Fairfield University DOLAN School of Business

Python for Analytics

The Pandas Library

[The Pandas Docs]

Learning Objectives

- **Theory:** You should be able to explain ...
  - The purpose of Pandas (versus, say, NumPy)
  - The Series and DataFrame data types
- **Skills:** You should know how to ...
  - How to create well-structured Series and DataFrames from lists, dicts, Numpy arrays, etc.
  - Use advanced selection techniques on tabular data
  - Work with time series data
  - Import and export data from/to various sources
  - Use NumPy with Pandas

Overview

Because NumPy Needs a Little Help Sometimes

What’s Pandas?

From the docs ...

> “**pandas** is a [Python](http://www.python.org/) package providing **fast, flexible, and expressive data structures** designed to **make working with 'relational' or 'labeled' data both easy and intuitive**.”

- Sounds a lot like a database management system, right?
- Why would we need that if we already have NumPy?

Information is more than just Data

NumPy's a very fast and efficient number crunching machine for structured data, which is great, but …

- NumPy arrays are not so good at preparing data for analysis
  - What if we only want a subset of the rows and columns?
  - What if our data tables have missing values?
  - What if we want more than `ints`, `floats`, and strings?
- NumPy does almost nothing about presentation of data for humans

Pandas to the Rescue

Pandas is built on top of NumPy to provide:

- **More flexible data structures** that can handle more data types and indexing schemes
- A wide variety of functions and methods for **slicing and dicing large data sets** into NumPy-friendly chunks
- Import and export facilities for **just about any data source** one might actually encounter

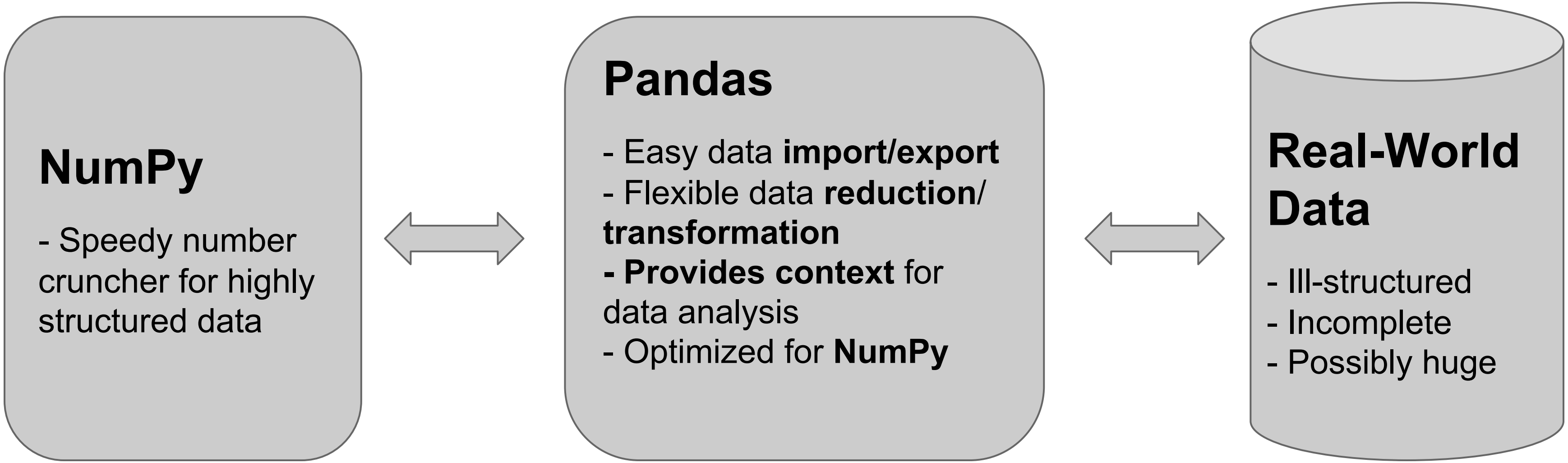
Why Do We Need NumPy Then?

Pandas’s flexibility and utility comes at the cost of performance:

- **Speed:** NumPy, with its highly structured and optimized numerical routines, is much faster than Pandas for most things.
- **Storage:** Pandas’s greater expressiveness also makes it more verbose and less space efficient

Nonetheless, Pandas makes some things possible that NumPy can’t do alone.

