Manipulating and analyzing data with pandas

\[ y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it} \]
Introduction

- **Pandas**: Python Data Analysis Library

- “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/)

- Sponsored by NumFOCUS, a non-profit organization in the US (like NumPy, Matplotlib, Jupyter, and Julia)

- Used in StatsModel, sklearn-pandas, Plotly, IPython, Jupyter, Spyder
  (http://pandas-docs.github.io/pandas-docs-travis/ecosystem.html)
Side remark: BSD licenses

- BSD = Berkeley Software Distribution
  The first software (an OS actually) to be distributed under BSD license
  “Permissive” license → can be used in a proprietary software

![BSD 3-Clause “New” or “Revised” License](https://github.com/pandas-dev/pandas/blob/master/LICENSE)
Introduction

• Built on top of **NumPy**

• Part of the **SciPy** ecosystem
  (Scientific Computing Tools for Python)

• Version history ([https://pandas.pydata.org/community.html#history-of-development](https://pandas.pydata.org/community.html#history-of-development))
  
  – Project initiated in 2008
  – Oldest version in the doc: 0.4.1 (September 2011)
  – Current version: 0.24.2 (March 2019)
Objectives of the presentation

- Explain when one can benefit from using pandas
- Describe the **data structures** in pandas
  - **Series** 1-dimensional array with labels
  - **DataFrame** 2-dimensional array with labels
  - **Panel** 3-dimensional array with labels
    (deprecated since version 0.20.0)
- Review the **data analysis tools** in pandas
  - Import and export data
  - Select data and reshape arrays
  - Merge, join, and concatenate arrays
  - Visualize data
  - ...
Two distinct questions

- **What is the advantage as a programmer?**
  Addressed in this presentation.

- **What is the speed of the obtained code?**
  Not addressed in this presentation. Two brief comments:
  - Pandas is an overlay on top of NumPy. Because of this, it may have a performance cost.
  - “pandas is fast. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance.”
    (http://pandas.pydata.org/pandas-docs/stable/getting_started/overview.html)
Outline

NumPy

Data structures in pandas
  Series
  DataFrame

Data analysis tools in pandas (10 minutes to pandas)
(http://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html)
Outline

NumPy

Data structures in pandas
  Series
  DataFrame

Data analysis tools in pandas (10 minutes to pandas)
(http://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html)
Overview

- **NumPy**: Numeric Python or Numerical Python
  (https://www.datacamp.com/community/tutorials/python-numpy-tutorial)

- **Data structure**: A fixed-size multidimensional array object called “ndarray”, for “N-dimensional array”, or just “array”

- **Tools**: “Routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.” (https://www.numpy.org/devdocs/user/whatisnumpy.html)

- Two important notions: **vectorizing** and **broadcasting**
Numpy ndarray structure

A fixed-size multidimensional array object

“NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.”

”The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.“

Advantage of this rigidity: (usually) contiguous block of memory → Faster code
Create an ndarray

```python
# define an array with two axes
> a = np.array([[3., 0.], [20., 230.], [21., 275.]])
array([[ 3.,  0.],
       [20., 230.],
       [21., 275.]])

> a[2,0]  # same result as a[2][0]
21.0

> a[:2, :]  # returns a view of the array
array([[ 3.,  0.],
       [20., 230.]])
```

Axis 0

Axis 1

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>20.</td>
<td>230.</td>
</tr>
<tr>
<td>2</td>
<td>21.</td>
<td>275.</td>
</tr>
</tbody>
</table>

11/50 © 2019 Nokia Public
Some attributes

> a.shape
(3, 2)
# axis 0 is of length 3, axis 1 is of length 2

> a.dtype
dtype('float64')
# data-type, specifies how to interpret each item
# (inferred from data if unspecified)

> a.itemize
8 # the size of each element of the array,
   # in bytes (8 x 8 = 64)
View: different numpy object but same data

```python
# create a view of the array
> b = a[:2, :]
array([[ 3.,  0.],
       [ 20., 230.]]

> b[0,0] = 0.

> a
array([[ 0.,  0.],
       [ 20., 230.],
       [ 21., 275.]]
```
Reshape an array

