These slides are generally based on the examples from VanderPlas (2017).

Index Object

- We saw that both the `Series` and `DataFrame` objects contain an explicit index which is useful for slicing.
- You can think of an index object as an immutable array.
- We saw different ways to slice `Series`/`DataFrame` objects.
- We can use the Python style indexing scheme or the explicit index associated with the `Series` and `DataFrame` objects.
- `loc` attribute allows indexing and slicing that always uses the explicit index.
- `iloc` attribute allows indexing and slicing that always uses the Pythonic index.
- A third indexing attribute is `ix` which is a hybrid of the two.

Missing Data

- Pandas handles missing values using NumPy package, which does not have a built-in notion of NA values for non-floating-point data types.
- Pandas utilizes sentinels for missing data by using two already existing Python null values: the special floating-point `NaN` value, and the Python `None` object.
- If you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error.
- The special floating-point `NaN` value is recognized as a number but behaves like a data virus: it infects any other object it interacts.
- If you perform aggregations like `sum()` or `min()` across an array with a `NaN` value, you will get `NaN` values.

Hierarchical Indexing

- We saw that the `Series` and `DataFrame` objects store one-dimensional and two-dimensional data, respectively.
- Higher dimensional data can be stored using hierarchical indexing (multi-indexing).
- It incorporates multiple index levels within a single index.
- These objects are `MultiIndex` objects.

Hierarchical Indexing

```
import pandas as pd
import numpy as np

#the bad way
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York',2010),
         ('Texas', 2000), ('Texas', 2010)]

populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]

pop = pd.Series(populations, index=index)
pop

Out[4]:
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64

index = pd.MultiIndex.from_tuples(index)
index

pop = pop.reindex(index)
pop

Out[5]:
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Hierarchical Indexing

- Note that the first two columns show the multiple index values, while the third column shows the data.
- Suppose you need to access all data for which the second index is 2010. You can use Pythonic notation.

```
pop[:, 2010]

Out[6]:
California     37253956
New York       19378102
Texas          25145561
dtype: int64
```

Hierarchical Indexing

- We can add more columns.

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
```

```
pop_df
```

```
Out[10]:
                      total   under18
California  2000   33871648   9267089
            2010   37253956   9284094
New York    2000   18976457   4687374
            2010   19378102   4318033
Texas       2000   20851820   5906301
            2010   25145561   6879014
```

- Let's compute fraction of people by year and present in a wide format.

```
fr_u18 = pop_df['under18']/pop_df['total']
fr_u18.unstack()
```

```
Out[12]:
                2000       2010
California  0.273594   0.249211
New York    0.247010   0.222831
Texas       0.283251   0.273568
```

Index resetting

- Index labels can be turned into data columns using `reset_index` method.

```
pop_reset = pop.reset_index(name='population')
pop_reset
```

```
Out[6]:
      level_0  level_1  population
0  California     2000    33871648
1  California     2010    37253956
2    New York     2000    18976457
3    New York     2010    19378102
4       Texas     2000    20851820
5       Texas     2010    25145561
```

- Often raw data will look like this. You can use `reset_index` method to build a `MultiIndex`.

```
pop_reset = pop.reset_index(name='population')
pop_reset = pop_reset.rename(columns = {'level_0':'state', 'level_1':'year'})
pop_reset.set_index(['state','year'])
pop_reset
```

```
Out[19]:
        state  year  population
0  California  2000    33871648
1  California  2010    37253956
2    New York  2000    18976457
3    New York  2010    19378102
4       Texas  2000    20851820
5       Texas  2010    25145561
```

