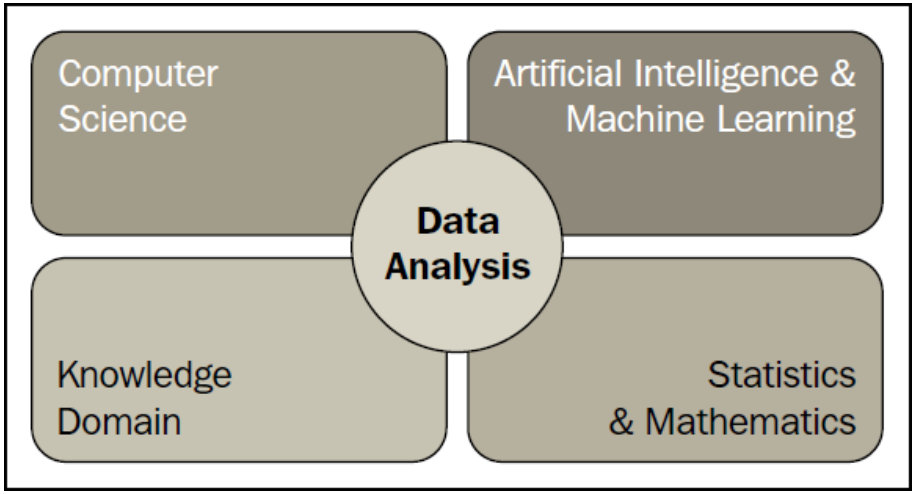
Section 3

Python Lirbrary for Data Analysis

Fityanul Akhyar, S.T., M.T
School of Electrical Engineering Telkom University

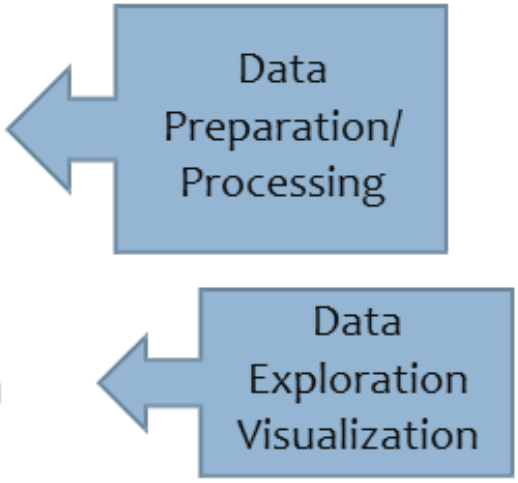
About Data Analysis

- Data analysis is the process in which raw data is ordered and organized, to be used in methods that help to explain the past and predict the future.
- **Data Analysis** is a multidisciplinary field, which combines **Computer Science, Artificial Intelligence & Machine Learning, Statistics & Mathematics,** and **Knowledge Domain.**



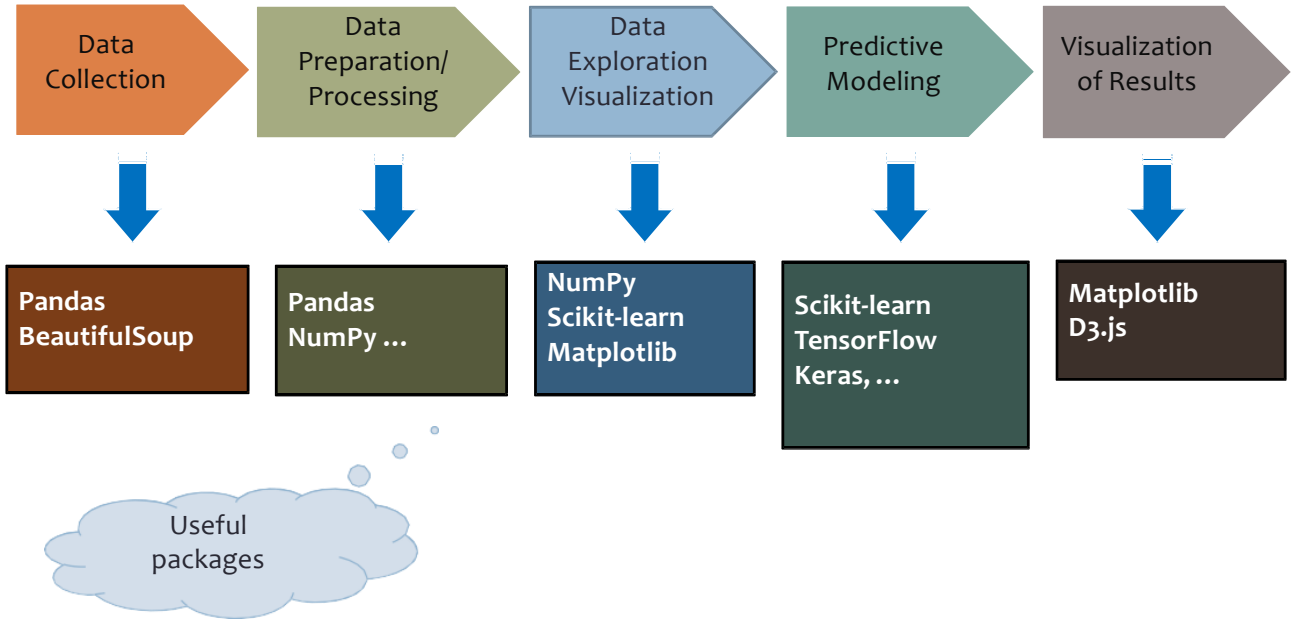
Data Analysis Process

- The data analysis process is composed of the following steps:
  - Understand the problem
  - Obtain your data
  - Clean the data
  - Normalize the data
  - Transform the data
  - Exploratory statistics
  - Exploratory visualization
  - Predictive modeling
  - Validate your model
  - Visualize and interpret your results
  - Deploy your solution



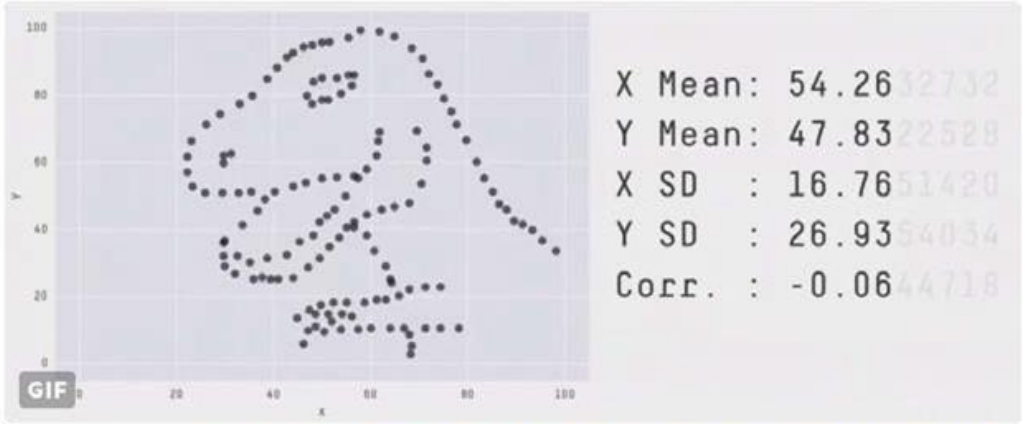
Data Analysis Process

- All these activities can be grouped as …



Importance of data visualization

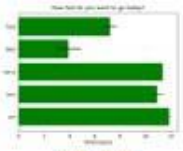
- Visualization almost always presents a more informative view of your data than statistics (the noun, not the field).

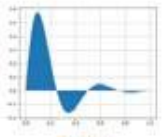


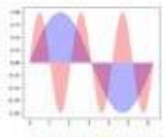
[Source: https://twitter.com/JustinMatejka/status/770682771656368128

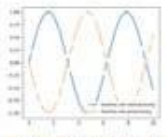
- Many kinds of data visualizations are available such as bar chart, histogram, line chart, pie chart, heat maps and so on, for one variable, two variables, and many variables in one, two, or three dimensions.

