

(2b) Element-wise multiplication and dot product

NumPy arrays support both element-wise multiplication and dot product. Element-wise multiplication occurs automatically when you use the * operator to multiply two `ndarray` objects of the same length.

To perform the dot product you can use either np.dot() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html#numpy.dot) or np.ndarray.dot() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.dot.html). For example, if you had NumPy arrays `x` and `y`, you could compute their dot product four ways: `np.dot(x, y)`, `np.dot(y, x)`, `x.dot(y)`, or `y.dot(x)`.

For this exercise, multiply the arrays `u` and `v` element-wise and compute their dot product.

In [18]:

```
# TODO: Replace <FILL IN> with appropriate code
# Create a ndarray based on a range and step size.
u = np.arange(0, 5, .5)
v = np.arange(5, 10, .5)

elementWise = u * v
dotProduct = np.dot(u, v)
print 'u: {0}'.format(u)
print 'v: {0}'.format(v)
print '\nelementWise\n{0}'.format(elementWise)
print '\ndotProduct\n{0}'.format(dotProduct)
```

```
u: [ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5]
v: [ 5.   5.5  6.   6.5  7.   7.5  8.   8.5  9.   9.5]

elementWise
[  0.     2.75   6.     9.75  14.    18.75  24.    29.75  36.    42.75]

dotProduct
183.75
```

In [19]:

```
# TEST Element-wise multiplication and dot product (2b)
Test.assertTrue(np.all(elementWise == [ 0., 2.75, 6., 9.75, 14., 18.75, 24., 29.75, 3
                'incorrect value for elementWise')
Test.assertEquals(dotProduct, 183.75, 'incorrect value for dotProduct')
```

```
1 test passed.
1 test passed.
```

(2c) Matrix math

With NumPy it is very easy to perform matrix math. You can use np.matrix() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html) to generate a NumPy matrix. Just pass a two-dimensional `ndarray` or a list of lists to the function. You can perform matrix math on NumPy matrices using *.

You can transpose a matrix by calling numpy.matrix.transpose() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.transpose.h or by using `.T` on the matrix object (e.g. `myMatrix.T`). Transposing a matrix produces a matrix where the new rows are the columns from the old matrix. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Inverting a matrix can be done using numpy.linalg.inv() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html). Note that only square matrices can be inverted, and square matrices are not guaranteed to have an inverse. If the inverse exists, then multiplying a matrix by its inverse will produce the identity matrix. $(\mathbf{A}^{-1}\mathbf{A} = \mathbf{I_n})$ The identity matrix $\mathbf{I_n}$ has ones along its diagonal and zero elsewhere.

$$\mathbf{I_n} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

For this exercise, multiply $\mathbf{A}$ times its transpose $\left(\mathbf{A}^\top\right)$ and then calculate the inverse of the result $\left(\left[\mathbf{A}\mathbf{A}^\top\right]^{-1}\right)$.

In [21]:

```
# TODO: Replace <FILL IN> with appropriate code
from numpy.linalg import inv

A = np.matrix([[1,2,3,4],[5,6,7,8]])
print 'A:\n{0}'.format(A)
# Print A transpose
print '\nA transpose:\n{0}'.format(A.T)

# Multiply A by A transpose
AAt = A*A.transpose()
print '\nAAt:\n{0}'.format(AAt)

# Invert AAt with np.linalg.inv()
AAtInv = inv(AAt)
print '\nAAtInv:\n{0}'.format(AAtInv)

# Show inverse times matrix equals identity
# We round due to numerical precision
print '\nAAtInv * AAt:\n{0}'.format((AAtInv * AAt).round(4))
```

```
A:
[[1 2 3 4]
 [5 6 7 8]]

A transpose:
[[1 5]
 [2 6]
 [3 7]
 [4 8]]

AAt:
[[ 30  70]
 [ 70 174]]

AAtInv:
[[ 0.54375 -0.21875]
 [-0.21875  0.09375]]

AAtInv * AAt:
[[ 1.  0.]
 [-0.  1.]]
```