Bridging an Architectural Gap



Standard Imports

The remaining slides assume that we have already imported NumPy and Pandas in the standard way.

```python
import numpy as np
import pandas as pd
```

Pandas `dtypes`

Pandas uses a relatively small but flexible set of data types to store individual data items:

- Primitives:`int`, `float`, and `bool`
- Date/Time: `datetime64` and `timedelta`
- Enumerations: `category`
- Serializables: `object` (e.g., string as 'S#')

Recall that NumPy has `int`, `float`, and string objects

Series

NumPy arrays (kind of) with a few useful tweaks

Series is a Container for 1D data

Each column of a table might be a series.

A series has a **common data type** (`int`, `float`, etc.) and possibly **a name** (e.g., the column header).

The **values in the series are indexed** either **by position** (0,1,2,3,4,etc.) or **by label** (‘a’, ‘b’, ‘c’, etc.).

Series are also **NumPy compatible**.

- Most NumPy functions can even take Pandas Series as arguments! (`Series` implements the same interface as `ndarray`.)

Creating a Series (data, index, name)

```
s1 = pd.Series([3.1, 2.8, 8.9])
s2 = pd.Series([3.1, 2.8, 8.9],
        index=['apatite', 'calcite', 'copper'])
s3 = pd.Series([3.1, 2.8, 8.9],
        index=['apatite', 'calcite', 'copper'],
        name= 'density')
s3      →  apatite    3.1
           calcite    2.8
           copper     8.9
           Name: density, dtype: float64
```

... from a NumPy array

```
s4=Series(np.array([3.1, 2.8, 8.9]),
              index=['apatite','calcite','copper'],
              name='density')
s4 →   apatite     3.1
       calcite     2.8
       cooper      8.9
       Name: density, dtype: float64
```

… from a `dict`

```
s5 = pd.Series({'apatite':3.1,'calcite':2.8,'copper':8.9},
                name='density')
s5 →  apatite    3.1
      calcite    2.8
      cooper     8.9
      Name: density, dtype: float64
```

Series are `array`-like and `dict`-like

```
# can slice like a NumPy array
s4[:2] →    apatite    3.1
            calcite    2.8
            Name: density, dtype: float64

# can use keys like a dict
s4['apatite'] → 3.1000000000000001
```

DataFrames

The workhorse of data science

DataFrames are for 2D data

Most similar to a database table:

- organized into rows and columns
- each column has a name and a data type
- each row has an index (numbered or labeled)

Convertible from/to NumPy Structured Arrays.

- Even the attributes (names) from `rec.array` translate

Have advanced indexing features to provide 'query-like' selections of data

Creating DataFrames

Lots and lots and lots of options:

- From a list of dictionaries (row-wise)
- From a `dict` of lists or `np.array`s (column-wise)
- From a NumPy `array` or `rec.array`
- From a **Series** (or `dict` of **Series**)
- Using `pd.from_dict()`, `pd.from_records()`, `pd.from_items()` functions
- ...

… From a List of `dicts`

```
planets_list_of_dicts =
[{'name':'Mercury','diam':4878,'spin':59,'orbit':88,'grav':0.38},
 {'name':'Venus','diam':12104,'spin':243,'orbit':224,'grav':0.9},
 {'name':'Earth','diam':12756,'spin':0.997,'orbit':365.25,'grav':1.0},
 {'name':'Mars','diam':6794,'spin':1.025,'orbit':687,'grav':0.38},
 {'name':'Jupiter','diam':142984,'spin':0.413,'orbit':4329,'grav':2.64},
 {'name':'Saturn','diam':120536,'spin':0.44375,'orbit':10592.25,'grav':1.16},
 {'name':'Uranus','diam':51118,'spin':0.71805,'orbit':30681,'grav':1.11},
 {'name':'Neptune','diam':49532,'spin':0.67153,'orbit':60193.2,'grav':1.21}
]
```

... From a List of `dicts` (2)

```
planets1 = pd.DataFrame(planets_list_of_dicts)
planets1 →
```

	diam	grav	**name**	orbit	spin
0	4878	0.38	**Mercury**	88.00	59.000000
1	12104	0.90	**Venus**	224.00	243.000000
2	12756	1.00	**Earth**	365.25	0.997000
3	6794	0.38	**Mars**	687.00	1.025690
4	142984	2.64	**Jupiter**	4329.00	0.413194
5	120536	1.16	**Saturn**	10592.25	0.443750
6	51118	1.11	**Uranus**	30681.00	0.718050
7	49532	1.21	**Neptune**	60193.20	0.671530

Notice how it alphabetizes the columns?

