Data Science Foundation
Lesson #3 - NumPy & Pandas

Ivanovitch Silva

August, 2017

Previously on last class

Lists, dictionaries, modules, functions, set, mission values, enumerate, list comprehension

Agenda

- Motivation
- Introduction to NumPy
- Creating arrays
- Inspecting data
- Array comparison
- Computing with NumPy
- Pros & Cons

Update the repository

```
git clone https://github.com/ivanovitchm/EEC2006.git
```

Ou ….

```
git pull
```

Motivation

```
areas = ["hallway", 11.25, "kitchen", 18.0, "living room", 20.0]
```

Python lists offer a few advantages when representing data:

- lists can contain mixed types
- lists can shrink and grow dynamically

On the other hand ...

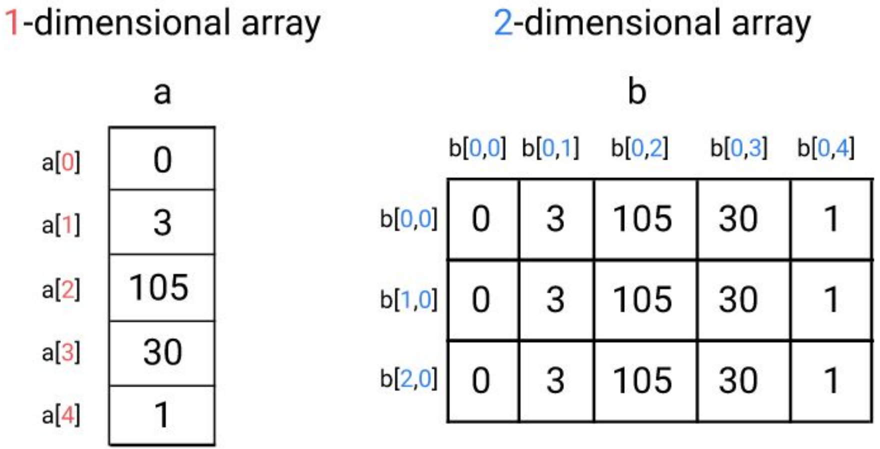
Using Python lists to represent and work with data also has a few key disadvantages:

- to support their flexibility, lists tend to consume lots of memory
- they struggle to work with medium and larger sized datasets

NumPy

- NumPy is the fundamental package for scientific computing with Python.
- It is a library that combines the flexibility and ease-of-use of Python with the speed of C

Creating arrays




```python
import numpy as np
a = np.array([0,3,105,30,1])
b = np.array([[0,3,105,30,1],[0,3,105,30,1],[0,3,105,30,1]])
```

Array shape

```
vector = np.array([1, 2, 3, 4])
print(vector.shape) #output: (4,)
matrix = np.array([[5, 10, 15], [20, 25, 30]])
print(matrix.shape) #output:  (2, 3)
```

Using NumPy

```
import numpy
data = numpy.genfromtxt("data.csv", delimiter=",")
```

World Health Organization

http://apps.who.int/gho/data/view.main.52160

"world_alcohol.csv'

Here's what each column represents:

- Year -- the year the data in the row is for.
- WHO Region -- the region in which the country is located.
- Country -- the country the data is for.
- Beverage Types -- the type of beverage the data is for.
- Display Value -- the number of liters, on average, of the beverage type a citizen of the country drank in the given year.

Inspecting data

world_alcohol

Header

```
array([[             nan,              nan,              nan,
                     nan,              nan],
       [  1.98600000e+03,              nan,              nan,
                     nan,   0.00000000e+00],
       [  1.98600000e+03,              nan,              nan,
                     nan,   5.00000000e-01],
       ...,
       [  1.98600000e+03,              nan,              nan,
                     nan,   2.54000000e+00],
       [  1.98700000e+03,              nan,              nan,
                     nan,   0.00000000e+00],
       [  1.98600000e+03,              nan,              nan,
                     nan,   5.15000000e+00]])
```

String

When NumPy can't convert a value to a numeric data type like float or integer, it uses a special nan value that stands for "not a number"

