Numpy Note

1. basics

1.1 type

- numpy has its own datatypes.
  - float: `np.float16`, `np.float32`, `np.float64`
  - int: `np.int8`, `np.int16`, `np.int32`,`np.int64`
  - unsigned_int: `np.uint8`, `np.uint16`, `np.uint32`,`np.uint64`
  - complex: `np.complex64`, `np.complex128`
  - bool: `np.bool`
  - byte: `np.byte`, signed char `np.ubyte`: unsigned char
  - more conventioned name from C: `np.short`, `np.long`, `np.double` etc.
- all above datatype can be used as function to convert type `np.float32(1.0)`
- like pandas, for given `np.array.astype('str')` convert types
- `np.ndarray.view(dtype = None)`: create a new view of the same data.
  - a.view(some_dtype) or a.view(dtype=some_dtype) constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.
  - a.view(ndarray_subclass) or a.view(type=ndarray_subclass) just returns an instance of ndarray_subclass that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

1.2 arrays:

- numpy's basic object is `np.ndarray`
- Common ways to build array:
  - from built-in python array_like object, use `np.array()`
  - numpy function to create arrays:
    - `np.zeros(size, dtype)`: zero array
    - `np.arange([start,]stop,[step,], dtype)`: syntax like `range()` but create corresponding array

- `np.linspace(start,stop,num=50,endpoint= True,retsetp=False, dtype,axis=0)`: numpy version of matlab linspace
  - `start`=, `stop`=: beginning and end point
  - `num`: int, number of points
  - `endpoint`: bool, if include `stop`
  - `retsetp`: bool, if true, return a tuple `(array,step)`
  - `axis`: axis to expand result. relevant only if `start`, `stop` are ndarray themselves.
- `np.indices(dimensions, dtype, sparse)`: return indices along all axies. Specifically, if the `dimensions = (d0,d1,...,d_n-1)`, the result is a `np.ndarray` with shape `(n,d0,d1,....d_n-1)`. The first dimension determines the axis for which indices are provided.

Example: (See also stackoverflow)

```
np.indices((2, 3))
array([[[0, 0, 0],
        [1, 1, 1]], # row index
       [[0, 1, 2],  # column index
        [0, 1, 2]]])
```

- other numpy functions' result, like functions in the `random` module below.

1.3 access array

Accessor method takes an array_like object to access part of data. It can be any array_like object other than tuple, which is reserved for special use.

There is no single index for multidimensional array as in matlab.

- basic accessor is bracket `np.ndarray[]`.
  - accept single integer, can be positive/negative, `a[-1]`
  - accept multidimensional index in the same bracket, `a[2,3]` if length of number < array dims, return subarray
  - For 2d array, `a[2,3] == a[2][3]` but the former is more efficient, just like in `pandas`
  - range slice works as expected, `a[:-1]` means exclude the last element. Note that it provide new view of data but not new copy

- array accept index arrays like matlab, `ndarray[index_array]`, this returns copy of data, this is one of the advanced indexing
    - for 1d array: return a new array of mapping: `index` -> `element` the index array can be multidimensional
    - for nd array: used less. Suppose array shape is (d0,d1,...d_n-1). The syntax should be `ndarray[index_array_1,index_array_2,...]`, to select certain dimension fully, use :
        - if each index array has the same shape `x` (shape match) and there are n index arrays. The result is an array of shape `x` with mapping `ndarray[index_1,index_2,..index_n]` -> `element`
        - if the index array does not have the same shape, numpy attempts broadcasting the match the shape
        - if there are l < n index arrays (partial indexing): the omitting axis's elements are all selected. `y[np.array([0,2,4])]` selects the 1,3,5 row of the 2d array y.
- array accepts boolean array, leads to new copy of data. this is the other one of the advanced indexing
    - if boolean array has the same shape as the input array. It returns 1d array with elements of true. This is equivalent to `y[np.nonzero(bool_array)]`
    - if boolean array has less dimension than the array, The result dim are all selected. The returned array has shape [# of true element, omitted dims] Note that any selected dimension should have the same shape between input array and index bool array.

