Lab 3

Python: NumPy and SciPy

Lab Objective: *Learn some other features available in NumPy and SciPy.*

Logic Operations

Logic operations return arrays of only true or false values depending on whether they satisfy a given condition. These arrays are often useful for masking values in other arrays.

```
>>> a = np.random.rand(5,5)
>>> a<.5
array([[ True, False, False,  True, False],
       [ True,  True, False,  True, False],
       [False,  True,  True, False, False],
       [False, False,  True, False,  True],
       [ True, False, False,  True, False]], dtype=bool)
>>> a[a<.5]
array([ 0.46121936,  0.11294639,  0.37868745,  0.23435659,  ↩
    0.25898226,
        0.09095808,  0.19124312,  0.41124911,  0.09823221,  ↩
            0.03739077,
        0.08655778])
>>> a[a<.3] = 0
>>> a
array([[ 0.46121936,  0.83080909,  0.5632045 ,  0.        ,  ↩
    0.59581868],
       [ 0.37868745,  0.        ,  0.54977124,  0.        ,  ↩
           0.753893  ],
       [ 0.79744663,  0.        ,  0.        ,  0.57981239,  ↩
           0.95839037],
       [ 0.66512744,  0.63471169,  0.41124911,  0.6466058 ,  0. ↩
                  ],
       [ 0.        ,  0.50692736,  0.54082953,  0.        ,  ↩
           0.5173614 ]])
```

The comparison operators can all be used like this with arrays. We can quickly test if all elements of a given axis evaluate to true with `np.all`. Likewise, we can test if any element evaluates to true with `np.any`. Because of the nature of floating point

numbers, it is next to impossible to accurately test the equality of elements of two arrays. NumPy provides a special function, `np.allclose`, to check if two arrays are *almost* the same (or within some specified tolerances). *Please note that in some rare cases `np.allclose(a, b)` will not match `np.allclose(b, a)`.* This is because the equation the function uses for checking closeness is not symmetric ($|a - b| \leq \text{atol} + \text{rtol} * |b|$).

```
>>> a = np.ones((5, 5))
>>> np.allclose(a, a+1e-5)
True
>>> np.allclose(a, a+1e-4)
False
```

NumPy also allows bitwise operations on arrays using the standard Python bitwise operators: `&`, `|`, and `^`. They are also available as NumPy functions: `np.bitwise_and`, `np.bitwise_or`, and `np.bitwise_xor` respectively. NumPy makes available logical, or boolean, operators as well: `np.logical_and`, `np.logical_or`, and `np.logical_xor`. These are analogous to Pythons logical operators: `and`, `or`. The difference between bitwise and logical operators is simple. Bitwise operators compare operands using their bit representations. Logical operators compare operands using their boolean values.

Broadcasting Array Dimensions

Most array operations require array sizes to be compatible. For example to add two arrays, they must be the same shape. Array broadcasting allows NumPy to work effectively with arrays of sizes that don't match exactly. This can be useful in a host of different situations, and often saves both the amount of typing that is necessary and the amount of memory required for computations. There are four basic rules to determine the behavior of broadcasted arrays.

1. All input arrays of lesser dimension than the input array with largest dimension have 1's prepended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input sizes in that dimension.
3. An input can be used in the calculation if its size in a particular dimension either matches the output size in that dimension, or has a size exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension.

One simple example is multiplying a two dimensional array by a set of numbers along its rows or columns. Run the following lines of code and consider their output:

```
>>> import numpy as np
>>> A = np.ones((3, 3))
>>> B = np.vstack([1, 2, 3])
>>> A * B #multiplies the rows of A by each entry of B
array([[ 1.,  1.,  1.],
       [ 2.,  2.,  2.],
       [ 3.,  3.,  3.]])
>>> A * B.T #multiplies the columns of A by each entry of B
array([[ 1.,  2.,  3.],
```

```
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
>>> A = np.array([1, 2, 3]).reshape((3,1))
>>> B = np.array([1, 2])
>>> A + B
array([[2, 3],
       [3, 4],
       [4, 5]])
>>> A = np.arange(3)
>>> B = np.arange(3, 6)
>>> A[np.newaxis,:] * B[:,np.newaxis] #np.newaxis can be used to add↩
    a new axis and affect broadcasting
array([[ 0,  3,  6],
       [ 0,  4,  8],
       [ 0,  5, 10]])
```

It is important to note that broadcasting does not explicitly construct the larger array. In fact, internally, broadcasting uses no extra memory. For a more detailed description of array broadcasting rules, see `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html`.

Problem 1. Create a $100 \times 100 \times 3$ array of integers taking values in the range $[0, 256]$ (using the function `numpy.random.randint` with appropriate arguments). Such an array can represent a RGB image of 100×100 pixels, where each pixel is associated with an array of three integers indicating the amounts of red, green, and blue color present in that pixel, respectively. Using array broadcasting, multiply the red and green values by .5. Such an operation will tone down the red and green colors and make the image appear more blue.