Reading the data correctly

```
world_alcohol = np.genfromtxt("world_alcohol.csv", delimiter=",", dtype="U75", skip_head
er=1)
[['1986' 'Western Pacific' 'Viet Nam' 'Wine' '0']
 ['1986' 'Americas' 'Uruguay' 'Other' '0.5']
 ['1985' 'Africa' "Cte d'Ivoire" 'Wine' '1.62']
 ...,
 ['1986' 'Europe' 'Switzerland' 'Spirits' '2.54']
 ['1987' 'Western Pacific' 'Papua New Guinea' 'Other' '0']
 ['1986' 'Africa' 'Swaziland' 'Other' '5.15']]
```

Arrays comparisons

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

```
array([False,  True, False, False], dtype=bool)
```

```
matrix = np.array([[5, 10, 15],
                   [20, 25, 30],
                   [35, 40, 45]]
                 )
matrix == 25
```

```
array([[False, False, False],
       [False,  True, False],
       [False, False, False]], dtype=bool)
```

Computing with NumPy

- sum()
- mean()
- median()
- max()
- min()

Executa funções sobre uma determinada dimensão do array

Computing with NumPy

```
vector = np.array([5, 10, 15, 20])
vector.sum()
```

```
50
```

```
matrix = np.array([
                   [5, 10, 15],
                   [20, 25, 30],
                   [35, 40, 45]
                ])
matrix.sum(axis=1)
```

```
array([ 30,  75, 120])
```

https://goo.gl/0eWPy6

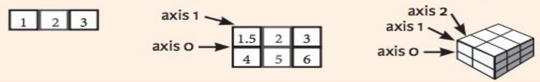
NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

`>>> np.zeros((3,4))`	Create an array of zeros
`>>> np.ones((2,3,4),dtype=np.int16)`	Create an array of ones
`>>> d = np.arange(10,25,5)`	Create an array of evenly spaced values (step value)
`>>> np.linspace(0,2,9)`	Create an array of evenly spaced values (number of samples)
`>>> e = np.full((2,2),7)`	Create a constant array
`>>> f = np.eye(2)`	Create a 2X2 identity matrix
`>>> np.random.random((2,2))`	Create an array with random values
`>>> np.empty((3,2))`	Create an empty array

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

`>>> np.int64`	Signed 64-bit integer types
`>>> np.float32`	Standard double-precision floating point
`>>> np.complex`	Complex numbers represented by 128 floats
`>>> np.bool`	Boolean type storing `TRUE` and `FALSE` values
`>>> np.object`	Python object type
`>>> np.string_`	Fixed-length string type
`>>> np.unicode_`	Fixed-length unicode type

Inspecting Your Array

`>>> a.shape`	Array dimensions
`>>> len(a)`	Length of array
`>>> b.ndim`	Number of array dimensions
`>>> e.size`	Number of array elements
`>>> b.dtype`	Data type of array elements
`>>> b.dtype.name`	Name of data type
`>>> b.astype(int)`	Convert an array to a different type

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

`>>> g = a - b`
`array([[-0.5, 0. , 0. ],`
`[-3. , -3. , -3. ]])`
	Subtraction

`>>> np.subtract(a,b)`	Subtraction

`>>> b + a`
`array([[ 2.5, 4. , 6. ],`
`[ 5. , 7. , 9. ]])`
	Addition

`>>> np.add(b,a)`	Addition

`>>> a / b`
`array([[ 0.66666667, 1. , 1. ],`
`[ 0.25 , 0.4 , 0.5 ]])`
	Division

`>>> np.divide(a,b)`	Division

`>>> a * b`
`array([[ 1.5, 4. , 9. ],`
`[ 4. , 10. , 18. ]])`
	Multiplication

`>>> np.multiply(a,b)`	Multiplication
`>>> np.exp(b)`	Exponentiation
`>>> np.sqrt(b)`	Square root
`>>> np.sin(a)`	Print sines of an array
`>>> np.cos(b)`	Element-wise cosine
`>>> np.log(a)`	Element-wise natural logarithm

`>>> e.dot(f)`
`array([[ 7., 7.],`
`[ 7., 7.]])`
	Dot product

Comparison

`>>> a == b`
`array([[False, True, True],`
`[False, False, False]], dtype=bool)`
	Element-wise comparison

