python™

Programming with Python for Data Science

Unit Topics

- NumPy
- Pandas

Learning objectives

- Describe the fundamental NumPy data types and the available ways for manipulating them.
- Know the basic characteristics of Pandas and get useful information about the data contained in a csv file

NumPy

- NumPy is a Linear Algebra Library for Python. It is so important because almost all of the libraries in the PyData ecosystem rely on NumPy as one of their main building block

- If you have installed Anaconda you have also NumPy already installed, otherwise you can install it from the command prompt:

  pip install numpy

Array

- NumPy arrays are the main way we will use NumPy
- NumPy arrays essentially come in two flavors: vectors and matrices.
- Vectors are strictly 1D arrays and matrices 2D arrays (but a matrix can still have only one row or one column).
- The number of dimensions is the ***rank*** of the array; the ***shape*** of an array is a tuple of integers giving the size of the array along each dimension.

Arrays from Python objects

```
import numpy as np
my_list = [1,2,3]
arr=np.array(my_list)      #cast
print(arr)
[1 2 3]
arr
array([1,2,3])

my_mat=[[1,2,3],[4,5,6],[7,8,9]]
mat=np.array(my_mat)
mat
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
print(mat)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

There is also a matrix data type, but the documentation recommends against use of matrix. The array has the transpose method (mat.T)

List vs Array

```
import numpy as np
L = [1,2,3]
A=np.array(L)          #cast
for item in L:
        print(item)
for item in A:    #the SAME
        print(item)

L.append(4)
L
[1,2,3,4]
A.append(4)
AttributeError: 'numpy.ndarray' object has no attribute 'append'

L = L + [5]
A = A + [4,5]
ValueError: operands could not be broadcast together with shapes (3,) (2,)

L2 = []
for e in L:
    L2.append( e + e )
A + A

# + sign     with list -> concatenation
#            with array -> vector (or matrix) elementwise addition
```

List vs Array

```
import numpy as np
L = [1,2,3]
A=np.array(L)          #cast
2*L                    #you must use a for loop or a map!
[1,2,3,1,2,3]
2*A
[2,4,6]

L**2              #use a for loop!
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'

A**2
array([1, 4, 9])

np.sqrt(A)
array([ 1.        ,  1.41421356,  1.73205081])

np.log(A)
array([ 0.        ,  0.69314718,  1.09861229])

np.exp(A)
array([  2.71828183,   7.3890561 ,  20.08553692])

# So a numpy array behaves like a vector
```

Dot (inner) product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

```
a = np.array([1,2])
b = np.array([2,1])
dot=0
for e, f in zip(a,b):
    dot += e*f
print(dot)
4

a*b                #sizes must be the same
array([2, 2])
np.sum(a*b)
4
(a*b).sum()
4

np.dot(a,b)
4
a.dot(b)
4
```

Dot (inner) product

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos(\theta)$$

```
a*b                #sizes must be the same
array([2, 2])
np.sum(a*b)
4

amag = np.sqrt((a*a).sum())
2.2360679774997898
amag = np.linalg.norm(a)

cosangle = a.dot(b) / (np.linalg.norm(a) * np.linalg.norm(b))
angle = np.arccos(cosangle)
print(angle)
0.6435011087932847
```

Code vectorization

```
import numpy as np
from datetime import datetime
a = np.random.randn(100)
b = np.random.randn(100)
T = 100000

def slow_dot_product(a,b):
    result = 0
    for e, f in zip(a,b):
        result += e*f
    return result

t0 = datetime.now()
for t in range(T):
    slow_dot_product(a,b)
dt1 = datetime.now()  - t0

t0 = datetime.now()
for t in range(T):
    a.dot(b)
dt2 = datetime.now()  - t0

print("dt1 / dt2:", dt1.total_seconds() / dt2.total_seconds())
```

dt1 / dt2: 28.53931436691435

Code vectorization

```
import numpy as np
import time                          # for measuring CPU time
a = np.linspace(0, 1, 1E+07)         # create some array
t0 = time.clock()

b = 3*a -1
t1 = time.clock()                    # t1-t0 is the CPU time of 3*a-1

for i in range(a.size):
    b[i] = 3*a[i] - 1
t2 = time.clock()

print('3*a-1: %g sec, loop: %g sec' % (t1-t0, t2-t1))

3*a-1: 0.050999 sec, loop: 5.08418 sec
```

Built-in arrays creation

```
>>> import numpy as np
>>> np.arange(0,12,2)                  #START, STOP, STEP
array([ 0, 2, 4, 6, 8, 10])

>>> np.zeros(3)
array([ 0., 0., 0.])
>>> np.zeros((2,3))                    #ROWS,COLUMNS
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])

>>> np.ones(4)
array([ 1., 1., 1., 1.])
>>> np.ones((2,3))
array([[ 1., 1., 1.],
       [ 1., 1., 1.]])

