Introduction to Pandas

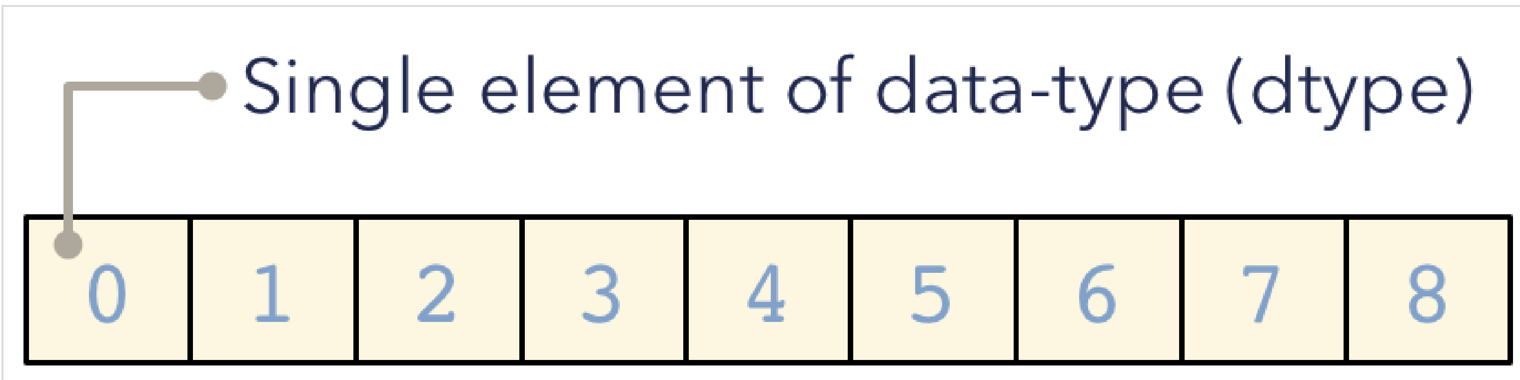
Continuum Analytics

First, we need a little

NumPy

What is NumPy?

Python library that provides multi-dimensional arrays, tables, and matrices for Python



