EML4930/EML6934: Lecture 13

Pandas - Python Data Analysis Library

Charles Jekel

November 30, 2017

Format of final

Sample Questions posted online.

Final will be 20 questions similar to the sample questions.

Monday December 12, 7:30 am - 9:30 am

Questions you will definitely be asked

- Differences between Python 2 and Python 3
- Anything about the Python syntax
- Loops in Python
- Functions in Python
- NumPy operations (arrays and multiplication)
- Matplotlib plotting
- Concept questions related to your understanding

Pandas - Python Data Analysis Library

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

`https://pandas.pydata.org/`

What does pandas solve?

> *Python has long been great for data munging and preparation, but less so for data analysis and modeling. pandas helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R.*

Useful resources on learning pandas

Python for Data Analysis, 2nd Edition Data Wrangling with Pandas, NumPy, and IPython, Wes McKinney
`http://shop.oreilly.com/product/0636920050896.do`

`https://github.com/wesm/pydata-book`

Python Data Science Handbook, Essential Tools for Working with Data, Jake VanderPlas
`http://shop.oreilly.com/product/0636920034919.do`

`https://github.com/jakevdp/PythonDataScienceHandbook`

`https://github.com/jvns/pandas-cookbook`

pandas consists of the following elements

- A set of labeled array data structures, the primary of which are Series and DataFrame
- Index objects enabling both simple axis indexing and multi-level / hierarchical axis indexing
- An integrated group by engine for aggregating and transforming data sets
- Date range generation (date_range) and custom date offsets enabling the implementation of customized frequencies
- Input/Output tools: loading tabular data from flat files (CSV, delimited, Excel 2003), and saving and loading pandas objects from the fast and efficient PyTables/HDF5 format.
- Memory-efficient "sparse" versions of the standard data structures for storing data that is mostly missing or mostly constant (some fixed value)
- Moving window statistics (rolling mean, rolling standard deviation, etc.)

pandas import

```python
import pandas as pd
# pd is the standard pandas alias
```

pandas objects

Object	Description
pd.Series	One-dimensional ndarray with axis labels.
pd.DataFrame	Two-dimensional size-mutable, row and column data.
pd.Index	Immutable ndarray for sorting axis labels.

pandas Series

```
# create arbitrary series from list
data = pd.Series([0.2, 0.4, 0.6, 0.27])

print(data)

# The Series will contain attributes named values and index
# data.values contains the numpy array
print(data.values)

# data.index contains the pandas Index object for the Series
print(data.index)

# you can slice pandas Series just like a numpy array
print(data[1])
print(data[0:2])

# so why Series over numpy array?
```

Explicit index definition for Series

```
# create arbitrary series from list
data = pd.Series([0.2, 0.4, 0.6, 0.27], index=['a','b','c','d'])

# data.index will now contain the letters a-d
print(data.index)

# you can still slice pandas Series just like a numpy array
print(data[0:2])

# or you can access the data with the index specific keys
print(data['a'])
print(data['b'])
# so this is kind of like a specialized dictionary...
# Dictionary: arbitrary keys -> set of arbitrary values
# Series: typed keys -> set of typed value
# Essentially Series are more efficient than Dictionaries
```

Building a series from dictionary

```
my_dict = {'Germany': 'sauerkraut', 'Spain': 'paella',
 'Italy': 'pizza', 'USA': 'Hamburger'}
my_series = pd.Series(my_dict)

# unlike dictionaries you can access a Series with slicing
print(my_series['Spain':'USA'])
```

So about this immutable Index object...

- thought of as immutable array and ordered multiset
- immutable = cannot be modified
- combination of Python set and 1D numpy array

Creating an Index object

```
# create arbitrary Index
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])

# You can access an index like you would a numpy array
print(indA[1:3])

# You can also use Pythons builtin set notation
print(indA & indB) # intersection
print(indA | indB) # union
print(indA ^ indB) # symmetric difference
```

What is a pandas DataFrame?