>>> np.full(10,5.)
array([ 5., 5., 5., 5., 5., 5., 5., 5., 5., 5.])

>>> np.linspace(0,5,10)                #evenly spaced numbers
array([ 0. , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
        2.77777778, 3.33333333, 3.88888889, 4.44444444, 5. ])
>>> np.linspace(0,5,100)               #START,STOP, POINTS
```

Built-in arrays creation

```
>>> import numpy as np
>>> np.eye(4)                    #identity matrix
array([[ 1., 0., 0., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.]])

>>> np.random.rand(5)            #random sample from uniform distribution from
                                 #0 to 1
array([ 0.44587486, 0.04230894, 0.04093282, 0.32328359, 0.74113171])
>>> np.random.rand(2,2)
array([[ 0.17902702, 0.47347058],
       [ 0.72107014, 0.89560959]])


>>> np.random.randn(5)           #random sample from GAUSSIAN distribution
                                 #centered around 0 with variance 1
array([-0.12815159, -0.39932564, 1.57836327, 0.97788626, -0.93179639])
>>> np.random.randn(2,2)
array([[ 1.07054136, 0.2550509 ],
       [ 0.13805842, -0.17739226]])
>>> np.random.randint(1,100,10)    #LOW, HIGH (excluded),NUMBER OF INTEGERS
array([94, 24, 22, 57, 76, 60, 66, 36, 25, 75])
```

Arrays methods and attributes

```
>>> arr=np.arange(25)
>>> arr
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24])
>>> ranarr=np.random.randint(0,50,10)
>>> ranarr
array([44, 23, 23, 46, 18, 21, 6, 36, 49, 7])

>>> arr.arange(25).reshape(5,5)
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> arr.dtype
dtype('int32')
>>> arr.shape
(5,5)

>>> arr.reshape(5,10)
ValueError: total size of new array must be unchanged

>>> ranarr.max()              #also min()
49
>>> ranarr.argmax()           #also argmin()
8
```

Arrays indexing

Numpy arrays can be sliced, with the same syntax used with the list.

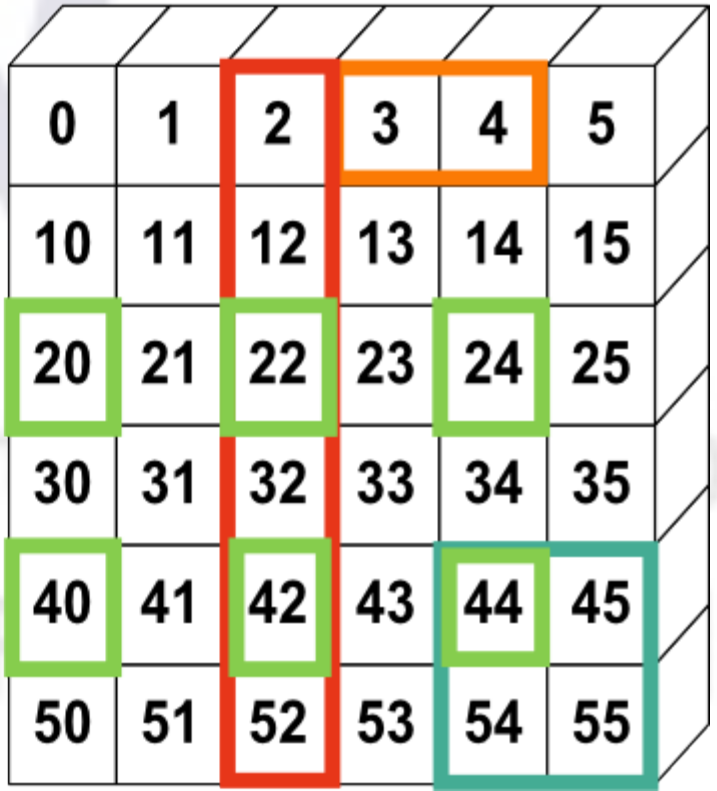
However a slice of an array is a VIEW into the data, so **if you modify a slice you're actually modifying the original data**, unlike what happens with the lists. You can copy an array with the method `copy()`

Since arrays may be multidimensional, you must specify a slice, or an index in general, for each dimension of the array.