Concat and Append

- Empirical analysis generally involves some form of `concat`, `merge` and `join` operations.
- Pandas has functions and methods that make this operations straightforward.

```
def make_dataframe(cols, ind):
    """ quick DataFrame composer """
    data = {c: [str(c) + str(i) for i in ind] for c in cols}
    return pd.DataFrame(data, ind)

df1 = make_dataframe('AB', [1,2])
df2 = make_dataframe('AB', [3,4])
pd.concat([df1,df2])

Out[21]:
    A   B
1   A1  B1
2   A2  B2
3   A3  B3
4   A4  B4
```

one-to-one join

- Pandas `merge` and `join` operations use a set of rules known as *relational algebra* to combine data.
- The relational algebra proposes several primitive operations which allows for handling more complicated operations.
- `merge()` functions implements several type of joins: the *one-to-one*, *many-to-one*, and *many-to-many*.

```
df1 = pd.DataFrame({'employee':['Bob','Jake','Lisa','Sue'],
                    'group':['accounting','engineering','engineering','hr']})
df2 = pd.DataFrame({'employee':['Lisa','Bob','Jake','Sue'],
                    'hire_date':[2004, 2008, 2012, 2014]})
df3 = pd.merge(df1, df2)
df3
Out[23]:
  employee        group  hire_date
0      Bob   accounting       2008
1     Jake  engineering       2012
2     Lisa  engineering       2004
3      Sue           hr       2014
```

many-to-one join

- *many-to-one* joins refer to cases where the key columns for merge contain duplicate entries.

```
df4 = pd.DataFrame({'group':['accounting','engineering','hr'],
                    'supervisor':['Carly','Guido','Steve']})

print(pd.merge(df3, df4))

  employee        group  hire_date supervisor
0      Bob   accounting       2008      Carly
1     Jake  engineering       2012      Guido
2     Lisa  engineering       2004      Guido
3      Sue           hr       2014      Steve
```

- We see that the resulting dataframe has the *supervisor* column where the information is repeated in one or more locations.

many-to-many join

- *many-to-many* joins involve cases where the left and right dataframes contain key columns with duplicate entries.

```
df5 = pd.DataFrame({'group':['accounting','accounting','engineering',
                             'engineering','hr','hr'],
                    'skills':['math','spreadsheets','coding','linux',
                              'spreadsheets','organization']})

print(pd.merge(df1, df5))
```

```
  employee        group        skills
0      Bob   accounting          math
1      Bob   accounting  spreadsheets
2     Jake  engineering        coding
3     Jake  engineering         linux
4     Lisa  engineering        coding
5     Lisa  engineering         linux
6      Sue           hr  spreadsheets
7      Sue           hr  organization
```

on keyword

- In the previous examples we have not specified the key columns in `pd.merge()`, the function automatically detected to intersection of columns common to the left and right data frames to join.
- Often it will be the case that the key columns are named differently in the left and right dataframes, and therefore we need to specify them in `pd.merge()`.

```
print(df1); print(df2);
print(pd.merge(df1, df2, on='employee'))

#df1
  employee        group
0      Bob   accounting
1     Jake  engineering
2     Lisa  engineering
3      Sue           hr
#df2
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
#merge(df1, df2, on='employee')
  employee        group  hire_date
0      Bob   accounting       2008
1     Jake  engineering       2012
2     Lisa  engineering       2004
3      Sue           hr       2014
```

left_on and right_on keywords

- We can specify the key columns for `pd.merge()` using `left_on` and `right_on` arguments corresponding to left and right dataframes.

```
df3 = pd.DataFrame({'name':['Bob','Jake','Lisa','Sue'],
                    'salary':[70000, 80000, 120000, 90000]})

print(df1); print(df3);
print(pd.merge(df1, df3, left_on='employee', right_on='name'))

#df1
  employee        group
0      Bob   accounting
1     Jake  engineering
2     Lisa  engineering
3      Sue           hr
#df3
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
#merge(df1, df3, left_on='employee', right_on='name')
  employee        group  name  salary
0      Bob   accounting   Bob   70000
1     Jake  engineering  Jake   80000
2     Lisa  engineering  Lisa  120000
3      Sue           hr   Sue   90000
```

- Similarly if left and right dataframes have indices set, you can use `left_index` and `right_index` arguments.

overlapping column names

- You may be asking what happens when we try to join two dataframes on a key column, but both dataframes also have conflicting columns.

