STAT606
Computing for Data Science and Statistics

Lecture 13-14: pandas
Pandas

Open-source library of data analysis tools
Low-level ops implemented in Cython (C+Python=Cython, often faster)
Database-like structures, largely similar to those available in R
Well integrated with numpy/scipy
Optimized for most common operations
E.g., vectorized operations, operations on rows of a table

From the documentation: pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.
Installing pandas

Using conda:
conda install pandas

Using pip:
pip install pandas

From binary (not recommended):
http://pypi.python.org/pypi/pandas

Warning: a few recent updates to pandas have been API-breaking changes, meaning they changed one or more functions (e.g., changed the number of arguments, their default values, or other behaviors). This shouldn’t be a problem for us, but you may as well check that you have the most recent version installed.
Basic Data Structures

Series: represents a one-dimensional labeled array
   Labeled just means that there is an index into the array
   Support vectorized operations

DataFrame: table of rows, with labeled columns
   Like a spreadsheet or an R data frame
   Support numpy ufuncs (provided data are numeric)
By default, indices are integers, starting from 0, just like you’re used to.

But we can specify a different set of indices if we so choose.

Warning: providing too few or too many indices is a ValueError.
**pandas Series**

```python
d = {'dog':3.1415, 'cat':42, 'bird':0, 'goat':1.618}
s = pd.Series(d)
s
bird    0.0000
cat     42.0000
dog     3.1415
goat    1.6180
dtype: float64

inds = ['dog', 'cat', 'bird', 'goat', 'cthulu']
s = pd.Series(d, index=inds)
s
dog     3.1415
cat     42.0000
bird    0.0000
goat    1.6180
cthulu  NaN
dtype: float64```

Can create a series from a dictionary. Keys become indices.

Index ‘cthulu’ doesn’t appear in the dictionary, so pandas assigns it NaN, the standard “missing data” symbol.
Indexing works like you’re used to and supports slices, but **not** negative indexing.

This object has type `np.int64`

This object is another `pandas Series`. 

```python
s = pd.Series([2, 3, 5, 7, 11])
s[0]
s[1:3]  
1 3
2 5
dtype: int64
```

```python
s[-1]
```

```
KeyError: 0
```

Traceback (most recent call last)
<ipython-input-22-0e2107f91cbd> in <module>()
    0 s[-1]
----> 1 s[-1]
pandas Series

Caution: indices need not be unique in pandas Series. This will only cause an error if/when you try to perform an operation that requires unique indices.
Series objects are like `np.ndarray` objects, so they support all the same kinds of slice operations, but note that the indices come along with the slices.

Series objects even support most `numpy` functions that act on arrays.
Series objects are \texttt{dict}-like, in that we can access and update entries via their keys.

\textbf{Not shown:} Series also support the \texttt{in} operator: \texttt{x in s} checks if \texttt{x} appears as an index of Series \texttt{s}. Series also supports the dictionary \texttt{get} method.

Like a dictionary, accessing a non-existent key is a \texttt{KeyError}.

\textbf{Note:} I cropped out a bunch of the error message, but you get the idea.
Entries of a Series can be of (almost) any type, and they may be mixed (e.g., some floats, some ints, some strings, etc), but they **can not** be sequences.

Series support universal functions, so long as all their entries support operations.

Series operations require that keys be shared. Missing values become NaN by default.

To reiterate, Series objects support most numpy ufuncs. For example, `np.sqrt(s)` is valid, so long as all entries are positive.
Series have an optional name attribute. After it is set, name attribute can be changed with rename method. Note: this returns a new Series. It does not change s.name. This will become especially useful when we start talking about DataFrames, because these name attributes will be column names.
Mapping and linking Series values

Series `map` method works analogously to Python’s `map` function. Takes a function and applies it to every entry.
Mapping and linking Series values

```python
s = pd.Series(['fruit', 'animal', 'animal', 'fruit', 'fruit'],
              index=['apple', 'cat', 'goat', 'banana', 'kiwi'])
s
apple    fruit
cat      animal
goat     animal
banana   fruit
kiwi     fruit
dtype: object
```

