Python R training course - Session 3

Van-Duyet LE, Thinh PHAM

August 12, 2017

Lecture list

1. Introduction to Python
2. The Python Standard Library, if/loops statements
3. **Vector/matrix structures,** `numpy` **library**
   - Vector/matrix: structures in Python, Operators for vector/matrix (access, comparation, search)
   - Introduce to module Numpy
4. Python data types, File Processing, `Pandas` library
5. Functions in Python, Debugging.
6. Introduction to R, R for Python programmers.
7. Import data, plot data.
8. Data Mining in Python/R.

1 Numpy

Numpy (`http://www.numpy.org/`) is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy.

1.1 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

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

        a = np.array([1, 2, 3])   # Create a rank 1 array
        print(type(a))            # Prints "<class 'numpy.ndarray'>"
        print(a.shape)            # Prints "(3,)"
        print(a[0], a[1], a[2])   # Prints "1 2 3"
        a[0] = 5                  # Change an element of the array
        print(a)                  # Prints "[5, 2, 3]"

        b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
        print(b.shape)                     # Prints "(2, 3)"
        print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
<type 'numpy.ndarray'>
(3,)
(1, 2, 3)
[5 2 3]
(2, 3)
(1, 2, 4)
```

Numpy also provides many functions to create arrays:

```
In [2]: import numpy as np

        a = np.zeros((2,2))   # Create an array of all zeros
        print(a)              # Prints "[[ 0.  0.]
                              #          [ 0.  0.]]"

        b = np.ones((1,2))    # Create an array of all ones
        print(b)              # Prints "[[ 1.  1.]]"

        c = np.full((2,2), 7)  # Create a constant array
        print(c)               # Prints "[[ 7.  7.]
                               #          [ 7.  7.]]"

        d = np.eye(2)         # Create a 2x2 identity matrix
        print(d)              # Prints "[[ 1.  0.]
                              #          [ 0.  1.]]"

        e = np.random.random((2,2))  # Create an array filled with random values
        print(e)
```

```
[[ 0.  0.]
 [ 0.  0.]]
[[ 1.  1.]]
[[7 7]
 [7 7]]
[[ 1.  0.]
```

```
 [ 0.  1.]]
[[ 0.7479854   0.47388464]
 [ 0.0728678   0.0334643 ]]
```

You can read about other methods of array creation in the documentation: http://docs.scipy.org/doc/numpy/user/basics.creation.html#arrays-creation.

1.2 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
In [3]: import numpy as np

        # Create the following rank 2 array with shape (3, 4)
        # [[ 1  2  3  4]
        #  [ 5  6  7  8]
        #  [ 9 10 11 12]]
        a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

        # Use slicing to pull out the subarray consisting of the first 2 rows
        # and columns 1 and 2; b is the following array of shape (2, 2):
        # [[2 3]
        #  [6 7]]
        b = a[:2, 1:3]

        # A slice of an array is a view into the same data, so modifying it
        # will modify the original array.
        print(a[0, 1])   # Prints "2"
        b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
        print(a[0, 1])   # Prints "77"
```

```
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

```
In [4]: import numpy as np

        # Create the following rank 2 array with shape (3, 4)
        # [[ 1  2  3  4]
        #  [ 5  6  7  8]
        #  [ 9 10 11 12]]
        a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
        # Two ways of accessing the data in the middle row of the array.
        # Mixing integer indexing with slices yields an array of lower rank,
        # while using only slices yields an array of the same rank as the
        # original array:
        row_r1 = a[1, :]    # Rank 1 view of the second row of a
        row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
        print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
        print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

        # We can make the same distinction when accessing columns of an array:
        col_r1 = a[:, 1]
        col_r2 = a[:, 1:2]
        print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
        print(col_r2, col_r2.shape)  # Prints "[[ 2]
                                     #          [ 6]
                                     #          [10]] (3, 1)"
```

```
(array([5, 6, 7, 8]), (4,))
(array([[5, 6, 7, 8]]), (1, 4))
(array([ 2,  6, 10]), (3,))
(array([[ 2],
       [ 6],
       [10]]), (3, 1))
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
In [5]: import numpy as np

        a = np.array([[1,2], [3, 4], [5, 6]])

        # An example of integer array indexing.
        # The returned array will have shape (3,) and
        print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

        # The above example of integer array indexing is equivalent to this:
        print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

        # When using integer array indexing, you can reuse the same
        # element from the source array:
        print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

        # Equivalent to the previous integer array indexing example
        print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

```
[1 4 5]
[1 4 5]
```

```
[2 2]
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```
In [6]: import numpy as np

        # Create a new array from which we will select elements
        a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

        print(a)  # prints "array([[ 1,  2,  3],
                  #                [ 4,  5,  6],
                  #                [ 7,  8,  9],
                  #                [10, 11, 12]])"

        # Create an array of indices
        b = np.array([0, 2, 0, 1])

        # Select one element from each row of a using the indices in b
        print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

        # Mutate one element from each row of a using the indices in b
        a[np.arange(4), b] += 10

        print(a)  # prints "array([[11,  2,  3],
                  #                [ 4,  5, 16],
                  #                [17,  8,  9],
                  #                [10, 21, 12]])
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[ 1  6  7 11]
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

```
In [7]: import numpy as np

        a = np.array([[1,2], [3, 4], [5, 6]])
        bool_idx = (a > 2)    # Find the elements of a that are bigger than 2;
                              # this returns a numpy array of Booleans of the same
                              # shape as a, where each slot of bool_idx tells
                              # whether that element of a is > 2.

        print(bool_idx)       # Prints "[[False False]
                              #          [ True  True]
                              #          [ True  True]]"

        # We use boolean array indexing to construct a rank 1 array
        # consisting of the elements of a corresponding to the True values
        # of bool_idx
        print(a[bool_idx])  # Prints "[3 4 5 6]"

        # We can do all of the above in a single concise statement:
        print(a[a > 2])     # Prints "[3 4 5 6]"
```

```
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
[3 4 5 6]
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation (http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html).