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass index (row labels) and columns (column labels) arguments.

and more:
`https://pandas.pydata.org/pandas-docs/stable/dsintro.html`

Creating a DataFrame

```
population_dict = {'California': 38332521,'Texas': 26448193,
 'New York': 19651127, 'Florida': 19552860,
 'Illinois': 12882135}
area_dict = {'California': 423967, 'Texas': 695662,
'New York': 141297, 'Florida': 170312, 'Illinois': 149995}
# let's first create a series from these dictionaries
population = pd.Series(population_dict)
area = pd.Series(area_dict)

# create a DataFrame from these two Series
states = pd.DataFrame({'population': population, 'area': area})
print(states)

# DataFrame will have index and values attributes
print(states.index) # pandas Index object
print(states.values) # Two dimensional numpy array
```

Access the 'rows' and 'columns'

```python
# you can think of the index as the rows of the table
print(states.index)

# to access a row you need to use .loc
print(states.loc['Florida'])

# and can find the columns by
print(states.columns)

# you can access the area column by
print(states['area'])
```

Ways to create DataFrame

- From Series object
- From list of dictionaries
- From dictionary of Series objects
- From two-dimensional numpy array
- From numpy structured array

Examples of creating DataFrames

```
# DataFrames will automatically fill missing values with NaNs
# integers are automatically used as the index (like Series)
in : pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
out:
   a    b   c
0  1.0  2  NaN
1  NaN  3  4.0

# You can feed a 2D numpy array, and manually pass the
# columns and index (rows) names
in : pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],
                  index=['a', 'b', 'c'])
out:
   foo       bar
a  0.610023  0.239564
b  0.141317  0.315237
c  0.221186  0.316919
```

Pandas uses int64 and float64 by default!

NumPy floats will be float64 by default, however NumPy integers will be int32 if small...

```
a = np.array((1,2,12,121))
print(a.dtype) # this will print int32!

b = np.array((2165156165151,15165,1561121261))
print(b.type) # this will print int64

# however looking at our DataFrame
print(states.dtypes)
# we'll see int64
```

DataFrame loc vs iloc

Recall how I said you can access the row of a DataFrame with loc?

```
mydf = pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],
                    index=['a', 'b', 'c'])

# Use .loc to access the explicit index
print(mydf.loc['b'])

# Use .iloc to access the implicit Python-style index
print(mydf.iloc[1])
```

Creating a new column in DataFrame

```
# recall our state data
states = pd.DataFrame({'population': population, 'area': area})

# we can make a new population density column by accessing the
# DataFrame like a dictionary
states = pd.DataFrame({'population': population, 'area': area})

print(states)

# NOTE:
# Even with Python 2 density will automatically be float64
# this happens by default in pandas
```

More ways to manipulate DataFrames

```
# swapping rows for columns using .T
print( states.T )

# accessing a row of the numpy array
print( states.values[0] )

# accessing the first three rows, and first two columns
print( states.iloc[:3, :2] )
```

Accessing data frames with masking

```
# you can select just the states that have a density > 100
in : states.loc[ states['density'] > 100 ]
out:
             area  population     density
Florida    170312    19552860  114.806121
New York   141297    19651127  139.076746

# masking and selection of columns
in : states.loc[ states['density'] > 100,['area', 'population']]
out:
             area  population
Florida    170312    19552860
New York   141297    19651127
```

How to add a row to a DataFrame

```
# let's say we have the following list which corresponds to the
# population, and density of Colorado
co_data = [269601, 5540545, 20.5509]

# We'll create a new Series from this list using the columns as
# and giving a name to the series as Colorado
new = pd.Series(co_data, index=states.columns, name='Colorado')

# you need to set states = states.append! as states.append won't
# DataFrame!
states = states.append(new)

# you'll you a new Row named Colorado
print( states )
```

How to remove duplicates from a DataFrame

```
                area  population     density
California  423967.0  38332521.0   90.413926
Florida     170312.0  19552860.0  114.806121
Illinois    149995.0  12882135.0   85.883763
New York    141297.0  19651127.0  139.076746
Texas       695662.0  26448193.0   38.018740
Colorado    269601.0   5540545.0   20.550900
Colorado    269601.0   5540545.0   20.550900

# This is an easy fix! just run drop_duplicates()
states = states.drop_duplicates()

# This will remove the duplicate Colorado row!
```

