Data Structures in Python

October 2, 2017
What is a data structure?

• Way to store data and have some method to retrieve and manipulate it

• Lots of examples in python:
  • List, dict, tuple, set, string
  • Array
  • Series, DataFrame

• Some of these are “built-in” (meaning you can just use them), others are contained within other python packages, like numpy and pandas
Basic Python Data Structures (built-in)

• List, dict, tuple, set, string

• Each of these can be accessed in a variety of ways

• Decision on which to use? Depends on what sort of features you need (easy indexing, immutability, etc)
  • Mutable vs immutable
    • Mutable – can change
    • Immutable – doesn’t change

```python
x = something # immutable type
print x
func(x)
print x # prints the same thing
```

```
x = something # mutable type
print x
print x
func(x)
print x # might print something different
```
Basic Structure: List

• Very versatile, can have items of different types, **is mutable**

• To create: use square brackets [] to contain comma separated values

• Example: `>> l = ['a', 'b', 123]`
  • `>> l`
    `['a', 'b', 123]`

• To get values out: `>> l[1]` (use index, starts with 0)
  `>> b`

• We saw these back in lab 3
Basic Structure: Set

• Set is an unordered collection with no duplicate values, **is mutable**

• Create using {}

• Example: >> s = {1, 2, 3}
  • >> s
    set([1,2,3])

• Useful for eliminating duplicate values from a list, doing operations like intersection, difference, union
Basic Structure: Tuple

- Tuple holds values separated by commas, are **immutable**
- Create using , or () to create empty
- Example: `>> t = 1,2,3`
  - `>> t`
    - (1,2,3)
  - `>> type(t)`
    - type ‘tuple’

- Useful when storing data that does not change, when needing to optimize performance of code (python knows how much memory needed)
Basic Structure: Dict

- Represented by key:value pair
  - Keys: can be any immutable type and unique
  - Values: can be any type (mutable or immutable)

- To create: use curly braces {} or dict() and list both key and value

```
>>> letters = {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> type(letters)
<type 'dict'>
```

- To access data in dictionary, call by the key
  - `>>> letters[2]`
    - 'b'

- Have useful methods like keys(), values(), iteritems(), itervalues() useful for accessing dictionary entries

- Useful when:
  - Need association between key:value pair
  - Need to quickly look up data based on a defined key
  - Values are modified
Array: Use NumPy!

• What is an array?
  • “list of lists”
  • Similar to Matlab in some ways
    • Create a 2x3 array
      • \([ 1 \ 2 \ 3; 4 \ 5 \ 6 ]\) : matlab
      • np.array([[1.,2.,3.],[4.,5.,6.]])

>>> import numpy as np
>>> y = np.array([[[1.,2.,3.],[4.,5.,6.]]])

• What is NumPy?
  • Numerical Python
  • Python library very useful for scientific computing

• How to access NumPy?
  • Need to import it into your python workspace or into your script

  • >> import numpy as np
Why use a NumPy array?

• What is it?
  • “multidimensional array of objects of all the same type”

• More compact for than list (don’t need to store both value and type like in a list)

• Reading/writing faster with NumPy

• Get a lot of vector and matrix operations
  • Can’t do “vectorized” operations on list (like element-wise addition, multiplication)

• Can also do the standard stuff, like indexing, comparisons, logical operations
Creating NumPy Arrays

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
```

Creating NumPy array and checking if each element is > 3

```python
>>> b = np.array([[1, 2, 3], [4, 5, 6]]) # Create a rank 2 array
>>> print(b.shape) # Prints "(2, 3)"
(2, 3)
>>> print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 4"
(1, 2, 4)
```

Create NumPy array, print out array dimensions, and use indexing tools

```python
>>> c = np.zeros((2, 2))
>>> print(c)
[[ 0.  0.]
 [ 0.  0.]]
```

Create 2x2 NumPy array with just zeros
More Creating NumPy Arrays

• `arange`: like “`range`”, returns an `ndarray`

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

• Use `reshape` to define/change shape of array

