04a_data_numpy

November 23, 2022

1 Doing More with Data: `numpy`

1.1 Introduction to Python

Data Sciences Institute, University of Toronto

Instructor: Kaylie Lau | TA: Salaar Liaqat

November - December 2022

2 Contents:

Dataset used: california_housing_test.csv

1. Setup
2. `numpy`
3. Intro and arrays
4. Loading `numpy`
5. Creating arrays
6. Reshaping
7. Basic operations
8. Logic and filtering
9. Loading data

3 `numpy`

3.1 What is `numpy`?

`numpy` is a Python package for scientific computing. It gives us *homogenous multidimensional arrays* -- a way of representing grids of elements where all elements are the same data type -- and functions to work efficiently with them. `numpy` both underpins and complements other data science libraries, including `pandas`, `matplotlib`, and `scikit-learn`.

3.2 Arrays vs lists

numpy arrays	Python list
all elements must be the same type	elements can be different types
fixed size	can change size
n-dimensional	1-dimensional
faster to process	slower to process
consumes less memory	consumes more memory

3.3 Loading numpy

We can `import` numpy like any other module. For convenience, `numpy` is typically loaded `as np` -- an alias that makes referencing it easier.

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

3.4 numpy arrays

The main object in `numpy` is the `ndarray`, also referred to as the `array`. Dimensions in an array are called *axes*. Most arrays we'll work with will be one-dimensional *vectors* and two-dimensional *matrices*.

We can create an array by calling `np.array()` and passing in data as a single value, like a list. Below is a matrix. The first axis has a length of two, and the second axis has a length of 3.

[ ]:
```
a = np.array([[1, 2, 3],
              [3, 2, 1]])
a
```

[ ]:
```
array([[1, 2, 3],
       [3, 2, 1]])
```

An `array` has an `ndim` attribute indicating the number of its axes, a `size` indicating the number of values it has, and a `shape` indicating its size in each dimension. It also has a `dtype` describing what data type all of the elements in the array are.

[ ]:
```
# number of dimensions
print(a.ndim)

# notice that the shape is rows x columns
print(a.shape)

# notice that the size is rows * columns
print(a.size)

# int32 is a numpy-provided dtype
print(a.dtype)
```

```
2
(2, 3)
6
int64
```

We can create arrays with placeholder content in several ways. This is useful when we know how many elements will be in an array, but not their values, as `numpy` arrays have fixed size. The full list is in numpy's documentation.

[ ]:
```
# create an 2x3x2 array of zeros. notice the double parentheses
np.zeros((2, 3, 2))
```

[ ]:
```
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

[ ]:
```
# create an array of ones based on the earlier a array
np.ones_like(a)
```

[ ]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

We can also create arrays by specifying a range of values through `arange()` or generating random ones through functions like `random.randint()` and `random.random()`.

[ ]:
```
# create a 1D array from 1 til 10 in steps of 2
np.arange(1, 10, 2)
```

[ ]:
```
array([1, 3, 5, 7, 9])
```

[ ]:
```
# create a 1D array from 0 to 1 in steps of 0.1
np.arange(0, 1, 0.1)
```

[ ]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

[ ]:
```
# seed is used to create a reproducible random example
np.random.seed(1)
# create a 3x4 array of random integers between 1 and 10
np.random.randint(1, 10, (3, 4))
```

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

We can repeat values and arrays to create bigger ones with `repeat()` and `tile()`.

[ ]:
```
# create a 1D array through repetition
np.repeat(10, 5)
```

[ ]:
```
array([10, 10, 10, 10, 10])
```

[ ]:
```
# create a 2D array through repetition
onedim_arr = np.array([1, 2, 3, 4, 5])
multidim_arr = np.tile(onedim_arr, (5,1))
multidim_arr
```

[ ]:
```
array([[1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5]])
```

[ ]:
```
# repeat onedim_arr in multiple directions
another_arr = np.tile(onedim_arr, (3,3))
another_arr
```

[ ]:
```
array([[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5],
       [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]])
```

3.4.1 Simulating data

Sometimes, it can be useful to simulate data to make sure an analysis works as expected. `numpy`'s `random` submodule provides support for drawing random samples from a variety of distributions. To generate random samples, we first call `default_rng()` to create a sample Generator. Then, we use the Generator's various distribution methods, like `normal()`, to create sample arrays.

[ ]:
```
# create the sample Generator
rng = np.random.default_rng(seed=42)