```python
t = pd.Series({'fruit': 0, 'animal': 1})
s.map(t)
```

apple    0
cat      1
goat     1
banana   0
kiwi     0
dtype: int64

Series `map` also allows us to change values based on another Series. Here, we're changing the fruit/animal category labels to binary labels.
pandas **DataFrames**

Fundamental unit of **pandas**
- Analogous to R data frame

2-dimensional structure (i.e., rows and columns)
- Columns, of potentially different types
  - Think: spreadsheet (or, better, database, but we haven’t learned those, yet)

Can be created from many different objects
- Dict of {ndarrays, Python lists, dicts, Series}
- 2-dimensional ndarray
- Series
Creating a DataFrame from a dictionary, the keys become the column names. Values become the columns of the dictionary.

Each column may have its own indices, but the resulting DataFrame will have a row for every index (i.e., every row name) that appears.

Note: in the code above, we specified the two columns differently. One was specified as a Series object, and the other as a dictionary. This is just to make the point that there is flexibility in how you construct your DataFrame. More options:

Indices that are unspecified for a given column receive NaN.
pandas DataFrames: creating DataFrames

Dictionary has 4 keys, so 4 columns.

```
     'PhD': {'Wilson':'Johns Hopkins'},
     'JD': {'Ford':'Yale', 'Obama':'Harvard'},
     'Terms': pd.Series([1,1,2,2], index=['Ford', 'Hoover', 'Wilson', 'Obama'])
}
presidents = pd.DataFrame(d)
presidents
```

**Note:** Dictionary includes both text and numeric columns

By default, rows and columns are ordered alphabetically.
### pandas DataFrames: row/column names

Row and column names accessible as the `index` and `column` attributes, respectively, of the DataFrame.

<table>
<thead>
<tr>
<th></th>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>1</td>
<td>UMich</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
<td>Stanford</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>2</td>
<td>Columbia</td>
</tr>
<tr>
<td>Wilson</td>
<td>NaN</td>
<td>Johns Hopkins</td>
<td>2</td>
<td>Princeton</td>
</tr>
</tbody>
</table>

```python
presidents.columns
Index([u'JD', u'PhD', u'Terms', u'Undergrad'], dtype='object')

presidents.index
Index([u'Ford', u'Hoover', u'Obama', u'Wilson'], dtype='object')
```
### pandas DataFrames: accessing/adding columns

#### Key Points:

- A DataFrame acts like a dictionary whose keys are column names, and values are Series.
- We can create new key-value pairs in a DataFrame, similar to creating new key-value pairs in a dictionary.

#### Example Code:

```python
presidents['PhD'] = NaN
Hoover['PhD'] = NaN
Obama['PhD'] = NaN
Wilson['PhD'] = Johns Hopkins
Name: PhD, dtype: object
```

#### Table Example:

<table>
<thead>
<tr>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>UMich</td>
<td>4</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>Stanford</td>
<td>4</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>Columbia</td>
<td>8</td>
</tr>
<tr>
<td>Wilson</td>
<td>NaN</td>
<td>Johns Hopkins</td>
<td>Princeton</td>
<td>8</td>
</tr>
</tbody>
</table>

**Note:** Technically, this isn’t quite correct, because Ford did not serve a full term. [https://en.wikipedia.org/wiki/Gerald_Ford](https://en.wikipedia.org/wiki/Gerald_Ford)
Since the row labels are ordered, we can specify a new column directly from a Python list, `numpy` array, etc. without having to specify indices.

**Note:** by default, new column are inserted at the end. See the `insert` method to change this behavior:

### pandas DataFrames: accessing/adding columns

Scalars are broadcast across the rows.