Universal Functions

NumPy and SciPy include a wide variety of functions that are designed to operate on arrays. Such functions, which take in an array and return an array of the same size and datatype, are called *universal functions*, or `ufuncs`. To illustrate this point, consider the universal function `numpy.sin` and the standard function `math.sin`. If A is an array of floats (of any size), `math.sin(A)` throws an error, whereas `numpy.sin(A)`. returns an array of floats containing the sines of the entries of A. Other simple examples from NumPy include `cos`, `sqrt`, `exp`, and `log` all the way to special functions like `polygamma` in the `scipy.misc` submodule. There are far more functions available in NumPy than could possibly be included here, so you will want to become familiar with the NumPy and SciPy documentation at `docs.scipy.org/doc/`. If you need to do any sort of simple operation on an array, there is very often a function there to do it. These functions are almost always faster and more convenient than iterating through the whole array.

Most of these functions also allow you to specify an array for the output. It is useful in cases where you would like to avoid unnecessary memory allocation. The

output array does need to be the correct shape to store the output. For example:

```
>>> np.exp(A, out=A) #take exp(A) and store result in A
```

Other useful examples are `max`, `min`, `absolute`, and `average`. Each of these operations also allows you to specify whether you want to operate across a particular axis or over the whole array. For example:

```
>>> np.max(A, axis=0) #max along axis 0
```

The above example returns a row of A which represents the maximum of all the rows of A. If we had set `axis=1`, it would have taken the maximum of all the columns. If, for purposes of broadcasting (discussed later) you need the output of one of these functions to have the same number of dimensions as the original array, you can also include the argument `keepdims=True`.

Universal functions are designed to apply elementwise operations on each element of an array. Because of this, there can be a significant overhead when using a `ufunc` on a single value.

```
>>> timeit np.sin(.5)
1000000 loops, best of 3: 1.37   s  per loop
>>> timeit np.math.sin(.5)
10000000 loops, best of 3: 144 ns per loop
```

We can see decent performance increases when we know how to use a `ufunc`. They were not designed to handle single values efficiently.

Linear Algebra

One of the most useful set of functions available in NumPy and SciPy are the linear algebra functions. Even though NumPy has linear algebra library, SciPy contains all the functions that NumPy has plus a few more advanced functions. The linear algebra library is typically imported as

```
from scipy import linalg
```

To shorten the amount of typing, it can be aliased as `from scipy import linalg as la`.

It is important to note that there exists a `matrix` class that is very similar to a NumPy array. The matrix class is convenient when doing matrix operations. It behaves much like MATLAB's matrix object. However, using the matrix class is generally discouraged. They add no extra benefit over using a standard 2D NumPy array. All of the other functions of NumPy and SciPy are written to take advantage of the features of the `ndarray`. All mentions of a matrix in these lab manuals either refers to the mathematical object or an `ndarray`.

The linear algebra library contains several functions to construct special matrices. These matrices are common in specific areas of interest. The functions for constructing these special matrices are located in `linalg.special_matrices`. The linear algebra functions available in SciPy are very feature rich. There are functions that will find inverses, determinants, norms, solutions to linear systems, solve least squares problems, and decompose matrices.

We can use `linalg.solve` to solve linear systems. A least squares solution can be found with `linalg.lstsq`. `linalg.det` will return the determinant of a matrix. `linalg.inv` will find the inverse of a matrix.

You can read more about the linear algebra capabilities of SciPy in the documentation for the `linalg` module

Problem 2. Block ciphers are ciphers that encode blocks of input symbols at a time instead of one symbol at a time. In the days before computers, the Hill cipher was the first cipher that allowed practical encoding of more than three symbols at a time. It was invented by Lester Hill in 1929. The Hill cipher is considered a classical substitution cipher. The entire cipher is based on linear algebra and uses a matrix key. All substitution ciphers work with the 26 letters. Thus, all our operations will be done $\mod 26$ (modulo 26). To do this, we introduce you to the `%` operator in Python. This new operator allows us to do modular arithmetic. When applied to an array, it takes the elementwise mod.

This problem has a number of parts. You will write a function that accepts *plaintext* and returns the encoded *ciphertext*. You will also write a decoder that will accept ciphertext and return plaintext.

The Encoder:

1. We must first gather the plaintext to encode and a block size, n. We must split this plaintext into blocks removing any spaces. We need to covert each character to a number. We use the index of `string.lowercase` (found in the `string` module of the Python standard library). We can easily build a lookup table that will let you easily find the index. With a lookup table, we map each letter to its index.

```
>>> from string import lowercase
>>> lut = {a:i for i, a in enumerate(lowercase)}
>>> s = "this is a message"
>>> s = "".join(s.split()) #removes all whitespace
>>> map(lut.__getitem__, s) #returns a list of indices
[19, 7, 8, 18, 8, 18, 0, 12, 4, 18, 18, 0, 6, 4]
```

   Another way is to use `lowercase.index()` in a loop to find the index each time. We need to split the list of indices into n-length arrays and store them in a list. If the input is not a multiple of n, you will need to pad the input until it is a multiple of n. Pick any character to pad the input (typically it is a rarely used letter). The `itertools` module is useful for this. One of the common recipes for doing this task is available in the `itertools` documentation.

```
def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx
    args = [iter(iterable)] * n
    return itertools.izip_longest(fillvalue=fillvalue, *↩
        args)
```

It will be useful to wrap all of this step in a separate function as we will need to do the same thing when decoding (except for removing whitespace).