- `pd.merge()` works and you will notice that the column will be displayed twice and the columns names will have suffixes `_x` and `_y` appended to them, respectively.

```
df6 = pd.DataFrame({'name':['Bob','Jake','Lisa','Sue'],
                    'rank':[1, 2, 2, 4]})
df7 = pd.DataFrame({'name':['Bob','Jake','Lisa','Sue'],
                    'rank':[3, 1, 4, 2]})
print(pd.merge(df6, df7, on='name'))
```

```
   name  rank_x  rank_y
0   Bob       1       3
1  Jake       2       1
2  Lisa       2       4
3   Sue       4       2
```

- It is possible customize suffixes using `suffixes` argument.

```
print(pd.merge(df6, df7, on='name', suffixes=['_L', '_R']))
```

```
   name  rank_L  rank_R
0   Bob       1       3
1  Jake       2       1
2  Lisa       2       4
3   Sue       4       2
```

group_by

- Often we will have to aggregate and/or obtain summary statistics data conditionally on some label or index.

- `Group-by` object in pandas allows for splitting data, applying functions to splits and combining the results from the apply step.

- The most important operations performed by `Group-by` objects are *aggregate*, *filter*, *transform* and *apply*.

- To understand these functionalities of a `Group-by` object, we will use the *planets* dataset available in `seaborn` module.

```
import seaborn as sns

planets = sns.load_dataset('planets')

planets.shape
Out[15]: (1035, 6)

planets.head()
Out[16]:
            method  number  orbital_period   mass  distance  year
0  Radial Velocity       1         269.300   7.10     77.40  2006
1  Radial Velocity       1         874.774   2.21     56.95  2008
2  Radial Velocity       1         763.000   2.60     19.84  2011
3  Radial Velocity       1         326.030  19.40    110.62  2007
4  Radial Velocity       1         516.220  10.50    119.47  2009
```

group_by

- Let's take a look at a table of descriptive statistics for the planets data.

```
planets.dropna().describe()
Out[18]:
          number  orbital_period        mass    distance         year
count  498.00000      498.000000  498.000000  498.000000   498.000000
mean     1.73494      835.778671    2.509320   52.068213  2007.377510
std      1.17572     1469.128259    3.636274   46.596041     4.167284
min      1.00000        1.328300    0.003600    1.350000  1989.000000
25%      1.00000       38.272250    0.212500   24.497500  2005.000000
50%      1.00000      357.000000    1.245000   39.940000  2009.000000
75%      2.00000      999.600000    2.867500   59.332500  2011.000000
max      6.00000    17337.500000   25.000000  354.000000  2014.000000
```

- Note that `describe()` can only be applied to numeric columns. `method` column is string data and we can calculate a frequency table of values it takes on.

```
planets.method.value_counts()

Out[19]:
Radial Velocity                  553
Transit                          397
Imaging                           38
Microlensing                      23
Eclipse Timing Variations          9
Pulsar Timing                      5
Transit Timing Variations          4
Orbital Brightness Modulation      3
Astrometry                         2
Pulsation Timing Variations        1
```

group_by

- Suppose you'd like to slice the data by the `method` column and calculate the median value of `orbital_period` variable across the slices.

```
planets.groupby('method')['orbital_period'].median()
```

```
Out[20]:
method
Astrometry                          631.180000
Eclipse Timing Variations          4343.500000
Imaging                           27500.000000
Microlensing                       3300.000000
Orbital Brightness Modulation         0.342887
Pulsar Timing                        66.541900
Pulsation Timing Variations        1170.000000
Radial Velocity                     360.200000
Transit                               5.714932
Transit Timing Variations            57.011000
```