```python
1 presidents["Fields Medals"] = 0
2 presidents
```

<table>
<thead>
<tr>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
<th>Nobels</th>
<th>Years</th>
<th>Fields Medals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>1</td>
<td>UMich</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
<td>Stanford</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>2</td>
<td>Columbia</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Wilson</td>
<td>NaN</td>
<td>Johns Hopkins</td>
<td>2</td>
<td>Princeton</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
Deleting columns

Delete columns identically to deleting keys from a dictionary. One can use the `del` keyword, or pop a key.
Indexing and selection

**df.loc** selects rows by their labels. **df.iloc** selects rows by their integer labels (starting from 0).
Indexing and selection

Select columns by their names.
Indexing and selection

Select rows by their numerical indices (again 0-indexed). This supports slices.

Note: one can also select slices with lists of column names, e.g., `presidents[['JD','PhD']]`. 
## Indexing and selection

<table>
<thead>
<tr>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
<th>Nobels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>1</td>
<td>UMich</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
<td>Stanford</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>2</td>
<td>Columbia</td>
</tr>
<tr>
<td>Wilson</td>
<td>NaN</td>
<td>Johns Hopkins</td>
<td>2</td>
<td>Princeton</td>
</tr>
</tbody>
</table>

```python
1 presidents['JD']
```

<table>
<thead>
<tr>
<th>JD</th>
<th>PhD</th>
<th>Terms</th>
<th>Undergrad</th>
<th>Nobels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>1</td>
<td>UMich</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
<td>Stanford</td>
</tr>
<tr>
<td>Obama</td>
<td>Harvard</td>
<td>NaN</td>
<td>2</td>
<td>Columbia</td>
</tr>
</tbody>
</table>

```python
1 presidents.loc['Obama']
```

```
JD | Harvard
PhD | NaN
Terms | 2
Undergrad | Columbia
Nobels | 1
Name: Obama, dtype: object
```

```python
1 presidents.iloc[1]
```

```
JD | NaN
PhD | NaN
Terms | 1
Undergrad | Stanford
Nobels | 0
Name: Hoover, dtype: object
```

```python
presidents.presidents['Terms']<2
```

```
JD  | PhD  | Terms | Undergrad | Nobels |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Yale</td>
<td>NaN</td>
<td>1</td>
<td>UMich</td>
</tr>
<tr>
<td>Hoover</td>
<td>NaN</td>
<td>NaN</td>
<td>1</td>
<td>Stanford</td>
</tr>
</tbody>
</table>
```

Select rows by Boolean expression.
# Indexing and selection

These expressions return Series objects.

```python
pseudidents['JD']
```

```python
pseudidents.loc['Obama']
```

```python
pseudidents.iloc[1]
```

```python
pseudidents[pseudidents['Terms']<2]
```
Indexing and selection

These expressions return Series objects.

These expressions return DataFrames.

More on indexing:
Arithmetic with DataFrames

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

df1 = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])
df1 + df2
```

Pandas tries to align the DataFrames as best it can, filling in non-alignable entries with NaN.

In this example, rows 0 through 4 and columns A through C exist in both DataFrames, so these entries can be successfully added. All other entries get NaN, because $x + \text{NaN} = \text{NaN}$. 
By default, Series are aligned to DataFrames via row-wise broadcasting.

`df.iloc[0]` is a Series representing the 0-th row of `df`. When we try to subtract it from `df`, pandas forces dimensions to agree by broadcasting the operation across all rows of `df`. 
Arithmetic with DataFrames

Scalar addition and multiplication works in the obvious way. DataFrames also support scalar division, exponentiation… Basically every numpy ufunc.

DataFrames also support entrywise Boolean operations.
Arithmetic with DataFrames

pandas DataFrames support numpy-like any and all methods.