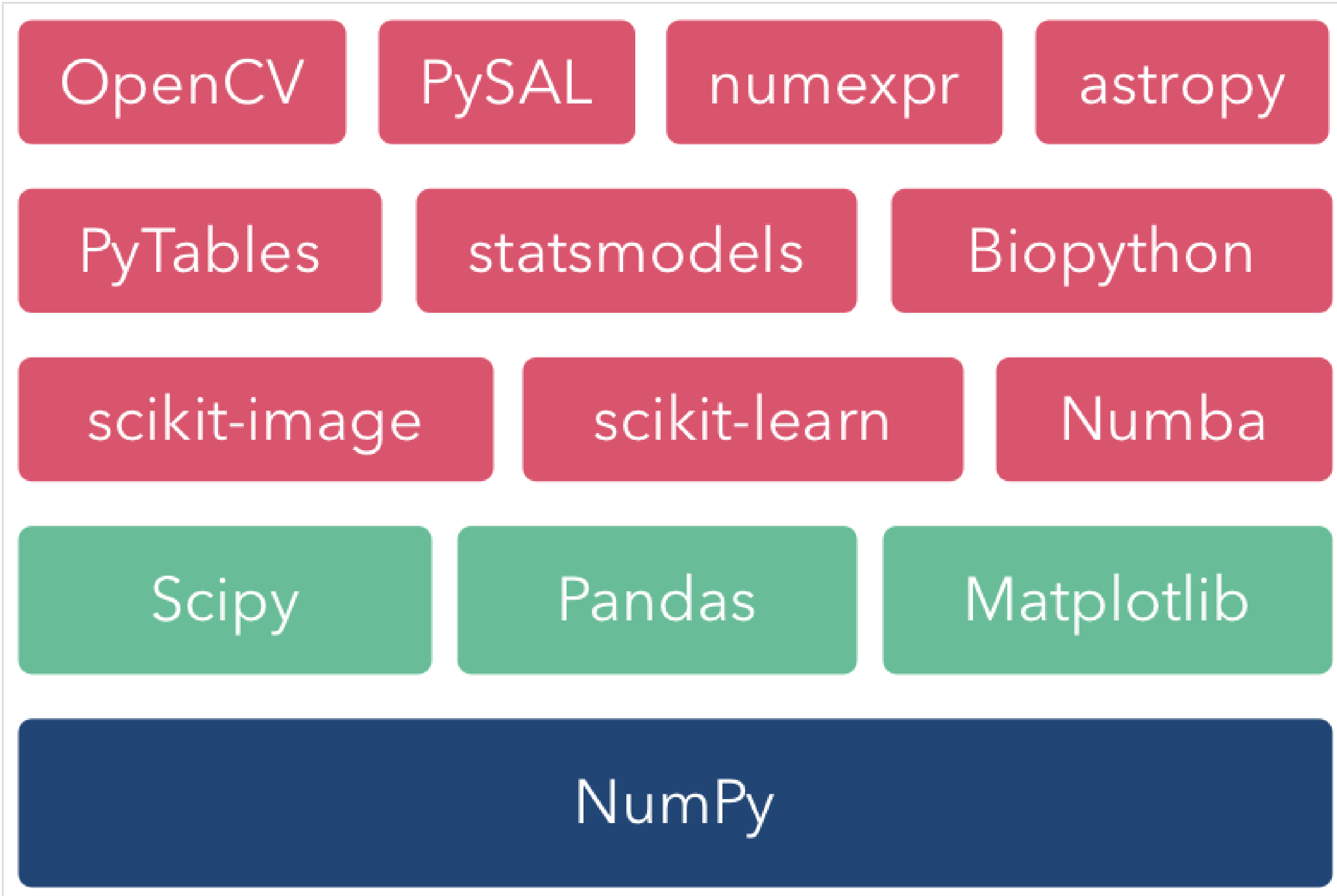
- Contiguous or strided arrays
- Homogeneous (but types can be algebraic)
    - Arrays of records and nested records
- Fast routines for array operations (C, ATLAS, MKL)

NumPy's Many Uses

- Image and signal processing
- Linear algebra
- Data transformation and query
- Time series analysis
- Statistical analysis
- Many more!

NumPy is the foundation of the Python scientific stack

NumPy Ecosystem



CONTINUUM ANALYTICS

Pandas

- Panel Data Structures
- Written by Wes McKinney, former quant at AQR
- Data alignment
- Date Time handling
- Moving window statistics
- Resampling/frequency conversion
- Easy and fast data access (hdf5, csv, sql)
- Integration with Matplotlib
- Statistical modeling
- Group by, joining, merging, pivoting

CONTINUUM
ANALYTICS

Pandas

- `Series` and `DataFrame` are the main structures
- Recursive nature of Pandas is key: operations on `Series` and `DataFrame` produce more `Series` and `DataFrame`
- No matter where you are in your analysis, you always have your full arsenal at your disposal
- Library is designed around comfort, rather than around programmatic consistency

CONTINUUM
ANALYTICS

Glossary

Recalling NumPy...

- Indexing - `a[2,3]`, selecting one value
- Slicing - `a[2:10]`, using slicing notation to select many values
- Fancy Indexing - `a[[2,3,4,5], 3]` or `a[[True, True], 3]`

Pandas Series

- like a NumPy array but with an index

```
In [25]: index = ['a','b','c','d','e']

In [28]: series = pandas.Series(np.arange(5), index=index)
Out[28]:
a     0
b     1
c     2
d     3
e     4

In [29]: series = pandas.Series(np.arange(5), index=index)

In [30]: series
Out[30]:
a     0
b     1
c     2
d     3
e     4
```