- `GroupBy` objects allows for iteration over the slices. Suppose you need to see the shape of each group when grouped by `method` column.

```
for (method, group) in planets.groupby('method'):
    print("{0:30s} shape = {1}".format(method,group.shape))
```

```
Astrometry                     shape = (2, 6)
Eclipse Timing Variations      shape = (9, 6)
Imaging                        shape = (38, 6)
Microlensing                   shape = (23, 6)
Orbital Brightness Modulation  shape = (3, 6)
Pulsar Timing                  shape = (5, 6)
Pulsation Timing Variations    shape = (1, 6)
Radial Velocity                shape = (553, 6)
Transit                        shape = (397, 6)
Transit Timing Variations      shape = (4, 6)
```

group_by

- Suppose you'd like to slice the data by the `method` column and obtain the descriptive statistics for the `distance` variable across groups.

```
planets.groupby('method')['distance'].describe()
Out[26]:
                               count         mean  ...        75%      max
method                                             ...
Astrometry                       2.0    17.875000  ...    19.3225    20.77
Eclipse Timing Variations        4.0   315.360000  ...   500.0000   500.00
Imaging                         32.0    67.715937  ...   132.6975   165.00
Microlensing                    10.0  4144.000000  ...  4747.5000  7720.00
Orbital Brightness Modulation    2.0  1180.000000  ...  1180.0000  1180.00
Pulsar Timing                    1.0  1200.000000  ...  1200.0000  1200.00
Pulsation Timing Variations      0.0          NaN  ...        NaN      NaN
Radial Velocity                530.0    51.600208  ...    59.2175   354.00
Transit                        224.0   599.298080  ...   650.0000  8500.00
Transit Timing Variations        3.0  1104.333333  ...  1487.0000  2119.00

[10 rows x 8 columns]
# also try
# planets.groupby('method')['distance'].describe().unstack()
```

group_by: aggregate

- aggregate() method can be combine with groupby(). It can take a string, a function, or a list, and compute all the aggregates at once.

- Let's calculate min, median, max for orbital_period, mass and distance by method.

```
planets.iloc[:,2:5].groupby(planets.iloc[:,0]).aggregate(['min',np.median,max])
Out[47]:
                           orbital_period              ...  distance
                                      min       median ...    median     max
method                                                 ...
Astrometry                     246.360000   631.180000 ...    17.875   20.77
Eclipse Timing Variations     1916.250000  4343.500000 ...   315.360  500.00
Imaging                       4639.150000 27500.000000 ...    40.395  165.00
Microlensing                  1825.000000  3300.000000 ...  3840.000 7720.00
Orbital Brightness Modulation    0.240104     0.342887 ...  1180.000 1180.00
Pulsar Timing                    0.090706    66.541900 ...  1200.000 1200.00
Pulsation Timing Variations   1170.000000  1170.000000 ...       NaN     NaN
Radial Velocity                  0.736540   360.200000 ...    40.445  354.00
Transit                          0.355000     5.714932 ...   341.000 8500.00
Transit Timing Variations       22.339500    57.011000 ...   855.000 2119.00

[10 rows x 9 columns]
```

group_by: filter

- `filter()` method allows for dropping data based on a user provided function. You need to define the function first and feed it into filter.

- Suppose you'd like to find the groups (based on `method`) such that there is no variation in `orbital_period`.

```
tmp = pd.DataFrame([planets.iloc[:,i].fillna(planets.iloc[:,i].dropna().mean()) \
                    for i in range(2,6)]).T
planets = pd.concat([planets.iloc[:,:2], tmp], axis = 1)

def my_filter(x):
    return np.isnan(x['orbital_period'].std())

planets.groupby('method').filter(my_filter)

Out[164]:
                          method  number  ...    distance    year
958  Pulsation Timing Variations       1  ...  264.069282  2007.0

[1 rows x 6 columns]
```

- You can confirm this again from the frequency table of `method`.

```
planets.method.value_counts()

Out[166]:
Radial Velocity                 553
Transit                         397
:
:
Pulsation Timing Variations       1
```

group_by: transform

- `transform()` allows for transforming full data. Therefore, the output will be the same shape as the input.
- Let's calculate the deviations from the mean for `orbital_period`, `mass` and `distance` after grouping by `method`.

```
planets.iloc[:,2:5].groupby(planets.iloc[:,0]).transform(lambda \
            x: x - x.mean()).head()

Out[10]:
   orbital_period       mass   distance
0      -554.05468   4.468721  16.962923
1        51.41932  -0.421279  -3.487077
2       -60.35468  -0.031279 -40.597077
3      -497.32468  16.768721  50.182923
4      -307.13468   7.868721  59.032923
```

group_by: apply

- `apply()` method allows for applying an arbitrary function to group results.