The `reshape` method returns its argument with a modified shape, whereas the `resize` method modifies the array itself:

```python
> b = a.reshape(2,3)
array([[  3.,   0.,  20.],
       [ 230.,  21.,  275.]]
> a
array([[  3.,   0.],
       [ 20., 230.],
       [ 21., 275.]]
> a.resize(2,3)
> a
array([[  3.,   0.,  20.],
       [ 230.,  21.,  275.]]
> np.resize(a, (2,3))
array([[  3.,   0.,  20.],
       [ 230.,  21.,  275.]]
```
Functionalities

- “Routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.”
  (https://www.numpy.org/devdocs/user/whatisnumpy.html)

- Summing the elements of an array
  
  ```python
  > a.sum()
  549.0
  > a.sum(axis=0)
  array([ 44., 505.])
  ```

- Taking the maximum of an array
  
  ```python
  > a.max() # and, similarly, a.max(axis=0)
  275.0
  ```
Structured arrays: addressing columns by name
(https://scipy-cookbook.readthedocs.io/items/Recarray.html)

```python
> a = np.array([(3., 0.), (20., 230.), (21., 275.)],
               dtype=np.dtype([('Age', int), ('Weight', float)]))
array([(  3,   0.), (20, 230.), (21, 275.)],
      dtype=[('Age', '<i8'), ('Weight', '<f8')])

> a['Age']
array([ 3, 20, 21])
```

Here, `a` is a 1-dimensional array with tuple elements.

```python
> a[0]
(3, 0.)
```
Outline

NumPy

Data structures in pandas
  Series
  DataFrame

Data analysis tools in pandas (10 minutes to pandas)
(http://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html)
Philosophy

- Arrays with **smaller dimensions**
  - Series: 1-dimensional
  - DataFrames: 2-dimensional

- Give a **semantical meaning** to the axes
  - Columns ≈ Variables
  - Lines ≈ Observations

- Other functionalities:
  - **Missing data**: Identified by NaN (np.nan).
  - **Mutability**: Add and remove columns in a DataFrame
  - **Data alignment**: Combine data based on the indices

<table>
<thead>
<tr>
<th>Observations</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>Weight</td>
</tr>
<tr>
<td>Bei Bei</td>
<td>3</td>
</tr>
<tr>
<td>Mei Xiang</td>
<td>20  230.</td>
</tr>
<tr>
<td>Tian Tian</td>
<td>21  275.</td>
</tr>
</tbody>
</table>
Data structures

**Series**  “One-dimensional ndarray with axis labels (including time series).”

**DataFrame**  “Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).  
Arithmetic operations align on both row and column labels.  
Can be thought of as a dict-like container for Series objects.  
The primary pandas data structure.”
Like ndarrays, the length of a Series cannot be modified after definition.

**Index**: Can be of any hashable type.

**Automatic data alignment**: “Data alignment is intrinsic. The link between labels and data will not be broken unless done so explicitly by you.”

**Missing data**: Represented as NaN (`np.nan`, a float!). “Statistical methods from ndarray have been overridden to automatically exclude missing data”.

<table>
<thead>
<tr>
<th>Weight (pounds)</th>
<th>Age (years)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis 0</td>
<td></td>
</tr>
<tr>
<td>Mei Xiang</td>
<td>230</td>
</tr>
<tr>
<td>Tian Tian</td>
<td>275</td>
</tr>
<tr>
<td>Bei Bei</td>
<td>3</td>
</tr>
<tr>
<td>Mei Xiang</td>
<td>20</td>
</tr>
<tr>
<td>Tian Tian</td>
<td>21</td>
</tr>
</tbody>
</table>
Creating a Series

```python
> s = pd.Series([3, 20, 21],
               index=['Bei Bei', 'Mei Xiang', 'Tian Tian'],
               name='Age')
Bei Bei    3
Mei Xiang  20
Tian Tian  21
Name: Age, dtype: int64
```

