Intro. Comp. for Data Science (FMI08)

Dr. Nono Saha

April 26, 2023

Max Planck Institute for Mathematics in the Sciences
University of Leipzig/ScaDS.AI

Spring 2023

Course plan

1. `NumPy` numerics
2. `NumPy` - Advanced indexing
3. `NumPy` - Broadcasting
4. `NumPy` - Basic file I/O
5. Structure of a Data Science (ML) project
6. Homework 3

`NumPy` numerics

NumPy numerics: basic operations

All basic mathematical operators in `Python` are implemented for arrays. They are applied element-wise to the array values.

```
np.arange(3) + np.arange(3)
## array([0, 2, 4])

np.arange(3) - np.arange(3)
## array([0, 0, 0])

np.arange(3) + 2
## array([2, 3, 4])

```

```
np.arange(3) * np.arange(3)
## array([0, 1, 4])

np.arange(1,4)/np.arange(1,4)
## array([1., 1., 1.])

np.arange(3) * 3
## array([0, 3, 6])
```

```
np.full((2,2), 2) ** np.arange(4).reshape((2,2))
np.full((2,2), 2) ** np.arange(4)
## Which of the two instructions will work?

```

NumPy numerics: mathematical functions

The package provides a wide variety of basic mathematical functions that are vectorized. In general, they will be faster than their base equivalents (e.g. `np.sum()` vs `sum()`).

```
np.sum(np.arange(1000))
## 499500

np.cumsum(np.arange(10))
## array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])

np.log10(np.arange(1,11))
## array([0., 0.30103, 0.47712125, 0.60205999, 0.69897,
## 0.77815125, 0.84509804, 0.90308999, 0.95424251, 1. ])

np.median(np.arange(10))
## 4.5

```

NumPy numerics: matrix multiplication

It is supported using the `matmul()` function or the operator,

```
x = np.arange(6).reshape(3,2)
y = np.tri(2,2)
x @ y
## array([[1., 1.], [5., 3.], [9., 5.]])

y.T @ y
## array([[2., 1.], [1., 1.]])

np.matmul(x.T, x)
## array([[20, 26], [26, 35]])

y @ x
##  Can this work?

```

NumPy numerics: other linear algebra functions

The standard linear algebra functions are (mostly) implemented in the `linalg` submodule. See here for more details.

```
np.linalg.det(y)
## 1.0

np.linalg.eig(x.T @ x)
## (array([ 0.43988174, 54.56011826]), array([[-0.79911221,
    -0.6011819 ], [ 0.6011819 , -0.79911221]]))

np.linalg.inv(x.T @ x)
## array([[ 1.45833333, -1.08333333], [-1.08333333,
    0.83333333]])

np.linalg.cholesky(x.T @ x)
## array([[4.47213595, 0.],[5.81377674, 1.09544512]])

```

NumPy numerics: random values

NumPy has another submodule called random for functions used to generate random values,

To use this, you should construct a generator via `default_rng()`, with or without a seed, and then use the generator's methods to obtain your desired random values.

```
rng = np.random.default_rng(seed = 1234)
rng.random(3) # ~ Uniform [0,1)
## array([0.97669977, 0.38019574, 0.92324623])

rng.normal(0, 2, size = (2,2))
## array([[ 0.30523839,  1.72748778],[ 5.82619845,
    -2.95764672]])

rng.binomial(n=5, p=0.5, size = 10)
## array([2, 4, 2, 2, 3, 4, 4, 3, 3, 3])

```

NumPy - Advanced indexing

From last time: subsetting with tuples

Unlike lists, a ndarray can be subset by a tuple containing integers

```
x = np.arange(6)
x
## array([0, 1, 2, 3, 4, 5])

x[(0,1,3),]

## array([0, 1, 3])

x[(0,1,3)]

## Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: too many indices for array: array is 1-
dimensional, but three were indexed

```

Question

What if we use the list instead?

NumPy - Advanced indexing: exercise

Given the following matrix,

```
    x = np.arange(16).reshape((4,4))
    x
    ## array([[ 0,  1,  2,  3], [ 4,  5,  6,  7], [ 8,  9, 10,
    11], [12, 13, 14, 15]])
```

Write an expression to obtain the centre 2x2 values (i.e. 5, 6, 9, 10 as a new matrix).