Matplotlib Gallery

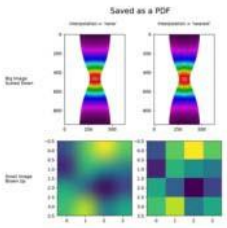

barh_demo


fill_demo

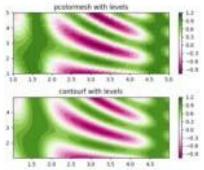

fill_demo_features


line_demo_dash_control

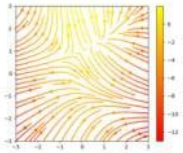

line

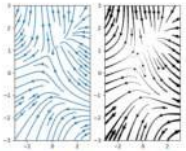


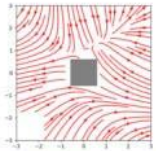
interpolation_none_vs_nearest

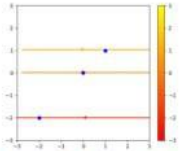


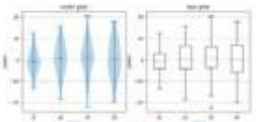
pcolormesh_levels

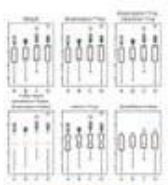

streamplot_demo_features

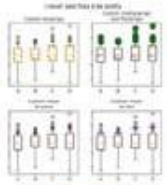

streamplot_demo_features

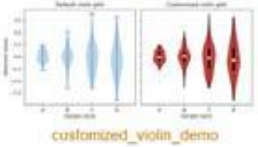

streamplot_demo_masking

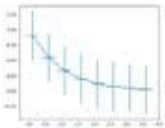

streamplot_demo_start_points

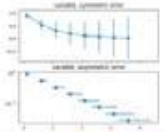

boxplot_vs_violin_demo

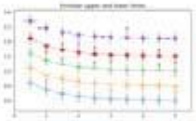

bxp_demo

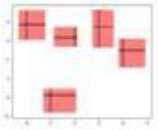

bxp_demo

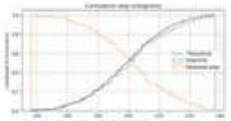

customized_violin_demo


errorbar_demo


errorbar_demo_features

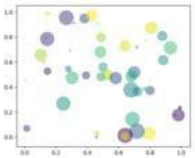

errorbar_limits


errorbars_and_boxes


histogram_demo_cumulative

artist_reference


path_patch_demo


scatter_demo

Open Data

- Taiwan Government's open data, https://data.gov.tw/
- U.S. Government's open data, https://www.data.gov/
- Kaggle, https://www.kaggle.com/
- Stanford Large Network Dataset Collection, https://snap.Stanford.edu/data/
- UC Irvine Machine Learning Repository, http://archive.ics.uci.edu/ml/
- Datahub, http://datahub.io/
- World Bank Open Data, http://data.worldbank.org/

Data Loading

- You can get data in one of several ways:
  - Directly download a data file (or files) manually
  - Query data from a database
  - Query an API (usually web-based)
  - Scrap data from a webpage
- We will discuss how to load data from standard formats such as
  - **CSV** (Comma-Separated Values) files
  - **JSON** (JavaScript Object Notation) files and strings
  - **HTML/XML** (hypertext markup language / extensible markup language) files and strings

Data formats: CSV

- **CSV** (Comma-Separated Values)
  - CSV is a very simple and common open format for table.
  - CSV is a plain text format this means that the file is a sequence of characters, with no data that has to be interpreted instead, for example, binary numbers.
  - Rows are called *tuples (or records)*

```
id,typeTwo,name,type
 001, Poison, Bulbasaur, Grass
 002, Poison, Ivysaur, Grass
 003, Poison, Venusaur, Grass
 006, Flying, Charizard, Fire
 012, Flying, Butterfree, Bugx
```

The first five records of the CSV file (pokemon.csv). First line is header.

- *You can read a csv file using csv module, NumPy, or pandas*

Parsing CSV using csv module

```
In [25]: import csv
In [26]: with open("pokemon.csv") as f:
    ...:     data = csv.reader(f)
    ...:     for line in data:
    ...:         print("id: {0} , typeTwo: {1}, name: {2}, type: {3}"
    ...:               .format(line[0],line[1],line[2],line[3]))
```

```
Output:
id: id , typeTwo: typeTwo, name: name, type: type
id: 001 , typeTwo: Poison, name: Bulbasaur, type: Grass
id: 002 , typeTwo: Poison, name: Ivysaur, type: Grass
…
```

header

```
In [3]: line
Out[3]: [' 649', ' Steel', ' Genesect', ' Bug']
```

The result is a list of str

Parsing CSV using NumPy

```
In [27]: import numpy as np
In [28]: data = np.genfromtxt("pokemon.csv"
    ...:                      ,skip_header=1
    ...:                      ,dtype=None
    ...:                      ,delimiter=',')
In [29]: print(data)
```

Python Style Rules

Skip header

The result is a list of tuples

Output:

```
[(1, b' Poison', b' Bulbasaur', b' Grass')
(2, b' Poison', b' Ivysaur', b' Grass')
(3, b' Poison', b' Venusaur', b' Grass')
(6, b' Flying', b' Charizard', b' Fire')
(12, b' Flying', b' Butterfree', b' Bug')
(13, b' Poison', b' Weedle', b' Bug')
```

Parsing CSV using NumPy

```
numpy.genfromtxt(fname, dtype=<type 'float'>, comments='#', delimiter=None,
skip_header=0, skip_footer=0, converters=None, missing_values=None,
filling_values=None, usecols=None, names=None, excludelist=None, deletechars=None,
replace_space='_', autostrip=False, case_sensitive=True, defaultfmt='f%i',
unpack=None, usemask=False, loose=True, invalid_raise=True, max_rows=None)
```

- Load data from a text file, with missing values handled as specified.
  - https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html
- **dtype** : dtype, optional
  - Data type of the resulting array. If None, the dtypes will be determined by the contents of each column, individually.
- **skip_header** : int, optional, The number of lines to skip at the beginning of the file.
- **usecols** : sequence, optional
  - Which columns to read, with **0** being the first. For example, usecols = (1, 4, 5) will extract the 2nd, 5th and 6th columns.

Parsing CSV using pandas

- *pandas* is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language
- The pandas library also offers similar functions to load MS Excel, HDFS, SQL, JSON, HTML, and Stata datasets.
  - http://pandas.pydata.org/

pandas.DataFrame

- Two-dimensional size-mutable, potentially heterogeneous **tabular data structure** with labeled axes (rows and columns).
- http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

```
In [4]: import pandas as pd
In [5]: dataframe = pd.read_csv('pokemon.csv')
In [6]: dataframe
Out[6]:
id typeTwo name type
1 1 Poison Bulbasaur Grass
2 2 Poison Ivysaur Grass
3 3 Poison Venusaur Grass
…
…
295  648   Fighting     Meloetta     Normal
296649        Steel     Genesect        Bug
[297 rows x 4 columns]
```

Parsing CSV using pandas

- **Parsing a CSV file using pandas**

```
import pandas as pd
dataframe = pd.read_csv('DATE0730.csv', encoding='big5')
```

Chinese with big5 encoding.

```
date,time,qe,surprise,VIX,CL前10分,CL後20分,CL前15分,CL後105分,HO前10分,HO後20分,HO前15分,HO後105分,NG前10分,NG後20分,NG前15分,NG後105分
20081125,815,1,0.552394541,60.9,,,,,,,,,,,,,,,,
20081201,1340,1,1.181464064,68.51,,,,,,,,,,,,,,,,
```

- List of Python standard encodings
  - https://docs.python.org/3/library/codecs.html#standard-encodings