> s.array # ``a thin (no copy) wrapper around numpy.ndarray''
```
<ndarray>
[3, 20, 21]
Length: 3, dtype: int64
```
Some attributes

```python
> s = pd.Series([3, 20, 21],
    index=['Bei Bei', 'Mei Xiang', 'Tian Tian'],
    name='Age')

> s.dtype  # default value: inferred from data
dtype('int64')  # usually of type numpy.dtype

> s.name  # default value: None
'Age'

> s.index  # default value: RangeIndex(start=0, stop=6, step=1)
Index(['Bei Bei', 'Mei Xiang', 'Tian Tian'], dtype='object')
```
Accessing data

```python
> s = pd.Series([3, 20, 21],
                   index=['Bei Bei', 'Mei Xiang', 'Tian Tian'],
                   name='Age')

> s['Mei Xiang']  # same as s[1]
20

> s['Mei Xiang':'Tian Tian']  # same as s[1:]  
Mei Xiang   20
Tian Tian   21
Name: Age, dtype: int64
```
Creating a View

```python
> t = s['Mei Xiang':'Tian Tian']  # another Series with the same data
Mei Xiang   20
Tian Tian   21
Name: Age, dtype: int64

> t['Tian Tian'] = 22

> s
Bei Bei    3
Mei Xiang  20
Tian Tian  22
Name: Age, dtype: int64
```
Adding two series (with automatic data alignment)

```python
> u = pd.Series([230., 275.],
    index=['Mei Xiang', 'Tian Tian'],
    name='Weight')
Mei Xiang 230.0
Tian Tian 275.0
Name: Weight, dtype: float64

> s.add(u)  # same as s + u, also the default in NumPy
Bei Bei NaN
Mei Xiang 250.0
Tian Tian 296.0
dtype: float64
```
Adding two series (with automatic data alignment)

```python
> u = pd.Series([230., 275.],
    index=['Mei Xiang', 'Tian Tian'],
    name='Weight')
Mei Xiang  230.0
Tian Tian  275.0
Name: Weight, dtype: float64

> s.add(u, fill_value=0)
Bei Bei   3.0
Mei Xiang 250.0
Tian Tian 296.0
dtype: float64
```
**DataFrame**


- “Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).”

- **Mutability**: Columns can have different dtypes and can be added and removed, but they have a fixed size.

- **Semantic**: Similar to a table in a relational database. Like in R language (https://en.wikipedia.org/wiki/R_(programming_language))
  - Columns ≈ Variables
  - Rows ≈ Observations of these variables
A dictionary of Series

- Column labels ≈ keys, columns ≈ values

- “You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations”.  

- In particular: access by key, `del`, `pop`.

<table>
<thead>
<tr>
<th></th>
<th>Age</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MX</td>
<td>20</td>
<td>230.</td>
</tr>
<tr>
<td>TT</td>
<td>21</td>
<td>275.</td>
</tr>
</tbody>
</table>

```python
BB  MX  TT
3   230. 275.
```
Creating a DataFrame

```python
> df = pd.DataFrame({
  'Age': [3, 20, 21],
  'Weight': [np.nan, 230., 275.],
},
index=['Bei Bei', 'Mei Xiang', 'Tian Tian'])
```

<table>
<thead>
<tr>
<th></th>
<th>Age</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bei Bei</td>
<td>3</td>
<td>NaN</td>
</tr>
<tr>
<td>Mei Xiang</td>
<td>20</td>
<td>230.0</td>
</tr>
<tr>
<td>Tian Tian</td>
<td>21</td>
<td>275.0</td>
</tr>
</tbody>
</table>

```python
> df.dtypes  # returns a Series
Age       int64
Weight    float64
dtype:    object
```

In general: list ≈ rows, dictionary ≈ columns.
Other attributes

> df.shape
(3, 2)

> df.size
6

> df.columns
Index(['Age', 'Weight'], dtype='object')

> df.index
Index(['Bei Bei', 'Mei Xiang', 'Tian Tian'], dtype='object')
Accessing data

There are many ways of accessing data (loc, iloc, at, iat).

```python
> df['Age']  # a column (Series)
Bei Bei 3
Mei Xiang 20
Tian Tian 21
Name: Age, dtype: int64
```

```python
> df['Mei Xiang':'Tian Tian']  # a range of rows (DataFrame)
Weight Age
Mei Xiang 230.0 20
Tian Tian 275.0 21
```

```python
> df['Mei Xiang']  # KeyError
```
Summing over columns and rows

Returns a Series
Excludes NaN values by default (**skipna**=**True**)  

```python
> df.sum()  # same as df.sum(axis=0)
Age    44.0
Weight 505.0
dtype: float64
```

```python
> df.sum(axis=1)
Bei Bei    3.0
Mei Xiang 250.0
Tian Tian 296.0
dtype: float64
```
Outline

NumPy

Data structures in pandas
  Series
  DataFrame

Data analysis tools in pandas (10 minutes to pandas)
(http://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html)
Overview