Column order does not matter unless we make it matter.

... From a `dict` of `dicts`

```
planets_dict_of_dicts= {
    'diam':{'Mercury':4878,'Venus':12104,'Earth':12756},
    'spin':{'Mercury':59,'Venus':243,'Earth':0.997},
    'orbit':{'Mercury':88,'Venus':0.9,'Earth':365.25}
}
```

one `dict` per column, with the planet names as keys.

```
planets2=pd.DataFrame(planets_dict_of_dicts)
print(planets2) →
          diam   orbit     spin
Earth    12756  365.25    0.997
Mercury   4878   88.00   59.000
Venus    12104    0.90  243.000
```

The planet names (keys) become the indexes (labels), with rows listed in alpha order.

… From a `dict` of NumPy Arrays

```
planets_dict_of_arrays = {
    'diam':np.array([4878,12104,12756]),
    'spin':np.array([59,243,0.997]),
    'orbit':np.array([88,0.9,365.25])
}
planets3=pd.DataFrame(planets_dict_of_arrays,
    index=['Mercury','Venus','Earth'])
print(planets3) →
            diam   orbit    spin
    Mercury  4878   88.00   59.000
    Venus   12104    0.90  243.000
    Earth   12756  365.25    0.997
```

one `array` per column.

specify names for index

Rows listed in the order given in the index

… From a 2D NumPy Array

```
planets_2d_array =
      np.array([[4878,12104,12756],
                [59,243,0.997],
                [88,0.9,365.25]])

planets4=pd.DataFrame(planets_2d_array,
              index=['Mercury','Venus','Earth'],
              columns=['diam','spin','orbit'])
print(planets4) →
               diam     spin     orbit
      Mercury  4878.0  12104.0  12756.000
      Venus      59.0    243.0      0.997
      Earth      88.0      0.9    365.250
```

2D array with `dtype=float`

specify both row indexes and column names

data appears in the same order as given by `index` and `columns`

... from a NumPy `rec.array`

`rec.array` with dtype used to spec column names and types

```
planets_rec_array =
    np.rec.array(
          [['Mercury',4878,12104,12756],
           ['Venus',59,243,0.997],
           ['Earth',88,0.9,365.25]],
           dtype=[('name','S10'),('diam',float),('spin',float),
                ('orbit',float)])
planets5=pd.DataFrame(planets_rec_array)
print(planets5) →
           name    diam     spin     orbit
    0  b'Mercury'  4878.0  12104.0  12756.000
    1    b'Venus'    59.0    243.0      0.997
    2    b'Earth'    88.0      0.9    365.250
```

Missing Data (NaN)

Sometimes, data tables are incomplete, with missing data. When this happens the DataFrame stores `NaN` ('not a number') instead of a number.

This is a feature, not a bug!

... with Missing data

```
planets_dict_of_dicts_with_missing_data= {
    'diam':{'Mercury':4878,'Venus':12104,'Earth':12756},
    'spin':{'Mercury':59,'Venus':243,'Earth':0.997},
    'orbit':{'Mercury':88,'Venus':0.9,'Earth':365.25},
    'pop':{'Earth':7500000000}
}
```

Note that the 'pop' `dict` only has one key-value pair.

```
planets6=pd.DataFrame(planets_dict_of_dicts_with_missing_data)
print(planets6) →
            diam   orbit           pop     spin
    Earth   12756  365.25  7.500000e+09    0.997
    Mercury  4878   88.00           NaN   59.000
    Venus   12104    0.90           NaN  243.000
```

NaN indicates that data is missing.
We just don't know, right? □

Adding/Deleting Columns

```
# Adding/deleting columns is just like a dict

# add the 'pop' column to planets2
planets2['pop'] =
    pd.Series({'Earth':7500000000},index=['Mercury','Venus','Earth'])
print(planets2) →
              diam   orbit     spin            pop
    Earth    12756  365.25    0.997   7.500000e+09
    Mercury   4878   88.00   59.000            NaN
    Venus    12104    0.90  243.000            NaN

# delete the 'pop' column from planets2
del planets2['pop']
```

Used a Series so that we could include missing data.