Parsing CSV using pandas

- 檔名路徑含有中文，read_csv()會有問題
  - df = pd.read_csv("D:/class/人數.csv")
  - OSError: Initializing from file failed
- Solution: 用open 開檔
  - f= open("D:/class/人數.csv")
  - df= pd.read_csv(f)
  - f.close()

Parsing CSV using pandas

- **pandas.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='.', lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=False, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True, delim_whitespace=False, as_recarray=False, compact_ints=False, use_unsigned=False, low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)

  http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html#pandas.read_csv

Pandas Input/Output

- **JSON**

read_json([path_or_buf, orient, typ, dtype, ...])	Convert a JSON string to pandas object
json_normalize(data[, record_path, meta, ...])	"Normalize" semi-structured JSON data into a flat table

- **Flat File**

read_table(filepath_or_buffer[, sep, ...])	Read general delimited file into DataFrame
read_csv(filepath_or_buffer[, sep, ...])	Read CSV (comma-separated) file into DataFrame
read_fwf(filepath_or_buffer[, colspecs, widths])	Read a table of fixed-width formatted lines into DataFrame
read_msgpack(path_or_buf[, encoding, iterator])	Load msgpack pandas object from the specified

- **Excel**

read_excel(io[, sheetname, header, ...])	Read an Excel table into a pandas DataFrame
ExcelFile.parse([sheetname, header, ...])	Parse specified sheet(s) into a DataFrame

- http://pandas.pydata.org/pandas-docs/stable/api.html#input-output

Pandas Input/Output

- **HTML**

read_html(io[, match, flavor, header, ...])	Read HTML tables into a list of DataFrame objects.

- **HDFStore: PyTables (HDF5)**

read_hdf(path_or_buf[, key])	read from the store, close it if we opened it
HDFStore.put(key, value[, format, append])	Store object in HDFStore
HDFStore.append(key, value[, format, ...])	Append to Table in file.
HDFStore.get(key)	Retrieve pandas object stored in file
HDFStore.select(key[, where, start, stop, ...])	Retrieve pandas object stored in file, optionally based on where

- **SQL**

read_sql_table(table_name, con[, schema, ...])	Read SQL database table into a DataFrame.
read_sql_query(sql, con[, index_col, ...])	Read SQL query into a DataFrame.
read_sql(sql, con[, index_col, ...])	Read SQL query or database table into a DataFrame.

pandas Data Structures: Series

- Series: A Series is a **one-dimensional** array-like object containing an array of data and an associated array of data labels, called its *index*.
- Constructing Series objects

```
pd.Series(data, index=index)
```

where index is an optional argument, and data can be one of many entities.

```
In [24]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
In [25]: data
Out[25]:
0 0.25
1 0.50
2 0.75
3 1.00
dtype: float64
```

we can access with the values and index attributes.

```
In [26]: data.values
Out[26]: array([ 0.25, 0.5 , 0.75, 1. ])
```

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html#pandas.Series

pandas Data Structures: Series

- *Explicitly defined* index associated with the values
  - The index need not be an integer, but can consist of values of any desired type.

```
In [27]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
    ...:                  index=['a', 'b', 'c', 'd'])

In [28]: data
Out[28]:
a 0.25
b 0.50
c 0.75
d 1.00
dtype: float64
```

The item access works as expected:

```
In [29]: data['b']
Out[29]: 0.5
```

pandas Data Structures: Series

- A Pandas Series is like a specialization of a Python dictionary.
  - A dictionary is a structure that maps arbitrary keys to a set of arbitrary values.
  - A Series is a structure that maps typed keys to a set of typed values.

```
In [30]: population_dict = {'California': 38332521,
    ...:                    'Texas': 26448193,
    ...:                    'New York': 19651127,
    ...:                    'Florida': 19552860,
    ...:                    'Illinois': 12882135}
In [31]: population = pd.Series(population_dict)
In [32]: population
Out[32]:
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
dtype: int64
```

pandas Data Structures: Series

- By default, a Series will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed:

```
In [33]: population['California']
Out[33]: 38332521
```

- Unlike a dictionary, though, the Series also supports array-style operations such as slicing:

```
In [34]:
population['California':'Illinois']
Out[34]:
California 38332521
Florida    19552860
Illinois   12882135
dtype: int64
```

pandas Data Structures: Series

- A Series's index can be altered in place by assignment

```
In [6]: population.index = ['CA', 'FL', 'IL', 'NY', 'TX']
In [7]: population
Out[7]:
CA    38332521
FL    19552860
IL    12882135
NY    19651127
TX    26448193
dtype: int64
```

pandas Data Structures: Series

- A DataFrame is a two-dimensional array with row indices and column names.
- A DataFrame is a sequence of aligned Series objects.
  - Here, by "aligned" we mean that they share the same index.
- A single-column DataFrame can be constructed from a single Series:

```
In [17]: pd.DataFrame(population, columns=['population'])
Out[17]:
             population
California     38332521
Florida        19552860
Illinois       12882135
New York       19651127
Texas          26448193
```

- http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

pandas Data Structures: Series

- Constructing from a dictionary of Series objects

```
In [6]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
   ...:              'Florida': 170312, 'Illinois': 149995}
   ...: area = pd.Series(area_dict)
In [7]: states = pd.DataFrame({'population': population, 'area': area})
In [8]: states
Out[8]:
                area   population
California    423967     38332521
Florida       170312     19552860
Illinois      149995     12882135
New York      141297     19651127
Texas         695662     26448193
```

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

pandas Data Structures: Series

- DataFrame has index and columns attributes

```
In [12]: states.index
Out[12]: Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')

In [13]: states.columns
Out[13]: Index(['area', 'population'], dtype='object')
```

- Both the rows and columns have a generalized index for accessing the data.

```
In [11]: states['area'][:2]
Out[11]:
California    423967
Florida       170312
Name: area, dtype: int64

In [15]: states['area']['Florida']
Out[15]: 170312
```

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

pandas Data Structures: Series

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:
- For example, asking for the 'area' attribute returns the Series object.

```
In [16]: states['area']
Out[16]:
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

```
In [23]: states.area
Out[23]:
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Note that the returned Series have the same index as the DataFrame, and their name attribute has been appropriately set.

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

pandas Data Structures: Series

- If you specify a sequence of columns, the DataFrame's columns will be exactly what you pass:

```
In [16]: pd.DataFrame(states, columns=
                      ['population', 'area'])
Out[16]:
             population    area
California     38332521 423967
Florida        19552860 170312
Illinois       12882135 149995
New York       19651127 141297
Texas          26448193 695662
```

- if you pass a column that isn't contained in data, it will appear with NA values in the result:

```
In [18]: frame2 = pd.DataFrame(states, columns=
                ['population', 'area', 'density'])

In [19]: frame2
Out[19]:
             population     area density
California     38332521   423967     NaN
Florida        19552860   170312     NaN
Illinois       12882135   149995     NaN
New York       19651127   141297     NaN
Texas          26448193   695662     NaN
```

Set index name:
`frame2.index.name = 'state'`

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

pandas Data Structures: Series

- Columns can be modified by assignment

```
In [25]: frame2['density'] = frame2['population']/frame2['area']
In [26]: frame2
Out[26]:
             population   area    density
California     38332521 423967  90.413926
Florida        19552860 170312 114.806121
Illinois       12882135 149995  85.883763
New York       19651127 141297 139.076746
Texas          26448193 695662  38.018740
```