2. Find a suitable cipher key. The keys of a Hill cipher are square matrices. Let K, be our key. K must be invertible mod 26. Remember from linear algebra, that the determinant of square matrix will tell you if that matrix is invertible. To find a matrix that is invertible mod 26, we need to find a matrix with a determinant that is relatively prime to 26 (they share no common factors). The Euclidean algorithm can be used to determine if two numbers are co-prime (i.e. $\gcd(d, 26) = 1$). Write a function that generate random integer matrices using NumPy, checking the determinant, and returns a suitable key, K.

3. Write a function that will accept a message and a key matrix. The message should already be broken into n length blocks (you can do this inside the encode function if needed) and the matrix should be $n \times n$. A Hill cipher is the dot product of the block with the key. Return a ciphertext that is letters (the numbers correspond the indices in `string.lower`).

4. Write a function that will compute $K^{-1} \pmod{26}$. This will necessarily be an integer inverse. Use `linalg.inv` to find the inverse of K. Then
$$K^{-1} = \det(K) K^{-1} \det(K)^{-1} \pmod{26}$$
where $\det(K)^{-1}$ is the inverse of the determinant mod 26. You will need to round the determinant to the nearest integer before doing these steps. You will also need to round the results of each of your multiplications. You can check that you have the integer inverse by checking $KK^{-1} = I$.

5. Write a function that will decode a message given a ciphertext and the key. You will need to invert the key before decoding. Break the message into blocks of size n and calculate the dot product of each block with the inverted key. Return a plaintext that is letters (the numbers, again, correspond to indices in `string.lower`).

Experiment with your Hill cipher. If you are in a classroom setting, try sending encoded messages to friends (they will need the key you used to encode).

Polynomials

Many other useful functions are available in NumPy. One of particular interest is the polynomial array. This is a convenience object that represents the coefficients of a polynomial. Polynomials can be represented in two ways in NumPy. First, and probably most convenient, is the `np.poly1d` object which will represent a polynomial

as a 1D array. Second, and slightly faster, an array that represents the coefficients of the terms in descending order.

```
>>> a = np.poly1d([3, 5, 1, 2, 0, 1])
>>> print a
   5     4     3     2
3 x + 5 x + 1 x + 2 x + 1
>>> b = np.array([3, 5, 1, 2, 0, 1])
>>> print b
[3 5 1 2 0 1]
```

Both objects represent the same polynomial, $3x^5 + 5x^4 + x^3 + 2x^2 + 1$. When representing a polynomial as just an array of coefficients, NumPy provides special methods to treat them as polynomials

Function	Description
`np.polyadd`	Add two polynomial arrays
`np.polyder`	Find the derivative of a polynomial array
`np.polydiv`	Divide two polynomial arrays
`np.polyfit`	Find a least squares polynomial fit
`np.polyint`	Find the integral of a polynomial array
`np.polymul`	Multiply two polynomial arrays
`np.polysub`	Subtract two polynomial arrays
`np.polyval`	Evaluate a polynomial at specific points

These polynomial objects make evaluating series approximations to functions very easy. For example, you are probably familiar with the fact that

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

This series can be evaluated very easily as follows:

```
import numpy as np
from scipy.misc import factorial
n = 18 # number of terms
p = 1. / factorial(np.arange(18, -1, -1)) # compute coefficients
X = np.random.rand(10000) # where to evaluate the series
P = np.poly1d(p) #make polynomial object
P(X)
```

The last two lines could be replaced by

```
np.polyval(p, X)
```

Problem 3. a) Use the NumPy's polynomial objects to approximate the following series.

$$\arcsin x = \sum_{n=0}^{\infty} \frac{(2n)! x^{2n+1}}{(2n+1)\,(n!)^2\, 4^n}$$

Use your approximation to find a close approximation of π.

b) The lambert W function is the inverse of xe^x. It's taylor series is

$$W(x) = \sum_{n=1}^{\infty} \frac{(-n)^{n-1} x^n}{n!}$$

This series has a radius of convergence of $\frac{1}{e}$. Use the series to find a number x such that $xe^x = \frac{1}{4}$. Verify that your computation is correct.

Useful Functions

Function	Description
`np.intersect1d`	Return the intersection of two flattened arrays.
`np.union`	Return the union of two flattened arrays.
`np.diff`	Calculates a discrete difference of order n.
`np.absolute`	Return the elementwise absolute value of an array.
`np.pad`	
`np.nonzero`	
`np.count_nonzero`	
`np.select`	
`np.nan`	Represent IEEE NAN (not-a-number).
`np.inf`	Represent IEEE INF (infinity).
`np.who`	Print information about defined NumPy arrays in a variable scope.
`np.unique`	Return a sorted array of unique elements of an array.

Table 3.1: Some useful NumPy functions. For more information please refer to the NumPy documentation.