Just like numpy, direct Boolean operations are not supported.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1.331635</td>
<td>-0.500870</td>
</tr>
<tr>
<td>1</td>
<td>1.111157</td>
<td>0.293138</td>
</tr>
<tr>
<td>2</td>
<td>-0.669850</td>
<td>0.456863</td>
</tr>
<tr>
<td>3</td>
<td>0.216643</td>
<td>-0.636942</td>
</tr>
</tbody>
</table>

```python
(df > 0).any()
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>True</td>
</tr>
<tr>
<td>B</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>dtype: bool</td>
</tr>
</tbody>
</table>

```python
(df > 0).all()
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>False</td>
</tr>
<tr>
<td>B</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>dtype: bool</td>
</tr>
</tbody>
</table>
Arithmetic with DataFrames

The `values` attribute stores the entries of the table in a numpy array. This is occasionally useful when you want to stop dragging the extra information around and just work with the numbers in the table.

```
array([[ 1.33163456, -0.50087024],
       [ 1.1115689 ,  0.29313846],
       [-0.66984966,  0.45686335],
       [ 0.21664278, -0.63694229]])
```
Arithmetic with DataFrames

DataFrames support entrywise multiplication. The \( T \) attribute is the transpose of the DataFrame.

DataFrames also support matrix multiplication via the `numpy-like dot` method. The DataFrame dimensions must be conformal, of course.

Note: Series also support a `dot` method, so you can compute inner products.
Removing NaNs

**DataFrame** `dropna` method removes rows or columns that contain NaNs.

**axis** argument controls whether we act on rows, columns, etc.

**how=‘any’** will remove all rows/columns that contain even one NaN. **how=‘all’** removes rows/columns that have all entries NaN.
pandas supports read/write for a wide range of different file formats. This flexibility is a major advantage of pandas.

### Reading/writing files

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

Reading/writing files

<table>
<thead>
<tr>
<th>Format Type</th>
<th>Data Description</th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>CSV</td>
<td>read_csv</td>
<td>to_csv</td>
</tr>
<tr>
<td>text</td>
<td>JSON</td>
<td>read_json</td>
<td>to_json</td>
</tr>
<tr>
<td>text</td>
<td>HTML</td>
<td>read_html</td>
<td>to_html</td>
</tr>
<tr>
<td>text</td>
<td>Local clipboard</td>
<td>read_clipboard</td>
<td>to_clipboard</td>
</tr>
<tr>
<td>binary</td>
<td>MS Excel</td>
<td>read_excel</td>
<td>to_excel</td>
</tr>
<tr>
<td>binary</td>
<td>Mspack</td>
<td>read_msgpack</td>
<td>to_msgpack</td>
</tr>
<tr>
<td>binary</td>
<td>Stata</td>
<td>read_stata</td>
<td>to_stata</td>
</tr>
<tr>
<td>binary</td>
<td>SAS</td>
<td>read_sas</td>
<td>to_sas</td>
</tr>
<tr>
<td>binary</td>
<td>Python Pickle Format</td>
<td>read_pickle</td>
<td>to_pickle</td>
</tr>
<tr>
<td>SQL</td>
<td>SQL</td>
<td>read_sql</td>
<td>to_sql</td>
</tr>
<tr>
<td>SQL</td>
<td>Google Big Query</td>
<td>read_gbq</td>
<td>to_gbq</td>
</tr>
</tbody>
</table>

*pandas* supports read/write for a wide range of different file formats. This flexibility is a major advantage of *pandas*.

*pandas* file I/O is largely similar to R `read.table` and similar functions, so I’ll leave it to you to read the *pandas* documentation as needed.

Table credit: [https://pandas.pydata.org/pandas-docs/stable/io.html](https://pandas.pydata.org/pandas-docs/stable/io.html)
Summarizing DataFrames

pd.read_csv() reads a comma-separated file into a DataFrame.

info() method prints summary data about the DataFrame. Number of rows, column names and their types, etc.

Note: there is a separate to_string() method that generates a string representing the DataFrame in tabular form, but this usually doesn’t display well if you have many columns.
Summarizing DataFrames

**head()** method displays just the first few rows of the DataFrame (5 by default; change this by supplying an argument). **tail()** displays the last few rows.

Note: R and pandas both supply **head/tail** functions, named after UNIX/Linux commands that displays the first/last lines of a file.
Comparing DataFrames

These two DataFrames *ought* to be equivalent...

