Fairfield University DOLAN School of Business

Python for Analytics

The NumPy Library

The [NumPy Quickstart Tutorial]
[NumPy Reference Docs]

Learning Objectives

- **Theory:** You should be able to explain ...
    - The benefits of NumPy for data analysts
    - Multidimensional Arrays, Type Coercion, Element-wise Operations, Universal Functions, Structured Arrays, etc.
    - Data types as column specs in NumPy
- **Skills:** You should know how to ...
    - Create and manipulate multidimensional arrays
    - Perform various array operations (without for loops)
    - Use universal functions to calculate descriptive stats
    - Import and export structured arrays

Overview

NumPy as a List Replacement

Lists are Great! … except when they're not

Lists are very flexible containers for ordered collections:

- Allow mixed data types
- Built in indexing and slicing schemes
- Easy concatenation and copying

However, lists are also somewhat inefficient:

- Handling mixed data types requires extra processing
- Most operations require **for loops** to iterate over the list tems; slicing and concatenating are the exceptions

What’s wrong with `for` loops?

```
for i in numbers:
for i in range(len(numbers)):
```

- Prone to programmer error
  - What type is the loop variable `i` in each case above?
- Serialized execution (one cycle at a time) doesn’t give the interpreter much room to optimize
  - Modern computers can do many things at once!

Standard Arrays to the Rescue?!?

Standard ('vanilla') arrays solve some of the inefficiencies associated with mixed data types by requiring all data to be of the same type:

```
from array import array
array('i',1,2)                # list of ints
array('f',1.0,5.0,7.8)        # list of floats
array('u','a','↑','☺')        # list of characters
```

However, they still require us to use loops!

Introducing NumPy Arrays

NumPy arrays are like vanilla arrays on steroids, with indexing, slicing, copying, etc. **plus**

- Type coercion so **you don't get errors** when you mix strings with floats
- Elementwise versions of ***, /, +, -,** boolean comparisons, etc. that **eliminate most uses for loops**
- Methods for **descriptive statistics** and other common calculations
- Support for **linear algebra** operations (dot prod, cross prod, etc.)
- **Streamlined file Input / Output** for 2D tabular data

Implications for Analysts

- Working with tabular data in **vanilla Python can be tedious and error-prone**.
- NumPy simplifies things by **automating away** most of the tedium of loops, if statements, etc.
- It makes tabular data **feel more like a spreadsheet** in Excel, only without the copy/paste and drag fills.
- Except, of course, NumPy can handle **any sized** data set (if you have enough time).

NumPy Arrays

The data type at the center of NumPy

Importing NumPy

NumPy is a third-party library that you have to install separately. (That's why we don't consider it vanilla.)

It's probably a good idea to import numpy near the top of every script/notebook that needs it.

For the remaining slides, assume that we have already imported numpy as follows:

```
import numpy as np    # np always refers to numpy
```

Creating a NumPy Array

We can create a new NumPy array from any ordered collection (list, tuple, etc.).

```
np_array = np.array([1,5,2,9]) # Note: from a list
print(type(np_array))    → <class 'numpy.ndarray'>
print(np_array)          → [1 5 2 9]
print(np_array[1])       → 5
print(len(np_array))     → 4
```

Type Coercion

To prevent mixed types within an array, NumPy will coerce (convert) all elements to a ‘**lowest common denominator**’ type that can represent the data, where int → float → str

```
x=np.array([1,2.0]) # coerces everything to float
print(x) → [ 1.  2.]
y=np.array([1,2.0,'3']) # coerces everything to str
print(y) → ['1' '2.0' '3']
```

Array Attributes

The `array` type keeps *metadata* about your arrays:

```
np_array = np.array([[1,5,2,9],[2,1,9,5]])
np_array.ndim  # dimensions 1D, 2D, 3D, etc.
    → 2
np_array.shape # rows and columns for 2D
    → (2,4)
np_array.dtype # data type; ‘int64’ is int
    → dtype('int64')
```

Note: there are no parentheses because these are metadata **attributes (data)**, **not methods (functions)**.

Basic Element-wise Operations

Arithmetic operations work element-wise, iterating over the elements one at a time (without a `for` loop!)

```
x =   np.array([[1,5, 2,9], [2, 1,9,5]])
y =   np.array([[1,3, 4,2], [0, 5,8,3]])

x-y   →  array([[0,2,-2,7], [2,-4,1,2]])  # pairwise
2*x   →  array([[2,10,4,18],[4, 2,18,10]]) # scalar
x.dot(np.transpose(y)) # dot product; not pairwise
   → array([[42, 68],[51, 92]])
```

Boolean Comparisons

The built-in boolean comparators **==**, **!=**, **>**, **>=**, **<=**, etc. also apply element-wise:

```
x = np.array([2,1,9,5])
x>2   → array([False,False,True,True],dtype=bool)
x==2  → array([True,False,False,False],dtype=bool)

y = np.array([1,3,7,2])
x>y   → array([True,False,True,True],dtype=bool)
```

In-Place Operations

If we want to do arithmetic on an array without creating a copy, we can use `*=`, `/=` , `+=`, and `-=`.

```
np_array = np.array([1,5,2,9])
np_array *= 3      # multiply each element by 3
np_array
      → array([3,15,6,27])
```

Array Operations

Methods and Functions

Array Methods

NumPy arrays have callable methods for descriptive statistics and other common ***unary*** calculations.

```
x = np.array([1,5,2,9])
x.sum()      → 17
x.min()      → 1
x.max()      → 9
x.mean()     → 4.25
x.sort()     → array([1,2,5,9]) # modifies x!
```

Ref: https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html

NumPy Universal Functions

In addition to array methods, NumPy also supplies a bunch of useful array functions. There are too many to cover here, but RTFM (below) for more.