Pandas Series

- Refer to content by named index or by range

```
In [1]: series['a']
Out[1]: 0

In [2]: series['b']
Out[2]: 1

In [3]: series['c']
Out[3]: 2

In [3]: series[2:4]
Out[3]:
c     2
d     3

In [4]: series * 2
Out[4]:
a     0
b     2
c     4
d     6
e     8
```

Pandas Series

- `Series` extends NumPy array
- If you don't pass in an index, one is created for you (equivalent of range(N), where N is the length of your data)
- Index used to implement fast lookups, data alignment and join operations
- Supports hierarchical indexes, where each label is a tuple
- *Try to avoid integer index names*

Series Construction

- Can be constructed with an array like object, or with a dict
  - With a dict, the keys are *sorted* and used as the index
  - With an array-like object, you can pass in another array-like object as the index

Series Indexing

- Indexing looks up value using the index (row label)
  - `myseries[0]`
  - `myseries['a']`
- Slicing with integers defaults to ignoring the index
  - `myseries[2:4]`
- Slicing with non-integers uses the index, and is inclusive
  - `myseries['a':'c']`
- Order matters
  - `myseries['a':'c']` is different from `myseries['c':'a']`
- *Try to avoid integer index names*

CONTINUUM ANALYTICS

Series Operations

- You can do math using series
  - When index values are different, default to an outer join

```
In [184]: myseries
Out[184]:
a    1
b    2
d    4
e    5

In [185]: myseries2
Out[185]:
a     1
b     2
d     4
f    10

In [186]: myseries + myseries2
Out[186]:
a     2
b     4
d     8
e   NaN
f   NaN
```

Hierarchical Indexes

```
In [15]: a = pandas.Series(np.random.random(4),
   ....:                        index=[('Banks', 'C'), ('Banks', 'JPM'),
   ....:                               ('Tech', 'GOOG'), ('Tech', 'MSFT')])

In [16]: a
Out[16]:
('Banks', 'C')      0.024134
('Banks', 'JPM')    0.339072
('Tech', 'GOOG')    0.946847
('Tech', 'MSFT')    0.722363
```

- Discussed in more detail in a later section

Demo 1

Pandas DataFrames

- `DataFrame` is a collection of Pandas `Series`
  - joined on index: DateTime, AlphaNumerical Index, etc

- This index is also referred to as a row label

- Pandas `DataFrame` objects have column names:
  - accessed attribute style: `prices.Close`
  - dictionary style: `prices['Adj Close']`

CONTINUUM
ANALYTICS

Pandas DataFrames

- `DataFrame` binary operations (+ - / *) defaults to outer join, on both columns as well as the index
- `NA` can be handled after join
- `DataFrame` objects are **NOT** NumPy arrays

DataFrame Example

```
In [14]: rawdata = {'a': np.random.random(5), 'b': np.random.random(5)}

In [15]: data = pandas.DataFrame(rawdata)
In [16]: data
Out[16]:
          a         b
0  0.266826  0.602288
1  0.338174  0.294303
2  0.019489  0.473737
3  0.876180  0.518681
4  0.901697  0.370186

In [17]: data[1:3]
Out[17]:
          a         b
1  0.338174  0.294303
2  0.019489  0.473737

In [18]: data[1:3].a
Out[18]:
1    0.338174
2    0.019489
Name: a
```

DataFrame Indexing

- `DataFrame` is dict-like in referring to column names

- Using a list of column names selects that list of columns

- Similar to `Series`, mathematical operations on `DataFrame` objects default to outer-joins
    - joins occur both row-wise, and column-wise

- `df.ix` for NumPy like indexing semantics

- `df.xs` for cross-section along a row

CONTINUUM ANALYTICS

DataFrame Indexing

- Referencing
  - first dimension refers to the index
  - second dimension refers to the columns
  - for single level indexes, more about this later
- Advanced DateTime Indexing
- *Try to avoid integer index names*