```python
>>> b = np.arange(12).reshape(4,3)
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```
Operations with NumPy Arrays

• Arithmetic operations (e.g. +, -, *, /, **) with scalars and between equal-size arrays – done element by element
  • A new array is created with the result

```python
>>> b = np.arange(12).reshape(4,3)
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> c = b + 5
>>> print(c)
[[ 5  6  7]
 [ 8  9 10]
 [11 12 13]
 [14 15 16]]
```

• Universal functions (for example: sin, cos, exp) also operate elementwise on an array, new array results
Be careful: * vs dot

- * is product operator, operates elementwise in NumPy arrays

```python
>>> A = np.array([[1,1],
                 ...  [0,1]] )
>>> B = np.array([[2,0],
                 ...  [3,4]] )
>>> A*B
array([[2, 0],
       [0, 4]])
>>> np.dot(A,B)
array([[5, 4],
       [3, 4]])
```

A*B – elementwise multiplication

.dot – matrix product
Other Useful NumPy Array Operations

• Sum, min, max: can be used to get values for all elements in array

```python
>>> a = np.random.random((2,3))
>>> a
array([[ 0.30541447,  0.64099062,  0.05487081],
       [ 0.9990191 ,  0.05537393,  0.38775904]])
```

Get sum of all elements in array, also min and max within array

```python
>>> a.sum()
2.4434279566031463
>>> a.min()
0.05487080789064569
>>> a.max()
0.9990190872534389
```

• Can use (axis=#) to specify certain rows and columns

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

Sum of each column (axis=0)

```python
>>> b.sum(axis=0)
array([12, 15, 18, 21])
```

Min of each row (axis = 1)

```python
>>> b.min(axis=1)
array([0, 4, 8])
```

Cumulative sum along each row

```python
>>> b.cumsum(axis=1)
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```
Indexing with NumPy Arrays

• 1D arrays (just like lists)

```python
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([  8,  27,  64])
```

Create array using arange
Pull out element at position 3
Pull out elements in positions starting at 3, before 6

• Multidimensional arrays: work with an index per axis

Element at row 3, column 4
Each row in 2\textsuperscript{nd} column
Each row in 2\textsuperscript{nd} column
Each column in 2\textsuperscript{nd} and 3\textsuperscript{rd} row

```python
>>> b
array([[  0,   1,   2,   3],
       [ 10,  11,  12,  13],
       [ 20,  21,  22,  23],
       [ 30,  31,  32,  33],
       [ 40,  41,  42,  43]])
>>> b[2,3]
23
>>> b[0:5, 1]
array([ 1, 11, 21, 31, 41])
>>> b[:,1]
array([ 1, 11, 21, 31, 41])
>>> b[1:3, :]
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```
What is pandas?

- Open source package with user friendly data structures and data analysis tools for Python
  - Built on top of NumPy, gives more tools

- Very useful for tabular data in columns (i.e. spreadsheets), time series data, matrix data, etc

- Two main data structures:
  - Series (1-dimensional)
  - DataFrame (2-dimensional)

- How to access:
  - Need to import it into your python workspace or into your script

  >> import pandas as pd
Pandas: Series

• Effectively a 1-D NumPy array with an index
• 1D labeled array that can hold any data type, with labels known as the “index”

```python
>>> import pandas as pd
>>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
>>> s
a   -0.896461
b   -0.268122
c   -1.097631
d   -2.069645
e   -0.289530
dtype: float64
```

data can be an array, scalar, or a dict

```python
>>> s = {'a': 2., 'b': 4., 'c': 6., 'd': 8.}
>>> pd.Series(s)
a   2.0
b   4.0
c   6.0
d   8.0
dtype: float64
```
Pandas: Series

• Can using slicing to grab out values

```python
>>> s = {'a': 2., 'b': 4., 'c': 6., 'd': 8.}
>>> pd.Series(s)
a  2.0
b  4.0
c  6.0
d  8.0
dtype: float64
```