Excel heads: notice anything in the function list below?

See also:
all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, *inv*, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where

Ref: https://docs.scipy.org/doc/numpy-1.13.0/reference/ufuncs.html#available-ufuncs

Indexing Tricks

Selecting just the items you need without a for loop

Indexing & Slicing

- All the usual indexing & slicing rules apply to 1D arrays:

```
x=np.array([1,2,3])
print(x[1:]) → [2 3]
```

- Comma-delimited slices for 2D and higher arrays:

```
x=np.array([[1,2,3],[4,5,6]])
print(x[1][1:])        → [5 6]  # vanilla Python
print(x[1,1:])         → [5 6]  # with commas
print(x[:1,1:])        → [2 3]
```

Indexed Selections

We can use an array of indexes to select elements from another array.

```
x = np.array([1,5,2,9])    # array of data
i = np.array([1,3])        # array of indexes
x[i]     → array([5,9])    # x reduced to i
```

Boolean Selections

Booleans can also be used as selectors.

```
x = np.array([2,1,9,5])
y=x>2     # y is an array of booleans
y         → array([False,False,True,True],dtype=bool)

x[y]      → array([9,5])
x[x>2]    → array([9,5]) # all in one statement
```

Note about Iterating in 2D, 3D, etc.

Take care when using `for` loops with NumPy arrays.
They always iterate over the **first axis** (dimension)!

```
x=np.array([[1,2,3],[4,5,6]])
for i in x:
   print(i) # i is an array, not a number

   → [1 2 3]
     [4 5 6]
```

Structured Arrays

When columns have names and types

Tables are more than just data ...

Structured arrays let us specify metadata like column names and data types, just like database tables.

ID	Last Name	First Name
1	Paca	Al
2	Loblaw	Bob

```
# a 2D (rows and columns) array of people data
people = np.array([(1,‘Paca’,‘Al’),(2,‘Loblaw’,‘Bob’)],
    dtype=[(‘id’,int),(‘lname’, ‘S25’),(‘fname’,‘S25’)])
```

The `dtype` Specification

Metadata for **each column** is encoded in the **`dtype`** spec, which is a list of a tuples:

(<col name>,<type spec>)

Common Type Specs

- `int`
- `float`
- `‘S#’` (string of up to # characters)

```
people = np.array([(1,‘Paca’,‘Al’),(2,‘Loblaw’,‘Bob’)],
    dtype=[(‘id’,int),(‘lname’, ‘S25’),(‘fname’,‘S25’)])
```

Record Arrays

Record arrays let us refer to columns as ***attributes*** with dot notation. All we have to do is use the **`np.rec.array`** type instead of `np.array`.

```
people =
    np.rec.array([(1,'Paca','Al'),(2,'Loblaw','Bob')],
    dtype=[('id',int),('lname', 'S25'),('fname','S25')])
print(people[1].lname) # refers to second column by name
    → ‘Loblaw’
```

File I/O

Fast and easy import and export of tabular data

Numpy Data Sources

NumPy can read and write data to:

- Strings (and streams)
- **CSV files**
- Formatted text files
- Binary files (raw, ‘pickled,’ or arrays)
- Zip files
- Web URLs (with ftp, sftp, http, https)

Reference: https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.io.html

The `genfromtxt()` function

The workhorse of NumPy I/O, **it makes reading from CSV files almost automatic**. No more opening, reading, splitting, stripping, closing…

```
my_table =
    np.genfromtxt(“my_file.csv”,delimiter=“,”)
print(type(my_table)) → <class 'numpy.ndarray'>
```

Ref: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.genfromtxt.html

`genfromtxt()` options

`genfromtxt()` has lots of ***optional*** arguments:

- `dtype` specifies the data type(s) of the columns
- `delimiter` specifies the column separator
- `autostrip` removes white space characters
- `skipheader` skips the indicated number of lines
- `usecols` indicates which columns to import
- names provides column names (not needed if the full `dtype` is given)

Many others are in the docs.

Examples

```
# people.csv is a csv file with cols ID,LName,FName

# This works if all data is numerical and no headers
people = np.genfromtxt(‘people.csv’,delimiter=‘,’)

# Skips the first line and uses mixed data types
people = np.genfromtxt('people.csv',skip_header=1,
    dtype=(int,'S25','S25'),names="id,lname,fname",
    delimiter=',')
```

Examples Continued

```
# Reads column names from the first line of the file
people =
    np.genfromtxt('people.csv',names=True,
        dtype=(int,'S25','S25'),delimiter=',')

# Shorthand function for CSV; returns a rec.array
people =
    np.recfromcsv('people.csv',names=True,
        dtype=(int,'S25','S25'))
```

Output with `savetxt()`

The `savetxt()` function does the reverse of `genfromtxt()`.

```
# people is a structured array
# Save to the CSV file “out.csv”
np.savetxt(“out.csv”, people, delimiter= ‘,’)

# Save as a gzip file, detected from the filename
np.savetxt(“out.csv.gz”, people, delimiter= ‘,’)
```

Fairfield University DOLAN School of Business

Python for Analytics

NumPy Basics

The [NumPy Quickstart Tutorial]

[NumPy Reference Docs]