# create a 5x4 array w/ normally distributed data
# distribution has mean 10, standard dev. 2.5
randrng = rng.normal(10, 2.5, (5, 4))
randrng
```

[ ]:
```
array([[10.7617927 ,  7.40003973, 11.87612799, 12.35141179],
       [ 5.12241203,  6.74455123, 10.31960101,  9.20939352],
       [ 9.95799711,  7.86739018, 12.19849494, 11.94447984],
       [10.16507674, 12.81810302, 11.16877336,  7.85176884],
       [10.92187696,  7.6027935 , 12.19612575,  9.87518522]])
```

[ ]:
```
# using the same Generator to draw a random sample from a Poisson distribution
poissonrng = rng.poisson(1, (5, 6))
poissonrng
```

[ ]:
```
array([[1, 3, 1, 0, 3, 1],
       [0, 0, 1, 1, 2, 2],
       [0, 1, 0, 0, 3, 1],
       [0, 0, 1, 2, 1, 0],
       [0, 1, 1, 0, 0, 0]])
```

3.5 Reshaping arrays

We can change the shape of an array in various ways, leaving the elements the same.

[ ]:
```
# get the transpose of an array
b = np.array([[5, 4, 3, 2, 1, 0],
              [10, 8, 6, 4, 2, 0]])
b.T
```

[ ]:
```
array([[ 5, 10],
       [ 4,  8],
       [ 3,  6],
       [ 2,  4],
       [ 1,  2],
       [ 0,  0]])
```

[ ]:
```
# change dimensions
b.reshape(4, 3)
```

[ ]:
```
array([[ 5,  4,  3],
       [ 2,  1,  0],
       [10,  8,  6],
       [ 4,  2,  0]])
```

[ ]:
```
# collapse an array to one dimension
b.flatten()
```

[ ]:
```
array([ 5,  4,  3,  2,  1,  0, 10,  8,  6,  4,  2,  0])
```

If two arrays share the same size along an axis, we can stack them with `np.hstack()` and `np.vstack()`. Notice that we pass the arrays to stack as a tuple.

[ ]:
```
# stack a and b horizontally
np.hstack((b, a))
```

[ ]:
```
array([[ 5,  4,  3,  2,  1,  0,  1,  2,  3],
       [10,  8,  6,  4,  2,  0,  3,  2,  1]])
```

[ ]:
```
# reshape b before stacking vertically
np.vstack((b.reshape(4, 3), a))
```

[ ]:
```
array([[ 5,  4,  3],
       [ 2,  1,  0],
       [10,  8,  6],
       [ 4,  2,  0],
       [ 1,  2,  3],
       [ 3,  2,  1]])
```

3.6 Basic operations

`numpy` arrays let us perform vector operations, manipulating all the elements in an axis without writing loops.

[ ]:
```
arr1 = np.array([5, 10, 15, 20])
arr1
```

[ ]:
```
array([ 5, 10, 15, 20])
```

[ ]:
```
arr2 = np.arange(5, 9)
arr2
```

[ ]:
```
array([5, 6, 7, 8])
```

[ ]:
```
np.sin(np.arange(0, 360, 45) * np.pi/180)
```

[ ]:
```
array([ 0.00000000e+00,  7.07106781e-01,  1.00000000e+00,  7.07106781e-01,
        1.22464680e-16, -7.07106781e-01, -1.00000000e+00, -7.07106781e-01])
```

We can perform operations when arrays are the same length along the axis in use, or when values can be *broadcast*, or repeated, along an axis.

[ ]:
```
# arr1 and arr2 are the same length
arr1 - arr2
```

[ ]:
```
array([ 0,  4,  8, 12])
```

[ ]:
```
results = []
for i, j in zip(arr1, arr2):
    results.append(i - j)

results
```

[ ]:
```
[0, 4, 8, 12]
```

[ ]:
```
# multiple each element in arr1 by 2
arr1 * 2
```

[ ]:
```
array([10, 20, 30, 40])
```

[ ]:
```
# incompatible shapes
arr2 + np.array([1, 2])
```