To add the column in the middle instead of the end, use the DataFrame `insert()` method.

Descriptive Stats

Pandas supports many of the NumPy universal functions and methods.

(In fact, when possible it just reuses the ones in NumPy.)

```
planets1['diam'].mean()
    → 50087.75
# describe() with mean, mode, ...
planets1.describe()
# histogram counts
planets1['spin'].value_counts(bins=3)
```

Function	Description
`count`	Number of non-null observations
`sum`	Sum of values
`mean`	Mean of values
`mad`	Mean absolute deviation
`median`	Arithmetic median of values
`min`	Minimum
`max`	Maximum
`mode`	Mode
`abs`	Absolute Value
`prod`	Product of values
`std`	Bessel-corrected sample standard deviation
`var`	Unbiased variance
`sem`	Standard error of the mean
`skew`	Sample skewness (3rd moment)
`kurt`	Sample kurtosis (4th moment)
`quantile`	Sample quantile (value at %)
`cumsum`	Cumulative sum
`cumprod`	Cumulative product
`cummax`	Cumulative maximum
`cummin`	Cumulative minimum

Selecting Datasets

[ ], loc(), iloc(), and join()

Selection with `[ ]`

```
# slice by rows
planets6[:2]
   →             diam   orbit    population    spin
      Earth     12756  365.25  7.500000e+09   0.997
      Mercury    4878   88.00           NaN  59.000
# can also select a column as a Series
planets6['diam']  → Earth       12756
                    Mercury      4878
                    Venus       12104
                    Name: diam, dtype: int64
```

Selection with `loc` and `iloc`

- **`loc`** is used to *select by labels*
- **`iloc`** is used to *select by position*
- Both attributes use 2D slicing notation

  [*from_row* **:** *to_row* **,** *from_col* **:** *to_col*]

```
planets6.loc['Earth':'Venus','diam':'orbit']
```

```
→              diam    orbit
    Earth     12756   365.25
    Mercury    4878    88.00
```

Advanced Selection Techniques

Pandas includes lots of other functions and methods for subsets of 1D and 2D data sets.

- Boolean selections with **`query()`**
- Masking with **`where()`**
- Lambda-based selections with **`select()`**
- Rowset and columnset selections with **`lookup()`**
- Set-based selections with **`isin()`**

Indexing docs: https://pandas.pydata.org/pandas-docs/stable/indexing.html

Database-Style DataFrame Joins

What if we need to cross-reference rows in one DataFrame with rows in another DataFrame?

The `merge()` function returns a SQL-style join on two DataFrames.

- **inner join**, **left outer join**, **right outer join**, etc.
- **`where`** is implemented with slicing, loc, iloc, etc.

Merge docs: http://pandas.pydata.org/pandas-docs/stable/merging.html
Compare to SQL: http://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html

Input/Output

Jupyter, HTML, CSV, Excel, SQL, JSON, Google Big Query, etc.

Easy HTML Tables in Jupyter

Just select the entire DataFrame with `[:]` or `.style`

You don't even have to print anything, but **it only works once per cell**.

In [12]:
```
import pandas as pd
planets_dict_of_dicts_with_missing_data= {
    'diam':{'Mercury':4878,'Venus':12104,'Earth':12756},
    'spin':{'Mercury':59,'Venus':243,'Earth':0.997},
    'orbit':{'Mercury':88,'Venus':0.9,'Earth':365.25},
    'population':{'Earth':7500000000}
}

planets6=pd.DataFrame(planets_dict_of_dicts_with_missing_data)
print(planets6)

# the easy way
planets6.style
```

```
         diam   orbit    population     spin
Earth    12756  365.25  7.500000e+09    0.997
Mercury   4878   88.00           NaN   59.000
Venus    12104    0.90           NaN  243.000
```

Out[12]:

	diam	orbit	population	spin
Earth	12756	365.25	7.5e+09	0.997
Mercury	4878	88	nan	59
Venus	12104	0.9	nan	243

Style docs: https://pandas.pydata.org/pandas-docs/stable/style.html

The More Traditional Way

IO Tools (Text, CSV, HDF5, ...)