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

pandas Data Structures: Series

- Transpose: `frame2.T`
- Like Series, the **values** attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [13]: frame2.values
Out[13]:
array([[  3.83325210e+07,   4.23967000e+05,   9.04139261e+01],
       [  1.95528600e+07,   1.70312000e+05,   1.14806121e+02],
       [  1.28821350e+07,   1.49995000e+05,   8.58837628e+01],
       [  1.96511270e+07,   1.41297000e+05,   1.39076746e+02],
       [  2.64481930e+07,   6.95662000e+05,   3.80187404e+01]])
```

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html#pandas.DataFrame

Data formats: JSON

Telkom University

- **JSON** (JavaScript Object Notation)
- A number of different data types can be represented
  - Number: 1.0 (always assumed to be floating point)
  - String: "string" or 'string'
  - Boolean: true or false
  - List (Array): [item1, item2, item3,...]
  - Dictionary (Object in Javascript): {"key":value}
- Lists and Dictionaries can be embedded within each other:
  - `[{"key":[value1, [value2, value3]]}]`

Data formats: JSON

- Braces {"key":value} indicates an object
  - Use colon to separate key and value
- Brackets [item1, item2, item3,...] indicates an array (or list)
- Ex. employees' value is an array (or list)
  - Array contains three objects which includes two keys: firstName and lastName

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

Data formats: JSON

- JSON example from Github API


```
{
"login":"zkolter",
"id":2465474,
"avatar_url":"https://avatars.githubusercontent.com/u/2465474?v=3",
"gravatar_id":"",
"url":"https://api.github.com/users/zkolter",
"html_url":"https://github.com/zkolter",
"followers_url":"https://api.github.com/users/zkolter/followers",
"following_url":"https://api.github.com/users/zkolter/following{/other_user}",
"gists_url":"https://api.github.com/users/zkolter/gists{/gist_id}",
"received_events_url":"https://api.github.com/users/zkolter/received_events",
"type":"User",
"site_admin":false,
"name":"Zico Kolter”
...
```

Parsing JSON using json module

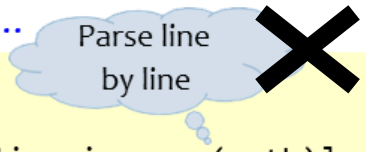
- **Parsing a JSON file using json module**

The first two records of the JSON file (pokemon.json) look as follows:

```
[
    {
        "id": " 001",
        "typeTwo": " Poison",
        "name": " Bulbasaur",
        "type": " Grass"
    },
    {
        "id": " 002",
        "typeTwo": " Poison",
        "name": " Ivysaur",
        "type": " Grass"
    },
```

```
In [19]: import json
In [20]: with open("pokemon.json") as f:
    ...:     data = json.loads(f.read())
In [21]: data[0]
Out[21]: {'id': ' 001', 'name': ' Bulbasaur', 'type': '
Grass', 'typeTwo': ' Poison'}
```

How about using **list comprehension** …




```
import json
path = 'pokemon.json'
records = [json.loads(line) for line in open(path)]

JSONDecodeError: Expecting value: line 2 column 1 (char 2)
```

Parsing JSON using json module

- If the json data like this, one line one json object.

```
{"id": " 001",   "typeTwo": " Poison",   "name": " Bulbasaur",   "type": "Grass" }
{"id": " 002",   "typeTwo": " Poison",   "name": " Ivysaur",     "type": " Grass" }
{"id": " 003",   "typeTwo": " Poison",   "name": " Venusaur",    "type": " Grass"}
{"id": " 006",   "typeTwo": " Flying",   "name": " Charizard",   "type": " Fire" }
{"id": " 012",   "typeTwo": " Flying",   "name": " Butterfree",  "type": " Bug"}
```

- Then, we can use list comprehension

```
In [21]: records = [json.loads(line) for line in open('pokemon_line.json')]
In [22]: records
Out[22]:
[{'id': ' 001', 'name': ' Bulbasaur', 'type': 'Grass', 'typeTwo': ' Poison'},
{'id': ' 002', 'name': ' Ivysaur', 'type': ' Grass', 'typeTwo': ' Poison'},
{'id': ' 003', 'name': ' Venusaur', 'type': ' Grass', 'typeTwo': ' Poison'},
{'id': ' 006', 'name': ' Charizard', 'type': ' Fire', 'typeTwo': ' Flying'},
{'id': ' 012', 'name': ' Butterfree', 'type': ' Bug', 'typeTwo': ' Flying'}]
```

Parsing JSON using pandas

- **Parsing a JSON file using pandas**

```
In [24]: import pandas as pd
In [25]: data = pd.read_json('pokemon.json')
In [26]: data
Out[26]:
   id       name   type typeTwo
0   1 Bulbasaur Grass  Poison
1   2    Ivysaur Grass  Poison
2   3  Venusaur Grass  Poison
…
```

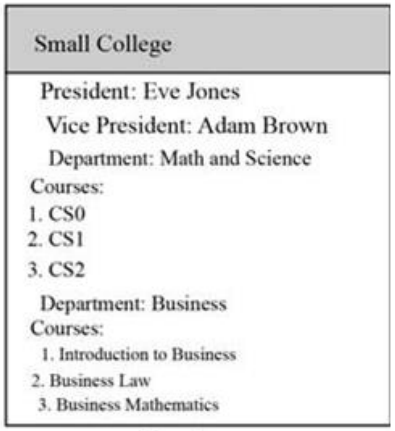
Data formats: HTML

- **HyperText Markup Language** (**HTML**) is the standard markup language for creating web pages and web applications.
- HTML elements are the building blocks of HTML pages.
  - It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items.
  - HTML elements are delineated by ***tags*** , written using **angle brackets <>**.
  - HTML tags most commonly come in pairs like <H1> … </H1>, … .
  - **HTML tags are predefined. You cannot define your own tags.**

Data formats: HTML

```
<HTML>
  <HEAD>
      <TITLE> Small College </TITLE>
  </HEAD>
  <BODY>
    <H1> President: Eve Jones </H1>
    <H1> Vice President: Adam Brown </H1>
    <H2> Department: Math and Science </H2>
    <H3> Courses </H3>
      <LI> CS0 </LI>
      <LI> CS1 </LI>
      <LI> CS2 </LI>
    <H2> Department: Business </H2>
    <H3> Courses </H3>
      <LI> Introduction to Business </LI>
      <LI> Business Law </LI>
      <LI> Business Mathematics </LI>
  </BODY>
</HTML>
```

HTML document

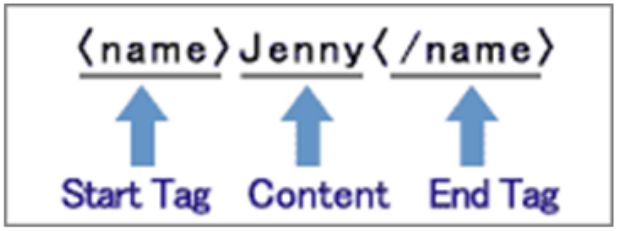


Result

Using tags to display web pages.
Ex: <HTML> </HTML>
<HEAD> </HEAD>
<BODY> </BODY>
<H1> </H1>

Data formats: HTML

- **What is XML?**
  - XML stands for E**X**tensible **M**arkup **L**anguage
  - XML is a markup language much like HTML.
  - XML was designed to describe data.
  - XML tags are **not predefined** in XML. You must define your own tags.
  - XML uses a DTD (Document Type Definition) to formally describe the data.

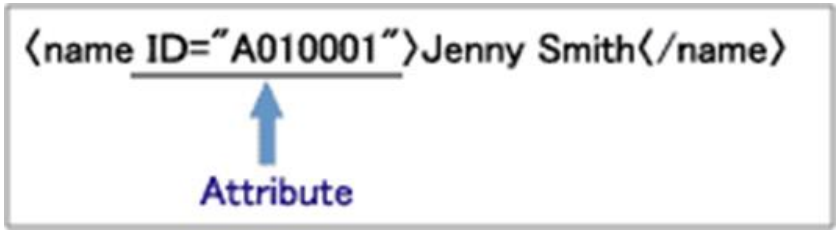