```
        ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-25-f38d1fa8d077> in <module>
         1 # incompatible shapes
   ----> 2 arr2 + np.array([1, 2])


       ValueError: operands could not be broadcast together with shapes (4,)␣
↪(2,)
```

[ ]:
```
short_output = []
for i, j in zip(arr2, np.array([1, 2])):
    short_output.append(i + j)

short_output
```

[ ]:
```
[6, 8]
```

We can also summarize the values in an array.

[ ]:
```
print(f'''arr1 sums to {arr1.sum()}.
Its max value is {arr1.max()}, and its mean is {arr1.mean()}.''')
```

```
arr1 sums to 50.
Its max value is 20, and its mean is 12.5.
```

Some descriptive statistics are only `numpy` functions, and are not available as `array` methods like we saw above.

[ ]:
```
np.median(arr1)
```

[ ]:
```
12.5
```

[ ]:
```
arr1.median()
```

```
↪---------------------------------------------------------------------------

    AttributeError                            Traceback (most recent call␣
↪last)

    <ipython-input-29-5f6234b1aec9> in <module>
----> 1 arr1.median()


    AttributeError: 'numpy.ndarray' object has no attribute 'median'
```

3.7 Operations in multiple dimensions

In a multi-dimensional array like a matrix, elements of same-sized arrays will be paired up for operations, just as with vectors. If we perform an operation with a matrix and a vector of the same size in one dimension, the vector can be *broadcast* to repeat along the other dimension.

```
[ ]: tens = np.arange(0, 120, 10).reshape(3, 4)
     tens
```

```
[ ]: array([[  0,  10,  20,  30],
            [ 40,  50,  60,  70],
            [ 80,  90, 100, 110]])
```

```
[ ]: horizontal = np.array([-5, -10, -15, -20])

     tens + horizontal
```

```
[ ]: array([[-5,  0,  5, 10],
            [35, 40, 45, 50],
            [75, 80, 85, 90]])
```

```
[ ]: vertical = np.array([[100],
                          [200],
                          [300]])
     tens + vertical
```

```
[ ]: array([[100, 110, 120, 130],
            [240, 250, 260, 270],
            [380, 390, 400, 410]])
```

We can still calculate statistics for multidimensional arrays, but we must specify the axis to calculate over. To calculate values for each column, we use `axis=0`. To calculate for each row, we use `axis=1`.

[ ]:
```
tens.mean(axis=0)
```

[ ]:
```
array([40., 50., 60., 70.])
```

[ ]:
```
tens.mean(axis=1)
```

[ ]:
```
array([15., 55., 95.])
```

3.8 Indexing, slicing, and iterating

We can index and slice arrays like we would a list.

[ ]:
```
arr1
```

[ ]:
```
array([ 5, 10, 15, 20])
```

[ ]:
```
arr1[1]
```

[ ]:
```
10
```

[ ]:
```
arr1[1:3]
```

[ ]:
```
array([10, 15])
```

We can iterate over arrays as well, though vectorized `numpy` operations are preferred when possible.

[ ]:
```
for i in arr1:
    print(i)
```

```
5
10
15
20
```

Multidimensional arrays like matrices have one index per axis. We can pass in more than one index within the square brackets.

[ ]:
```
tens
```

[ ]:
```
array([[  0,  10,  20,  30],
       [ 40,  50,  60,  70],
       [ 80,  90, 100, 110]])
```

[ ]:
```
# indexing goes row, column
tens[1, 2]
```

[ ]:
```
60
```

[ ]:
```
# get the first row
tens[0]
```

[ ]:
```
array([ 0, 10, 20, 30])
```

[ ]:
```
# get the first column
tens[:,0]
```

[ ]:
```
array([ 0, 40, 80])
```

[ ]:
```
# slice rows 1-2, columns 2-3
tens[0:2, 1:3]
```

[ ]:
```
array([[10, 20],
       [50, 60]])
```

3.9 Mutations and copies

We can also update individual elements in an array. Note that any variables that refer to that array will also change, just like with mutating lists. To make an independent copy of an array, use the `.copy()` method.

[ ]:
```
# create a 3x4 array of random integers
matrix = np.random.randint(1, 11, 12).reshape(3, 4)
matrix2 = matrix

