NumPy Basics: Arrays and Vectorized Computation

Part 6

File Input and Output with Arrays

- `np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk.
- Arrays are saved by default in an uncompressed raw binary format with file extension `.npy`:

```
In [171]: arr = np.arange(10)
          np.save('some_array', arr)
```

- If the file path does not already end in `.npy`, the extension will be appended.

- The array on disk can then be loaded with `np.load`:

```
In [172]: np.load('some_array.npy')
Out[172]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- You save multiple arrays in an uncompressed archive using `np.savez` and passing the arrays as keyword arguments:

```
In [173]: np.savez('array_archive.npz', a=arr, b=arr)
```

- When loading an `.npz` file, you get back a dict-like object that loads the individual arrays lazily:

```
In [174]: arch = np.load('array_archive.npz')
          arch['b']

Out[174]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- If your data compresses well, you may wish to use `numpy.savez_compressed` instead:

```
In [176]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)

In [177]: arch = np.load('arrays_compressed.npz')
          arch['b']

Out[177]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Linear Algebra

- Unlike some languages like MATLAB, multiplying two two-dimensional arrays with `*` is an element-wise product instead of a matrix dot product.
- Thus, there is a function `dot`, both an array method and a function in the numpy namespace, for matrix multiplication:

```
In [178]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
          y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [179]: x

Out[179]: array([[1., 2., 3.],
                 [4., 5., 6.]])

In [180]: y

Out[180]: array([[ 6., 23.],
                 [-1.,  7.],
                 [ 8.,  9.]])

In [181]: x.dot(y)

Out[181]: array([[ 28.,  64.],
                 [ 67., 181.]])
```

- `x.dot(y)` is equivalent to `np.dot(x, y)`:

```
In [182]: np.dot(x, y)

Out[182]: array([[ 28.,  64.],
                 [ 67., 181.]])
```

- A matrix product between a two-dimensional array and a suitably sized one-dimensional array results in a one-dimensional array:

```
In [183]: x
Out[183]: array([[1., 2., 3.],
                 [4., 5., 6.]])

In [184]: np.dot(x, np.ones(3))
Out[184]: array([ 6., 15.])
```

- The `@` symbol (as of Python 3.5) also works as an infix operator that performs matrix multiplication:

```
In [185]: x @ np.ones(3)
Out[185]: array([ 6., 15.])
```

- `numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant.
- These are implemented under the hood via the same industry-standard linear algebra libraries used in other languages like MATLAB and R, such as BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library):

```
In [186]: from numpy.linalg import inv, qr
          X = np.random.randn(5, 5)
          mat = X.T.dot(X)

In [187]: mat

Out[187]: array([[ 1.27621004, -0.22921106, -1.3508425 , -0.28065747,  0.35316522],
                 [-0.22921106,  4.50289341, -4.78223736, -3.70596593, -0.31637721],
                 [-1.3508425 , -4.78223736,  8.32288414,  3.88402481, -0.14646479],
                 [-0.28065747, -3.70596593,  3.88402481,  3.86668698, -0.12498055],
                 [ 0.35316522, -0.31637721, -0.14646479, -0.12498055,  0.59264934]])

In [188]: inv(mat)

Out[188]: array([[ 8.40733937, 11.48186812,  4.80214104,  6.91283976,  3.7640077 ],
                 [11.48186812, 18.41089423,  7.2222652 , 11.45668531,  7.18715273],
                 [ 4.80214104,  7.2222652 ,  3.13060837,  4.21181815,  2.65575288],
                 [ 6.91283976, 11.45668531,  4.21181815,  7.66054936,  4.65294087],
                 [ 3.7640077 ,  7.18715273,  2.65575288,  4.65294087,  4.91865248]])

In [189]: mat.dot(inv(mat))

Out[189]: array([[ 1.00000000e+00,  1.78722830e-14,  2.57802495e-15,
                   8.90489092e-15, -3.19358913e-16],
                 [-3.98405420e-15,  1.00000000e+00,  1.28203364e-15,
                  -6.94044070e-15,  2.90691061e-15],
                 [ 2.29019354e-15, -3.67406512e-15,  1.00000000e+00,
                   2.50187304e-15,  6.98117978e-16],
                 [ 1.53361685e-15,  1.36113570e-14, -2.63505700e-15,
                   1.00000000e+00,  1.03599526e-15],
                 [ 2.33255645e-16,  2.98486440e-15, -7.92357654e-16,
                   1.36178826e-15,  1.00000000e+00]])
```

