3 Introduction to NumPy

Lab Objective: *NumPy is a powerful Python package for manipulating data with multi-dimensional vectors. Its versatility and speed makes Python an ideal language for applied and computational mathematics. In this lab we introduce basic NumPy data structures and operations as a first step to numerical computing in Python.*

Arrays

In many algorithms, data can be represented mathematically as a *vector* or a *matrix*. Conceptually, a vector is just a list of numbers and a matrix is a two-dimensional list of numbers (a list of lists). However, even basic linear algebra operations like matrix multiplication are cumbersome to implement and slow to execute when data is stored this way. The *NumPy* module[1] [Oli06, ADH+01, Oli07] offers a much better solution.

The basic object in NumPy is the *array*, which is conceptually similar to a matrix. The NumPy array class is called `ndarray` (for "*n*-dimensional array"). The simplest way to explicitly create a 1-D `ndarray` is to define a list, then cast that list as an `ndarray` with NumPy's `array()` function.

```
>>> import numpy as np

# Create a 1-D array by passing a list into NumPy's array() function.
>>> np.array([8, 4, 6, 0, 2])
array([8, 4, 6, 0, 2])

# The string representation has no commas or an array() label.
>>> print(np.array([1, 3, 5, 7, 9]))
[1 3 5 7 9]
```

The alias "`np`" for numpy is standard in the Python community.

An `ndarray` can have arbitrarily many dimensions. A 2-D array is a 1-D array of 1-D arrays (like a list of lists), a 3-D array is a 1-D array of 2-D arrays (a list of lists of lists), and, more generally, an *n*-dimensional array is a 1-D array of $(n-1)$-dimensional arrays (a list of lists of lists of lists...). Each dimension is called an *axis*. For a 2-D array, the 0-axis indexes the rows and the 1-axis indexes the columns. Elements are accessed using brackets and indices, with the axes separated by commas.

---

[1] NumPy is *not* part of the standard library, but it is included in most Python distributions.

```
# Create a 2-D array by passing a list of lists into array().
>>> A = np.array( [ [1, 2, 3],[4, 5, 6] ] )
>>> print(A)
[[1 2 3]
 [4 5 6]]

# Access elements of the array with brackets.
>>> print(A[0, 1], A[1, 2])
2 6

# The elements of a 2-D array are 1-D arrays.
>>> A[0]
array([1, 2, 3])
```

Problem 1. There are two main ways to perform matrix multiplication in NumPy: with NumPy's `dot()` function (`np.dot(A, B)`), or with the `@` operator (`A @ B`). Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 3 & -1 & 4 \\ 1 & 5 & -9 \end{bmatrix} \qquad\qquad B = \begin{bmatrix} 2 & 6 & -5 & 3 \\ 5 & -8 & 9 & 7 \\ 9 & -3 & -2 & -3 \end{bmatrix}$$

Return the matrix product AB.

For examples of array initialization and matrix multiplication, use object introspection in IPython to look up the documentation for `np.ndarray`, `np.array()` and `np.dot()`.

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

In [2]: np.array?        # press 'enter'
```

ACHTUNG!

The `@` operator was not introduced until Python 3.5. It triggers the `__matmul__()` magic method,[a] which for the `ndarray` is essentially a wrapper around `np.dot()`. If you are using a previous version of Python, always use `np.dot()` to perform basic matrix multiplication.

---
[a]See the lab on Object Oriented Programming for an overview of magic methods.

Basic Array Operations

NumPy arrays behave differently with respect to the binary arithmetic operators + and * than Python lists do. For lists, + concatenates two lists and * replicates a list by a scalar amount (strings also behave this way).

```
# Addition concatenates lists together.
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

# Mutliplication concatenates a list with itself a given number of times.
>>> [1, 2, 3] * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

NumPy arrays act like mathematical vectors and matrices: + and * perform component-wise addition or multiplication.