How to add a column to a DataFrame

```
# let's add the electoral votes to the DataFrame
# First let's create a dictionary of what we want to add
votes = {'California': 55, 'Colorado': 9, 'Florida': 29,
 'Illinois': 20, 'New York': 29, 'Texas': 38}
# Now we'll use the assign function to add a column named
# electoral to the DataFrame
states = states.assign(electoral=votes)

# we now have:
                area  population     density  electoral
California  423967.0  38332521.0   90.413926         55
Florida     170312.0  19552860.0  114.806121         29
Illinois    149995.0  12882135.0   85.883763         20
New York    141297.0  19651127.0  139.076746         29
Texas       695662.0  26448193.0   38.018740         38
Colorado    269601.0   5540545.0   20.550900          9
```

Other useful DataFrame functions

Function	Description
head	Returns the first *n* rows (default 5)
tail	Returns the last *n* rows (default 5)
describe	Statistic summary of the data
group_by	Group by a series or column
plot	DataFrame plotting method
value_counts	Count the number of occurrences in a series
replace	replace(oldvalue, newvalue) in a DataFrame
dropna	remove all rows from DataFrame that contain a NaN value

So where will I most likely use DataFrames?

- If you have spreadsheet data (csv, xls, ...)
- SQL data
- Various other databases...

To read and write CSV files.

How to open a CSV file as DataFrame

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html This has more options than any other Python function I've ever seen...

```
pd.read_csv(filepath_or_buffer, sep=', ', delimiter=None,
header='infer', names=None, index_col=None, usecols=None,
 squeeze=False, prefix=None, mangle_dupe_cols=True,
 dtype=None, engine=None, converters=None, true_values=None,
 false_values=None, skipinitialspace=False, skiprows=None,
 nrows=None, na_values=None, keep_default_na=True,
 na_filter=True, verbose=False, skip_blank_lines=True,
 parse_dates=False, infer_datetime_format=False,
 keep_date_col=False, date_parser=None, dayfirst=False,
 iterator=False, chunksize=None, compression='infer',
 thousands=None, decimal=b'.', lineterminator=None,
 quotechar='"', quoting=0, escapechar=None, comment=None,
 encoding=None, dialect=None, tupleize_cols=None,
 error_bad_lines=True, warn_bad_lines=True, skipfooter=0,
```

Basic pd.read_csv

```
# TSLA is the stock data for Tesla
# create a DataFrame named tsla from TSLA.csv
tsla = pd.read_csv('TSLA.csv')

# pandas will automatically create an Index
print(tsla.head())
```

```
         Date       Open   High        Low      Close  Adj Close
0  2010-06-29  19.000000  25.00  17.540001  23.889999  23.889999
1  2010-06-30  25.790001  30.42  23.299999  23.830000  23.830000
2  2010-07-01  25.000000  25.92  20.270000  21.959999  21.959999
3  2010-07-02  23.000000  23.10  18.709999  19.200001  19.200001
4  2010-07-06  20.000000  20.00  15.830000  16.110001  16.110001
```

You can explicitly load the csv stating the index column

```python
# create a DataFrame named tsla from TSLA.csv
# let's use Date as the index of the DataFrame
tsla = pd.read_csv('TSLA.csv', index_col='Date')

print(tsla.head())
```

```
                 Open   High        Low      Close  Adj Close
Date
2010-06-29  19.000000  25.00  17.540001  23.889999  23.889999  1
2010-06-30  25.790001  30.42  23.299999  23.830000  23.830000  1
2010-07-01  25.000000  25.92  20.270000  21.959999  21.959999
2010-07-02  23.000000  23.10  18.709999  19.200001  19.200001
2010-07-06  20.000000  20.00  15.830000  16.110001  16.110001
```