#make a copy
matrix3 = matrix.copy()

matrix
```

[ ]:
```
array([[ 5,  3,  5,  8],
       [ 8, 10,  2,  8],
       [ 1,  7, 10, 10]])
```

[ ]:
```
# replace the second row
matrix2[1] = [0, 0, 0, 0]
matrix2
```

[ ]:
```
array([[ 5,  3,  5,  8],
       [ 0,  0,  0,  0],
       [ 1,  7, 10, 10]])
```

[ ]:
```
# the original also changed
matrix
```

```
[ ]: array([[ 5,  3,  5,  8],
            [ 0,  0,  0,  0],
            [ 1,  7, 10, 10]])
```

```
[ ]: # the copy did not
     matrix3
```

```
[ ]: array([[ 5,  3,  5,  8],
            [ 8, 10,  2,  8],
            [ 1,  7, 10, 10]])
```

3.10 Logic and filtering

`numpy` arrays work with boolean expressions. Each element is checked, and the result is an array of `True`/`False` values. They resulting arrays sometimes called *masks* because they are used to mask, or filter, data.

```
[ ]: tens
```

```
[ ]: array([[  0,  10,  20,  30],
            [ 40,  50,  60,  70],
            [ 80,  90, 100, 110]])
```

```
[ ]: # evaluate whether each element is divisible by 3
     tens % 3 == 0
```

```
[ ]: array([[ True, False, False,  True],
            [False, False,  True, False],
            [False,  True, False, False]])
```

```
[ ]: # the same thing with for loops
     masked = []

     for row in tens:
         masked_row = []
         for col in row:
             masked_row.append(col % 3 == 0)
         masked.append(masked_row)

     masked
```

```
[ ]: [[True, False, False, True],
      [False, False, True, False],
      [False, True, False, False]]
```

To filter use a boolean expression as a mask, pass it into square brackets after the array to mask.

[ ]:
```
tens[tens % 3 == 0]
```

[ ]: array([0, 30, 60, 90])

[ ]:
```
# also works
mask = tens % 3 == 0
tens[mask]
```

[ ]: array([0, 30, 60, 90])

[ ]:
```
# comparison in standard python
filtered_data = []

for row in tens:
    for col in row:
        if col % 3 == 0:
            filtered_data.append(col)

filtered_data
```

[ ]: [0, 30, 60, 90]

We can even use masks to generate new arrays with conditionals. `np.where()` takes as its arguments a boolean expression, an expression to evaluate if `True`, and an expression to evaluate elsewise. This is analagous to creating a new array based on an old one with a `for` loop and `if/else` statements, but much faster.

[ ]:
```
np.where(tens % 3 == 0,  # condition
         tens,  # return the element if True
         0)  # return 0 if False
```

[ ]:
```
array([[ 0,  0,  0, 30],
       [ 0,  0, 60,  0],
       [ 0, 90,  0,  0]])
```

[ ]:
```
result = []

for row in tens:
    result_row = []
    for col in row:
        if col % 3 == 0:
            result_row.append(col)
        else:
            result_row.append(0)
    result.append(result_row)

result
```

```
[ ]: [[0, 0, 0, 30], [0, 0, 60, 0], [0, 90, 0, 0]]
```

3.11 Loading flat files to numpy arrays

We can also load data from files into `numpy` arrays. Recall the California housing csv we loaded earlier:

```
[ ]: with open('/content/data/california_housing_test.csv', 'r') as f:
         # print the first five lines
         for i in range(5):
             print(f.readline())
```

```
"longitude","latitude","housing_median_age","total_rooms","total_bedrooms","popu
lation","households","median_income","median_house_value"