• Can also use index to grab out values

```python
>>> sd = pd.Series(s)
>>> sd[1]
4.0

>>> sd['b']
4.0
```

```python
>>> sd
a  2.0
b  4.0
c  6.0
d  8.0
dtype: float64
>>> type(sd)
<class 'pandas.core.series.Series'>
```
Pandas: DataFrame

• Most commonly used pandas object

• DataFrame is basically a table made up of named columns of series
  • Think spreadsheet or table of some kind
  • Can take data from
    • Dict of 1D arrays, lists, dicts, Series
    • 2D numpy array
    • Series
    • Another DataFrame
  • Can also define index (row labels) and columns (column labels)
  • Series can be dynamically added to or removed from the DataFrame
Creating DataFrames

- From dict of Series or dicts:

```python
>>> d = {'one': pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
    ...
    'two': pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
>>> df = pd.DataFrame(d)
```

Have 2 series (one and two)

New DataFrame (df) is union of the 2 Series indices

Output includes row labels (index) and column labels as specified

Note the NaN reported because of no 4\textsuperscript{th} value in “one”

Using arrays/lists is similar:

```python
>>> d = {'one': [1., 2., 3., 4.],
    ...
    'two': [4., 3., 2., 1.]}  
```

If no index is given, index will be range(n)
where n is array length
Accessing DataFrame Info

```python
>>> df
  one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

>>> pd.DataFrame(d, index=['d', 'b', 'a'])
  one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

>>> pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
  two  three
a  1.0  NaN
b  2.0  NaN
c  3.0  NaN
d  4.0  NaN

>>> df['two']
a  1.0
b  2.0
c  3.0
d  4.0
Name: two, dtype: float64
```

Can access specific rows
Can access specific rows and columns
Grab specific column from existing DataFrame
Accessing DataFrame Info

Grab specific column from existing DataFrame

```
[>>> df['two']]  
a 1.0  
b 2.0  
c 3.0  
d 4.0  
Name: two, dtype: float64
```

Make a new column through operations on others

```
[>>> df['three'] = df['one'] * df['two']]  
[>>> df]  
one  two  three
a  1.0  1.0  1.0  
b  2.0  2.0  4.0  
c  3.0  3.0  9.0  
d  NaN  4.0  NaN
```

Get rid of columns

```
[>>> del df['two']]  
[>>> df]  
one  three
a  1.0  1.0  
b  2.0  4.0  
c  3.0  9.0  
d  NaN  NaN
```

The basics of indexing are as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select column</td>
<td>df[col]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by label</td>
<td>df.loc[label]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by integer location</td>
<td>df.iloc[loc]</td>
<td>Series</td>
</tr>
<tr>
<td>Slice rows</td>
<td>df[5:10]</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Select rows by boolean vector</td>
<td>df[bool_vec]</td>
<td>DataFrame</td>
</tr>
</tbody>
</table>
Working with DataFrames

```python
>>> df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
>>> df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
>>> df
   A         B         C         D
0  0.925605 -0.101996 -0.856505  0.041030
1  0.683239 -0.666745 -0.746350 -0.765129
2 -1.149823  0.256815 -1.844288 -0.182581
3  1.283709 -0.421631 -0.439489  1.059241
4  0.072631  0.604832  0.033367 -0.484844
5  1.429966 -0.863785  1.076024 -2.696585
6 -1.175667  0.911034  0.156114  0.054385
7 -1.452214  1.480580 -1.615911  0.298720
8 -1.630938  0.176167  0.269268 -0.822671
9  1.782930 -0.029016 -0.268635  0.623897
```

Add the dataframes together

Note elementwise addition, with the result having the union of row and column labels, even if you don’t have values in each position

```python
>>> df + df2
   A         B         C         D
0  1.092601 -0.791951 -1.112835  NaN
1  1.107048 -1.329862 -1.895610  NaN
2 -2.904277  1.115999 -2.264664  NaN
3  1.617319 -0.971522  0.781058  NaN
4 -0.704926  1.176177  0.298219  NaN
5  2.522225 -1.283977  1.387164  NaN
6 -0.138694  0.523581 -0.342428  NaN
7   NaN      NaN      NaN      NaN
8   NaN      NaN      NaN      NaN
9   NaN      NaN      NaN      NaN
```