In [22]:

```
# TEST Matrix math (2c)
Test.assertTrue(np.all(AAt == np.matrix([[30, 70], [70, 174]])), 'incorrect value for
Test.assertTrue(np.allclose(AAtInv, np.matrix([[0.54375, -0.21875], [-0.21875, 0.0937
                'incorrect value for AAtInv')
```

```
1 test passed.
1 test passed.
```

Part 3: Additional NumPy and Spark linear algebra

(3a) Slices

You can select a subset of a one-dimensional NumPy `ndarray`'s elements by using slices. These slices operate the same way as slices for Python lists. For example, `[0, 1, 2, 3][:2]` returns the first two elements `[0, 1]`. NumPy, additionally, has more sophisticated slicing that allows slicing across multiple dimensions; however, you'll only need to use basic slices in future labs for this course.

Note that if no index is placed to the left of a `:`, it is equivalent to starting at 0, and hence `[0, 1, 2, 3][:2]` and `[0, 1, 2, 3][0:2]` yield the same result. Similarly, if no index is placed to the right of a `:`, it is equivalent to slicing to the end of the object. Also, you can use negative indices to index relative to the end of the object, so `[-2:]` would return the last two elements of the object.

For this exercise, return the last 3 elements of the array `features`.

In [23]:

```
# TODO: Replace <FILL IN> with appropriate code
features = np.array([1, 2, 3, 4])
print 'features:\n{0}'.format(features)

# The last three elements of features
lastThree = features[-3:]
print '\nlastThree:\n{0}'.format(lastThree)
```

```
features:
[1 2 3 4]

lastThree:
[2 3 4]
```

In [24]:

```
# TEST Slices (3a)
Test.assertTrue(np.all(lastThree == [2, 3, 4]), 'incorrect value for lastThree')
```

```
1 test passed.
```

(3b) Combining `ndarray` objects

NumPy provides many functions for creating new arrays from existing arrays. We'll explore two functions: np.hstack() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html), which

allows you to combine arrays column-wise, and np.vstack() (http://docs.scipy.org/doc/numpy/reference/generated/numpy.vstack.html), which allows you to combine arrays row-wise. Note that both `np.hstack()` and `np.vstack()` take in a tuple of arrays as their first argument. To horizontally combine three arrays `a`, `b`, and `c`, you would run `np.hstack((a, b, c))`.

If we had two arrays: `a = [1, 2, 3, 4]` and `b = [5, 6, 7, 8]`, we could use `np.vstack((a, b))` to produce the two-dimensional array:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

For this exercise, you'll combine the `zeros` and `ones` arrays both horizontally (column-wise) and vertically (row-wise).

Note that the result of stacking two arrays is an `ndarray`. If you need the result to be a matrix, you can call `np.matrix()` on the result, which will return a NumPy matrix.

In [26]:

```
# TODO: Replace <FILL IN> with appropriate code
zeros = np.zeros(8)
ones = np.ones(8)
print 'zeros:\n{0}'.format(zeros)
print '\nones:\n{0}'.format(ones)

zerosThenOnes = np.hstack((zeros, ones))   # A 1 by 16 array
zerosAboveOnes = np.vstack((zeros, ones))  # A 2 by 8 array

print '\nzerosThenOnes:\n{0}'.format(zerosThenOnes)
print '\nzerosAboveOnes:\n{0}'.format(zerosAboveOnes)
```

```
zeros:
[ 0.  0.  0.  0.  0.  0.  0.  0.]

ones:
[ 1.  1.  1.  1.  1.  1.  1.  1.]

zerosThenOnes:
[ 0.  0.  0.  0.  0.  0.  0.  0.  1.  1.  1.  1.  1.  1.  1.  1.]

zerosAboveOnes:
[[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.]]
```