```
In [190]: q, r = qr(mat)

In [191]: q

Out[191]: array([[-0.66261815,  0.3373003 , -0.40912897, -0.40055033,  0.34545421],
                 [ 0.11900816, -0.71055747, -0.01232091, -0.2137491 ,  0.65962463],
                 [ 0.7013679 ,  0.37498715, -0.55031942, -0.07215828,  0.24374047],
                 [ 0.14571954,  0.47304521,  0.72542733, -0.21537628,  0.42703898],
                 [-0.18336612,  0.13019004, -0.05801208,  0.86155499,  0.45142554]])

In [192]: r

Out[192]: array([[-1.9260113 , -3.14836607,  6.75620622,  3.05542789, -0.50127477],
                 [ 0.        , -6.8644337 ,  7.88164064,  5.80794189,  0.3070401 ],
                 [ 0.        ,  0.        , -1.14258003,  0.83528246, -0.18503478],
                 [ 0.        ,  0.        ,  0.        , -0.31617053,  0.47425139],
                 [ 0.        ,  0.        ,  0.        ,  0.        ,  0.0917783 ]])
```

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudo-inverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

Pseudorandom Number Generation

- The `numpy.random` module supplements the built-in Python `random` with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.
- For example, you can get a 4 × 4 array of samples from the standard normal distribution using `normal`:

```
In [193]: samples = np.random.normal(size=(4, 4))
          samples

Out[193]: array([[-0.48623968,  0.19549196, -0.53056981, -0.67304926],
                 [ 0.67414711,  1.01547146, -0.22468823, -0.65790939],
                 [ 2.776713  , -0.96051976, -0.2147762 , -1.92574968],
                 [ 0.26452563,  1.67697861,  1.17274609,  1.46183675]])
```

- Python's built-in `random` module, by contrast, only samples one value at a time.
- As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [194]: from random import normalvariate
          N = 1000000

In [195]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
          617 ms ± 4.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [196]: %timeit np.random.normal(size=N)
          39.2 ms ± 565 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

- We say that these are *pseudorandom* numbers because they are generated by an algorithm with deterministic behavior based on the *seed* of the random number generator.
- You can change NumPy's random number generation seed using `np.random.seed`:

```
In [197]: np.random.seed(1234)
```

- The data generation functions in `numpy.random` use a global random seed.
- To avoid global state, you can use `numpy.random.RandomState` to create a random number generator isolated from others:

```
In [198]: rng = np.random.RandomState(1234)
          rng.randn(10)

Out[198]: array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873,
                  0.88716294,  0.85958841, -0.6365235 ,  0.01569637, -2.24268495])
```

Function	Description
`seed`	Seed the random number generator
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in-place
`rand`	Draw samples from a uniform distribution
`randint`	Draw random integers from a given low-to-high range
`randn`	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
`binomial`	Draw samples from a binomial distribution
`normal`	Draw samples from a normal (Gaussian) distribution
`beta`	Draw samples from a beta distribution
`chisquare`	Draw samples from a chi-square distribution
`gamma`	Draw samples from a gamma distribution
`uniform`	Draw samples from a uniform [0, 1) distribution

Example: Random Walks

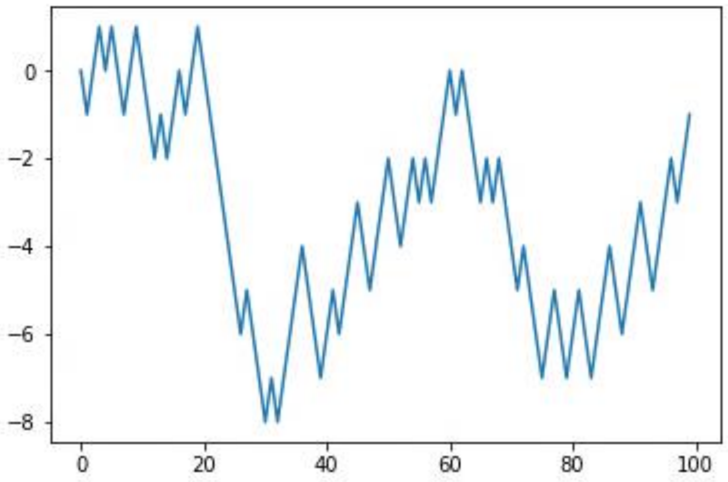
- The simulation of random walks provides an illustrative application of utilizing array operations.
- Let's first consider a simple random walk starting at `0` with steps of `1` and `−1` occurring with equal probability.
- Here is a pure Python way to implement a single random walk with `1,000` steps using the built-in `random` module:

```
In [199]: import random
          position = 0
          walk = [position]
          steps = 1000
          for i in range(steps):
              step = 1 if random.randint(0, 1) else -1
              position += step
              walk.append(position)
```

- The following figure is an example plot of the first 100 values on one of these random walks:

```
In [200]: plt.figure()
Out[200]: <Figure size 432x288 with 0 Axes>
          <Figure size 432x288 with 0 Axes>

In [201]: plt.plot(walk[:100])
Out[201]: [<matplotlib.lines.Line2D at 0x7fcd5528c390>]
```



- You might make the observation that `walk` is simply the cumulative sum of the random steps and could be evaluated as an array expression.
- Thus, I use the `np.random` module to draw `1,000` coin flips at once, set these to `1` and `−1`, and compute the cumulative sum:

```
In [204]: nsteps = 1000
          draws = np.random.randint(0, 2, size=nsteps)
          steps = np.where(draws > 0, 1, -1)
          walk = steps.cumsum()
```

- From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [205]: walk.min()
Out[205]: -3

In [206]: walk.max()
Out[206]: 31
```

- A more complicated statistic is the *first crossing time*, the step at which the random walk reaches a particular value.
- Here we might want to know how long it took the random walk to get at least `10` steps away from the origin `0` in either direction.
- `np.abs(walk) >= 10` gives us a boolean array indicating where the walk has reached or exceeded `10`, but we want the index of the first `10` or `−10`.
- Turns out, we can compute this using `argmax`, which returns the first index of the maximum value in the boolean array (`True` is the maximum value):

```
In [207]: (np.abs(walk) >= 10).argmax()

Out[207]: 37
```

- Note that using `argmax` here is not always efficient because it always makes a full scan of the array.
- In this special case, once a `True` is observed we know it to be the maximum value.

Simulating Many Random Walks at Once

- If your goal was to simulate many random walks, say `5,000` of them, you can generate all of the random walks with minor modifications to the preceding code.
- If passed a 2-tuple, the `numpy.random` functions will generate a two-dimensional array of draws, and we can compute the cumulative sum across the rows to compute all `5,000` random walks in one shot:

```
In [208]: nwalks = 5000
          nsteps = 1000
          draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1
          steps = np.where(draws > 0, 1, -1)
          walks = steps.cumsum(1)
          walks

Out[208]: array([[  1,   0,   1, ...,   8,   7,   8],
                 [  1,   0,  -1, ...,  34,  33,  32],
                 [  1,   0,  -1, ...,   4,   5,   4],
                 ...,
                 [  1,   2,   1, ...,  24,  25,  26],
                 [  1,   2,   3, ...,  14,  13,  14],
                 [ -1,  -2,  -3, ..., -24, -23, -22]])
```

- Now, we can compute the maximum and minimum values obtained over all of the walks:

```
In [209]: walks.max()
Out[209]: 138

In [210]: walks.min()
Out[210]: -133
```

- Out of these walks, let’s compute the minimum crossing time to 30 or -30.
- This is slightly tricky because not all $5{,}000$ of them reach 30.
- We can check this using the `any` method:

```
In [211]: hits30 = (np.abs(walks) >= 30).any(1)
          hits30

Out[211]: array([False,  True, False, ..., False,  True, False])

In [212]: hits30.sum() # Number that hit 30 or -30

Out[212]: 3410
```

- We can use this boolean array to select out the rows of walks that actually cross the absolute `30` level and call `argmax` across axis `1` to get the crossing times:

```
In [213]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)
          crossing_times.mean()

Out[213]: 498.8897360703812
```