The pandas I/O API is a set of top level `reader` functions accessed like `pd.read_csv()` that generally return a `pandas` object. The corresponding `writer` functions are object methods that are accessed like `df.to_csv()`

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

I/O docs: http://pandas.pydata.org/pandas-docs/stable/io.html#io-tools-text-csv-hdf5

HTML Files

To write an HTML table to a file (or a string or a stream), just use the DataFrame's `to_html()` method:

```
planets6.to_html("planets.html")
```

planets6.html

```html
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>diam</th>
      <th>orbit</th>
      <th>population</th>
      <th>spin</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>Earth</th>
      <td>12756</td>
      <td>365.25</td>
      <td>7.500000e+09</td>
      <td>0.997</td>
    </tr>
    <tr>
      <th>Mercury</th>
      <td>4878</td>
      <td>88.00</td>
      <td>NaN</td>
      <td>59.000</td>
    </tr>
    <tr>
```

Again, in Jupyter

For multiple tables in a Jupyter cell, use the IPython (Jupyter) **`HTML()`** and **`display()`** functions to render the HTML.

In [22]:
```
from IPython.core.display import display, HTML
display(HTML(planets6.to_html())) #first table
display(HTML(planets7.to_html())) #second table
```

	diam	orbit	population	spin
Earth	12756	365.25	7.500000e+09	0.997
Mercury	4878	88.00	NaN	59.000
Venus	12104	0.90	NaN	243.000

	Unnamed: 0	diam	orbit	population	spin
0	Earth	12756	365.25	7.500000e+09	0.997
1	Mercury	4878	88.00	NaN	59.000
2	Venus	12104	0.90	NaN	243.000

CSV Files

```
# write to a CSV file called "planets.csv"
planets6.to_csv("planets.csv")

# reading is also very straightforward
planets7 = pd.read_csv("planets.csv",index_col=0)

# To read a CSV file over the web just use a URL
planets9 =
   pd.read_csv("https://planets.org/data.csv")
```

Lots of other optional arguments are defined in the docs!

CSV docs: http://pandas.pydata.org/pandas-docs/stable/io.html#io-read-csv-table

And so forth

By providing a consistent interface for the I/O functions and methods, Pandas makes it pretty easy to guess how to deal with new formats.

There may be some optional arguments that vary according to format, but usually the defaults do pretty much what you expect them to do.

As always, RTFM if you need something special.

NumPy with Pandas

We don’t have to choose one over the other

Pandas → NumPy → Pandas

NumPy → Pandas is easy

- Pandas `pd.Series()` or `pd.DataFrame()` constructors are designed to convert from NumPy arrays.

Pandas → NumPy can take some work

- While NumPy does not reciprocate by taking Pandas Series or DataFrames in its `np.array()` constructor, there are several handy methods for exporting to NumPy arrays.

The `values` Attribute

Pandas Series and DataFrames use NumPy arrays internally to store data. These arrays can be accessed directly using the `values` attribute.

```
# Access an interval ndarray
planets_ndarray = planets6.values

# planets_ndarray is now an alias (or view)
planets_ndarray[1][2] = 5   # modifies planets6
```

The `lookup()` method

We saw `lookup()` before. It's used to select explicit sets of rows and columns.

The result is always a NumPy **`array`**.

The `to_records()` Method

The `to_records()` method returns a NumPy `rec.array` with NumPy-style `dtype` specs.

```
planets_recarray = planets6.to_records()
planets_recarray
```

→ **rec.array**([('Earth', 12756, 365.25, 7.50000000e+09, 0.997),
('Mercury', 4878, 88. , nan, 59.),
('Venus', 12104, 0.9 , nan, 243.)],
dtype=[('index', 'O'), ('diam', '<i8'), ('orbit', '<f8'), ('pop', '<f8'), ('spin', '<f8')])

Python for Analytics

The Pandas Library

[The Pandas Docs]