`>>> a < 2` `array([True, False, False], dtype=bool)`	Element-wise comparison
`>>> np.array_equal(a, b)`	Array-wise comparison

Aggregate Functions

`>>> a.sum()`	Array-wise sum
`>>> a.min()`	Array-wise minimum value
`>>> b.max(axis=0)`	Maximum value of an array row
`>>> b.cumsum(axis=1)`	Cumulative sum of the elements
`>>> a.mean()`	Mean
`>>> b.median()`	Median
`>>> a.corrcoef()`	Correlation coefficient
`>>> np.std(b)`	Standard deviation

Copying Arrays

`>>> h = a.view()`	Create a view of the array with the same data
`>>> np.copy(a)`	Create a copy of the array
`>>> h = a.copy()`	Create a deep copy of the array

Sorting Arrays

`>>> a.sort()`	Sort an array
`>>> c.sort(axis=0)`	Sort the elements of an array's axis

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

`>>> a[2]` `3`		Select the element at the 2nd index
`>>> b[1,2]` `6.0`		Select the element at row 0 column 2 (equivalent to `b[1][2]`)

Slicing

`>>> a[0:2]` `array([1, 2])`		Select items at index 0 and 1
`>>> b[0:2,1]` `array([ 2., 5.])`		Select items at rows 0 and 1 in column 1
`>>> b[:1]` `array([[1.5, 2., 3.]])`		Select all items at row 0 (equivalent to `b[0:1, :]`)

`>>> c[1,...]`
`array([[[ 3., 2., 1.],`
`[ 4., 5., 6.]]])`
		Same as `[1,:,:]`

`>>> a[ : :-1]` `array([3, 2, 1])`		Reversed array `a`

Boolean Indexing

`>>> a[a<2]` `array([1])`		Select elements from `a` less than 2

Fancy Indexing

`>>> b[[1, 0, 1, 0], [0, 1, 2, 0]]` `array([ 4. , 2. , 6. , 1.5])`	Select elements `(1,0)`, `(0,1)`, `(1,2)` and `(0,0)`

`>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]`
`array([[ 4. ,5. , 6. , 4. ],`
`[ 1.5, 2. , 3. , 1.5],`
`[ 4. , 5. , 6. , 4. ],`
`[ 1.5, 2. , 3. , 1.5]])`
	Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

`>>> i = np.transpose(b)`	Permute array dimensions
`>>> i.T`	Permute array dimensions

Changing Array Shape

`>>> b.ravel()`	Flatten the array
`>>> g.reshape(3,-2)`	Reshape, but don't change data

Adding/Removing Elements

`>>> h.resize((2,6))`	Return a new array with shape (2,6)
`>>> np.append(h,g)`	Append items to an array
`>>> np.insert(a, 1, 5)`	Insert items in an array
`>>> np.delete(a,[1])`	Delete items from an array

Combining Arrays

`>>> np.concatenate((a,d),axis=0)` `array([ 1, 2, 3, 10, 15, 20])`	Concatenate arrays

`>>> np.vstack((a,b))`
`array([[ 1. , 2. , 3. ],`
`[ 1.5, 2. , 3. ],`
`[ 4. , 5. , 6. ]])`
	Stack arrays vertically (row-wise)

`>>> np.r_[e,f]`	Stack arrays vertically (row-wise)

`>>> np.hstack((e,f))`
`array([[ 7., 7., 1., 0.],`
`[ 7., 7., 0., 1.]])`
	Stack arrays horizontally (column-wise)

`>>> np.column_stack((a,d))`
`array([[ 1, 10],`
`[ 2, 15],`
`[ 3, 20]])`
	Create stacked column-wise arrays

`>>> np.c_[a,d]`	Create stacked column-wise arrays

Splitting Arrays

`>>> np.hsplit(a,3)` `[array([1]),array([2]),array([3])]`	Split the array horizontally at the 3rd index

`>>> np.vsplit(c,2)`
`[array([[[ 1.5, 2. , 1. ],`
`[ 4. , 5. , 6. ]]]),`
`array([[[ 3., 2., 3.],`
`[ 4., 5., 6.]]])]`
	Split the array vertically at the 2nd index