...but they aren’t.
Comparing DataFrames

These two DataFrames *ought* to be equivalent...

...but they aren’t.

The problem comes from the fact that NaNs are not equal to one another.

**Solution:** DataFrames have a separate `equals()` method for checking the kind of equality that we meant above.
Comparing DataFrames

There is a solid design principle behind this. If there are NaNs in our data, we want to err on the side of being overly careful about what operations we perform on them. We see similar ideas in numpy and in R.

**Solution:** DataFrames have a separate `equals()` method for checking the kind of equality that we meant above.
Statistical Operations on DataFrames

Getting means of DataFrame rows/columns using numpy is possible, but tedious.

```
DataFrame.mean method is a cleaner way to do the same thing. Argument picks out which axis to take means on: rows (1) or columns (0).
```
Statistical Operations on DataFrames

Of course, DataFrames also support a bunch of related functions, that work similarly: sum, min, max, std, var etc. All of these functions take an optional Boolean argument skipna. If True, NaNs are **not included** in the computation. If False, NaNs are included (which can mean either that the computation doesn’t work at all, or changes the value only slightly). More information: https://pandas.pydata.org/pandas-docs/stable/basics.html#descriptive-statistics

DataFrame.mean method is a cleaner way to do the same thing. Argument picks out which axis to take means on: rows (1) or columns (0).
Summarizing DataFrames

`DataFrame.describe()` is similar to the R `summary()` function. Non-numeric data will get statistics like counts, number of unique items, etc. If a DataFrame has mixed types (both numeric and non-numeric), the non-numeric data is excluded by default.

Details and optional arguments:
https://pandas.pydata.org/pandas-docs/stable/basics.html#summarizing-data-describe
Row- and column-wise functions: `apply()`

`DataFrame.apply()` takes a function and applies it to each column of the DataFrame.

Axis argument is 0 by default (column-wise). Change to 1 for row-wise application.
Row- and column-wise functions: `apply()`

Numpy ufuncs take vectors and spit out vectors, so using `df.apply()` to apply a ufunc to every row or column in effect ends up applying the ufunc to every element.
Row- and column-wise functions: `apply()`

We can pass positional and keyword arguments into the function via `df.apply`. `Args` is a tuple of the positional arguments (in order), followed by the keyword arguments.

```
def quadratic(x, a, b, c=1):
    return a*x**2 + b*x + c
```

```
df.apply(quadratic, args=(1,2), c=5)
```

Note: "`apply()` takes an argument `raw` which is `False` by default, which converts each row or column into a Series before applying the function. When set to `True`, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality." This can be useful if your function is meant to work specifically with Series.
Element-wise function application

```
1 df = pd.DataFrame({'A': ['cat', 'dog', 'bird'],
                   'B': ['unicorn','chupacabra','pixie']})
1 df

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>cat</td>
<td>unicorn</td>
</tr>
<tr>
<td>1</td>
<td>dog</td>
<td>chupacabra</td>
</tr>
<tr>
<td>2</td>
<td>bird</td>
<td>pixie</td>
</tr>
</tbody>
</table>
```

This causes an error, because `apply` thinks that its argument should be applied to Series (i.e., columns), not to individual entries.
Element-wise function application

applymap works similarly to Python’s map function (and the Series map method). Applies its argument function to every entry of the DataFrame.
Here we have a function composition applied to a DataFrame. This is perfectly valid code, but pandas supports another approach.
Tablewise Function Application

The DataFrame `pipe` method is built for a pattern called **method chaining**. The `pipe` method has better support for passing additional arguments around than does the function composition to the right. This pattern using `pipe` is also more conducive to functional programming patterns.
Recap

Previously: basics of pandas
Series and DataFrames
Indexing, changing entries
Function application

Next: more complicated operations
Statistical computations
Group-By operations
Reshaping, stacking and pivoting
Recap

Previously: basics of pandas
Series and DataFrames
Indexing, changing entries
Function application

Next: more complicated operations
Statistical computations
Group-By operations
Reshaping, stacking and pivoting

Caveat: pandas is a large, complicated package, so I will not endeavor to mention every feature here. These slides should be enough to get you started, but there’s no substitute for reading the documentation.
Pct_change over time

The `pct_change` method is supported by both Series and DataFrames. Series.pct_change returns a new Series representing the step-wise percent change.