NumPy - Advanced indexing: boolean indexing

Lists or ndarrays of boolean values can also be used to subset, positions with True are kept, and False are discarded.

```
    x = np.arange(6)
    ## array([0, 1, 2, 3, 4, 5])

    x[[True, False, True, False, True, False]]
    ## array([0, 2, 4])

    x[np.array([True, True, False, False, True, False])]
    ## array([0, 1, 4])

```

The utility comes from vectorized comparison operations,

```
    x > 3
    ## array([False, False, False, False,  True,  True])
    x[x>3]
    ## array([4, 5])
    x % 2 == 1
    ## array([False,  True, False,  True, False,  True])

```

NumPy - Advanced indexing: boolean operators

If we want to use a boolean operator on an array, we need to use &, |, and $\sim$ instead of `and`, `or`, `and not` respectively.

```
x = np.arange(6)
x
## array([0, 1, 2, 3, 4, 5])

y = x % 2 == 0
y
## array([ True, False,  True, False,  True, False])

~y
## array([False,  True, False,  True, False,  True])

y & (x > 3)
## array([False, False, False, False,  True, False])

y | (x > 3)
## array([ True, False,  True, False,  True,  True])

```

NumPy - meshgrid

One other useful function in NumPy is meshgrid(), which generates all possible combinations between the input vectors,

```
pts = np.arange(3)
x, y = np.meshgrid(pts, pts)
x
## array([[0, 1, 2], [0, 1, 2], [0, 1, 2]])

y
## array([[0, 0, 0], [1, 1, 1], [2, 2, 2]])

np.sqrt(x**2 + y**2)

## array([[0.        , 1.        , 2.        ],
##        [1.        , 1.41421356, 2.23606798],
##        [2.        , 2.23606798, 2.82842712]])

```

NumPy - meshgrid: exercise

We will now use this to attempt a simple brute force approach to numerical optimization, define a grid of points using `meshgrid()` to approximate the minima of the following function:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

Considering values of $x, y \in (-1, 3)$, which values of x, y minimize this function?

NumPy - Broadcasting

NumPy - Broadcasting: general broadcasting

When operating on two arrays, `NumPy` compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when

- they are equal, or
- one of them is 1

If these conditions are not met, a `ValueError`: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes.

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

x + np.array([1,2,3])

```

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

x + np.array([1,2,3])
```

NumPy - Broadcasting: mechanism

```
x = np.arange(12).reshape((4,3))
y = 1
x+y

x    (2d array): 4 x 3
y    (1d array):     1
----------------------
x+y  (2d array): 4 x 3


x = np.arange(12).reshape((4,3))
y = np.array([1,2,3])
x+y

x    (2d array): 4 x 3
y    (1d array):     3
----------------------
x+y  (2d array): 4 x 3

```

```
x = np.arange(12).reshape((3,4))
y = np.array([1,2,3])
x+y

x    (2d array): 3 x 4
y    (1d array):     3
----------------------
x+y  (2d array): Error

x = np.arange(12).reshape((3,4))
y = np.array([1,2,3]).reshape
    ((3,1))
x+y

x    (2d array): 3 x 4
y    (1d array): 3 x 1
----------------------
x+y  (2d array): 3 x 4
```

Please, check the official NumPy user guide - Broadcasting

NumPy - Broadcasting: example for data standardizing

Below we generate a data set with 3 columns of random normal values. Each column has a different mean and standard deviation which we can check with `mean()` and `std()`.

```
    rng = np.random.default_rng(1234)
    d = rng.normal(loc=[-1,0,1], scale=[1,2,3], size=(1000,3))
    d.mean(axis=0)
    ## array([-1.0294382 , -0.01396257,  1.01241784])

    d.std(axis=0)
    ## array([0.99674719, 2.03222595, 3.10625219])

```

Use broadcasting to standardize all three columns to have a mean of 0 and a standard deviation of 1.

Check the new data set using `mean()` and `std()`.

NumPy - Broadcasting: exercises

For each of the following combinations, determine what the resulting dimension will be:

- $A(128 \times 128 \times 3) + B(3)$
- $A(8 \times 1 \times 6 \times 1) + B(7 \times 1 \times 5)$
- $A(2 \times 1) + B(8 \times 4 \times 3)$
- $A(3 \times 1) + B(15 \times 3 \times 5)$
- $A(3) + B(4)$

NumPy - Basic file I/O

NumPy - Basic file I/O: reading and writing arrays

We will not spend much time on this as most data you will encounter is more likely to be in a tabular format (e.g. data frame), and tools like `Pandas` are more appropriate.

For basic saving and loading of `NumPy` arrays, there are the `save()` and `load()` functions, which use a built-in binary format.

```python
x = np.arange(1e5)
np.save("data/x.npy", x)
new_x = np.load("data/x.npy")
np.all(x == new_x)

## True

```

Additional functions for saving (`savez()`, `savez_compressed()`, `savetxt()`) exist for saving multiple arrays or saving a text representation of an array.

If you need to read delimited (CSV, tsv, etc.) data into a `NumPy` array, you can use `genfromtxt()`.