1.3 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

```
In [8]: import numpy as np

        x = np.array([1, 2])   # Let numpy choose the datatype
        print(x.dtype)         # Prints "int64"

        x = np.array([1.0, 2.0])   # Let numpy choose the datatype
        print(x.dtype)             # Prints "float64"

        x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
        print(x.dtype)                         # Prints "int64"
```

```
int64
float64
```

```
int64
```

You can read all about numpy datatypes in the documentation (http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html).

1.4 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
In [9]: import numpy as np

        x = np.array([[1,2],[3,4]], dtype=np.float64)
        y = np.array([[5,6],[7,8]], dtype=np.float64)

        # Elementwise sum; both produce the array
        # [[ 6.0  8.0]
        #  [10.0 12.0]]
        print(x + y)
        print(np.add(x, y))

        # Elementwise difference; both produce the array
        # [[-4.0 -4.0]
        #  [-4.0 -4.0]]
        print(x - y)
        print(np.subtract(x, y))

        # Elementwise product; both produce the array
        # [[ 5.0 12.0]
        #  [21.0 32.0]]
        print(x * y)
        print(np.multiply(x, y))

        # Elementwise division; both produce the array
        # [[ 0.2         0.33333333]
        #  [ 0.42857143  0.5       ]]
        print(x / y)
        print(np.divide(x, y))

        # Elementwise square root; produces the array
        # [[ 1.          1.41421356]
        #  [ 1.73205081  2.        ]]
        print(np.sqrt(x))
```

```
[[  6.   8.]
 [ 10.  12.]]
[[  6.   8.]
 [ 10.  12.]]
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[  5.  12.]
 [ 21.  32.]]
[[  5.  12.]
 [ 21.  32.]]
[[ 0.2         0.33333333]
 [ 0.42857143  0.5       ]]
[[ 0.2         0.33333333]
 [ 0.42857143  0.5       ]]
[[ 1.          1.41421356]
 [ 1.73205081  2.        ]]
```

Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the **dot** function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. **dot** is available both as a function in the numpy module and as an instance method of array objects:

```
In [10]: import numpy as np

         x = np.array([[1,2],[3,4]])
         y = np.array([[5,6],[7,8]])

         v = np.array([9,10])
         w = np.array([11, 12])

         # Inner product of vectors; both produce 219
         print(v.dot(w))
         print(np.dot(v, w))

         # Matrix / vector product; both produce the rank 1 array [29 67]
         print(x.dot(v))
         print(np.dot(x, v))

         # Matrix / matrix product; both produce the rank 2 array
         # [[19 22]
         #  [43 50]]
         print(x.dot(y))
         print(np.dot(x, y))
```

```
219
219
[29 67]
[29 67]
[[19 22]
```

```
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is sum:

```
In [11]: x = np.array([[1,2],[3,4]])

         print(np.sum(x))  # Compute sum of all elements; prints "10"
         print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
         print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation (http://docs.scipy.org/doc/numpy/reference/routines.math.html).

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

```
In [12]: x = np.array([[1,2], [3,4]])
         print(x)    # Prints "[[1 2]
                     #          [3 4]]"
         print(x.T)  # Prints "[[1 3]
                     #          [2 4]]"
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
```

2 References

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

- Python Numpy Tutorial: http://cs231n.github.io/python-numpy-tutorial/#numpy
- Python for Data Analysis (Wes McKinney, O'Reilly Media, Inc., 2012).
- Guide to NumPy (Travis Oliphant, 2006).

3 Exercise

Using `numpy`:

3.1 Create a null vector of size 10 but the fifth value which is 1

Expect output:

```
[ 0 0 0 0 1 0 0 0 0 0]
```

3.2 Create a vector with values ranging from 10 to 49

Expect output:

```
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
```

3.3 Create a 3x3 matrix with values ranging from 0 to 8

Expect output:

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

3.4 Given a 1D array, negate all elements which are between 3 and 8, in place.

Expect output:

```
[ 0  1  2  3 -4 -5 -6 -7 -8  9 10]
```

3.5 Create a 5x5 matrix with row values ranging from 0 to 4

Expect output:

```
[[ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]
 [ 0.  1.  2.  3.  4.]]
```

3.6 Create a random vector of size 10 and sort it

3.7 Compute a matrix rank

Expect output:

```
array([[ 0.55295659,  0.24760051,  0.21054281],
       [ 0.80339823,  0.58569875,  0.43019202],
       [ 0.58991629,  0.86724474,  0.1964042 ]])

# Rank = 3
```

3.8 Find the max value in a numpy array

Expect output:

```
Z = np.array([ 0.55295659,  0.24760051,  0.21054281])

# Max: 0.55295659
```