-122.050000,37.370000,27.000000,3885.000000,661.000000,1537.000000,606.000000,6.
608500,344700.000000

-118.300000,34.260000,43.000000,1510.000000,310.000000,809.000000,277.000000,3.5
99000,176500.000000

-117.810000,33.780000,27.000000,3589.000000,507.000000,1484.000000,495.000000,5.
793400,270500.000000

-118.360000,33.820000,28.000000,67.000000,15.000000,49.000000,11.000000,6.135900
,330000.000000
```

We can load this data to a `numpy` array using `genfromtxt()`, which takes a path to a file as an argument. We should also include a `delimiter` argument, indicating how values are separated. Useful optional arguments include `names`, which we can use to tell `numpy` that the first row contains column names; `skip_header`, which we can use to skip the first few lines; and arguments for how to handle missing values.

There is also `loadtxt()`, which is simpler than `genfromtxt()`. The former does not have options for using header names or handling missing values. To fill missing values, we need to set the `usemask` parameter to `True`. The result will be a masked array in that case, which we can then convert to a regular array with the `filled(np.nan)` method. Note that this only works for float columns.

```
[ ]: hd = np.loadtxt('/content/data/california_housing_test.csv',
                     delimiter=',',
                     skiprows=1)
     hd.shape
```

```
[ ]: (3000, 9)
```

[ ]:
```
housing_data = np.genfromtxt('/content/data/california_housing_test.csv',
                             delimiter=',',
                             #skip_header=1,  # let's use names instead
                             names=True,
                             missing_values=500001.0,
                             usemask=True  # replace missing values
                            )

housing_data.filled(np.nan)
```

[ ]:
```
array([(-122.05, 37.37, 27., 3885., 661., 1537., 606., 6.6085, 344700.),
       (-118.3 , 34.26, 43., 1510., 310.,  809., 277., 3.599 , 176500.),
       (-117.81, 33.78, 27., 3589., 507., 1484., 495., 5.7934, 270500.),
       …,
       (-119.7 , 36.3 , 10.,  956., 201.,  693., 220., 2.2895,  62000.),
       (-117.12, 34.1 , 40.,   96.,  14.,   46.,  14., 3.2708, 162500.),
       (-119.63, 34.42, 42., 1765., 263.,  753., 260., 8.5608,     nan)],
      dtype=[('longitude', '<f8'), ('latitude', '<f8'), ('housing_median_age',
'<f8'), ('total_rooms', '<f8'), ('total_bedrooms', '<f8'), ('population',
'<f8'), ('households', '<f8'), ('median_income', '<f8'), ('median_house_value',
'<f8')])
```

If you look closely, the array we got back is not a matrix. Each row looks like a tuple with comma-separated values. If we check the **shape**, we see 3000 rows and what looks like no columns. The **dtype** attribute lists all our field names.

[ ]:
```
housing_data.shape
```

[ ]:
```
(3000,)
```

[ ]:
```
housing_data.dtype
```

[ ]:
```
dtype([('longitude', '<f8'), ('latitude', '<f8'), ('housing_median_age', '<f8'),
('total_rooms', '<f8'), ('total_bedrooms', '<f8'), ('population', '<f8'),
('households', '<f8'), ('median_income', '<f8'), ('median_house_value', '<f8')])
```

In this case, **genfromtxt** returned a *structured array*, a different type of array than the ones we have seen so far. We can refer to fields by putting the field name as a string in square brackets, similar to referencing a dictionary key. However, we will soon see a data type in another package, **pandas**, that is even better suited to working with columns in tabular data like this.

[ ]:
```
np.median(housing_data['housing_median_age'])
```

```
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:755:
UserWarning: Warning: 'partition' will ignore the 'mask' of the MaskedArray.
  a.partition(kth, axis=axis, kind=kind, order=order)
```

[ ]:
```
29.0
```

[ ]:
```python
housing_data['median_income'].mean()
```

[ ]: 3.8072717999999997

4 References

- NumPy. *Basic Numpy*. https://numpy.org/devdocs/user/quickstart.html
- NumPy. *Numpy Routines*. https://numpy.org/doc/stable/reference/routines.html