Note: pandas has extensive support for time series data, which we mostly won’t talk about in this course. Refer to the documentation for more.
Percent change over time

`pct_change` operates on columns of a DataFrame, by default. Periods argument specifies the time-lag to use in computing percent change. So periods=2 looks at percent change compared to two time steps ago.

`pct_change` includes control over how missing data is imputed, how large a time-lag to use, etc. See documentation for more detail: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.pct_change.html
Computing covariances

`cov` method computes covariance between a Series and another Series.

```python
1 s1 = pd.Series(np.random.randn(1000))
2 s2 = pd.Series(0.1*s1+np.random.randn(1000))
3 s1.cov(s2)
```

0.1522727637202401

`cov` method is also supported by DataFrame, but instead computes a new DataFrame of covariances between columns.

`cov` supports extra arguments for further specifying behavior:
Pairwise correlations

Dataframe `corr` method computes correlations between columns (use `axis` keyword to change this behavior). Method argument controls which correlation score to use (default is Pearson’s correlation.)
Ranking data

The `rank` method returns a new Series whose values are the data ranks. Ties are broken by assigning the mean rank to both values.
By default, `rank` ranks columns of a DataFrame individually.

Rank rows instead by supplying an `axis` argument.

**Note:** more complicated ranking of whole rows (i.e., sorting whole rows rather than sorting columns individually) is possible, but requires we define an ordering on Series.
Aggregating data

This command creates time series data, with rows indexed by year-month-day timestamps.

Supplying a list of functions to `agg` will apply each function to each column of the DataFrame, with each function getting a row in the resulting DataFrame.

`agg` is an alias for the method `aggregate`. Both work exactly the same.
Aggregating data

`agg` can, alternatively, take a dictionary whose keys are column names, and values are functions.

Note that the values here are strings, not functions! `pandas` supports dispatch on strings. It recognizes certain strings as referring to functions. `apply` supports similar behavior.
Aggregating data

```
df = pd.DataFrame({'A': [1, 2, 3],
                   'B': [1., 2., 3.],
                   'C': ['foo', 'bar', 'baz']})
```

*df contains mixed data types.*

```
    A   B C
   0  1.0 foo
   1  2.0 bar
   2  3.0 baz
```

*agg (and similarly apply) will only try to apply these functions on the columns of types supported by those functions.*

```
df.agg(['mean', 'max'])
```

*Note: the DataFrame `transform` method provides generally similar functionality to the `agg` method.*

```
     A   B   C
  max  3.0  3.0  foo
mean  2.0  2.0  NaN
```

*pandas doesn’t know how to compute a mean string, so it doesn’t try.*
Iterating over Series and DataFrames

Iterating over a Series gets an iterator over the values of the Series.

```
for x in s:
    print x
```

Iterating over a DataFrame gets an iterator over the column names.

```
for x in df:
    print x
```
Iterating over Series and DataFrames

The `iteritems()` method is supported by both Series and DataFrames. Returns an iterator over the key-value pairs. In the case of Series, these are (index, value) pairs. In the case of DataFrames, these are (colname, Series) pairs.

```python
for x in df.iteritems():
    print(x)
```

```
for x in df.iteritems():
    print(x)
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>-2.072339</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.587874</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-0.164436</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-1.215576</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-0.350299</td>
<td></td>
</tr>
</tbody>
</table>

Name: A, dtype: float64)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>-1.282539</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.517591</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.450398</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-0.671235</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.958805</td>
<td></td>
</tr>
</tbody>
</table>

Name: B, dtype: float64)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>-1.241128</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.394561</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-0.975424</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.394053</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.467778</td>
<td></td>
</tr>
</tbody>
</table>

Name: C, dtype: float64)
Iterating over Series and DataFrames

