L02_NumPy

December 15, 2015

```
In [ ]: # suppress deprecated warnings
        import warnings
        warnings.filterwarnings('ignore')
```

1 NumPy

Kevin Stratford kevin@epcc.ed.ac.uk Emmanouil Farsarakis farsarakis@epcc.ed.ac.uk
Other course authors: Neelofer Banglawala Andy Turner Arno Proeme

www.archer.ac.uk support@archer.ac.uk

[NumPy] Introducing NumPy

- Core Python provides lists
- Lists are slow for many numerical algorithms
- NumPy provides fast precompiled functions for numerical routines:
- multidimensional arrays : faster than lists
- matrices and linear algebra operations
- random number generation
- Fourier transforms and much more...
- https://www.numpy.org/

[NumPy] Calculating π

If we know the area A of square length R, and the area Q of the quarter circle with radius R, we can calculate π : \$ Q/A = $\pi R2/4R2\$, so$

\$ $\pi = 4, Q/A\$$

[NumPy] Calculating π : monte carlo method

We can use the monte carlo method to determine areas A and Q and approximate π. For N iterations

1. randomly generate the coordinates (x, y), where $0 \leq x, y < R$
2. Calculate distance \$ r = x^2 + y^2 $.Check if ($x,,y)\$ lies within radius of circle
3. Check if \$ r \$ lies within radius R of circle i.e. if $r \leq R$
4. if yes, add to count for approximating area of circle

The numerical approximation of π is then : 4 * (count/N)

[NumPy] Calculating π : a solution

```
In [ ]: # calculate pi
        import numpy as np

        # N : number of iterations
        def calc_pi(N):
            x = np.random.ranf(N);
            y = np.random.ranf(N);
            r = np.sqrt(x*x + y*y);
            c=r[ r <= 1.0 ]
            return 4*float((c.size))/float(N)

        # time the results
        pts = 6; N = np.logspace(1,8,num=pts);
        result = np.zeros(pts); count = 0;
        for n in N:
            result = %timeit -o -n1 calc_pi(n)
            result[count] = result.best
            count += 1

        # and save results to file
        np.savetxt('calcpi_timings.txt', np.c_[N,results],
                   fmt='%1.4e %1.6e');
```

[NumPy] Creating arrays I

```
In [ ]: # import numpy as alias np
        import numpy as np
```

```
In [ ]: # create a 1d array with a list
        a = np.array( [-1,0,1] ); a
```

[NumPy] Creating arrays II

```
In [ ]: # use arrays to create arrays
        b = np.array( a ); b
```

```
In [ ]: # use numpy functions to create arrays
        # arange for arrays, range for lists!
        a = np.arange( -2, 6, 2 ); a
```

[NumPy] Creating arrays III

```
In [ ]: # between start, stop, sample step points
        a = np.linspace(-10, 10, 5);
        a;
```

```
In [ ]: # Ex: can you guess these functions do?
        b = np.zeros(3); print b
        c = np.ones(3); print c
```

```
In [ ]: # Ex++: what does this do? Check documentation!
        h = np.hstack( (a, a, a), 0 ); print h
```

[NumPy] Array characteristics

```
In [ ]: # array characteristics such as:
        print a
        print a.ndim  # dimensions
        print a.shape # shape
        print a.size  # size
        print a.dtype # data type
```

```
In [ ]: # can choose data type
        a = np.array( [1,2,3], np.int16 ); a.dtype
```

[NumPy] Multi-dimensional arrays I

```
In [ ]: # multi-dimensional arrays e.g. 2d array or matrix
        # e.g. list of lists
        mat = np.array( [[1,2,3], [4,5,6]]);
        print mat; print mat.size; mat.shape
```

```
In [ ]: # join arrays along first axis (0)
        d = np.r_[np.array([1,2,3]), 0, 0, [4,5,6]];
        print d; d.shape
```

[NumPy] Multi-dimensional arrays II

```
In [ ]: # join arrays along second axis (1)
        d = np.c_[np.array([1,2,3]), [4,5,6]];
        print d; d.shape
```

```
In [ ]: # Ex: use r_, c_ with nd (n>1) arrays
```

```
In [ ]: # Ex: can you guess the shape of these arrays?
        h = np.array( [1,2,3,4,5,6] );
        i = np.array( [[1,1],[2,2],[3,3],[4,4],[5,5],[6,6]] );
        j = np.array( [[[1],[2],[3],[4],[5],[6]]] );
        k = np.array( [[[[1],[2],[3],[4],[5],[6]]]] );
```

[NumPy] Reshaping arrays I

```
In [ ]: # reshape 1d arrays into nd arrays original matrix unaffected
        mat = np.arange(6); print mat
        print mat.reshape( (3, 2) )
        print mat; print mat.size;
        print mat.shape
```

```
In [ ]: # can also use the shape, this modifies the original array
        a = np.zeros(10); print a
        a.shape = (2,5)
        print a; print a.shape;
```