In [27]:

```
# TEST Combining ndarray objects (3b)
Test.assertTrue(np.all(zerosThenOnes == [0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1]),
                'incorrect value for zerosThenOnes')
Test.assertTrue(np.all(zerosAboveOnes == [[0,0,0,0,0,0,0,0],[1,1,1,1,1,1,1,1]]),
                'incorrect value for zerosAboveOnes')
```

```
1 test passed.
1 test passed.
```

(3c) PySpark's DenseVector

PySpark provides a DenseVector (https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.li class within the module pyspark.mllib.linalg (https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#module-pyspark.mllib.linalg). `DenseVector` is used to store arrays of values for use in PySpark. `DenseVector` actually stores values in a NumPy array and delegates calculations to that object. You can create a new `DenseVector` using `DenseVector()` and passing in an NumPy array or a Python list.

`DenseVector` implements several functions. The only function needed for this course is `DenseVector.dot()`, which operates just like `np.ndarray.dot()`.

Note that `DenseVector` stores all values as `np.float64`, so even if you pass in an NumPy array of integers, the resulting `DenseVector` will contain floating-point numbers. Also, `DenseVector` objects exist locally and are not inherently distributed. `DenseVector` objects can be used in the distributed setting by either passing functions that contain them to resilient distributed dataset (RDD) transformations or by distributing them directly as RDDs. You'll learn more about RDDs in the spark tutorial.

For this exercise, create a `DenseVector` consisting of the values `[3.0, 4.0, 5.0]` and compute the dot product of this vector with `numpyVector`.

In [29]:

```
from pyspark.mllib.linalg import DenseVector
```

In [31]:

```
# TODO: Replace <FILL IN> with appropriate code
numpyVector = np.array([-3, -4, 5])
print '\nnumpyVector:\n{0}'.format(numpyVector)

# Create a DenseVector consisting of the values [3.0, 4.0, 5.0]
myDenseVector = DenseVector([3.0, 4.0, 5.0])
# Calculate the dot product between the two vectors.
denseDotProduct = DenseVector.dot(myDenseVector, numpyVector)

print 'myDenseVector:\n{0}'.format(myDenseVector)
print '\ndenseDotProduct:\n{0}'.format(denseDotProduct)
```

```
numpyVector:
[-3 -4  5]
myDenseVector:
[3.0,4.0,5.0]

denseDotProduct:
0.0
```

In [32]:

```
# TEST PySpark's DenseVector (3c)
Test.assertTrue(isinstance(myDenseVector, DenseVector), 'myDenseVector is not a Dense
Test.assertTrue(np.allclose(myDenseVector, np.array([3., 4., 5.])),
                'incorrect value for myDenseVector')
Test.assertTrue(np.allclose(denseDotProduct, 0.0), 'incorrect value for denseDotProdu
```

```
1 test passed.
1 test passed.
1 test passed.
```

Part 4: Python lambda expressions

(4a) Lambda is an anonymous function

We can use a lambda expression to create a function. To do this, you type `lambda` followed by the names of the function's parameters separated by commas, followed by a `:`, and then the expression statement that the function will evaluate. For example, `lambda x, y: x + y` is an anonymous function that computes the sum of its two inputs.

Lambda expressions return a function when evaluated. The function is not bound to any variable, which is why lambdas are associated with anonymous functions. However, it is possible to assign the function to a variable. Lambda expressions are particularly useful when you need to pass a simple function into another

function. In that case, the lambda expression generates a function that is bound to the parameter being passed into the function.

Below, we'll see an example of how we can bind the function returned by a lambda expression to a variable named `addSLambda`. From this example, we can see that `lambda` provides a shortcut for creating a simple function. Note that the behavior of the function created using `def` and the function created using `lambda` is equivalent. Both functions have the same type and return the same results. The only differences are the names and the way they were created.

For this exercise, first run the two cells below to compare a function created using `def` with a corresponding anonymous function. Next, write your own lambda expression that creates a function that multiplies its input (a single parameter) by 10.