```
<note>
   <to>Tove</to>
   <from>Jani</from>
   <heading>Reminder</heading>
   <body>Don't forget me this weekend!</body>
</note>
```

https://www.tutorialspoint.com/xml/index.htm

Data formats: HTML

- **The main difference between XML and HTML**
  - XML is **not a replacement** for HTML.
  - XML and HTML were designed with **different goals:**
    - XML was designed to **describe data** and to focus on **what data is.**
    - HTML was designed to **display data** and to focus on **how data looks.**
  - HTML is about **displaying** information, XML is about **describing** information.
- When required, an "attribute" can be described in the start tag of an element.



XML – DTDs (Document Type Definition)

- The purpose of a DTD is to define the legal building blocks of an XML document.
- It defines the document structure with a list of legal elements.
- A DTD can be declared **inline** in your XML document, or as an **external** reference.
- Basic syntax of a DTD is as follows:

```
<!DOCTYPE element DTD identifier
[
   declaration1
   declaration2
   ........
]>
```

- The syntax of **internal** DTD is as shown:

```
<!DOCTYPE root-element [element-declarations]>
```

XML – DTDs (Document Type Definition)

- The purpose of a DTD is to define the legal building blocks of an XML document.
- It defines the document structure with a list of legal elements.
- A DTD can be declared **inline** in your XML document, or as an **external** reference.
- Basic syntax of a DTD is as follows:

```
<!DOCTYPE element DTD identifier
[
   declaration1
   declaration2
   ........
]>
```

- The syntax of **internal** DTD is as shown:

```
<!DOCTYPE root-element [element-declarations]>
```

XML – DTDs (Document Type Definition)

- An example of internal DTD

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE address [
   <!ELEMENT address (name,company,phone)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT company (#PCDATA)>
   <!ELEMENT phone (#PCDATA)>
]>
<address>
   <name>Tanmay Patil</name>
   <company>TutorialsPoint</company>
   <phone>(011) 123-4567</phone>
</address>
```

Start Declaration- Begin the XML declaration with following statement

DTD Body- The DOCTYPE declaration is followed by body of the DTD

<!ELEMENT name (#PCDATA)> defines the element *name* to be of type "#PCDATA".
Here #PCDATA means parse-able text data.

https://www.tutorialspoint.com/xml/xml_dtds.htm

Parsing XML using xml module

- Output:

```
id: 001
typeTwo: Poison
name: Bulbasaur
type: Grass

id: 002
typeTwo: Poison
name: Ivysaur
type: Grass

id: 003
typeTwo: Poison
name: Venusaur
type: Grass
```

The first two records of the XML file (pokemon.xml) look as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<pokemon>
        <row>
                <id> 001</id>
                <typeTwo> Poison</typeTwo>
                <name> Bulbasaur</name>
                <type> Grass</type>
        </row>
        <row>
                <id> 002</id>
                <typeTwo> Poison</typeTwo>
                <name> Ivysaur</name>
                <type> Grass</type>
        </row>
...
</pokemon>
```

Importing XML to Pandas DataFrame

```
from xml.etree import ElementTree
import pandas as pd
with open("pokemon.xml") as f:
    dfcols = ['id', 'typeTwo', 'name', 'type']
    df_xml = pd.DataFrame(columns=dfcols)

    doc = ElementTree.parse(f)
    for node in doc.findall('row'):
        nid = node.find('id').text
        ntypeTwo = node.find('typeTwo').text
        nname = node.find('name').text
        ntype = node.find('type').text
        df_xml = df_xml.append(pd.Series([nid,ntypeTwo, nname, ntype ], index=dfcols),
ignore_index=True)

print(df_xml)
```

	id	typeTwo	name	type
1	001	Poison	Bulbasaur	Grass
2	002	Poison	Ivysaur	Grass
3	003	Poison	Venusaur	Grass
4	006	Flying	Charizard	Fire
5	012	Flying	Butterfree	Bug
6	013	Poison	Weedle	Bug
7	014	Poison	Kakuna	Bug
8	015	Poison	Beedrill	Bug
9	016	Flying	Pidgey	Normal
10	017	Flying	Pidgeotto	Normal
11	018	Flying	Pidgeot	Normal

Append rows of *other* to the end of caller, returning a new object.

Parsing XML using BeautifulSoup

- Beautiful Soup is a Python library for pulling data out of HTML and XML files.
  - https://www.crummy.com/software/BeautifulSoup/

```
In [30]: from bs4 import BeautifulSoup
    ...: infile = open('pokemon.xml')
    ...: soup = BeautifulSoup(infile,'xml')
In [31]: names = soup.find_all('name')
In [32]: for name in names:
    ...:     print(name.get_text())
Bulbasaur
Ivysaur
Venusaur
Charizard
Butterfree
Weedle
…
```

```
In [33]: for name in names:
    ...:     print(name)
    ...:
<name> Bulbasaur</name>
<name> Ivysaur</name>
<name> Venusaur</name>
<name> Charizard</name>
<name> Butterfree</name>
<name> Weedle</name>
```

Parsing XML using BeautifulSoup

- The first argument to the BeautifulSoup constructor is a string or an open file handle–the markup you want parsed.
- The second argument is how you'd like the markup parsed.
  - If you don't specify anything, you'll get the best HTML parser that's installed.

```
infile = open('pokemon.xml')
soup = BeautifulSoup(infile,'xml')
```

  - Currently supported are "html", "xml", and "html5".

Dealing with problematic data

- if the CSV file contains a header, and some missing values and dates.

```
Date,Temperature_city_1,Temp_city_2,Temp_city_3,Which_destination
20140910,80,32,40,1
20140911,100,50,36,2
20140912,102,55,46,1
20140913,60,20,35,3
20140914,60,,32,3
20140915,,57,42,2
```

```
In [33]: fake_data = pd.read_csv('a_loading_example_1.csv', sep=',')
In [34]: fake_data
Out[34]:
       Date  Temperature_city_1  Temp_city_2  Temp_city_3  Which_destination
0  20140910                80.0         32.0           40                  1
1  20140911               100.0         50.0           36                  2
2  20140912               102.0         55.0           46                  1
3  20140913                60.0         20.0           35                  3
4  20140914                60.0          NaN           32                  3
5  20140915                 NaN         57.0           42                  2
```

all the data, even the **dates**, has been parsed as integers

NaN (Not a Number)

Dealing with problematic data

- If the format of the dates is not very strange, you can try the auto detection routines that specify the column that contains the date data.

```
In [35]: fake_data = pd.read_csv('a_loading_example_1.csv', parse_dates=[0])
In [36]: fake_data
Out[36]:
         Date   Temperature_city_1   Temp_city_2   Temp_city_3   Which_destination
0 2014-09-10                 80.0          32.0            40                   1
1 2014-09-11                100.0          50.0            36                   2
2 2014-09-12                102.0          55.0            46                   1
3 2014-09-13                 60.0          20.0            35                   3
4 2014-09-14                 60.0           NaN            32                   3
5 2014-09-15                  NaN          57.0            42                   2
```

Dealing with problematic data

- To get rid of the missing data that is indicated as NaN, replace them with a more meaningful number (let's say 50 Fahrenheit, for example).