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

# Addition or multiplication by a scalar acts on each element of the array.
>>> x + 10                          # Add 10 to each entry of x.
array([11, 12, 13])
>>> x * 4                           # Multiply each entry of x by 4.
array([ 4,  8, 12])

# Add two arrays together (component-wise).
>>> x + y
array([5, 7, 9])

# Multiply two arrays together (component-wise).
>>> x * y
array([ 4, 10, 18])
```

Problem 2. Write a function that defines the following matrix as a NumPy array.

$$A = \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ -5 & 3 & 1 \end{bmatrix}$$

Return the matrix $-A^3 + 9A^2 - 15A$.

In this context, $A^2 = AA$ (the matrix product, not the component-wise square). The somewhat surprising result is a demonstration of the Cayley-Hamilton theorem.

Array Attributes

An `ndarray` object has several attributes, some of which are listed below.

Attribute	Description
dtype	The type of the elements in the array.
ndim	The number of axes (dimensions) of the array.
shape	A tuple of integers indicating the size in each dimension.
size	The total number of elements in the array.

```
>>> A = np.array([[1, 2, 3],[4, 5, 6]])

# 'A' is a 2-D array with 2 rows, 3 columns, and 6 entries.
>>> print(A.ndim, A.shape, A.size)
2 (2, 3) 6
```

Note that `ndim` is the number of entries in `shape`, and that the `size` of the array is the product of the entries of `shape`.

Array Creation Routines

In addition to casting other structures as arrays via `np.array()`, NumPy provides efficient ways to create certain commonly-used arrays.

Function	Returns
`arange()`	Array of sequential integers (like `list(range())`).
`eye()`	2-D array with ones on the diagonal and zeros elsewhere.
`ones()`	Array of given shape and type, filled with ones.
`ones_like()`	Array of ones with the same shape and type as a given array.
`zeros()`	Array of given shape and type, filled with zeros.
`zeros_like()`	Array of zeros with the same shape and type as a given array.
`full()`	Array of given shape and type, filled with a specified value.
`full_like()`	Full array with the same shape and type as a given array.

Each of these functions accepts the keyword argument `dtype` to specify the data type. Common types include `np.bool_`, `np.int64`, `np.float64`, and `np.complex128`.

```
# A 1-D array of 5 zeros.
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

# A 2x5 matrix (2-D array) of integer ones.
>>> np.ones((2,5), dtype=np.int)     # The shape is specified as a tuple.
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])

# The 2x2 identity matrix.
>>> I = np.eye(2)
>>> print(I)
[[ 1.  0.]
 [ 0.  1.]]

# Array of 3s the same size as 'I'.
>>> np.full_like(I, 3)               # Equivalent to np.full(I.shape, 3).
array([[ 3.,  3.],
       [ 3.,  3.]])
```

Unlike native Python data structures, **all elements of a NumPy array must be of the same data type**. To change an existing array's data type, use the array's `astype()` method.

```python
# A list of integers becomes an array of integers.
>>> x = np.array([0, 1, 2, 3, 4])
>>> print(x)
[0 1 2 3 4]
>>> x.dtype
dtype('int64')

# Change the data type to one of NumPy's float types.
>>> x = x.astype(np.float64)        # Equivalent to x = np.float64(x).
>>> print(x)
[ 0.  1.  2.  3.  4.]               # Floats are displayed with periods.
>>> x.dtype
dtype('float64')
```

The following functions are for dealing with the diagonal, upper, or lower portion of an array.

Function	Description
diag()	Extract a diagonal or construct a diagonal array.
tril()	Get the lower-triangular portion of an array by replacing entries above the diagonal with zeros.
triu()	Get the upper-triangular portion of an array by replacing entries below the diagonal with zeros.

```python
>>> A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Get only the upper triangular entries of 'A'.
>>> np.triu(A)
array([[1, 2, 3],
       [0, 5, 6],
       [0, 0, 9]])

# Get the diagonal entries of 'A' as a 1-D array.
>>> np.diag(A)
array([1, 5, 9])

# diag() can also be used to create a diagonal matrix from a 1-D array.
>>> np.diag([1, 11, 111])
array([[  1,   0,   0],
       [  0,  11,   0],
       [  0,   0, 111]])
```

See http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html for the official documentation on NumPy's array creation routines.

Problem 3. Write a function that defines the following matrices as NumPy arrays using the functions presented in this section (not `np.array()`). Calculate the matrix product ABA. Change the data type of the resulting matrix to `np.int64`, then return it.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} -1 & 5 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Data Access

Array Slicing

Indexing for a 1-D NumPy array uses the slicing syntax `x[start:stop:step]`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed. For multi-dimensional arrays, use a comma to separate slicing syntax for each axis.

