# numpy

October 20, 2022

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

[2]:
```
# basics of number arrays
my_arr = np.array([1,2,3,4])
print(my_arr)
print([1,2,3,4])
print(type(my_arr))
my_arr_dtype = np.array([1,2,3,4],dtype='U')
print(my_arr_dtype)
print(type(my_arr_dtype[0]))
```

```
[1 2 3 4]
[1, 2, 3, 4]
<class 'numpy.ndarray'>
['1' '2' '3' '4']
<class 'numpy.str_'>
```

[3]:
```
# math on numpy arrays vs trying to do math on a python list
my_arr[0:2]
print(my_arr * 7)
print([1,2,3,4] * 2)
```

```
[ 7 14 21 28]
[1, 2, 3, 4, 1, 2, 3, 4]
```

[4]:
```
# basic array opperations

# still an array
print(np.array_split(my_arr, 4))

my_arr2 = np.array([5,6,7,8])

# concatenate 2 arrays
print(np.concatenate([my_arr,my_arr2]))
```

```
[array([1]), array([2]), array([3]), array([4])]
[1 2 3 4 5 6 7 8]
```

## 0.1 Random

[5]:
```python
# set a random seed so results are reproducible
np.random.seed(7)
# samples from a uniform distribution
print(np.random.rand(5,5))

# sample from an array I made
print(np.random.choice(my_arr, 8))

# sample from a normal distribution
print(np.random.normal(10, 7, 8))

# samples from an exponential dist
print(np.random.exponential(10, 8))
```

```
[[0.07630829 0.77991879 0.43840923 0.72346518 0.97798951]
 [0.53849587 0.50112046 0.07205113 0.26843898 0.4998825 ]
 [0.67923    0.80373904 0.38094113 0.06593635 0.2881456 ]
 [0.90959353 0.21338535 0.45212396 0.93120602 0.02489923]
 [0.60054892 0.9501295  0.23030288 0.54848992 0.90912837]]
[2 4 4 4 1 4 1 2]
[-0.0472817   8.90226968 -6.30190164  8.82041138 -5.7523431   7.34556866
 13.33911061 14.2000969 ]
[ 3.76869892  8.50094209  3.23031635  6.03019374  4.35375548 10.71190125
  4.6259289   6.14507878]
```

[6]:
```python
# let demonstrate the speed advantage of numpy

# create 2 large groups of numbers
size = 1000000
arr1 = np.random.rand(size)
arr2 = np.random.normal(2022,10,size)
print(type(arr1))
list1 = list(arr1)
list2 = list(arr2)
print(type(list1))

# speed of a for loop
start = time.time()
list3 = []
for i in range(len(list1)):
    list3.append(list1[i] * list2[i])
finish = time.time()
print(finish - start, 'seconds For loop math')

# speed of pre-allocated for loop
list3 = [0] * size
```

```python
start = time.time()
for i in range(len(list1)):
    list3[i] = list1[i] * list2[i]
finish = time.time()
print(finish - start, 'seconds For pre allocated math')

# list comprehension
start = time.time()
list4 = [ x * y for x,y in zip(list1,list2)]
finish = time.time()
print(finish - start, 'seconds list comprehesion')

# np arrays - this one wins!
start = time.time()
arr3 = arr1 * arr2
finish = time.time()
print(finish - start, 'seconds numpy math')
```

```
<class 'numpy.ndarray'>
<class 'list'>
0.22275495529174805 seconds For loop math
0.2063148021697998 seconds For pre allocated math
0.09261894226074219 seconds list comprehesion
0.005018949508666992 seconds numpy math
```

[7]:
```python
# check in 1 find the median of 1 million numbers samples from a beta␣
↪distribution with alpha=21, beta=6 using seed 94
np.random.seed(94)
check1_arr = np.random.beta(21,6,size)
print(np.median(check1_arr))
```

```
0.7846848917973409
```