  For example, using a 2-D boolean array of shape (2,3) with four True elements to select rows from a 3-D array of shape (2,3,5) results in a 2-D result of shape (4,5). But a 2-D boolean array of shape (2,4) is invalid
- `np.newaxis` can be used to expand array. For example

```
y = np.arange(35).reshape(5,7)
y[:,np.newaxis,:].shape # 5,1,7
```

- `...` may be used to indicate selecting in full any remaining unspecified dimensions.

```
y = np.arange(30).reshape(1,1,2,3,5)
y[0,...,1,1].shape # 1,2
```

- all access methods if used in assignment changes original data.

```
>>> x = np.arange(0, 50, 10)
>>> x
array([ 0, 10, 20, 30, 40])
>>> x[np.array([1, 1, 3, 1])] += 1
>>> x
array([ 0, 11, 20, 31, 40])
```

Where people expect that the 1st location will be incremented by 3. In fact, it will only be incremented by 1. The reason is because a new array is extracted from the original (as a temporary) containing the values at 1, 1, 3, 1, then the value 1 is added to the temporary, and then the temporary is assigned back to the original array. Thus the value of the array at x[1]+1 is assigned to x[1] three times, rather than being incremented 3 times.

- tuple when supplied is not interpreted as an index array but a list of indices, i.e. indicate single element access described in the first section. `a[(1,1)] == a[1,1]`

1.4 array methods:

Shape manipulation

- `ndarray.shape`: attribute, shapes of the array
- `np.reshape(array,newshape, order = {'C','F','A'})`: reshape, return new view whenever possible.
    - `newshape` is a tuple
    - `order` specifies how ndarray is arranged in memory, row-major('C') or column major ('F')
    - also available as `ndarray.reshape()`
- `ndarray.flat`: 1d iterator over array, a `numpy.flattier` instance
- `ndarray.flatten(order = )`: collapse data into 1d array

Transpose

- `ndarray.T`: transpose of array
- `np.transpose(array, axes=)`: permute dimensions. by default, reverse dimensions.
    - `axes =`: list of int, order of axes, starting from 0
- `np.swapaxes(array, axis1,axis2)`: swap axes,

- `np.moveaxis(a,start,dest)`: move axis at `start` to `dest`, keep other axes' relative orders intact.

Change Dimension:

- `ndarray.ndims`: attribute, number of dims
- `np.expand_dims(array, axis)`: add dimension at location
- `np.squeeze(array, axis)`: remove dims with size 1
    - `axis` =: if not specified, remove all.
- `np.broadcast_to(array,shape)`: broadcast array to new shape
- `np.broadcast_arrays(array1,array2,...)`: broad cast to make all arrays match. return list of arrays

Stack and merge

- `numpy.concatenate(list of arrays,axis= 0)`: concatenate array1,array2,... along axis
- `numpy.stack(list of arrays, axis = 0)`: join arrays along a new axis

Tiling arrays

- `np.tile(array,reps)`: if `reps = [x0,x1,...,x_n-1]`repeat array in ith dimension for `x_i` times.
    - if `array.ndims < n`: prepend new axis to array to match n. So a shape (3,) array is promoted to (1, 3) for 2-D replication, or shape (1, 1, 3) for 3-D replication.
    - if `array.ndims > n`: prepend reps with 1 to match `array.ndims`.for an A of shape (2, 3, 4, 5), a reps of (2, 2) is treated as (1, 1, 2, 2).
- `np.repeat(array,repeats, axis=None)`: repeat array along a single axis.
    - if `axis = None`, use `array.flatten()`as input, return 1d array
    - `repeats` =: int, number of repetition

Add and remove elements

- `np.unique(array,...)`: find unique elements. merge `df.value_counts()`, `df.nunique` and other functions
    - `return_index` =: bool, return indices of input array that give unique value

- `return_inverse` =: bool, return indices of unique array to reconstruct original array. `unique_array[inverse_index]` = `original_array`.
- `return_counts`: counts of frequency for each unique value
- `axis`: axis to operate on, if =`None`,flatten array first. if specific axis, along that axis, flatten all subarrays.
- return tuple of `(unique_array, indice, indice_inverse, counts)`