```
# Make an array of the integers from 0 to 10 (exclusive).
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Access elements of the array with slicing syntax.
>>> x[3]                              # The element at index 3.
3
>>> x[:3]                             # Everything before index 3.
array([0, 1, 2])
>>> x[3:]                             # Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]                            # Everything between index 3 and index 8 (↩
    exclusive).
array([3, 4, 5, 6, 7])

>>> A = np.array([[0,1,2,3,4],[5,6,7,8,9]])
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Use a comma to separate the dimensions for multi-dimensional arrays.
>>> A[1, 2]                           # The element at row 1, column 2.
7
>>> A[:, 2:]                          # All of the rows, from column 2 on.
array([[2, 3, 4],
       [7, 8, 9]])
```

> **NOTE**
>
> Indexing and slicing operations return a *view* of the array. Changing a view of an array also changes the original array. In other words, **arrays are mutable**. To create a copy of an array, use `np.copy()` or the array's `copy()` method. Changes to a copy of an array does not affect the original array, but copying an array uses more time and memory than getting a view.

Fancy Indexing

So-called *fancy indexing* is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a *mask*) to extract specific elements.

```python
>>> x = np.arange(0, 50, 10)           # The integers from 0 to 50 by tens.
>>> x
array([ 0, 10, 20, 30, 40])

# An array of integers extracts the entries of 'x' at the given indices.
>>> index = np.array([3, 1, 4])        # Get the 3rd, 1st, and 4th elements.
>>> x[index]                           # Same as np.array([x[i] for i in index]).
array([30, 10, 40])

# A boolean array extracts the elements of 'x' at the same places as 'True'.
>>> mask = np.array([True, False, False, True, False])
>>> x[mask]                            # Get the 0th and 3rd entries.
array([ 0, 30])
```

Fancy indexing is especially useful for extracting or changing the values of an array that meet some sort of criterion. Use comparison operators like < and == to create masks.

```python
>>> y = np.arange(10, 20, 2)           # Every other integers from 10 to 20.
>>> y
array([10, 12, 14, 16, 18])

# Extract the values of 'y' larger than 15.
>>> mask = y > 15                      # Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False,  True,  True], dtype=bool)
>>> y[mask]                            # Same as y[y > 15]
array([16, 18])

# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14  100  100]
```

While indexing and slicing always return a view, fancy indexing always returns a copy.

Problem 4. Write a function that accepts a single array as input. Make a copy of the array, then use fancy indexing to set all negative entries of the copy to 0. Return the copy.

Array Manipulation

Shaping

An array's `shape` attribute describes its dimensions. Use `np.reshape()` or the array's `reshape()` method to give an array a new shape. The total number of entries in the old array and the new array must be the same in order for the shaping to work correctly. Using a `-1` in the new shape tuple makes the specified dimension as long as necessary.

```
>>> A = np.arange(12)                    # The integers from 0 to 12 (exclusive).
>>> print(A)
[ 0  1  2  3  4  5  6  7  8  9 10 11]

# 'A' has 12 entries, so it can be reshaped into a 3x4 matrix.
>>> A.reshape((3,4))                      # The new shape is specified as a tuple.
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Reshape 'A' into an array with 2 rows and the appropriate number of columns.
>>> A.reshape((2,-1))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

Use `np.ravel()` to flatten a multi-dimensional array into a 1-D array and `np.transpose()` or the `T` attribute to transpose a 2-D array in the matrix sense.

```
>>> A = np.arange(12).reshape((3,4))
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Flatten 'A' into a one-dimensional array.
>>> np.ravel(A)                           # Equivalent to A.reshape(A.size)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# Transpose the matrix 'A'.
>>> A.T                                   # Equivalent to np.transpose(A).
array([[ 0,  4,  8],
```

```
       [ 1,   5,   9],
       [ 2,   6,  10],
       [ 3,   7,  11]])
```