Lots of NumPy elementwise functions work on DataFrames, as do operations like transpose (.T), .dot
Other cool things to do with DataFrames

```python
>>> df.describe()

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>10.00000</td>
<td>10.00000</td>
<td>10.00000</td>
<td>10.00000</td>
</tr>
<tr>
<td>mean</td>
<td>0.062418</td>
<td>0.134626</td>
<td>-0.423640</td>
<td>-0.287454</td>
</tr>
<tr>
<td>std</td>
<td>1.318020</td>
<td>0.721369</td>
<td>0.883781</td>
<td>1.031773</td>
</tr>
<tr>
<td>min</td>
<td>-1.630938</td>
<td>-0.863785</td>
<td>-1.844288</td>
<td>-2.696585</td>
</tr>
<tr>
<td>25%</td>
<td>-1.169206</td>
<td>-0.341722</td>
<td>-0.828966</td>
<td>-0.695058</td>
</tr>
<tr>
<td>50%</td>
<td>0.305304</td>
<td>0.073575</td>
<td>-0.354062</td>
<td>-0.070775</td>
</tr>
<tr>
<td>75%</td>
<td>1.194183</td>
<td>0.517827</td>
<td>0.125428</td>
<td>0.237637</td>
</tr>
<tr>
<td>max</td>
<td>1.782930</td>
<td>1.480580</td>
<td>1.076024</td>
<td>1.059241</td>
</tr>
</tbody>
</table>
```

Basic statistics

```python
>>> df

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.925605</td>
<td>-0.101996</td>
<td>-0.856505</td>
<td>0.041030</td>
</tr>
<tr>
<td>1</td>
<td>0.683239</td>
<td>-0.666745</td>
<td>-0.746358</td>
<td>-0.765129</td>
</tr>
<tr>
<td>2</td>
<td>-1.149823</td>
<td>0.256815</td>
<td>-1.844288</td>
<td>-0.182581</td>
</tr>
<tr>
<td>3</td>
<td>1.283709</td>
<td>-0.421631</td>
<td>-0.439489</td>
<td>1.059241</td>
</tr>
<tr>
<td>4</td>
<td>-0.072631</td>
<td>0.604832</td>
<td>0.033367</td>
<td>-0.484844</td>
</tr>
<tr>
<td>5</td>
<td>1.429966</td>
<td>-0.863785</td>
<td>1.076024</td>
<td>-2.696585</td>
</tr>
<tr>
<td>6</td>
<td>-1.175667</td>
<td>0.911034</td>
<td>0.156114</td>
<td>0.054385</td>
</tr>
<tr>
<td>7</td>
<td>-1.452214</td>
<td>1.480580</td>
<td>-1.615911</td>
<td>0.298720</td>
</tr>
<tr>
<td>8</td>
<td>-1.630938</td>
<td>0.176167</td>
<td>0.269268</td>
<td>-0.822671</td>
</tr>
<tr>
<td>9</td>
<td>1.782930</td>
<td>-0.029016</td>
<td>-0.268635</td>
<td>0.623897</td>
</tr>
</tbody>
</table>

>>> df.sort_values(by='B')

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.429966</td>
<td>-0.863785</td>
<td>1.076024</td>
<td>-2.696585</td>
</tr>
<tr>
<td>1</td>
<td>0.683239</td>
<td>-0.666745</td>
<td>-0.746358</td>
<td>-0.765129</td>
</tr>
<tr>
<td>3</td>
<td>1.283709</td>
<td>-0.421631</td>
<td>-0.439489</td>
<td>1.059241</td>
</tr>
<tr>
<td>0</td>
<td>0.925605</td>
<td>-0.101996</td>
<td>-0.856505</td>
<td>0.041030</td>
</tr>
<tr>
<td>9</td>
<td>1.782930</td>
<td>-0.029016</td>
<td>-0.268635</td>
<td>0.623897</td>
</tr>
<tr>
<td>8</td>
<td>-1.630938</td>
<td>0.176167</td>
<td>0.269268</td>
<td>-0.822671</td>
</tr>
<tr>
<td>2</td>
<td>-1.149823</td>
<td>0.256815</td>
<td>-1.844288</td>
<td>-0.182581</td>
</tr>
<tr>
<td>4</td>
<td>-0.072631</td>
<td>0.604832</td>
<td>0.033367</td>
<td>-0.484844</td>
</tr>
<tr>
<td>6</td>
<td>-1.175667</td>
<td>0.911034</td>
<td>0.156114</td>
<td>0.054385</td>
</tr>
<tr>
<td>7</td>
<td>-1.452214</td>
<td>1.480580</td>
<td>-1.615911</td>
<td>0.298720</td>
</tr>
</tbody>
</table>
```