[NumPy] Reshaping arrays II

```
In [ ]: # Ex: what do flatten() and ravel()?
        # use online documentation, or '?'
        mat2 = mat.flatten()
        mat2 = mat.ravel()
```

```
In [ ]: # Ex: split a martix? Change the cuts and axis values
        # need help?: np.split?
        cuts=2;
        np.split(mat, cuts, axis=0)
```

[NumPy] Functions for you to explore

```
In [ ]: # Ex: can you guess what these functions do?
        # np.copyto(b, a);
        # v = np.vstack( (arr2d, arr2d) ); print v; v.ndim;
        # c0 = np.concatenate( (arr2d, arr2d), axis=0); c0;
        # c1 = np.concatenate(( mat, mat ), axis=1); print "c1:", c1;
```

```
In [ ]: # Ex++: other functions to explore
        #
        # stack(arrays[, axis])
        # tile(A, reps)
        # repeat(a, repeats[, axis])
        # unique(ar[, return_index, return_inverse, ...])
        # trim_zeros(filt[, trim]), fill(scalar)
        # xv, yv = meshgrid(x,y)
```

[NumPy] Accessing arrays I

```
In [ ]: # basic indexing and slicing we know from lists
        a = np.arange(8); print a
        a[3]
```

```
In [ ]: # a[start:stop:step] --> [start, stop every step)
        print a[0:7:2]
        print a[0::2]
```

```
In [ ]: # negative indices are valid!
        # last element index is -1
        print a[2:-3:2]
```

[NumPy] Accessing arrays II

```
In [ ]: # basic indexing of a 2d array : take care of each dimension
        nd = np.arange(12).reshape((4,3)); print nd;
        print nd[2,2];
        print nd[2][2];
```

```
In [ ]: # get corner elements 0,2,9,11
        print nd[0:4:3, 0:3:2]
```

```
In [ ]: # Ex: get elements 7,8,10,11 that make up the bottom right corner
        nd = np.arange(12).reshape((4,3));
        print nd; nd[2:4, 1:3]
```

[NumPy] Slices and copies I

```
In [ ]: # slices are views (like references)
        # on an array, can change elements
        nd[2:4, 1:3] = -1; nd
```

```
In [ ]: # assign slice to a variable to prevent this
        s = nd[2:4, 1:3]; print nd;
        s = -1; nd
```

[NumPy] Slices and copies II

```
In [ ]: # Care - simple assignment between arrays
        # creates references!
        nd = np.arange(12).reshape((4,3))
        md = nd
        md[3] = 1000
        print nd
```

```
In [ ]: # avoid this by creating distinct copies
        # using copy()
        nd = np.arange(12).reshape((4,3))
        md = nd.copy()
        md[3] = 999
        print nd
```

[NumPy] Fancy indexing I

```
In [ ]: # advanced or fancy indexing lets you do more
        p = np.array( [[0,1,2], [3,4,5], [6,7,8], [9,10,11]] );
        print p
```

```
In [ ]: rows = [0,0,3,3]; cols = [0,2,0,2];
        print p[rows, cols]
```

```
In [ ]: # Ex: what will this slice look like?
        m = np.array( [[0,-1,4,20,99], [-3,-5,6,7,-10]] );
        print m[[0,1,1,1], [1,0,1,4]];
```

[NumPy] Fancy indexing II

```
In [ ]: # can use conditionals in indexing
        # m = np.array([[0,-1,4,20,99],[-3,-5,6,7,-10]]);
        m[ m < 0 ]
```

```
In [ ]: # Ex: can you guess what this does? query: np.sum?
        y = np.array([[0, 1], [1, 1], [2, 2]]);
        rowsum = y.sum(1);
        y[rowsum <= 2, :]
```

```
In [ ]: # Ex: and this?
        a = np.arange(10);
        mask = np.ones(len(a), dtype = bool);
        mask[[0,2,4]] = False; print mask
        result = a[mask]; result
```

```
In [ ]: # Ex: r=np.array([[0,1,2],[3,4,5]]);
        xp = np.array( [[[1,11],[2,22],[3,33]], [[4,44],[5,55],[6,66]]] );
        xp[slice(1), slice(1,3,None), slice(1)]; xp[:1, 1:3:, :1];
        print xp[[1,1,1],[1,2,1],[0,1,0]]
```

[NumPy] Manipulating arrays

```
In [ ]: # add an element with insert
        a = np.arange(6).reshape([2,3]); print a
        np.append(a, np.ones([2,3]), axis=0)
```

```
In [ ]: # inserting an array of elements
        np.insert(a, 1, -10, axis=0)
```

```
In [ ]: # can use delete, or a boolean mask, to delete array elements
        a = np.arange(10)
        np.delete(a, [0,2,4], axis=0)
```

[NumPy] Vectorization I

```
In [ ]: # vectorization allows element-wise operations (no for loop!)
        a = np.arange(10).reshape([2,5]); b = np.arange(10).reshape([2,5]);
```

```
In [ ]: -0.1*a
```

```
In [ ]: a*b
```

```
In [ ]: a/(b+1)  #.astype(float)
```