```
In [37]: fake_data.fillna(50)
Out[37]:
         Date   Temperature_city_1   Temp_city_2   Temp_city_3   Which_destination
0 2014-09-10                 80.0          32.0            40                   1
1 2014-09-11                100.0          50.0            36                   2
2 2014-09-12                102.0          55.0            46                   1
3 2014-09-13                 60.0          20.0            35                   3
4 2014-09-14                 60.0          50.0            32                   3
5 2014-09-15                 50.0          57.0            42                   2
```

Dealing with problematic data

- NaN values can also be replaced by the column mean or median value as a way to minimize the guessing error:

```
In [38]: fake_data.fillna(fake_data.mean(axis=0))
Out[38]:
         Date   Temperature_city_1   Temp_city_2   Temp_city_3   Which_destination
0 2014-09-10                 80.0          32.0            40                   1
1 2014-09-11                100.0          50.0            36                   2
2 2014-09-12                102.0          55.0            46                   1
3 2014-09-13                 60.0          20.0            35                   3
4 2014-09-14                 60.0          42.8            32                   3
5 2014-09-15                 80.4          57.0            42                   2
```

  - The .mean method calculates the mean of the specified axis.
  - Please note that **axis= 0** implies the calculation of the means that span the rows.
    - (80+100+102+60+60)/5 = 80.4
  - Instead, axis=1 spans columns and therefore, row-wise results are obtained.

Dealing with problematic data

Telkom University

- Another possible problem when handling real world datasets is the loading of a dataset with **errors or bad lines**.
- In this case, the default behavior of the load_csv method is to stop and raise an exception.
- A possible workaround, which is not always feasible, is to ignore this line.
- **pd.read_csv() parameter** error_bad_lines : boolean, default True
  - Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned.
  - If False, then these "bad lines" will dropped from the DataFrame that is returned. (Only valid with C parser)

Dealing with problematic data

error_bad_lines

```
Val1,Val2,Val3
0,0,0
1,1,1
2,2,2,2
3,3,3
```

```
In [40]: bad_dataset = pd.read_csv('a_loading_example_2.csv',error_bad_lines=False)
b'Skipping line 4: expected 3 fields, saw 4\n'
In [41]: bad_dataset
Out[41]:
   Val1  Val2  Val3
0     0     0     0
1     1     1     1
2     3     3     3
```

Dealing with big datasets

Telkom University

- If the dataset you want to load is too big to fit in the memory, you actually can load the data in chunks.
- With pandas, there are two ways to chunk and load a file.
  - The first way to do this is by loading the dataset in chunks of the same size;
  - Each chunk is a piece of the dataset that contains all the columns and the number of lines set in the function call (the chunksize parameter).
  - Note that the output of the read_csv function in this case is not a pandas DataFrame but an iterator-like object.
  - In fact, to get the results, you need to iterate that object.

Dealing with big datasets

- The first method: loading the dataset in chunks of the same size

```
In [42]: import pandas as pd
In [46]: iris_chunks = pd.read_csv(
                    'datasets-uci-iris.csv', header=None,
                    names=['C1', 'C2', 'C3', 'C4', 'C5'],
                    chunksize=10)
In [48]: for chunk in iris_chunks:
    ...:     print(chunk.shape)
    ...:     print(chunk)
    ...:
```

To get the results, you need to iterate that object

Out:

```
(10, 5)
    C1   C2   C3   C4           C5
0  5.1  3.5  1.4  0.2  Iris-setosa
1  4.9  3.0  1.4  0.2  Iris-setosa
2  4.7  3.2  1.3  0.2  Iris-setosa
3  4.6  3.1  1.5  0.2  Iris-setosa
4  5.0  3.6  1.4  0.2  Iris-setosa
5  5.4  3.9  1.7  0.4  Iris-setosa
6  4.6  3.4  1.4  0.3  Iris-setosa
7  5.0  3.4  1.5  0.2  Iris-setosa
8  4.4  2.9  1.4  0.2  Iris-setosa
9  4.9  3.1  1.5  0.1  Iris-setosa
(10, 5)
     C1   C2   C3   C4           C5
10  5.4  3.7  1.5  0.2  Iris-setosa
```

Dealing with big datasets

- The second method: loading the dataset in chunks of the different size

```
iris_iterator = pd.read_csv('datasets-uci-iris.csv',
                header=None,
                names=['C1', 'C2', 'C3', 'C4', 'C5'],
                iterator=True)
In [51]: print (iris_iterator.get_chunk(10).shape)
(10, 5)

In [52]: piece = iris_iterator.get_chunk(2)
In [53]: piece
Out[53]:
     C1   C2   C3   C4           C5
10  5.4  3.7  1.5  0.2  Iris-setosa
11  4.8  3.4  1.6  0.2  Iris-setosa
```

- In this example, we first got the **iterator**.
- Then, we got a piece of data with 10 lines.
- We then got 2 further rows.

- In addition to pandas, you can also use the **csv module** that offers two functions to iterate small chunks of data from files: the **reader** and the **DictReader** functions.

INTRODUCTORY EXAMPLE

Python for Data Analysis, Wes McKinney, O'Reilly Media, Inc., 2013

- 1.usa.gov data from bit.ly
  - In 2011, URL shortening service bit.ly partnered with the United States government website usa.gov to provide a feed of anonymous data gathered from users who shorten links ending with .gov or .mil.
  - http://1usagov.measuredvoice.com/
- Each line in each file contains a common form of JSON.

USA.gov Data

- The JSON data dictionary is as follows:

```
{
    "a": USER_AGENT,
    "c": COUNTRY_CODE, # 2-character iso code
    "nk": KNOWN_USER,  # 1 or 0\. 0=this is the first time we've seen this browser
    "g": GLOBAL_BITLY_HASH,
    "h": ENCODING_USER_BITLY_HASH,
    "l": ENCODING_USER_LOGIN,
    "hh": SHORT_URL_CNAME,
    "r": REFERRING_URL,
    "u": LONG_URL,
    "t": TIMESTAMP,
    "gr": GEO_REGION,
    "ll": [LATITUDE, LONGITUDE],
    "cy": GEO_CITY_NAME,
    "tz": TIMEZONE # in http://en.wikipedia.org/wiki/Zoneinfo format
    "hc": TIMESTAMP OF TIME HASH WAS CREATED,
    "al": ACCEPT_LANGUAGE http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.4
}
```

USA.gov Data

- if we read just the first line of a file you may see something like

```
In [15]: path = 'usagov_bitly_data2012-03-16-1331923249.txt'

In [16]: open(path).readline()
Out[16]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1,
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u":
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

USA.gov Data

- Python has numerous built-in and 3rd party modules for converting a JSON string into a Python **dictionary** object.
- Use the json module and its loads function invoked on each line in the sample file :

```
import json
path = 'usagov_bitly_data2012-03-16.txt'
records = [json.loads(line) for line in open(path)]
```

  - The last expression here is called a *list comprehension*, which is a concise way of applying an operation (like json.loads) to a collection of strings or other objects.

```
----> 1 records = [json.loads(line) for line in open(path)]
UnicodeDecodeError: 'cp950' codec can't decode byte 0xe2 in position 6987: illegal multibyte sequence
Try this:
records = [json.loads(line) for line in open(path, encoding='utf-8')]
```

USA.gov Data

- The resulting object records is now a list of Dictionaries:

```
>>> print (records[0])
{'ll': [4.6492, -74.062798], 'hc': 1356983161, 'a': 'MOT-
MB525/Blur_Version.34.4.802.MB525.AmericaMovil.en.01 Mozilla/5.0 (Linux; U;
Android 2.2.2; en-us; MB525 Build/3.4.2-108_JDNL-2_R01) AppleWebKit/533.1 (KHTML,
like Gecko) Version/4.0 Mobile Safari/533.1', 'h': 'VUas6m', 'r':
'http://t.co/0NoHYs9t', 't': 1356995456, 'u':
'http://www.nasa.gov/mission_pages/station/expeditions/expedition34/newyear.html
', 'al': 'en-US', 'g': '10sGgGU', 'hh': 'go.nasa.gov', 'c': 'CO', 'gr': '34',
'nk': 1, 'cy': 'Bogotá', 'tz': 'America/Bogota', 'l': 'nasatwitter'}
>>>
```

- Dictionary example
  - `D = {'spam': 2, 'ham': 1, 'eggs': 3}`

USA.gov Data

- Access individual values within records by passing a string for the key you wish to access:

```
In [19]: records[0]['tz']
Out[19]: 'America/New_York'
```

- Counting Time Zones in Pure Python
  - Suppose we were interested in the most often-occurring time zones in the data set (the tz field).
  - First, let's extract a list of time zones again using a list comprehension:

```
In [25]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------KeyError
Traceback (most recent call last)
/home/wesm/book_scripts/whetting/<ipython> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

Counting Time Zones

- Not all of the records have a time zone field.
  - This is easy to handle as we can add the check if 'tz' in rec at the end of the list comprehension:

```
In [26]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]
In [27]: time_zones[:10]
Out[27]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

looking at the first 10 time zones

some of them are empty

Counting Time Zones in Pure Python

- To produce counts by time zone
  - Using Python standard library
  - Using pandas
- Using Python standard library
  - **collections.Counter class**
  - **Counter.most_common([n])**
    - Return a **list** of the n most common elements and their counts from the most common to the least.
    - If n is omitted or None, most_common() returns all elements in the counter.

```
In [54]: from collections import Counter
In [55]: counts = Counter(time_zones)
In  [56]:  counts.most_common(10)
Out[56]:
[('America/New_York', 1251),
('', 521),
('America/Chicago', 400),
('America/Los_Angeles', 382),
('America/Denver', 191),
('Europe/London', 74),
('Asia/Tokyo', 37),
('Pacific/Honolulu', 36),
('Europe/Madrid', 35),
('America/Sao_Paulo', 33)]
```

collections

- This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

https://docs.python.org/3.5/library/collections.html

collections.Counter

- A Counter is a dict subclass for counting hashable objects.
- It is an unordered collection where elements are stored as **dictionary** keys and their counts are stored as dictionary values.

```
In [57]: c = Counter('gallahad')
In [58]: c
Out[58]: Counter({'a': 3, 'd': 1, 'g': 1, 'h': 1, 'l': 2})

In [61]: c = Counter({'red': 4, 'blue': 2})
In [62]: c
Out[62]: Counter({'blue': 2, 'red': 4})

In [63]: c = Counter(cats=4, dogs=8)
In [64]: c
Out[64]: Counter({'cats': 4, 'dogs': 8})
```

Counting Time Zones with pandas

- The main pandas data structure is the *DataFrame*
  - You can think of as representing a table or spreadsheet of data.
  - Creating a DataFrame from the original set of records is simple:

```
In [290]: import pandas as pd
In [291]: frame = pd.DataFrame(records)
```

```
In [293]: frame['tz'][:10]
Out[293]:
1 America/New_York
2 America/Denver
3 America/New_York
4 America/Sao_Paulo
5 America/New_York
6 America/New_York
7Europe/Warsaw
7
8
9
Name: tz, dtype: object
```

Counting Time Zones with pandas

- The Series object returned by frame['tz'] has a method value_counts that gives us what we're looking for:

```
In [294]: tz_counts = frame['tz'].value_counts()
In [295]: tz_counts[:10]
Out[295]:
America/New_York    1251
                     521
America/Chicago      400
 America/Los_Angeles 382
America/Denver       191
Europe/London         74
Asia/Tokyo            37
Pacific/Honolulu      36
Europe/Madrid         35
America/Sao_Paulo     33
```

value_counts()

- Returns a series object containing counts of unique values.
- The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.value_counts.html#pandas.Series.value_counts

Counting Time Zones with pandas

- The **fillna** function can replace missing (NA) values
- Unknown (empty strings) values can be replaced by boolean array indexing:

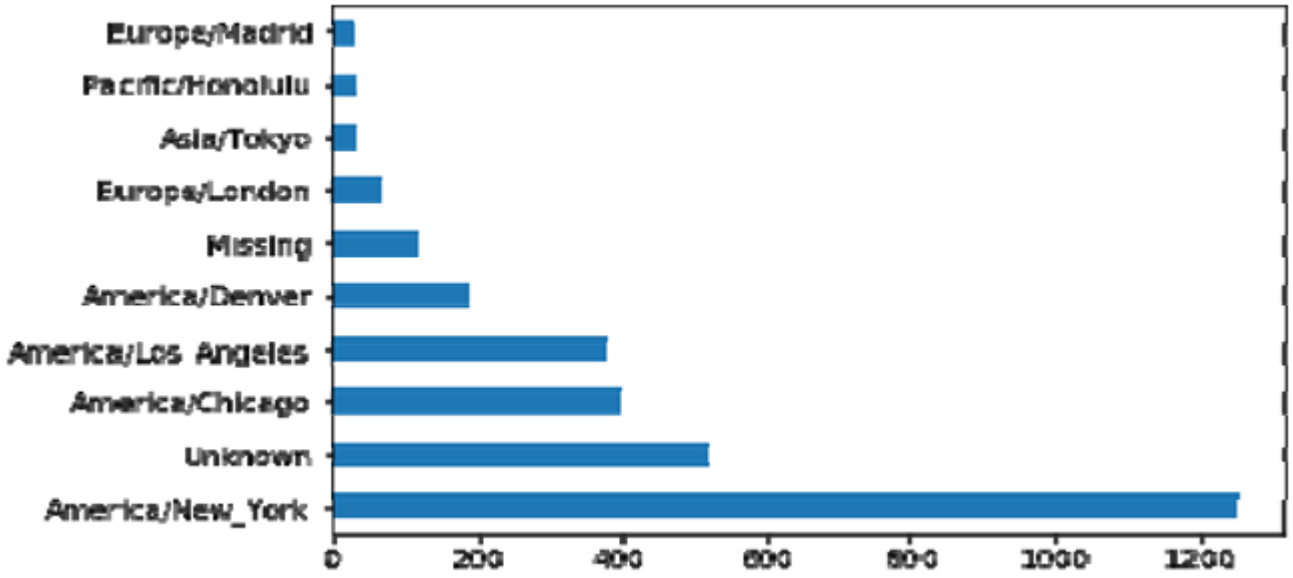
```
In [296]: clean_tz = frame['tz'].fillna('Missing')
In [297]: clean_tz[clean_tz == ''] = 'Unknown'
In [298]: tz_counts = clean_tz.value_counts()
In [299]: tz_counts[:10]
Out[299]:
America/New_York 1251
Unknown 521
America/Chicago 400
America/Los_Angeles 382
America/Denver 191
Missing 120
…
```

Counting Time Zones with pandas

- Make a plot of this data using plotting library, matplotlib
  - Making a horizontal bar plot can be accomplished using the plot method on the counts objects:

```
In [21]: %matplotlib inline
In [22]: tz_counts[:10].plot(kind='barh', rot=0)
```

IPython Magic Commands prefixed by the % character.



Series.plot

- Series.plot is both a callable method and a namespace attribute for specific plotting methods of the form Series.plot.<kind>.

Series.plot([kind, ax, figsize, ...])	Series plotting accessor and method
Series.plot.area(**kwds)	Area plot
Series.plot.bar(**kwds)	Vertical bar plot
Series.plot.barh(**kwds)	Horizontal bar plot
Series.plot.box(**kwds)	Boxplot
Series.plot.density(**kwds)	Kernel Density Estimate plot
Series.plot.hist([bins])	Histogram
Series.plot.kde(**kwds)	Kernel Density Estimate plot
Series.plot.line(**kwds)	Line plot
Series.plot.pie(**kwds)	Pie chart
Series.hist([by, ax, grid, xlabelsize, ...])	Draw histogram of the input series using matplotlib