[8]:
```python
# load data using numpy, load only the numerical columns
iris = np.loadtxt('iris.data', delimiter=',', usecols=[0,1,2,3])
print(iris.shape)
print(iris)
```

```
(150, 4)
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
```

```
[4.9 3.1 1.5 0.1]
[5.4 3.7 1.5 0.2]
[4.8 3.4 1.6 0.2]
[4.8 3.  1.4 0.1]
[4.3 3.  1.1 0.1]
[5.8 4.  1.2 0.2]
[5.7 4.4 1.5 0.4]
[5.4 3.9 1.3 0.4]
[5.1 3.5 1.4 0.3]
[5.7 3.8 1.7 0.3]
[5.1 3.8 1.5 0.3]
[5.4 3.4 1.7 0.2]
[5.1 3.7 1.5 0.4]
[4.6 3.6 1.  0.2]
[5.1 3.3 1.7 0.5]
[4.8 3.4 1.9 0.2]
[5.  3.  1.6 0.2]
[5.  3.4 1.6 0.4]
[5.2 3.5 1.5 0.2]
[5.2 3.4 1.4 0.2]
[4.7 3.2 1.6 0.2]
[4.8 3.1 1.6 0.2]
[5.4 3.4 1.5 0.4]
[5.2 4.1 1.5 0.1]
[5.5 4.2 1.4 0.2]
[4.9 3.1 1.5 0.1]
[5.  3.2 1.2 0.2]
[5.5 3.5 1.3 0.2]
[4.9 3.1 1.5 0.1]
[4.4 3.  1.3 0.2]
[5.1 3.4 1.5 0.2]
[5.  3.5 1.3 0.3]
[4.5 2.3 1.3 0.3]
[4.4 3.2 1.3 0.2]
[5.  3.5 1.6 0.6]
[5.1 3.8 1.9 0.4]
[4.8 3.  1.4 0.3]
[5.1 3.8 1.6 0.2]
[4.6 3.2 1.4 0.2]
[5.3 3.7 1.5 0.2]
[5.  3.3 1.4 0.2]
[7.  3.2 4.7 1.4]
[6.4 3.2 4.5 1.5]
[6.9 3.1 4.9 1.5]
[5.5 2.3 4.  1.3]
[6.5 2.8 4.6 1.5]
[5.7 2.8 4.5 1.3]
[6.3 3.3 4.7 1.6]
```

```
[4.9 2.4 3.3 1. ]
[6.6 2.9 4.6 1.3]
[5.2 2.7 3.9 1.4]
[5.  2.  3.5 1. ]
[5.9 3.  4.2 1.5]
[6.  2.2 4.  1. ]
[6.1 2.9 4.7 1.4]
[5.6 2.9 3.6 1.3]
[6.7 3.1 4.4 1.4]
[5.6 3.  4.5 1.5]
[5.8 2.7 4.1 1. ]
[6.2 2.2 4.5 1.5]
[5.6 2.5 3.9 1.1]
[5.9 3.2 4.8 1.8]
[6.1 2.8 4.  1.3]
[6.3 2.5 4.9 1.5]
[6.1 2.8 4.7 1.2]
[6.4 2.9 4.3 1.3]
[6.6 3.  4.4 1.4]
[6.8 2.8 4.8 1.4]
[6.7 3.  5.  1.7]
[6.  2.9 4.5 1.5]
[5.7 2.6 3.5 1. ]
[5.5 2.4 3.8 1.1]
[5.5 2.4 3.7 1. ]
[5.8 2.7 3.9 1.2]
[6.  2.7 5.1 1.6]
[5.4 3.  4.5 1.5]
[6.  3.4 4.5 1.6]
[6.7 3.1 4.7 1.5]
[6.3 2.3 4.4 1.3]
[5.6 3.  4.1 1.3]
[5.5 2.5 4.  1.3]
[5.5 2.6 4.4 1.2]
[6.1 3.  4.6 1.4]
[5.8 2.6 4.  1.2]
[5.  2.3 3.3 1. ]
[5.6 2.7 4.2 1.3]
[5.7 3.  4.2 1.2]
[5.7 2.9 4.2 1.3]
[6.2 2.9 4.3 1.3]
[5.1 2.5 3.  1.1]
[5.7 2.8 4.1 1.3]
[6.3 3.3 6.  2.5]
[5.8 2.7 5.1 1.9]
[7.1 3.  5.9 2.1]
[6.3 2.9 5.6 1.8]
[6.5 3.  5.8 2.2]
```

```
 [7.6 3.  6.6 2.1]
 [4.9 2.5 4.5 1.7]
 [7.3 2.9 6.3 1.8]
 [6.7 2.5 5.8 1.8]
 [7.2 3.6 6.1 2.5]
 [6.5 3.2 5.1 2. ]
 [6.4 2.7 5.3 1.9]
 [6.8 3.  5.5 2.1]
 [5.7 2.5 5.  2. ]
 [5.8 2.8 5.1 2.4]
 [6.4 3.2 5.3 2.3]
 [6.5 3.  5.5 1.8]
 [7.7 3.8 6.7 2.2]
 [7.7 2.6 6.9 2.3]
 [6.  2.2 5.  1.5]
 [6.9 3.2 5.7 2.3]
 [5.6 2.8 4.9 2. ]
 [7.7 2.8 6.7 2. ]
 [6.3 2.7 4.9 1.8]
 [6.7 3.3 5.7 2.1]
 [7.2 3.2 6.  1.8]
 [6.2 2.8 4.8 1.8]
 [6.1 3.  4.9 1.8]
 [6.4 2.8 5.6 2.1]
 [7.2 3.  5.8 1.6]
 [7.4 2.8 6.1 1.9]
 [7.9 3.8 6.4 2. ]
 [6.4 2.8 5.6 2.2]
 [6.3 2.8 5.1 1.5]
 [6.1 2.6 5.6 1.4]
 [7.7 3.  6.1 2.3]
 [6.3 3.4 5.6 2.4]
 [6.4 3.1 5.5 1.8]
 [6.  3.  4.8 1.8]
 [6.9 3.1 5.4 2.1]
 [6.7 3.1 5.6 2.4]
 [6.9 3.1 5.1 2.3]
 [5.8 2.7 5.1 1.9]
 [6.8 3.2 5.9 2.3]
 [6.7 3.3 5.7 2.5]
 [6.7 3.  5.2 2.3]
 [6.3 2.5 5.  1.9]
 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]
```