NumPy strengths and weaknesses

Strengths

- It's easy to perform computations on data.
- Data indexing and slicing is faster and easier.
- We can convert data types quickly

Weaknesses

- All of the items in an array must have the same data type.
- Columns and rows must be referred to by number

NumPy.ipynb

Agenda

- Introduction to Pandas
- Reading a CSV file
- Lendo CSV
- Exploring the dataframe
- Indexing
- Series vs Dataframe
- Selecting rows and columns
- Data manipulation

Motivation

- Pandas is a library that **unifies the most common workflows** that data analysts and data scientists previously relied on many different libraries for.
- To represent tabular data, pandas uses a custom data structure called a **dataframe**
- One of the biggest advantages that pandas has over NumPy is the ability to **store mixed data types** in rows and columns.
- Pandas dataframes can also **handle missing values** gracefully

Introduction to the data

USDA National Nutrient Database for Standard Reference

NDB_No	Shrt_Desc	Water_(g)	Energy_Kcal	Protein_(g)	Lipid_Tot_(g)	Ash_(g)	Carbohydrt_(g)	Fiber_TD_(g)
1001	BUTTER WITH SALT	15.87	717	0.85	81.11	2.11	0.06	0.0
1002	BUTTER WHIPPED WITH SALT	15.87	717	0.85	81.11	2.11	0.06	0.0
1003	BUTTER OIL ANHYDROUS	0.24	876	0.28	99.48	0.00	0.00	0.0
1004	CHEESE BLUE	42.41	353	21.40	28.74	5.11	2.34	0.0
1005	CHEESE BRICK	41.11	371	23.24	29.68	3.18	2.79	0.0

Read a CSV file

```
import pandas as pd
food_info = pd.read_csv("food_info.csv")
```

Exploring the dataframe

```
food_info.head()
```

	NDB_No	Shrt_Desc	Water_(g)	Energ_Kcal	Protein_(g)	Lipid_Tot_(g)	Ash_(g)	Carbohydrt_(g)	Fiber_TD_(g)	Sugar_Tot_(g)
0	1001	BUTTER WITH SALT	15.87	717	0.85	81.11	2.11	0.06	0.0	0.06
1	1002	BUTTER WHIPPED WITH SALT	15.87	717	0.85	81.11	2.11	0.06	0.0	0.06
2	1003	BUTTER OIL ANHYDROUS	0.24	876	0.28	99.48	0.00	0.00	0.0	0.00
3	1004	CHEESE BLUE	42.41	353	21.40	28.74	5.11	2.34	0.0	0.50
4	1005	CHEESE BRICK	41.11	371	23.24	29.68	3.18	2.79	0.0	0.51

Which columns?

```
print(food_info.columns)
```

```
Index(['NDB_No', 'Shrt_Desc', 'Water_(g)', 'Energ_Kcal', 'Protein_(g)',
       'Lipid_Tot_(g)', 'Ash_(g)', 'Carbohydrt_(g)', 'Fiber_TD_(g)',
       'Sugar_Tot_(g)', 'Calcium_(mg)', 'Iron_(mg)', 'Magnesium_(mg)',
       'Phosphorus_(mg)', 'Potassium_(mg)', 'Sodium_(mg)', 'Zinc_(mg)',
       'Copper_(mg)', 'Manganese_(mg)', 'Selenium_(mcg)', 'Vit_C_(mg)',
       'Thiamin_(mg)', 'Riboflavin_(mg)', 'Niacin_(mg)', 'Vit_B6_(mg)',
       'Vit_B12_(mcg)', 'Vit_A_IU', 'Vit_A_RAE', 'Vit_E_(mg)', 'Vit_D_mcg',
       'Vit_D_IU', 'Vit_K_(mcg)', 'FA_Sat_(g)', 'FA_Mono_(g)', 'FA_Poly_(g)',
       'Cholestrl_(mg)'],
      dtype='object')
```

Shape?

```
# Returns the tuple (8618,36) and assigns to `dimensions`.
dimensions = food_info.shape
# The number of rows, 8618.
num_rows = dimensions[0]
# The number of columns, 36.
num_cols = dimensions[1]
```

Indexing

column labels
(column index)

row
labels
(row
index)

	NDB_No	Shrt_Desc	Water_(g)	Energy_Kcal	Protein_(g)
0					
1					
2					

Series vs Dataframe

Dataframe

NDB_No	Shrt_Desc	Water_(g)	Energy_Kcal	Protein_(g)	...
1001	BUTTER WITH SALT	15.87	717	0.85	...