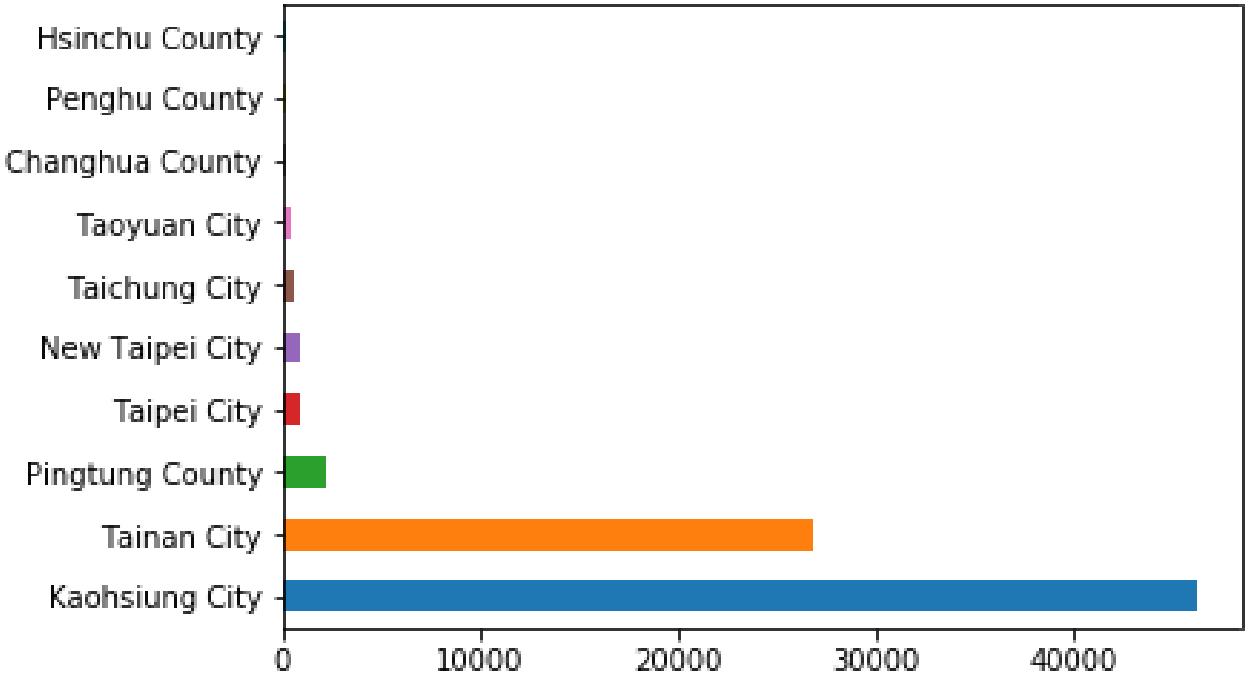
http://pandas.pydata.org/pandas-docs/stable/api.html#plotting

Homework 4: Dengue fever dataset

- Dengue fever
  - **Dengue fever** is a mosquito-borne tropical disease caused by the dengue virus. Each year between 50 and 528 million people are infected and approximately 10,000 to 20,000 die.
- Fields include
  "Date_Onset","Date_Confirmation","Date_Notification","Sex","Age_Group","County_living","Township_living","Village_Living","Village_Living_Code","Enumeration_unit","Enumeration_unit_long","Enumeration_unit_lat","First_level_dissemination_unit","Second_level_dissemination_unit","County_infected","Township_infected","Village_infected","Village_infected_Code","Imported","Country_infected","Number_of_confirmed_cases"
- Make statistics and plot the figures
  - Number of cases for each county
  - Number of cases for each month
  - Number of cases for each Age_Group

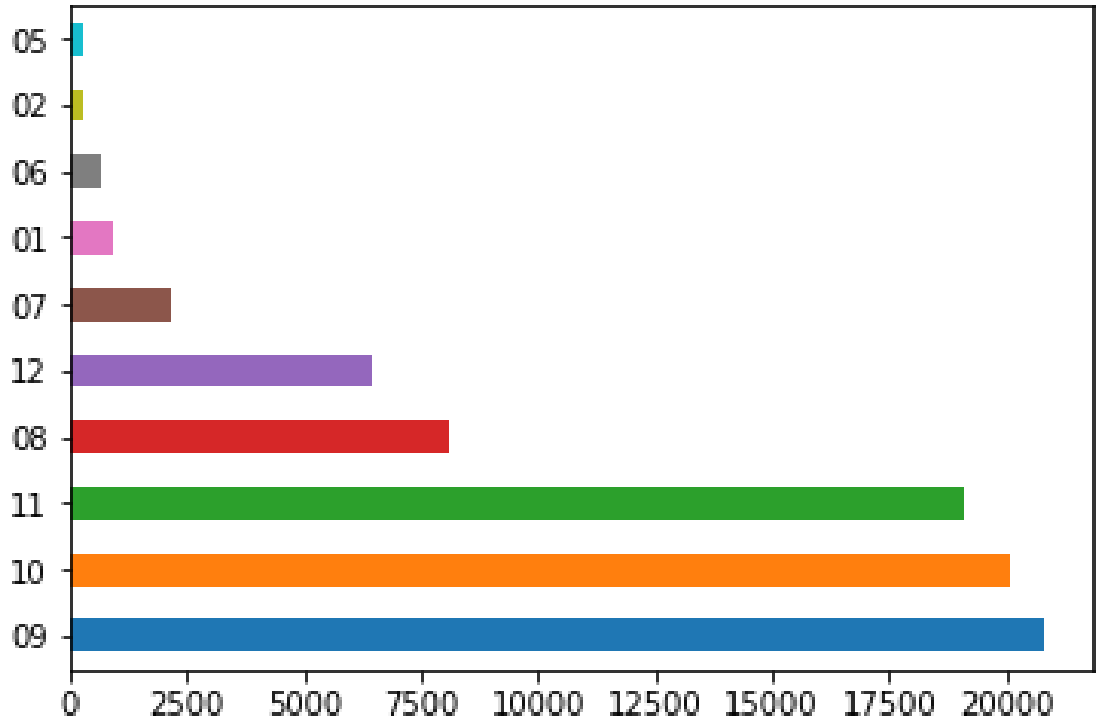
Homework 4: Dengue fever dataset

"Answer of County"



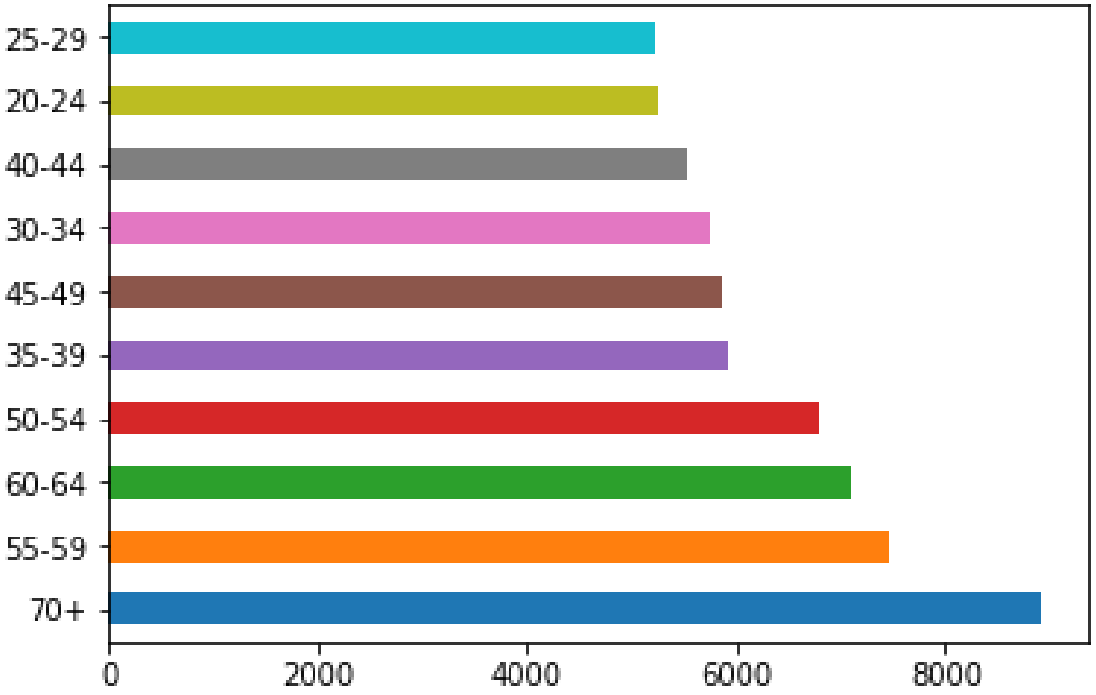
Homework 4: Dengue fever dataset

"Answer of Month"



Homework 4: Dengue fever dataset

"Answer of Age_Group"



Thank You