```
>>> arr=np.arange(25)
>>> arr[8]                    #Get a value at an index
8
>>> arr[1:5]                  #Get values in a range
array([1, 2, 3, 4])

>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> b = a[:2, 1:3]
>>> print(a[0, 1])
2
>>> b[0, 0] = 77
>>> print(a[0, 1])
77
```

```
>>> a = np.array([[1,2], [3, 4], [5, 6]])
>>> print(a[[0, 1, 2], [0, 1, 0]])
[1 4 5]
>> print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
[1 4 5]

>>> print(a[[0, 0], [1, 1]])
[2 2]
>>> print(np.array([a[0, 1], a[0, 1]]))
[2 2]
```

```
>>> a[0,3:5]
array([3,4])


>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



Matrix multiplication

Requirement: inner dimensions must match! Remember that:

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

So the (I,j)th element of the product is the dot product of row A(I,:) and column B(:,j). You can use the dot function:

AB = A.dot(B)

If you want to do a elementwise multiplication, you can use the * operator; both arrays must have same size:

$(AB)_{i,j} = A_{i,j} * B_{i,j}$

```
A=np.array([[1,2],[3,4]])
a = np.array([1,2])
b = np.array([3,4])

Ainv = np.linalg.inv(A)   #inverse
Ainv.dot(A)               #check of the Identity Matrix
np.linalg.det(A)          #determinant
-2
np.diag(A)                #diagonal elements
array([1,4])
np.diag([1,4])
array([[1, 0],
       [0, 4]])
np.outer(a,b)             #Outer (C_ij = a_i b_j) product
array([[3, 4],
       [6, 8]])
np.inner(a,b)             #Inner (C = sum_i(a_i b_i)) product
11
np.diag(A).sum()          #Trace
5
np.trace(A)

arr=np.arange(0,5)
arr/arr
array([ nan, 1., 1., 1., 1.]) # nan stands for Not A Number,
1/arr
array([ inf, 1. , 0.5 , 0.33333333, 0.25 ])
```

Eigenvalues and Eigenvectors

```
X = np.random.randn(100,3)      #(100 "samples" of 3 "features"

cov = np.cov(X)                 #covariance
cov.shape                       #wrong shape…
(100,100)
cov = np.cov(X.T)               #(or cov = np.cov(X, rowvar=False))

eigenvalues, eigenvectors = np.linalg.eig(cov)

eigenvalues, eigenvectors = np.linalg.eigh(cov)

#eigh is for simmetric or hermitian (self-adjoint) matrices
#Symmetric   A = A^T
#Hermitian   A = A^H
#                   where A^H = conjugate transpose of A

#Solving a linear sistem
x = np.linalg.solve(A, b)
```

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.

```
arr=np.arange(25)
arr[0:5]=100                    #Setting a value with index range

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v                  # Add v to each row of x using broadcasting
array([[ 2,  2,  4],
       [ 5,  5,  7],
       [ 8,  8, 10],
       [11, 11, 13]])
```

y = x + v works even though x has shape (4, 3) and v has shape (3,) due to broadcasting; this line works as if v actually had shape (4, 3), where each row of the matrix v was a copy of v, and the sum was performed elementwise
`(y[i, :] = x[i, :] + v)`

Selection

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
>>> arr=np.arange(11)

>>> arr > 5
array([False, False, False, False, False, False, True, True, True, True,
True], dtype=bool)

>>> arr[arr>3]
array([ 4, 5, 6, 7, 8, 9, 10])

>>> a = np.array([[1,2], [3, 4], [5, 6]])
>>> a.shape
(3,2)
>>> b = a[a>2]
>>> b.shape
(4,)
```