> ## NOTE
>
> By default, all NumPy arrays that can be represented by a single dimension, including column slices, are automatically reshaped into "flat" 1-D arrays. For example, by default an array will have 10 elements instead of 10 arrays with one element each. Though we usually represent vectors vertically in mathematical notation, NumPy methods such as `dot()` are implemented to purposefully work well with 1-D "row arrays".
>
> ```
> >>> A = np.arange(10).reshape((2,5))
> >>> A
> array([[0, 1, 2, 3, 4],
>        [5, 6, 7, 8, 9]])
>
> # Slicing out a column of A still produces a "flat" 1-D array.
> >>> x = A[:,1]                       # All of the rows, column 1.
> >>> x
> array([1, 6])                        # Not array([[1],
> >>> x.shape                          #               [6]])
> (2,)
> >>> x.ndim
> 1
> ```
>
> However, it is occasionally necessary to change a 1-D array into a "column array". Use `np.reshape()`, `np.vstack()`, or slice the array and put `np.newaxis` on the second axis. Note that `np.transpose()` does not alter 1-D arrays.
>
> ```
> >>> x = np.arange(3)
> >>> x
> array([0, 1, 2])
>
> >>> x.reshape((-1,1))                # Or x[:,np.newaxis] or np.vstack(x).
> array([[0],
>        [1],
>        [2]])
> ```
>
> Do not force a 1-D vector to be a column vector unless necessary.

Stacking

NumPy has functions for *stacking* two or more arrays with similar dimensions into a single block matrix. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

Function	Description
concatenate()	Join a sequence of arrays along an existing axis
hstack()	Stack arrays in sequence horizontally (column wise).
vstack()	Stack arrays in sequence vertically (row wise).
column_stack()	Stack 1-D arrays as columns into a 2-D array.

```
>>> A = np.arange(6).reshape((2,3))
>>> B = np.zeros((4,3))

# vstack() stacks arrays vertically (row-wise).
>>> np.vstack((A,B,A))
array([[ 0.,   1.,   2.],           # A
       [ 3.,   4.,   5.],
       [ 0.,   0.,   0.],           # B
       [ 0.,   0.,   0.],
       [ 0.,   0.,   0.],
       [ 0.,   0.,   0.],
       [ 0.,   1.,   2.],           # A
       [ 3.,   4.,   5.]])

>>> A = A.T
>>> B = np.ones((3,4))

# hstack() stacks arrays horizontally (column-wise).
>>> np.hstack((A,B,A))
array([[ 0.,   3.,   1.,   1.,   1.,   1.,   0.,   3.],
       [ 1.,   4.,   1.,   1.,   1.,   1.,   1.,   4.],
       [ 2.,   5.,   1.,   1.,   1.,   1.,   2.,   5.]])

# column_stack() stacks arrays horizontally, including 1-D arrays.
>>> np.column_stack((A, np.zeros(3), np.ones(3), np.full(3, 2)))
array([[ 0.,   3.,   0.,   1.,   2.],
       [ 1.,   4.,   0.,   1.,   2.],
       [ 2.,   5.,   0.,   1.,   2.]])
```

See http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html for more array manipulation routines and documentation.

Problem 5. Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & 0 & 0 \\ 3 & 3 & 0 \\ 3 & 3 & 3 \end{bmatrix} \qquad C = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

Use NumPy's stacking functions to create and return the block matrix:

$$\begin{bmatrix} \mathbf{0} & A^\mathsf{T} & I \\ A & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & C \end{bmatrix},$$

where I is the 3×3 identity matrix and each $\mathbf{0}$ is a matrix of all zeros of appropriate size.
A block matrix of this form is used in the interior point method for linear optimization.

Array Broadcasting

Many matrix operations make sense only when the two operands have the same shape, such as element-wise addition. *Array broadcasting* extends such operations to accept some (but not all) operands with different shapes, and occurs automatically whenever possible.

Suppose, for example, that we would like to add different values to the columns of an $m \times n$ matrix A. Adding a 1-D array x with the n entries to A will automatically do this correctly. To add different values to the different rows of A, first reshape a 1-D array of m values into a column array. Broadcasting then correctly takes care of the operation.

Broadcasting can also occur between two 1-D arrays, once they are reshaped appropriately.

```
>>> A = np.arange(12).reshape((4,3))
>>> x = np.arange(3)
>>> A
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> x
array([0, 1, 2])

# Add the entries of 'x' to the corresponding columns of 'A'.
>>> A + x
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10],
       [ 9, 11, 13]])

>>> y = np.arange(0, 40, 10).reshape((4,1))
>>> y
array([[ 0],
       [10],
       [20],
       [30]])

# Add the entries of 'y' to the corresponding rows of 'A'.
>>> A + y
array([[ 0,  1,  2],
       [13, 14, 15],
       [26, 27, 28],
```

```
         [39, 40, 41]])

# Add 'x' and 'y' together with array broadcasting.
>>> x + y
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Numerical Computing with NumPy

Universal Functions

A *universal function* is one that operates on an entire array element-wise. Universal functions are significantly more efficient than using a loop to operate individually on each element of an array.

Function	Description
abs() or absolute()	Calculate the absolute value element-wise.
exp() / log()	Exponential (e^x) / natural log element-wise.
maximum() / minimum()	Element-wise maximum / minimum of two arrays.
sqrt()	The positive square-root, element-wise.
sin(), cos(), tan(), etc.	Element-wise trigonometric operations.