[9]:
```
# print out the mean of each column
sepal_width = 0
sepal_length = 1
petal_width = 2
petal_length = 3

print(iris[:,sepal_width].mean(),'mean sepal_width')
print(iris[:,sepal_length].mean(),'mean sepal_length')
print(iris[:,petal_width].mean(),'mean petal_width')
print(iris[:,petal_length].mean(),'mean petal_length')
```

```
5.843333333333334 mean sepal_width
3.0540000000000003 mean sepal_length
3.758666666666666 mean petal_width
1.1986666666666668 mean petal_length
```

[10]:
```
# NumPy is great for math but bad when it comes to categorical data, pandas␣
↪works well with both
import pandas as pd
```

[11]:
```
# load the whole data with pandas
iris_df = pd.read_csv('iris.data',sep=',',header=None)
# name the columns for ease of use
iris_df.columns =␣
↪['sepal_width','sepal_length','petal_width','petal_length','name']
print(iris_df.shape)
print(iris_df)
print(iris_df['sepal_length'][5:10])
```

```
(150, 5)
     sepal_width  sepal_length  petal_width  petal_length            name
0            5.1           3.5          1.4           0.2     Iris-setosa
1            4.9           3.0          1.4           0.2     Iris-setosa
2            4.7           3.2          1.3           0.2     Iris-setosa
3            4.6           3.1          1.5           0.2     Iris-setosa
4            5.0           3.6          1.4           0.2     Iris-setosa
..           …             …            …             …               …
145          6.7           3.0          5.2           2.3  Iris-virginica
146          6.3           2.5          5.0           1.9  Iris-virginica
147          6.5           3.0          5.2           2.0  Iris-virginica
148          6.2           3.4          5.4           2.3  Iris-virginica
149          5.9           3.0          5.1           1.8  Iris-virginica

[150 rows x 5 columns]
5    3.9
6    3.4
7    3.4
```

```
8    2.9
9    3.1
Name: sepal_length, dtype: float64
```

[12]:

```
# how to save a dataframe
np.savetxt('np_iris.csv',iris,delimiter=',')
iris_df.to_csv('pd_iris.tsv',sep='\t')
```