Here are some additional references that explain lambdas: Lambda Functions (http://www.secnetix.de/olli/Python/lambda_functions.hawk), Lambda Tutorial (https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/), and Python Functions (http://www.bogotobogo.com/python/python_functions_lambda.php).

In [33]:

```
# Example function
def addS(x):
    return x + 's'
print type(addS)
print addS
print addS('cat')
```

```
<type 'function'>
<function addS at 0xb0e924fc>
cats
```

In [34]:

```
# As a lambda
addSLambda = lambda x: x + 's'
print type(addSLambda)
print addSLambda
print addSLambda('cat')
```

```
<type 'function'>
<function <lambda> at 0xb0e925dc>
cats
```

In [36]:

```
# TODO: Replace <FILL IN> with appropriate code
# Recall that: "lambda x, y: x + y" creates a function that adds together two numbers
multiplyByTen = lambda x: x * 10
print multiplyByTen(5)

# Note that the function still shows its name as <lambda>
print '\n', multiplyByTen
```

```
50

<function <lambda> at 0xb0e9279c>
```

In [37]:

```
# TEST Python lambda expressions (4a)
Test.assertEquals(multiplyByTen(10), 100, 'incorrect definition for multiplyByTen')
```

```
1 test passed.
```

(4b) `lambda` fewer steps than `def`

`lambda` generates a function and returns it, while `def` generates a function and assigns it to a name. The function returned by `lambda` also automatically returns the value of its expression statement, which reduces the amount of code that needs to be written.

For this exercise, recreate the `def` behavior using `lambda`. Note that since a lambda expression returns a function, it can be used anywhere an object is expected. For example, you can create a list of functions where each function in the list was generated by a lambda expression.

In [38]:

```
# Code using def that we will recreate with lambdas
def plus(x, y):
    return x + y

def minus(x, y):
    return x - y

functions = [plus, minus]
print functions[0](4, 5)
print functions[1](4, 5)
```

```
9
-1
```

In [39]:

```
# TODO: Replace <FILL IN> with appropriate code
# The first function should add two values, while the second function should subtract
# value from the first value.
lambdaFunctions = [lambda x, y: x + y ,  lambda x, y: x - y]
print lambdaFunctions[0](4, 5)
print lambdaFunctions[1](4, 5)
```

```
9
-1
```

In [40]:

```
# TEST lambda fewer steps than def (4b)
Test.assertEquals(lambdaFunctions[0](10, 10), 20, 'incorrect first lambdaFunction')
Test.assertEquals(lambdaFunctions[1](10, 10), 0, 'incorrect second lambdaFunction')
```

```
1 test passed.
1 test passed.
```

(4c) Lambda expression arguments

Lambda expressions can be used to generate functions that take in zero or more parameters. The syntax for `lambda` allows for multiple ways to define the same function. For example, we might want to create a function that takes in a single parameter, where the parameter is a tuple consisting of two values, and the function adds the two values together. The syntax could be either: `lambda x: x[0] + x[1]` or `lambda (x0, x1): x0 + x1`. If we called either function on the tuple `(3, 4)` it would return `7`. Note that the second `lambda` relies on the tuple `(3, 4)` being unpacked automatically, which means that `x0` is assigned the value `3` and `x1` is assigned the value `4`.

As an other example, consider the following parameter lambda expressions: `lambda x, y: (x[0] + y[0], x[1] + y[1])` and `lambda (x0, x1), (y0, y1): (x0 + y0, x1 + y1)`. The result of applying either of these functions to tuples `(1, 2)` and `(3, 4)` would be the tuple `(4, 6)`.

For this exercise: you'll create one-parameter functions `swap1` and `swap2` that swap the order of a tuple; a one-parameter function `swapOrder` that takes in a tuple with three values and changes the order to: second element, third element, first element; and finally, a three-parameter function `sumThree` that takes in three tuples, each with two values, and returns a tuple containing two values: the sum of the first element of each tuple and the sum of second element of each tuple.