```
>>> x = np.arange(-2,3)
>>> print(x, np.abs(x))              # Like np.array([abs(i) for i in x]).
[-2 -1  0  1  2] [2 1 0 1 2]

>>> np.sin(x)                        # Like np.array([math.sin(i) for i in x]).
array([-0.90929743, -0.84147098,  0.        ,  0.84147098,  0.90929743])
```

See http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs for a more comprehensive list of universal functions.

> **ACHTUNG!**
>
> The math module has many useful functions for numerical computations. However, most of these functions can only act on single numbers, not on arrays. NumPy functions can act on either scalars or entire arrays, but math functions tend to be a little faster for acting on scalars.
>
> ```
> >>> import math
>
> # Math and NumPy functions can both operate on scalars.
> >>> print(math.exp(3), np.exp(3))
> 20.085536923187668 20.0855369232
>
> # However, math functions cannot operate on arrays.
> ```

```
>>> x = np.arange(-2, 3)
>>> np.tan(x)
array([ 2.18503986, -1.55740772,  0.        ,  1.55740772, -2.18503986])
>>> math.tan(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Always use universal NumPy functions, not the `math` module, when working with arrays.

Other Array Methods

The `np.ndarray` class itself has many useful methods for numerical computations.

Method	Returns
`all()`	`True` if all elements evaluate to `True`.
`any()`	`True` if any elements evaluate to `True`.
`argmax()`	Index of the maximum value.
`argmin()`	Index of the minimum value.
`argsort()`	Indices that would sort the array.
`clip()`	restrict values in an array to fit within a given range
`max()`	The maximum element of the array.
`mean()`	The average value of the array.
`min()`	The minimum element of the array.
`sort()`	Return nothing; sort the array in-place.
`std()`	The standard deviation of the array.
`sum()`	The sum of the elements of the array.
`var()`	The variance of the array.

Each of these `np.ndarray` methods has an equivalent NumPy function. For example, `A.max()` and `np.max(A)` operate the same way. The one exception is the `sort()` function: `np.sort()` returns a sorted copy of the array, while `A.sort()` sorts the array in-place and returns nothing.

Every method listed can operate *along an axis* via the keyword argument `axis`. If `axis` is specified for a method on an n-D array, the return value is an $(n-1)$-D array, the specified axis having been collapsed in the evaluation process. If `axis` is not specified, the return value is usually a scalar. Refer to the NumPy Visual Guide in the appendix for more visual examples.

```
>>> A = np.arange(9).reshape((3,3))
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# Find the maximum value in the entire array.
>>> A.max()
8

# Find the minimum value of each column.
```

```
>>> A.min(axis=0)                        # np.array([min(A[:,i]) for i in range(3)])
array([0, 1, 2])

# Compute the sum of each row.
>>> A.sum(axis=1)                        # np.array([sum(A[i,:]) for i in range(3)])
array([3, 12, 21])
```

See `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html` for a more comprehensive list of array methods.

Problem 6. A matrix is called *row-stochastic*[a] if its rows each sum to 1. Stochastic matrices are fundamentally important for finite discrete random processes and some machine learning algorithms.

Write a function than accepts a matrix (as a 2-D array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting and the `axis` argument instead of a loop.

---

[a]Similarly, a matrix is called *column-stochastic* if its columns each sum to 1.

Problem 7. This problem comes from `https://projecteuler.net`.

In the 20×20 grid below, four numbers along a diagonal line have been marked in red.

```
08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
```

The product of these numbers is $26 \times 63 \times 78 \times 14 = 1788696$. Write a function that returns the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the grid.

For convenience, this array has been saved in the file `grid.npy`. Use the following syntax to extract the array:

```
>>> grid = np.load("grid.npy")
```

One way to approach this problem is to iterate through the rows and columns of the array, checking small slices of the array at each iteration and updating the current largest product. Array slicing, however, provides a much more efficient solution.