sorting
**Other cool things to do with DataFrames**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.925605</td>
<td>-0.101996</td>
<td>-0.856505</td>
<td>0.041830</td>
</tr>
<tr>
<td>1</td>
<td>0.683239</td>
<td>-0.666745</td>
<td>-0.746350</td>
<td>-0.765129</td>
</tr>
<tr>
<td>2</td>
<td>1.283709</td>
<td>-0.421631</td>
<td>-0.439489</td>
<td>1.059241</td>
</tr>
<tr>
<td>3</td>
<td>1.429966</td>
<td>-0.863785</td>
<td>1.076024</td>
<td>-2.696585</td>
</tr>
<tr>
<td>4</td>
<td>1.782930</td>
<td>-0.029016</td>
<td>-0.268635</td>
<td>0.623897</td>
</tr>
</tbody>
</table>

Grabbing data that meet a certain condition

```
>>> df[df.A > 0]
```

Add a new column at end of dataframe

```
>>> df2 = df.copy()
>>> df2['E'] = ['one', 'one', 'two', 'three', 'four', 'five', 'four', 'one', 'two', 'four']
>>> df2
```

Filtering data to grab only data that contains certain values using .isin

```
>>> df2[df2['E'].isin([ 'two', 'four' ])]
```

```python
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-1.149823</td>
<td>0.256815</td>
<td>-1.844288</td>
<td>-0.182581</td>
<td>two</td>
</tr>
<tr>
<td>4</td>
<td>-0.072631</td>
<td>0.604832</td>
<td>0.033367</td>
<td>-0.484844</td>
<td>four</td>
</tr>
<tr>
<td>6</td>
<td>-1.175667</td>
<td>0.911034</td>
<td>0.156114</td>
<td>0.054385</td>
<td>four</td>
</tr>
<tr>
<td>7</td>
<td>-1.452214</td>
<td>1.480580</td>
<td>-1.615911</td>
<td>0.298720</td>
<td>one</td>
</tr>
<tr>
<td>8</td>
<td>-1.630938</td>
<td>0.176167</td>
<td>0.269268</td>
<td>-0.822671</td>
<td>two</td>
</tr>
<tr>
<td>9</td>
<td>1.782930</td>
<td>-0.029016</td>
<td>-0.268635</td>
<td>0.623897</td>
<td>four</td>
</tr>
</tbody>
</table>
DataFrames: groupby

- This allows you to split up data into groups based on some criteria, apply some function, and get a result.

Using “groupby” to select rows that contain same value in E, then sum those values.
Plotting Data in Series

```python
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))
>>> ts = ts.cumsum()
>>> ts.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0x10f74d690>
>>> plt.show()
```

Created a series of 1000 random numbers, with an index of dates starting at 1/1/2000

Plotted the cumulative sum of those random numbers
Plotting Data in DataFrames

Using `df.plot()` with DataFrames will plot all of the columns with labels.
Next up:

• Lab today – working with data structures

• Next week: how to get data into and out of python (I/O topics)