- Review the data analysis tools provided by pandas.
- The organization and most of the examples of this section come from the official tutorial **10 minutes to pandas**.  
  (http://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html)
- Some examples originate from the **User Guide**.  
Object creation

```python
> dates = pd.date_range('20130101', periods=5)
               dtype='datetime64[ns]', freq='D')
```

```python
> df = pd.DataFrame(np.random.randn(5, 4), index=dates,
                   columns=list('ABCD'))
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>1.501942</td>
<td>-1.459551</td>
<td>-0.376242</td>
<td>0.410211</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>0.803188</td>
<td>-0.651458</td>
<td>1.457657</td>
<td>1.575324</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.398711</td>
<td>-0.496614</td>
<td>1.032707</td>
<td>0.343666</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.101690</td>
<td>0.982808</td>
<td>-1.049312</td>
<td>0.535201</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.271261</td>
<td>-0.557231</td>
<td>0.307699</td>
<td>0.626503</td>
</tr>
</tbody>
</table>
Viewing Data

```r
> df.head(3)  # default value: 5

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01</td>
<td>1.501942</td>
<td>-1.459551</td>
<td>-0.376242</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>0.803188</td>
<td>-0.651458</td>
<td>1.457657</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>-0.398711</td>
<td>-0.496614</td>
<td>1.032707</td>
</tr>
</tbody>
</table>

> df.tail(3)  # default value: 5

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-03</td>
<td>-0.398711</td>
<td>-0.496614</td>
<td>1.032707</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.101690</td>
<td>0.982808</td>
<td>-1.049312</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>-0.271261</td>
<td>-0.557231</td>
<td>0.307699</td>
</tr>
</tbody>
</table>
```
Viewing Data

> df.index
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05'], dtype='datetime64[ns]', freq='D')

> df.columns
Index(['A', 'B', 'C', 'D'], dtype='object')

> df.A # same as df['A']
2013-01-01  0.000000
2013-01-02  -0.718574
...
2013-01-05  0.593794
Freq: D, Name: A, dtype: float64
Viewing Data

to_numpy returns an ndarray that contains the data.
May require a copy if the data is heterogeneous.

```python
> a = df.to_numpy()
array([[ -2.35406005,  -0.31282731,   0.19482154,   1.14387112],
       [  1.70706975,  -0.78209048,   0.06241179,  -0.00753477],
       [ -0.21252435,   0.06799263,   1.03563884,  -0.67680038],
       [  0.65801543,  -0.39368803,   0.56542520,  -1.32672643],
       [-1.30699305,  -0.06174394,   0.09464223,  -0.97696831]])

> a[0,0] = 0
> df['A'][0]
0.0
```
Selection

Obtain a view of the data. By **label** (loc, at), by **index** (iloc, iat), or by **boolean indexing**.

```R
> df[df.A > 0]  # show the rows where A is positive
          A    B       C       D
2013-01-05 0.593794 -0.191118  0.622146  1.325086

> df = df[df > 0]  # replaces non-positive values with NaN
         A    B       C       D
2013-01-01 NaN  1.112209  0.277689  1.300440
2013-01-02 NaN    NaN  1.728119    NaN
2013-01-03 NaN    NaN  0.056013  0.970420
2013-01-04 NaN  0.364966    NaN    NaN
2013-01-05 0.593794    NaN  0.622146  1.325086
```
Missing Data

Remove or replace missing data:

- `df.dropna`: Deletes columns or rows that contain missing values (NaN).
- `df.fillna`: Fills the NaN with the provided value.
- `df.isna` or `pd.isna(df)`: Returns a DataFrame of the same size as `df` with boolean values that say if the original value in `df` is NaN.