```python
for x in df.iterrows():
    print(x)
```

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.
Iterating over Series and DataFrames

```python
for x in df.iterrows():
    print(x)
```

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.

Note: DataFrames are designed to make certain operations (mainly vectorized operations) fast. This implementation has the disadvantage that iteration over a DataFrame is slow. It is usually best to avoid iterating over the elements of a DataFrame or Series, and instead find a way to compute your quantity of interest using a vectorized operation or a map/reduce operation.
Group By: reorganizing data

“Group By” operations are a concept from databases
Splitting data based on some criteria
Applying functions to different splits
Combining results into a single data structure

Fundamental object: pandas GroupBy objects
Group By: reorganizing data

```python
df = pd.DataFrame({'A': ['plant', 'animal', 'plant', 'plant'],
                  'B': ['apple', 'goat', 'kiwi', 'grape'],
                  'C': np.random.randn(4),
                  'D': np.random.randn(4)})

df.groupby('A')
```

DataFrame `groupby` method returns a `pandas groupby` object.
Group By: reorganizing data

Every `groupby` object has an attribute `groups`, which is a dictionary with maps group labels to the indices in the DataFrame.

In this example, we are splitting on the column `'A'`, which has two values: `‘plant’` and `‘animal’`, so the groups dictionary has two keys.
Group By: reorganizing data

Every `groupby` object has an attribute `groups`, which is a dictionary with maps group labels to the indices in the DataFrame.

In this example, we are splitting on the column `A`, which has two values: `plant` and `animal`, so the `groups` dictionary has two keys.

The important point is that the `groupby` object is storing information about how to partition the rows of the original DataFrame according to the argument(s) passed to the `groupby` method.
Group By: aggregation

<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>plant</td>
<td>apple</td>
<td>0.529326</td>
<td>-0.796997</td>
</tr>
<tr>
<td>1</td>
<td>animal</td>
<td>goat</td>
<td>-0.901377</td>
<td>-0.670747</td>
</tr>
<tr>
<td>2</td>
<td>plant</td>
<td>kiwi</td>
<td>1.203032</td>
<td>1.162924</td>
</tr>
<tr>
<td>3</td>
<td>plant</td>
<td>grape</td>
<td>-0.740208</td>
<td>1.184488</td>
</tr>
</tbody>
</table>

Split on group ‘A’, then compute the means within each group. Note that columns for which means are not supported are removed, so column ‘B’ doesn’t show up in the result.

```python
1 df.groupby('A').mean()
```

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>animal</td>
<td>-0.901377</td>
<td>-0.670747</td>
</tr>
<tr>
<td>plant</td>
<td>0.330717</td>
<td>0.516805</td>
</tr>
</tbody>
</table>
Group By: aggregation

Here we’re building a hierarchically-indexed Series (i.e., multi-indexed), recording (fictional) scores of students by major and handedness.

Suppose I want to collapse over handedness to get average scores by major. In essence, I want to group by major and ignore handedness.
Suppose I want to collapse over handedness to get average scores by major. In essence, I want to group by major and ignore handedness.

Group by the 0-th level of the hierarchy (i.e., `major`), and take means.

We could have equivalently written `groupby('major')`, here.
groupby.get_group lets us pick out an individual group. Here, we’re grabbing just the data from the ‘econ’ group, after grouping by ‘major’.
Group By: aggregation

Similar aggregation to what we did a few slides ago, but now we have a DataFrame instead of a Series.
Group By: aggregation

Similar aggregation to what we did a few slides ago, but now we have a DataFrame instead of a Series.

Groupby objects also support the `aggregate` method, which is often more convenient.
Transforming data

From the documentation: “The transform method returns an object that is indexed the same (same size) as the one being grouped.”

Building a time series, indexed by year-month-day.

Suppose we want to standardize these scores within each year.

Group the data according to the output of the key function, apply the given transformation within each group, then un-group the data.

Important point: the result of groupby.transform has the same dimension as the original DataFrame or Series.
Filtering data

