Lab: Numerics with Numpy

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

Day 1
	numpy is one of the most popular numeric libraries in the world. Using the numpy.array type, we can do element-wise array operations without using a loop:

\[
x = [5,1,2,3,4]
\]

# simple method of adding 7 to every element in x:
x_plus_seven = []
for val in x:
    x_plus_seven.append(val+7)
print x_plus_seven

# list comprehension version:
x_plus_seven = [ val+7 for val in x ]
print x_plus_seven

# numpy version
import numpy
x_plus_seven = numpy.array(x) + 7
print x_plus_seven

We can use multiple arrays when we map operations over the array:

# add element wise:
# (prints [10, 10, 7])
print numpy.array([1,3,4]) + numpy.array([9,7,3])

# multiply element wise:
# (prints [9, 21, 12])
print numpy.array([1,3,4]) * numpy.array([9,7,3])

# divide element wise (integer division):
# (prints [3,2,1])
print numpy.array([9,4,3]) / numpy.array([3,2,2])
# divide element wise (float division):
# (prints [3,2,1.5])
print numpy.array([9,4,3], float) / numpy.array([3,2,2])

We can cast an array \( x \) of \texttt{int} types to an array of \texttt{float} types by using \texttt{numpy.array(x, float)}.

We can generate evenly spaced values using \texttt{arange}, \texttt{numpy}'s equivalent of \texttt{range}:

```python
# generate x from 1.5 to <2.0 in steps of 0.1:
x = numpy.arange(0.0, 10.0, 0.1)
print x
```

We can do something similar using \texttt{linspace}, which accepts the length of the result instead of the step size:

```python
# generate x from 1.0 to <=5.0 using 95 evenly spaced values:
x = numpy.linspace(1.0, 5.0, 95)
print x
```

We can call some functions on \texttt{array} types and they will broadcast over the array element wise. These values can be plotted using \texttt{pylab}:

```python
import pylab

# generate x from 1.0 to <=5.0 using 5 evenly spaced values:
x = numpy.linspace(-10, 10, 1000)
y = -13*x**3 + 7*x**2 - 5*x + 13
```

Element wise operations make it easy to create and plot polynomials:
pylab.plot(x, y, label=r'$-13x^3 + 7x^2 - 5x + 13$')
pylab.legend()
pylab.show()

We can likewise and operate on multidimensional arrays as follows:

```python
x = numpy.array([[1,2],[3,4],[5,6]])
y = numpy.array([[1,4],[9,16],[25,36]])
print x+y
```

Alternatively, we can easily construct multidimensional arrays using the `reshape` function:

```python
shape = (3,2)
flat_size = numpy.product(shape)
# [1,2,...6]:
x = numpy.arange(1,7).reshape( shape )
y = x**2
print x+y
```

To create a multidimensional array of a given shape filled with zeros, you can use the `zeros` and `ones` functions:

```python
print numpy.zeros( (3,2) )
print numpy.ones( (3,2) )
```

One of the most powerful tools available to `numpy` is to broadcast indexing operations over an of indices:

```python
full_array = numpy.array([10,6,4,11,2,31,4,0])
indices = numpy.array([3,1,4,2])
print full_array[indices]
```

Likewise, we can do the same with boolean values. The following will only index the array where the index array is `True`:

```python
x = numpy.array([1,10,3,12,4,9])
print x
print x>8
print x[x>8]

This can be used to perform element wise operations only on certain parts of the array and do so without using any if-statements. For instance, we can make a scatter plot that only plots the points where a function is nonnegative:

```python
# generate 100 evenly spaced points from 0 to <=4pi:
x = numpy.linspace(0, 4*numpy.pi, 100)
y = numpy.sin(x)
# plot x and y:
pylab.plot(x, y)

y_is_nonneg = y>0.0
# scatter plot x and y only where y is nonnegative:
pylab.scatter(x[y_is_nonneg], y[y_is_nonneg])
pylab.show()
```

We can cast these booleans to integers to form an array of 0s and 1s:

```python
x = numpy.array([1,10,3,12,4,9])
print x
print x>8
print int(x>8)
```

We can generate a single random value using functions from `numpy.random`:

```python
for i in range(10):
    print numpy.random.uniform(0.0,1.0)
```

But even better, we can generate many random values at once and use them to populate an array: We can generate a single random value using functions from `numpy.random`:

```python
n=32

# generate n random uniform values:
x = numpy.random.uniform(0.0,1.0,n)
print x
```

`numpy` can encode polynomials using an array of their coefficients. For example, the following encodes $2x^4 - x^3 + 5x^2 - 8x + 2$: 

```python
print x[x>8]
```
coeffs = numpy.array([2, -1, 5, -8, 2])

# evaluate the polynomial at x=3:
print numpy.polyval(coeffs, 3)

# verify that this matches the above:
print 2*3**4 - 3**3 + 5*3**2 - 8*3 + 2

We can use this to plot \(2x^4 - x^3 + 5x^2 - 8x + 2\) with \(x\) from -4 to 4:

```python
x = numpy.linspace(-4, 4, 1024)
y = numpy.polyval(coeffs, x)
pylab.plot(x, y)
pylab.show()
```

We can use `numpy` to find the best-fit polynomial for given data:

```python
x_observed = numpy.linspace(-4, 0, 10)
y_observed = 3*x**2 - 2
pylab.scatter(x_observed, y_observed)
coeffs = numpy.polyfit(x_observed, y_observed, 2)
print coeffs
x_full = numpy.linspace(-4, 4, 100)
y_interpolate = numpy.polyval(coeffs, x_full)
pylab.plot(x_full, y_interpolate)
pylab.show()
```

1. Generate an array \(x\) with 1024 elements evenly spaced with \(x\) from -4 to 10 (inclusive).
2. Generate an array \(y = x^3 + x^2 + x + 1\).
3. Add uniform noise (from -1 to 1) to \(y\).
4. Fit a cubic polynomial to the \(x\) and \(y\) data.
5. Scatter plot the \(x\) and \(y\) data and superimpose a (line) plot of the cubic polynomial fit. How accurate is the fit to the original polynomial \(x^3 + x^2 + x + 1\)?
Day 2

1. Generate an array of \( n = 10^6 \) uniform values between 0.0 and 1.0 (named \( x \)). Generate another array of \( n \) uniform values between 0.0 and 1.0 (named \( y \)).

2. Generate an array of booleans showing where \( x^2 + y^2 \leq 1 \). Call this array \( \text{in\_quarter\_circle} \).

3. Cast the boolean array, \( \text{in\_quarter\_circle} \), as an array of integers.

4. Use the \texttt{numpy.sum} function to count the number of instances where \( \text{in\_quarter\_circle} \) is True.

5. Use the resulting sum to compute a Monte Carlo estimate of \( \pi \). Name two ways explaining how this code is improved over your original Monte Carlo estimate of \( \pi \).