The naïve method for computing the greatest product of four adjacent numbers in a horizontal row might be as follows:

```
>>> winner = 0
>>> for i in range(20):
...     for j in range(17):
...         winner = max(np.prod(grid[i,j:j+4]), winner)
...
>>> winner
48477312
```

Instead, use array slicing to construct a single array where the (i, j)th entry is the product of the four numbers to the right of the (i, j)th entry in the original grid. Then find the largest element in the new array.

```
>>> np.max(grid[:,:-3] * grid[:,1:-2] * grid[:,2:-1] * grid[:,3:])
48477312
```

Use slicing to similarly find the greatest products of four vertical, right diagonal, and left diagonal adjacent numbers.
(Hint: Consider drawing the portions of the grid that each slice in the above code covers, like the examples in the visual guide. Then draw the slices that produce vertical, right diagonal, or left diagonal sequences, and translate the pictures into slicing syntax.)

ACHTUNG!

All of the examples in this lab use NumPy arrays, objects of type `np.ndarray`. NumPy also has a "matrix" data structure called `np.matrix` that was built specifically for MATLAB users who are transitioning to Python and NumPy. It behaves slightly differently than the regular array class, and can cause some unexpected and subtle problems.

For consistency (and your sanity), **never** use a NumPy matrix; **always** use NumPy arrays. If necessary, cast a matrix object as an array with `np.array()`.

Additional Material

Random Sampling

The submodule `np.random` holds many functions for creating arrays of random values chosen from probability distributions such as the uniform, normal, and multinomial distributions. It also contains some utility functions for getting non-distributional random samples, such as random integers or random samples from a given array.

Function	Description
choice()	Take random samples from a 1-D array.
random()	Uniformly distributed floats over $[0, 1)$.
randint()	Random integers over a half-open interval.
random_integers()	Random integers over a closed interval.
randn()	Sample from the standard normal distribution.
permutation()	Randomly permute a sequence / generate a random sequence.

Function	Distribution
beta()	Beta distribution over $[0, 1]$.
binomial()	Binomial distribution.
exponential()	Exponential distribution.
gamma()	Gamma distribution.
geometric()	Geometric distribution.
multinomial()	Multivariate generalization of the binomial distribution.
multivariate_normal()	Multivariate generalization of the normal distribution.
normal()	Normal / Gaussian distribution.
poisson()	Poisson distribution.
uniform()	Uniform distribution.

Note that many of these functions have counterparts in the standard library's `random` module. These NumPy functions, however, are much better suited for working with large collections of random samples.

```
# 5 uniformly distributed values in the interval [0, 1).
>>> np.random.random(5)
array([ 0.21845499,  0.73352537,  0.28064456,  0.66878454,  0.44138609])


# A 2x5 matrix (2-D array) of integers in the interval [10, 20).
>>> np.random.randint(10, 20, (2,5))
array([[17, 12, 13, 13, 18],
       [16, 10, 12, 18, 12]])
```

Saving and Loading Arrays

It is often useful to save an array as a file for later use. NumPy provides several easy methods for saving and loading array data.

Function	Description
save()	Save a single array to a .npy file.
savez()	Save multiple arrays to a .npz file.
savetxt()	Save a single array to a .txt file.
load()	Load and return an array or arrays from a .npy or .npz file.
loadtxt()	Load and return an array from a text file.

```python
# Save a 100x100 matrix of uniformly distributed random values.
>>> x = np.random.random((100,100))
>>> np.save("uniform.npy", x)         # Or np.savetxt("uniform.txt", x).

# Read the array from the file and check that it matches the original.
>>> y = np.load("uniform.npy")        # Or np.loadtxt("uniform.txt").
>>> np.allclose(x, y)                 # Check that x and y are close entry-wise.
True
```

To save several arrays to a single file, specify a keyword argument for each array in np.savez(). Then np.load() will return a dictionary-like object with the keyword parameter names from the save command as the keys.

```python
# Save two 100x100 matrices of normally distributed random values.
>>> x = np.random.randn(100,100)
>>> y = np.random.randn(100,100)
>>> np.savez("normal.npz", first=x, second=y)

# Read the arrays from the file and check that they match the original.
>>> arrays = np.load("normal.npz")
>>> np.allclose(x, arrays["first"])
True
>>> np.allclose(y, arrays["second"])
True
```

Bibliography

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001. [1]

[Oli06]   Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006. [1]

[Oli07]   Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3), 2007. [1]