DataFrame Updates

- Adding New Columns:
  - zero fill: `df['var'] = 0`
  - values from NumPy array: `df['my_data'] = data`
  - note: `df.var` construct can not create a column by that name; only used to access existing columns by name
- Deleting Columns:
  - `df.drop(['var','new_data'], axis=1)`

Demo 2

Dataframe Construction

- Dict of array like objects -- keys are column names
- Nested dict of values
- CSV
  - Excel Files (requires `xlrd`)
- HDF5
- SQL

Dataframe Extraction

- Important to get data out of `DataFrame`
- `df.values` returns underlying NumPy array
- to_method
  - `df.to_csv`
  - `df.to_excel`
  - `df.to_html`
  - `df.to_latex`
  - ...

Working with CSV Data

- Basic Usage:
  - `pandas.read_csv(file_path, sep=',')`
- Almost every option for messy CSVs
- Noteable Options
  - `index_col`
  - `parse_dates`
  - `header`
  - `skip_footer`
  - `gzip loading`
  - `na_values` (for missing values)
  - Integrated DateTime Indexing

Working with Missing Data

- When loading with read_csv use `na_values`
    - Pandas only understands **NaN**
    - `na_values` defines your version of NaN
- Drop all rows with missing values: `dropna`
- Fill in missing values: `fillna`
    - zero fill: `fillna(0)`
    - forward fill but only up to 5 at a time: `fillna(method='pad', limit=5)`
    - interpolation: `pandas.Series.interpolate`

```
df = pandas.read_csv(file_path, sep=',', na_values='nil')
```

Working with XLSX Data

- More work than CSV

- XLSX documents have sheets

- Select sheets then parse

```
xlsx = pd.io.parsers.ExcelFile('../data/geo_loc.xlsx')
sheet = xlsx.sheet_names[0]
df = xlsx.parse(sheet)
```

- parse has same additional arguments as `read_csv`

Builtin Math

- Many Standard Computational tools
  - `rolling_count`: Number of non-null observations
  - `rolling_sum`: Sum of values
  - `rolling_mean`: Mean of values
  - `rolling_median`: Arithmetic median of values
  - `rolling_window`: Moving window function
  - `rolling_apply`: Generic apply
  - ...
  - basic usage: `pandas.rolling_sum(df, window)`
- Each method returns a new `DataFrame`

Integrated Plotting

- Uses Matplotlib backend for easy plotting

- Examples
  - plot entire data frame `DataFrame`: `df.plot()`
  - or column `df['VOL'].plot()`
  - or statistic `df.rolling_mean().plot()`

```python
import pandas as pd
import matplotlib.pyplot as plt

ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', p
eriods=1000))
ts = ts.cumsum()

ts.plot(style='k--')

pd.rolling_mean(ts, 60).plot(style='k')

plt.show()
```

CONTINUUM
ANALYTICS

The Index

- `set_index` versus `reindex`
  - `set_index` replaces the index with some new values, you can refer to them by column name, or pass an array
  - `reindex` subselects from the `DataFrame`, padding any values that are necessary with `NA`

DateTime Indexing and Resampling

- Built-in logic for standard time chunks
  - microsecond, millisecond, minute, hour, etc.
  - `df.index.day`, `df.index.dayofweek`
- Resampling for non-standard time chunks (up- or down-sampling)
  - `df.resample(time,fill_method)`
- Can build index for any time chunk

```
df.resample('1min', fill_method='pad')
min35 = pandas.dateoffset(minutes=35)
df[datetime(2010,10,10,0,0,0) + min35]
```

Exercise 1

Split—Apply—Combine

- Split the data into chunks
- Apply some transformation or aggregation onto the chunks
- Pull the computed values back into a data structure
- Repeat again and again
- You often end up with very simple code for each step, rather than one hunk of complicated code
- This is much easier to reason about, and debug
- A bit tricky to get used to at first
- Similar mental process to understanding vectorized computing