Sometimes null or na strings will mess up the data type

```
# so let's say our csv file has a bunch of 'null' strings
# that are messing up our analysis...
             Open   High        Low      Close  Adj Close    Vol
Date
2010-06-29   null  25.00  17.540001  23.889999  23.889999  18766
2010-06-30  25.79  30.42  23.299999  23.830000  23.830000  17187

# The Open column will have an Object data type because it has
# strings and floats, one way to get rid of this is to load the
# this will setup all 'null' strings to be a np.NaN (a float)
tsla =pd.read_csv('TSLA.csv',index_col='Date',na_values='null')

print(tsla.head())
             Open   High        Low      Close  Adj Close    Vol
Date
2010-06-29    NaN  25.00  17.540001  23.889999  23.889999  18766
2010-06-30  25.79  30.42  23.299999  23.830000  23.830000  17187
```

Saving the csv file

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html

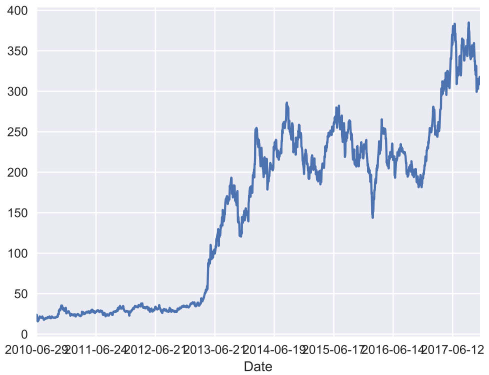
```
# so let's create a new column based on the other columns
tsla['avg'] = .25*(tsla['Open'] + tsla['High'] \
              + tsla['Low'] + tsla['Close'])

# we can create a new csv by running
tsla.to_csv('my_new.csv')
```

Plotting data with from a DataFrame

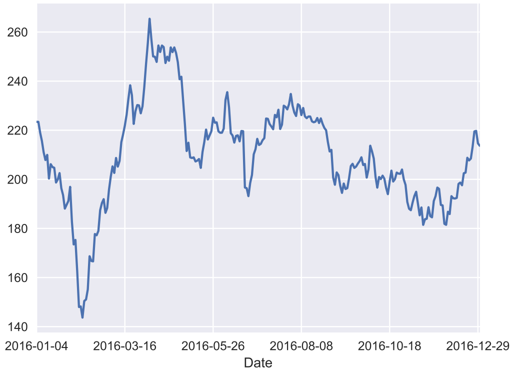
```
import matplotlib.pyplot as plt
import seaborn # for styling
seaborn.set()  # for styling

tsla.['Close'].plot()
```



Pandas handles Time Series data really well

```
# plot just the close price for 2016
tsla['Close'].loc['2016':'2017'].plot()
```



Creating date-time sequences with pandas

```
in : pd.date_range('2017-01-03', '2017-01-10')
out: DatetimeIndex(['2017-01-03', '2017-01-04', '2017-01-05',
 '2017-01-06', '2017-01-07', '2017-01-08', '2017-01-09',
 '2017-01-10'],dtype='datetime64[ns]', freq='D')

in : pd.date_range('2017-07-03', periods=8)
out: DatetimeIndex(['2017-07-03', '2017-07-04', '2017-07-05',
 '2017-07-06','2017-07-07', '2017-07-08', '2017-07-09',
 '2017-07-10'], dtype='datetime64[ns]', freq='D')

in : pd.date_range('1999-07-03', periods=4, freq='H')
out:
DatetimeIndex(['1999-07-03 00:00:00', '1999-07-03 01:00:00',
               '1999-07-03 02:00:00', '1999-07-03 03:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Creating date-time sequences with pandas

```
in : pd.period_range('1943-03', periods=8, freq='M')
out: PeriodIndex(['1943-03', '1943-04', '1943-05', '1943-06',
     '1943-07', '1943-08', '1943-09', '1943-10'],
     dtype='period[M]', freq='M')

in : pd.timedelta_range(0, periods=10, freq='H')
out: TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00',
    '03:00:00', '04:00:00', '05:00:00', '06:00:00',
    '07:00:00', '08:00:00', '09:00:00'],
     dtype='timedelta64[ns]', freq='H')
```

So Why use Pandas

- Useful function for working with databases
- DataFrames can be manipulated easier than lists
- For exporting and importing CSV files in Python
- Working with time series data

Thanks for taking my class!

Hopefully you've learned something about Python.