- Let's normalize `orbital_period` by subtracting its mean and then dividing by its standard deviation, after grouping by `method`.

```
def my_normalize(x):
    tmp1 = x['orbital_period'].mean()
    tmp2 = x['orbital_period'].std()
    x['orbital_period'] -= tmp1
    x['orbital_period'] /= tmp2
    return x

planets.groupby('method').apply(my_normalize)
Out[12]:
             method  number  orbital_period       mass  distance    year
0   Radial Velocity       1        0.751004   7.100000     77.40  2006.0
1   Radial Velocity       1        1.167158   2.210000     56.95  2008.0
2   Radial Velocity       1        1.090333   2.600000     19.84  2011.0
3   Radial Velocity       1        0.789995  19.400000    110.62  2007.0
4   Radial Velocity       1        0.920717  10.500000    119.47  2009.0
5   Radial Velocity       1        0.693640   4.800000     76.39  2008.0
6   Radial Velocity       1        1.784802   4.640000     18.15  2002.0
7   Radial Velocity       1        1.114733   2.638161     21.41  1996.0
8   Radial Velocity       1        1.248623  10.300000     73.10  2008.0
⋮
```

pivot_table

- A *pivot table* operation is similar to `GroupBy` but operates on tabular data.
- It is easier to think about it as a *multidimensional* version of `GroupBy` aggregation.
- Both the *split* and *combine* happen across not a one-dimensional index, but across a two-dimensional grid.
- We will use the *titanic* dataset available in `seaborn` module.

```
import pandas as pd
import numpy as np
import seaborn as sns

titanic = sns.load_dataset('titanic')
titanic.columns

Out[18]:
Index(['survived', 'pclass', 'sex', 'age', 'sibsp', 'parch', 'fare',
       'embarked', 'class', 'who', 'adult_male', 'deck', 'embark_town',
       'alive', 'alone'],
      dtype='object')
```

pivot_table

- Let's take a look at survival rate by gender.

```
titanic.groupby('sex')['survived'].mean()

Out[20]:
sex
female     0.742038
male       0.188908
```

- Approximately, three of every four females on board survived, while only one in five males survived.

- Let's go one step further and look at survival by both `sex` and `class`.

- We *group by* class and gender, *select* survival, and *apply* a mean aggregate.

```
titanic.groupby(['sex','class'])['survived'].aggregate('mean').unstack()

Out[21]:
class       First    Second     Third
sex
female   0.968085  0.921053  0.500000
male     0.368852  0.157407  0.135447
```

pivot_table

- This is an example of two-dimensional group by and the code is starting to look a bit cluttered.
- Pandas offers a convenient tool, `pivot_table`, which succintly handles this type of multi-dimensional aggreagtion.

```
titanic.pivot_table('survived', index='sex', columns='class')

Out[22]:
class      First     Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

- We can easily add another dimension to our survival analysis, say, age.
- First, we will generate a discrete age variable using the original age variable.

```
age = pd.cut(titanic['age'],[0, 18, 80])
titanic.pivot_table('survived', index=['sex', age], columns='class')

Out[27]:
class               First     Second     Third
sex    age
female (0, 18]   0.909091  1.000000  0.511628
       (18, 80]  0.972973  0.900000  0.423729
male   (0, 18]   0.800000  0.600000  0.215686
       (18, 80]  0.375000  0.071429  0.133663
```

Bibliography I

VanderPlas, J. (2017). *Python Data Science Handbook: Essential Tools for Working with Data*, O'Reilly, California.