```python
> df.fillna(value=0)
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-01-01 0.00000</td>
<td>1.171513</td>
<td>0.000000</td>
<td>0.298407</td>
</tr>
<tr>
<td>2013-01-02 0.00000</td>
<td>0.893041</td>
<td>2.136786</td>
<td>0.000000</td>
</tr>
<tr>
<td>2013-01-03 0.00000</td>
<td>0.030041</td>
<td>0.131783</td>
<td>0.000000</td>
</tr>
<tr>
<td>2013-01-04 0.46075</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>2013-01-05 0.00000</td>
<td>0.953238</td>
<td>0.778675</td>
<td>1.109996</td>
</tr>
</tbody>
</table>
Getting Data In/Out

The basic functions are `df.to_csv` and `df.read_csv`.

```python
> df.to_csv('foo.csv')

> df = pd.read_csv('foo.csv', index_col=0)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>2013-01-01</td>
<td>NaN</td>
<td>1.171513</td>
<td>NaN</td>
<td>0.298407</td>
</tr>
<tr>
<td>2013-01-02</td>
<td>NaN</td>
<td>0.893041</td>
<td>2.136786</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-03</td>
<td>NaN</td>
<td>0.030041</td>
<td>0.131783</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-04</td>
<td>0.46075</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2013-01-05</td>
<td>NaN</td>
<td>0.953238</td>
<td>0.778675</td>
<td>1.109996</td>
</tr>
</tbody>
</table>
```
Hierarchical indexing (MultiIndex)
(http://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html)

Paves the way for higher dimensional data.

Example with two levels of indices:

```python
> arrays = [np.array(['bar', 'bar', 'foo', 'foo']),
            np.array(['one', 'two', 'one', 'two'])]

> df = pd.DataFrame(np.random.randn(4, 3), index=arrays)

        0   1   2
bar one -0.783896 -1.033699 0.113092
  two  0.376749 -0.617641 1.858707
foo one -0.345071  0.288537 0.251429
two  1.391096 -0.053008 -1.290041
```
Various operations

```
> df.mean()  # same as df.mean(axis=0)
A   0.593794
B   0.738587
C   0.670992
D   1.198649
dtype: float64
```

Other examples:

- **sub**  Subtracts another Series or DataFrame (broadcasting)
- **apply**  Applies a function such as `np.cumsum`
- **value_counts**  Counts the number of occurrences of each value
Gather Series or DataFrames

Concatenate:

concat  General-purpose function
        Concatenate Series or DataFrames along columns or rows

append  Append rows to a DataFrame
        Equivalent to concat along axis 0

Join / Merge:

merge   Database-like join operations
Grouping

> df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar'],
                     'B': np.random.randn(4), 'C': np.random.randn(4)})

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>0</td>
<td>foo</td>
<td>-0.124831</td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>-0.755884</td>
</tr>
<tr>
<td>2</td>
<td>foo</td>
<td>-0.736155</td>
</tr>
<tr>
<td>3</td>
<td>bar</td>
<td>-0.318638</td>
</tr>
</tbody>
</table>

> df.groupby('A').sum() # the elements of A are the indices

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bar</td>
<td>-1.074521</td>
<td>2.653020</td>
</tr>
<tr>
<td>foo</td>
<td>-0.860986</td>
<td>0.791651</td>
</tr>
</tbody>
</table>
Time Series

```python
rng = pd.date_range('1/1/2012', periods=4, freq='S')

ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
Freq: S, dtype: int64

ts.resample('2S').sum()
Freq: 2S, dtype: int64
```

Support for time zone representation, converting to another time zone, and converting between time span representations.
Categoricals
(http://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html)

Similar to categorical variables used in statistics. Practical for saving memory and sorting data. “Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.”

```python
> s = pd.Series(["a", "b", "c", "a"], dtype="category")
0   a
1   b
2   c
3   a
dtype: category
Categories (3, object): [a, b, c]
```
Plotting

Based on the API matplotlib.pyplot.

```python
> ts = pd.Series(np.random.randn(1000),
               index=pd.date_range('1/1/2000', periods=1000))

> ts.head(3)
2000-01-01   0.310037
2000-01-02   1.747102
2000-01-03  -2.121889
Freq: D, dtype: float64

> ts = ts.cumsum()
> ts.plot()
```
Resources

Python Data Analysis Library (pandas)

- Documentation
- GitHub awesome-pandas
  Links towards videos, cheat sheets, tutorials, books, and papers
    → Video Pandas from the Inside by Stephen Simmons at PyData 2017
      About implementation and performance
    → Cheat sheet Data Wrangling with pandas

Numerical Python (NumPy)

- What makes Numpy Arrays Fast: Memory and Strides by Jessica Yung
- From Python to Numpy by Nicolas P. Rougier
Resources

Bei Bei, Mei Xiang, and Tian Tian

- Smithsonian’s National Zoo’s Panda Cams
- Their age and weight were found on Wikipedia