In [41]:

```
# Examples.  Note that the spacing has been modified to distinguish parameters from t

# One-parameter function
a1 = lambda x: x[0] + x[1]
a2 = lambda (x0, x1): x0 + x1
print 'a1( (3,4) ) = {0}'.format( a1( (3,4) ) )
print 'a2( (3,4) ) = {0}'.format( a2( (3,4) ) )

# Two-parameter function
b1 = lambda x, y: (x[0] + y[0], x[1] + y[1])
b2 = lambda (x0, x1), (y0, y1): (x0 + y0, x1 + y1)
print '\nb1( (1,2), (3,4) ) = {0}'.format( b1( (1,2), (3,4) ) )
print 'b2( (1,2), (3,4) ) = {0}'.format( b2( (1,2), (3,4) ) )
```

```
a1( (3,4) ) = 7
a2( (3,4) ) = 7

b1( (1,2), (3,4) ) = (4, 6)
b2( (1,2), (3,4) ) = (4, 6)
```

In [44]:

```
# TODO: Replace <FILL IN> with appropriate code
# Use both syntaxes to create a function that takes in a tuple of two values and swap
# E.g. (1, 2) => (2, 1)
swap1 = lambda x: (x[1], x[0])
swap2 = lambda (x0, x1): (x1, x0)
print 'swap1((1, 2)) = {0}'.format(swap1((1, 2)))
print 'swap2((1, 2)) = {0}'.format(swap2((1, 2)))

# Using either syntax, create a function that takes in a tuple with three values and
# of (2nd value, 3rd value, 1st value).  E.g. (1, 2, 3) => (2, 3, 1)
swapOrder = lambda (x, y, z): (y, z, x)
print 'swapOrder((1, 2, 3)) = {0}'.format(swapOrder((1, 2, 3)))

# Using either syntax, create a function that takes in three tuples each with two val
# function should return a tuple with the values in the first position summed and the
# second position summed. E.g. (1, 2), (3, 4), (5, 6) => (1 + 3 + 5, 2 + 4 + 6) => (9
sumThree = lambda x, y, z: (x[0] + y[0] + z[0], x[1] + y[1] + z[1])
print 'sumThree((1, 2), (3, 4), (5, 6)) = {0}'.format(sumThree((1, 2), (3, 4), (5, 6)
```

```
swap1((1, 2)) = (2, 1)
swap2((1, 2)) = (2, 1)
swapOrder((1, 2, 3)) = (2, 3, 1)
sumThree((1, 2), (3, 4), (5, 6)) = (9, 12)
```

In [45]:

```
# TEST Lambda expression arguments (4c)
Test.assertEquals(swap1((1, 2)), (2, 1), 'incorrect definition for swap1')
Test.assertEquals(swap2((1, 2)), (2, 1), 'incorrect definition for swap2')
Test.assertEquals(swapOrder((1, 2, 3)), (2, 3, 1), 'incorrect definition fo swapOrder
Test.assertEquals(sumThree((1, 2), (3, 4), (5, 6)), (9, 12), 'incorrect definition fo
```

```
1 test passed.
1 test passed.
1 test passed.
1 test passed.
```

(4d) Restrictions on lambda expressions

Lambda expressions (https://docs.python.org/2/reference/expressions.html#lambda) consist of a single expression statement (https://docs.python.org/2/reference/simple_stmts.html#expression-statements) and cannot contain other simple statements (https://docs.python.org/2/reference/simple_stmts.html). In short, this means that the lambda expression needs to evaluate to a value and exist on a single logical line. If more complex logic is necessary, use `def` in place of `lambda`.

Expression statements evaluate to a value (sometimes that value is None). Lambda expressions automatically return the value of their expression statement. In fact, a `return` statement in a `lambda` would raise a `SyntaxError`.

The following Python keywords refer to simple statements that cannot be used in a lambda expression: `assert`, `pass`, `del`, `print`, `return`, `yield`, `raise`, `break`, `continue`, `import`, `global`, and `exec`. Also, note that assignment statements (=) and augmented assignment statements (e.g. +=) cannot be used either.