Series

```
NDB_No                      1001
Shrt_Desc       BUTTER WITH SALT
Water_(g)                  15.87
Energ_Kcal                   717
Protein_(g)                 0.85
  ...                       ...
```

- Series = collection of values
- Dataframe = collection of series

Selectiong a row

```
# Series object representing the row at index 0.
food_info.loc[0]
# Series object representing the seventh row.
food_info.loc[6]
# Will throw an error: "KeyError: 'the label [8620] is not in the [index]'"
food_info.loc[8620]
```

Selecting multiple rows

```
# DataFrame containing the rows at index 3, 4, 5, and 6 returned.
food_info.loc[3:6]
# DataFrame containing the rows at index 2, 5, and 10 returned. Either of th
e following work.
# Method 1
two_five_ten = [2,5,10]
food_info.loc[two_five_ten]
# Method 2
food_info.loc[[2,5,10]]
```

Selectiong a column

```
# Series object representing the "NDB_No" column.
ndb_col = food_info["NDB_No"]
# You can instead access a column by passing in a string variable.
col_name = "NDB_No"
ndb_col = food_info[col_name]
```

Selectiong multiples columns

```
columns = ["Zinc_(mg)", "Copper_(mg)"]
zinc_copper = food_info[columns]
# Skipping the assignment.
zinc_copper = food_info[["Zinc_(mg)", "Copper_(mg)"]]
```

Data manipulation with Pandas

$$Score = 2 \times (Protein\_(g)) - 0.75 \times (Lipid\_Tot\_(g))$$

NDB_No	Shrt_Desc	Water_(g)	Energy_Kcal	Protein_(g)	Lipid_Tot_(g)	Ash_(g)	Carbohydrt_(g)	Fiber_TD_(g)
1001	BUTTER WITH SALT	15.87	717	0.85	81.11	2.11	0.06	0.0
1002	BUTTER WHIPPED WITH SALT	15.87	717	0.85	81.11	2.11	0.06	0.0
1003	BUTTER OIL ANHYDROUS	0.24	876	0.28	99.48	0.00	0.00	0.0
1004	CHEESE BLUE	42.41	353	21.40	28.74	5.11	2.34	0.0
1005	CHEESE BRICK	41.11	371	23.24	29.68	3.18	2.79	0.0

Performing math with colum

```
# Adds 100 to each value in the column and returns a Series object.
add_100 = food_info["Iron_(mg)"] + 100
# Subtracts 100 from each value in the column and returns a Series object.
sub_100 = food_info["Iron_(mg)"] - 100
# Multiplies each value in the column by 2 and returns a Series object.
mult_2 = food_info["Iron_(mg)"]*2
```

Performing math with multiples columns

```
water_energy = food_info["Water_(g)"] x food_info["Energ_Kcal"]
```

water_energy		food_info["Water_(g)"]		food_info["Energ_Kcal"]
11378.79	=	15.87	x	717
11378.79	=	15.87	x	717
210.24	=	0.24	x	876
14970.73	=	42.41	x	353
15251.81	=	41.11	x	371
...		...		...

Normalize columns in a dataset

```python
# The largest value in the "Energ_Kcal" column.
max_calories = food_info["Energ_Kcal"].max()
# Divide the values in "Energ_Kcal" by the largest value.
normalized_calories = food_info["Energ_Kcal"] / max_calories
```

Create a new column

```
iron_grams = food_info["Iron_(mg)"] / 1000
food_info["Iron_(g)"] = iron_grams
```

Sorting a dataframe by a column

```
# Sorts the DataFrame in-place, rather than returning a new DataFrame.
food_info.sort_values("Sodium_(mg)", inplace=True)
# Sorts by descending order, rather than ascending.
food_info.sort_values("Sodium_(mg)", inplace=True, ascending=False)
```

Pandas.ipynb