```
1 sf = pd.Series([1, 1, 2, 2, 3, 3])
2 sf
0   1
1   1
2   2
3   2
4   3
5   3
dtype: int64
```

```
1 sf.groupby(sf).filter(lambda x: x.sum() > 2)
2 2
3 2
4 3
5 3
```

**From the documentation:** “The argument of filter must be a function that, applied to the group as a whole, returns True or False.”

So this will throw out all the groups with sum <= 2.

Like `transform`, the result is ungrouped.
Combining DataFrames

`pandas.concat` function concatenates DataFrames into a single DataFrame.

Repeated indices remain repeated in the resulting DataFrame.

`pandas.concat` accepts numerous optional arguments for finer control over how concatenation is performed. See the documentation for more.

Missing values get NaN.
Merges and joins

**pandas** DataFrames support many common database operations. Most notably, join and merge operations.

We’ll learn about these when we discuss SQL later in the semester. So we won’t discuss them here.

**Important:** What we learn for SQL later has analogues in **pandas**.

If you are already familiar with SQL, you might like to read this: [https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html](https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html)
Pivoting and Stacking

Data in this format is usually called **stacked**. It is common to store data in this form in a file, but once it’s read into a table, it often makes more sense to create columns for A, B and C. That is, we want to **unstack** this DataFrame.
Pivoting and Stacking

The **pivot** method takes care of unstacking DataFrames. We supply indices for the new DataFrame, and tell it to turn the variable column in the old DataFrame into a set of column names in the unstacked one.

https://en.wikipedia.org/wiki/Pivot_table
How do we stack this? That is, how do we get a non-pivot version of this DataFrame? The answer is to use the DataFrame `stack` method.

```python
tuples = list(zip(*[['bird', 'bird', 'goat', 'goat'], ['x', 'y', 'x', 'y']]))
index = pd.MultiIndex.from_tuples(tuples, names=['animal', 'cond'])
df = pd.DataFrame(np.random.randn(4, 2),
                 index=index, columns=['A', 'B'])
```

Note: we could also construct the index set using `itertools`. 

<table>
<thead>
<tr>
<th>animal</th>
<th>cond</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bird</td>
<td>x</td>
<td>0.699732</td>
<td>-1.407296</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>0.810211</td>
<td>1.249299</td>
</tr>
<tr>
<td>goat</td>
<td>x</td>
<td>-0.909280</td>
<td>0.184450</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>-0.755891</td>
<td>-0.957222</td>
</tr>
</tbody>
</table>
Pivoting and Stacking

The DataFrame `stack` method makes a stacked version of the calling DataFrame. In the event that the resulting column index set is trivial, the result is a Series. Note that `df.stack()` no longer has columns A or B. The column labels A and B have become an extra index.
Pivoting and Stacking

Here is a more complicated example. Notice that the column labels have a three-level hierarchical structure.

There are multiple ways to stack this data. At one extreme, we could make all three levels into columns. At the other extreme, we could choose only one to make into a column.
Pivoting and Stacking

Stack only according to level 1 (i.e., the animal column index).

Missing animal x cond x hair_length conditions default to NaN.
Pivoting and Stacking

Stacking across all three levels yields a Series, since there is no longer any column structure. This is often called flattening a table.

Notice that the NaN entries are not necessary here, since we have an entry in the Series only for entries of the original DataFrame.
Plotting DataFrames

def = pd.DataFrame(np.random.randn(1000, 3),
                   index=pd.date_range('1/1/2000', periods=1000),
                   columns=['cat', 'bird', 'goat'])
def = df.cumsum()
_ = df.plot()

cumsum gets partial sums, just like in numpy.

Note: this requires that you have imported matplotlib.

Note that legend is automatically populated and x-ticks are automatically date formatted.