In [46]:

```
# Just run this code
# This code will fail with a syntax error, as we can't use print in a lambda expressi
import traceback
try:
    exec "lambda x: print x"
except:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "<ipython-input-46-989250748d81>", line 5, in <module>
    exec "lambda x: print x"
  File "<string>", line 1
    lambda x: print x
                  ^
SyntaxError: invalid syntax
```

(4e) Functional programming

The `lambda` examples we have shown so far have been somewhat contrived. This is because they were created to demonstrate the differences and similarities between `lambda` and `def`. An excellent use case for lambda expressions is functional programming. In functional programming, you will often pass functions to other functions as parameters, and `lambda` can be used to reduce the amount of code necessary and to make the code more readable.

Some commonly used functions in functional programming are map, filter, and reduce. Map transforms a series of elements by applying a function individually to each element in the series. It then returns the series of transformed elements. Filter also applies a function individually to each element in a series; however, with filter, this function evaluates to `True` or `False` and only elements that evaluate to `True` are retained. Finally, reduce operates on pairs of elements in a series. It applies a function that takes in two values and returns a single value. Using this function, reduce is able to, iteratively, "reduce" a series to a single value.

For this exercise, you'll create three simple `lambda` functions, one each for use in map, filter, and reduce. The map `lambda` will multiply its input by 5, the filter `lambda` will evaluate to `True` for even numbers, and the reduce `lambda` will add two numbers. Note that we have created a class called `FunctionalWrapper` so that the syntax for this exercise matches the syntax you'll see in PySpark.

Note that map requires a one parameter function that returns a new value, filter requires a one parameter function that returns `True` or `False`, and reduce

requires a two parameter function that combines the two parameters and returns a new value.

In [47]:

```
# Create a class to give our examples the same syntax as PySpark
class FunctionalWrapper(object):
    def __init__(self, data):
        self.data = data
    def map(self, function):
        """Call `map` on the items in `data` using the provided `function`"""
        return FunctionalWrapper(map(function, self.data))
    def reduce(self, function):
        """Call `reduce` on the items in `data` using the provided `function`"""
        return reduce(function, self.data)
    def filter(self, function):
        """Call `filter` on the items in `data` using the provided `function`"""
        return FunctionalWrapper(filter(function, self.data))
    def __eq__(self, other):
        return (isinstance(other, self.__class__)
            and self.__dict__ == other.__dict__)
    def __getattr__(self, name):  return getattr(self.data, name)
    def __getitem__(self, k):  return self.data.__getitem__(k)
    def __repr__(self):  return 'FunctionalWrapper({0})'.format(repr(self.data))
    def __str__(self):  return 'FunctionalWrapper({0})'.format(str(self.data))
```

In [48]:

```
# Map example

# Create some data
mapData = FunctionalWrapper(range(5))

# Define a function to be applied to each element
f = lambda x: x + 3

# Imperative programming: loop through and create a new object by applying f
mapResult = FunctionalWrapper([])  # Initialize the result
for element in mapData:
    mapResult.append(f(element))  # Apply f and save the new value
print 'Result from for loop: {0}'.format(mapResult)

# Functional programming: use map rather than a for loop
print 'Result from map call: {0}'.format(mapData.map(f))

# Note that the results are the same but that the map function abstracts away the imp
# and requires less code
```

```
Result from for loop: FunctionalWrapper([3, 4, 5, 6, 7])
Result from map call: FunctionalWrapper([3, 4, 5, 6, 7])
```