[NumPy] Random number generation

```
In [ ]: # random floats
        a = np.random.ranf(10); a
```

```
In [ ]: # create random 2d int array
        a = np.random.randint(0, high=5, size=25).reshape(5,5);
        print a;
```

In [ ]:
```
# generate sample from normal distribution
# (mean=0, standard deviation=1)
s = np.random.standard_normal((5,5)); s;
```

In [ ]:
```
# Ex: what other ways are there to generate random numbers?
# What other distributions can you sample?
```

[NumPy] File IO

In [ ]:
```
# easy way to save data to text file
pts = 5; x = np.arange(pts); y = np.random.random(pts);
```

In [ ]:
```
# format specifiers: d = int, f = float, e = scientific
np.savetxt('savedata.txt', np.c_[x,y], header = 'DATA', footer = 'END',
           fmt = '%d %1.4f')
```

In [ ]:
```
!cat savedata.txt
# One could do ...
# p = np.loadtxt('savedata.txt')
```

In [ ]:
```
# ...but much more flexibility with genfromtext
p = np.genfromtxt('savedata.txt', skip_header=2, skip_footer=1); p
```

In [ ]:
```
# Ex++: what do numpy.save, numpy.load do ?
```

[NumPy] Polynomials I
Can represent polynomials with the numpy class Polynomial from numpy.polynomial.polynomial.
Polynomial([a, b, c, d, e]) is equivalent to $p(x) = a\,+\,b\,x\,+\,c\,x^2\,+\,d\,x^3\,+\,e\,x^4$. For example:

- Polynomial([1,2,3]) is equivalent to $p(x) = 1\,+\,2\,x\,+\,3\,x^2$
- Polynomial([0,1,0,2,0,3]) is equivalent to $p(x) = x\,+\,2\,x^3\,+\,3\,x^5$

[NumPy] Polynomials II
Can carry out arithmetic operations on polynomials, as well integrate and differentiate them.
Can also use the polynomial package to find a least-squares fit to data.
[NumPy] Polynomials : calculating π I
The Taylor series expansion for the trigonometric function $\arctan(y)$ is :
$\arctan(y)\;=\;y - \frac{y^3}{3} + \frac{y^5}{5} - \frac{y^7}{7} + \dots$
Now, $\arctan(1) = \frac{\pi}{4}$, so ...
\$ $\pi = 4, \big(- \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \big)$\$
We can represent the series expansion using a numpy Polynomial, with coefficients: $p(x) =$ [0, 1, 0, -1/3, 0, 1/5, 0, -1/7,...], and use it to approximate π.
[NumPy] Polynomials : calculating π II

In [ ]:
```
# calculate pi using polynomials
# import Polynomial class
from numpy.polynomial import Polynomial as poly;
num = 100000;
denominator = np.arange(num);

denominator[3::4] *= -1 # every other odd coefficient is -ve
numerator = np.ones(denominator.size);

# avoid dividing by zero, drop first element denominator
```

```
almost = numerator[1:]/denominator[1:];

# make even coefficients zero
almost[1::2] = 0

# add back zero coefficient
coeffs = np.r_[0,almost];

p = poly(coeffs);
4*p(1) # pi approximation
```

[NumPy] Performance I

Python has a convenient timing function called timeit.

- Can use this to measure the execution time of small code snippets.
- To use timeit function
- import module timeit and use timeit.timeit or
- use magic command %timeit in an IPython shell

[NumPy] Performance II

By default, timeit:

- Takes the best time out of 3 repeat tests (-r)
- takes the average time for a number of iterations (-n) per repeat

In an IPython shell:

- %timeit -n<iterations> -r<repeats> <code>
- query %timeit? for more information
- https://docs.python.org/2/library/timeit.html

[NumPy] Performance : experiments I

Here are some timeit experiments for you to run.

```
In [ ]: # accessing a 2d array
        nd = np.arange(100).reshape((10,10))

        # accessing element of 2d array
        %timeit -n10000000 -r3 nd[5][5]
        %timeit -n10000000 -r3 nd[(5,5)]
```

```
In [ ]: # Ex: multiplying two vectors
        x=np.arange(10E7)
        %timeit -n1 -r10 x*x
        %timeit -n1 -r10 x**2

        # Ex++: from the linear algebra package
        %timeit -n1 -r10 np.dot(x,x)
```

[NumPy] Performance : experiments II

In [ ]:
```
import numpy as np
# Ex: range functions and iterating in  for loops
size = int(1E6);

%timeit for x in range(size): x ** 2

# faster than range for very large arrays?
%timeit for x in xrange(size): x ** 2

%timeit for x in np.arange(size): x ** 2

%timeit np.arange(size) ** 2
```

In [3]:
```
# Ex: look at the calculating pi code
# Make sure you understand it. Time the code.
```

[NumPy] Summary

- NumPy introduces multi-dimensional arrays to Python, which is crucial for efficient scientific computing
- It also provides fast numerical routines for scientific computation
- Next up: Matplotlib