- `np.trim_zeros(1darray,trim = {'f','b','fb'})`: trim front and back zeros of given 1 d array.
- `np.delete(array, obj, axis = )`: return a new array where along new `axis` subarrays indexed by list of integers `obj` are removed.
- `np.insert(array, obj, axis = )`: opposite of `np.delete`

2. random module

Random module is used to generate random samples from various distributions

2.1 random sampling from uniform/normal

- to generate random sample use `np.random` module
- `random.rand(d0,d1,...,dn)`: random sampling from uniform distribution of [0,1] with shape [d0, d1,...,dn]
- `random.randn(d0,d1,..,dn)`: random sampling from standard normal
- `random.randint(low =, high =, size =)`: random sampling from uniform integer distribution. Right end is excluded. [low,high)
  - `low`: int, if `high = None`, this is the maximal integer. draw from [1,low], otherwise from [low, high]
  - `high`: int, the maximum
  - `size`: list/tuple, shape of np.array
- `random.random_integers(low= ,high= ,size= )`: (deprecated) similar to `randint`, but both ends are inclusive. [low,high]

- `random.random_sample(size)`: uniform from half-open interval [0,1), similar to `rand` but accepts tuple. Equivalent to `random.random(size)`, `random.ranf(size)`, `random.sample(size)`.
- `random.choice(a= , size= , replace =, p =)`: select from 1d array `a`, with given probability `p`, allow replacement if `replace = True`
    - `a`: 1d array-like object
    - `p`: 1d array-like object, if not given, assume uniform

2.2 permutation

- `random.shuffle(x)` : shuffle array-like `x` along first axis. The modification is inplace

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

- `random.permutation(x)` : similar to `random.shuffle()`, but make a copy and return result

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

2.3 Seed control

There are two ways to work with set seed.

1. Set the global seed. Use `random.seed(int)`. `python np.random.seed(0)np.rand()np.random.seed(0)np.rand()# should be the same result`
2. Create a container `random.RandomState(seed=int)` to fix seed for this particular container. Use member methods to generate random sample. For example `RandomState(1000).randint(0,1,[10])`, all functions covered in section 2.2 are available in the object's memeber methods. `python container_A = random.RandomState(10)container_A.randn(10)container_B = random.RandomState(100)container_B.randn(100)` Useful for managing multiple seeds.

2.4 Other distributions

There are other distributions like `logistic()`, `beta()`, `binominal()` and `poisson()`. The syntaxs are similar. Consists of `(distribution_para, size)`

I list some here:

- `standard_normal(size)`
- `standard_t(size)`
- `f(dfnum =,dfden =,size)`: `dfnum` is for numerator. `dfden` is for denominator
- `lognormal(mean,sigma,size)`:
- `uniform([low,high,size])`: half-open interval [low,high), float

3 Advanced mechanism

3.1 memory layout

- An instance of class `ndarray` consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps N integers into the location of an item in the block.

3.2 view and copy

- simple selection/slicing returns view of data. indexing based on index array /boolean array returns copy of data. See example below

```
Z = np.zeros(9)
Z_view = Z[0:3] # simple selection
Z_view[...] = 1
print(Z) # [1. 1. 1. 0. 0. 0. 0. 0. 0.]

Z = np.zeros(9)
Z_copy = Z[[0,1,2]] # advanced index
Z_copy[...] = 1
print(Z) # [0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

- to check if the result is a view or copy use `.base` to return the orginal data and `is` to check

```
Z = np.random.uniform(0,1,(5,5))
Z1 = Z[:3,:]
Z2 = Z[[0,1,2], :]
```

```python
Z1.base is Z # True
Z2.base is Z # False
Z2.base is None # True
```

- See function's document for whether a view or copy is returned
  - `np.reshape()` return new view whenever possible
  - `np.flatten()` returns new copy
- In arithmetic calculations between arrays, temporary copies are implicitly made. If arrays are big, think about other possible ways. For example, use `np.add/minus/...(,out=)` the keyword `out=` specifies the output numpy array, if the intermediate result need not to be saved, we can speed up slightly via coding the intermediate result with `out` on the same `ndarray`.