In [51]:

```
# TODO: Replace <FILL IN> with appropriate code
dataset = FunctionalWrapper(range(10))

# Multiply each element by 5
mapResult = dataset.map(lambda x: x * 5)
# Keep the even elements
# Note that "x % 2" evaluates to the remainder of x divided by 2
filterResult = dataset.filter(lambda x: x % 2 == 0)
# Sum the elements
reduceResult = dataset.reduce(lambda x, y: x + y)

print 'mapResult: {0}'.format(mapResult)
print '\nfilterResult: {0}'.format(filterResult)
print '\nreduceResult: {0}'.format(reduceResult)
```

```
mapResult: FunctionalWrapper([0, 5, 10, 15, 20, 25, 30, 35, 40, 45])

filterResult: FunctionalWrapper([0, 2, 4, 6, 8])

reduceResult: 45
```

In [52]:

```
# TEST Functional programming (4e)
Test.assertEquals(mapResult, FunctionalWrapper([0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
                  'incorrect value for mapResult')
Test.assertEquals(filterResult, FunctionalWrapper([0, 2, 4, 6, 8]),
                  'incorrect value for filterResult')
Test.assertEquals(reduceResult, 45, 'incorrect value for reduceResult')
```

```
1 test passed.
1 test passed.
1 test passed.
```

(4f) Composability

Since our methods for map and filter in the `FunctionalWrapper` class return `FunctionalWrapper` objects, we can compose (or chain) together our function calls. For example, `dataset.map(f1).filter(f2).reduce(f3)`, where `f1`, `f2`, and `f3` are functions or lambda expressions, first applies a map operation to `dataset`, then filters the result from map, and finally reduces the result from the first two operations.

Note that when we compose (chain) an operation, the output of one operation becomes the input for the next operation, and operations are applied from left to right. It's likely you've seen chaining used with Python strings. For example, `'Split this'.lower().split(' ')` first returns a new string object `'split this'` and then `split(' ')` is called on that string to produce `['split',`

'this'].

For this exercise, reuse your lambda expressions from (4e) but apply them to `dataset` in the sequence: map, filter, reduce. Note that since we are composing the operations our result will be different than in (4e). Also, we can write our operations on separate lines to improve readability.

In [53]:

```
# Example of a mult-line expression statement
# Note that placing parentheses around the expression allow it to exist on multiple l
# causing a syntax error.
(dataset
 .map(lambda x: x + 2)
 .reduce(lambda x, y: x * y))
```

Out[53]:

```
39916800
```

In [54]:

```
# TODO: Replace <FILL IN> with appropriate code
# Multiply the elements in dataset by five, keep just the even values, and sum those
finalSum = dataset.map(lambda x: x * 5).filter(lambda x: x % 2 == 0).reduce(lambda x,
print finalSum
```

```
100
```

In [55]:

```
# TEST Composability (4f)
Test.assertEquals(finalSum, 100, 'incorrect value for finalSum')
```

```
1 test passed.
```

Part 5: CTR data download

Lab four will explore website click-through data provided by Criteo. To obtain the data, you must first accept Criteo's data sharing agreement. Below is the agreement from Criteo. After you accept the agreement, you can obtain the download URL by right-clicking on the "Download Sample" button and clicking "Copy link address" or "Copy Link Location", depending on your browser. Paste the URL into the `# TODO` cell below. The file is 8.4 MB compressed. The script below will download the file to the virtual machine (VM) and then extract the data.

If running the cell below does not render a webpage, open the Criteo agreement (http://labs.criteo.com/downloads/2014-kaggle-display-advertising-challenge-

dataset/) in a separate browser tab. After you accept the agreement, you can obtain the download URL by right-clicking on the "Download Sample" button and clicking "Copy link address" or "Copy Link Location", depending on your browser. Paste the URL into the `# TODO` cell below.

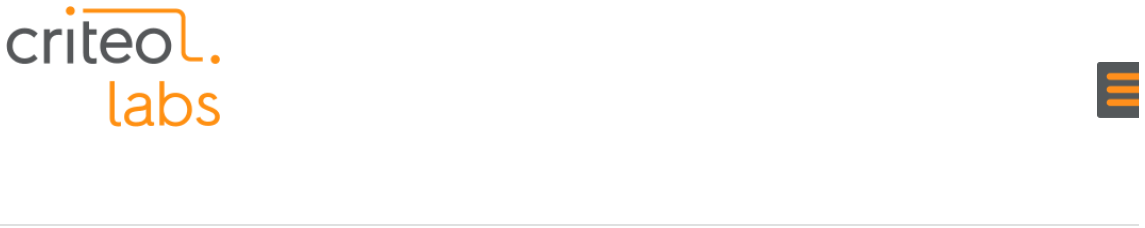
Note that the download could take a few minutes, depending upon your connection speed.

In [56]:

```
# Run this code to view Criteo's agreement
# Note that some ad blocker software will prevent this IFrame from loading.
# If this happens, open the webpage in a separate tab and follow the instructions fro
from IPython.lib.display import IFrame

IFrame("http://labs.criteo.com/downloads/2014-kaggle-display-advertising-challenge-da
       600, 350)
```

Out[56]:



In [57]:

```
# TODO: Replace <FILL IN> with appropriate code
# Just replace <FILL IN> with the url for dac_sample.tar.gz
import glob
import os.path
import tarfile
import urllib
import urlparse

# Paste url, url should end with: dac_sample.tar.gz
url = 'http://labs.criteo.com/wp-content/uploads/2015/04/dac_sample.tar.gz'

url = url.strip()
baseDir = os.path.join('data')
inputPath = os.path.join('cs190', 'dac_sample.txt')
fileName = os.path.join(baseDir, inputPath)
inputDir = os.path.split(fileName)[0]

def extractTar(check = False):
    # Find the zipped archive and extract the dataset
    tars = glob.glob('dac_sample*.tar.gz*')
    if check and len(tars) == 0:
      return False

    if len(tars) > 0:
        try:
            tarFile = tarfile.open(tars[0])
        except tarfile.ReadError:
            if not check:
                print 'Unable to open tar.gz file.  Check your URL.'
            return False

        tarFile.extract('dac_sample.txt', path=inputDir)
        print 'Successfully extracted: dac_sample.txt'
        return True
    else:
        print 'You need to retry the download with the correct url.'
        print ('Alternatively, you can upload the dac_sample.tar.gz file to your Jupy
               'directory')
        return False


if os.path.isfile(fileName):
    print 'File is already available. Nothing to do.'
elif extractTar(check = True):
    print 'tar.gz file was already available.'
elif not url.endswith('dac_sample.tar.gz'):
    print 'Check your download url.  Are you downloading the Sample dataset?'
else:
    # Download the file and store it in the same directory as this notebook
    try:
        urllib.urlretrieve(url, os.path.basename(urlparse.urlsplit(url).path))
    except IOError:
        print 'Unable to download and store: {0}'.format(url)

    extractTar()
```

```
Successfully extracted: dac_sample.txt
```

In [59]:

```
import os.path
baseDir = os.path.join('data')
inputPath = os.path.join('cs190', 'dac_sample.txt')
fileName = os.path.join(baseDir, inputPath)

if os.path.isfile(fileName):
    rawData = (sc
               .textFile(fileName, 2)
               .map(lambda x: x.replace('\t', ',')))  # work with either ',' or '\t'

print rawData.take(1)
rawDataCount = rawData.count()
print rawDataCount
# This line tests that the correct number of observations have been loaded
assert rawDataCount == 100000, 'incorrect count for rawData'
if rawDataCount == 100000:
    print 'Criteo data loaded successfully!'
```

```
[u'0,1,1,5,0,1382,4,15,2,181,1,2,,2,68fd1e64,80e26c9b,fb936136,7b4723c
4,25c83c98,7e0ccccf,de7995b8,1f89b562,a73ee510,a8cd5504,b2cb9c98,37c9c1
64,2824a5f6,1adce6ef,8ba8b39a,891b62e7,e5ba7672,f54016b9,21ddcdc9,b1252
a9d,07b5194c,,3a171ecb,c5c50484,e8b83407,9727dd16']
100000
Criteo data loaded successfully!
```