Python String Interpolation

- A process substituting values of variables into placeholders in a string. For instance,

  - "Hello {Name of person}, nice to meet you!“
  - You would like to replace the placeholder for name of person with an actual name.
  - This process is called string interpolation.

f-strings

- Added in Python 3.6
- It provides access to embedded Python expressions inside string constants.

```
name = 'World'
program = 'Python'
print(f'Hello {name}! This is {program}')
```

```
Hello World! This is Python
```

Cont.

```
a = 12
b = 3
print(f'12 multiply 3 is {a * b}.')
```

%-formatting

- Strings in Python have a unique built-in operation that can be accessed with the % operator. Using % we can do simple string interpolation very easily.

```
print("%s %s" %('Hello','World',))
```

```
name = 'world'
program ='python'
print('Hello %s! This is %s.'%(name,program))
```

Str.format()

- In this string formatting we use format() function on a string object and braces {}, the string object in format() function is substituted in place of braces {}. We can use the format() function to do simple positional formatting, just like % formatting.

```
name = 'world'
print('Hello, {}'.format(name))
```

Python zip()

- The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.

```
languages = ['Java', 'Python', 'JavaScript']
versions = [14, 3, 6]

result = zip(languages, versions)
print(list(result))

# Output: [('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

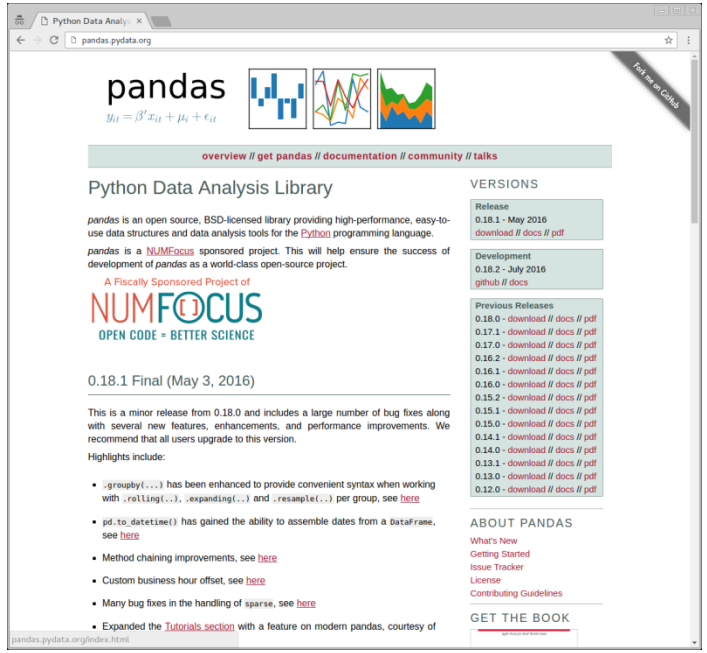
Pandas

Trainer: Mirza Touseef

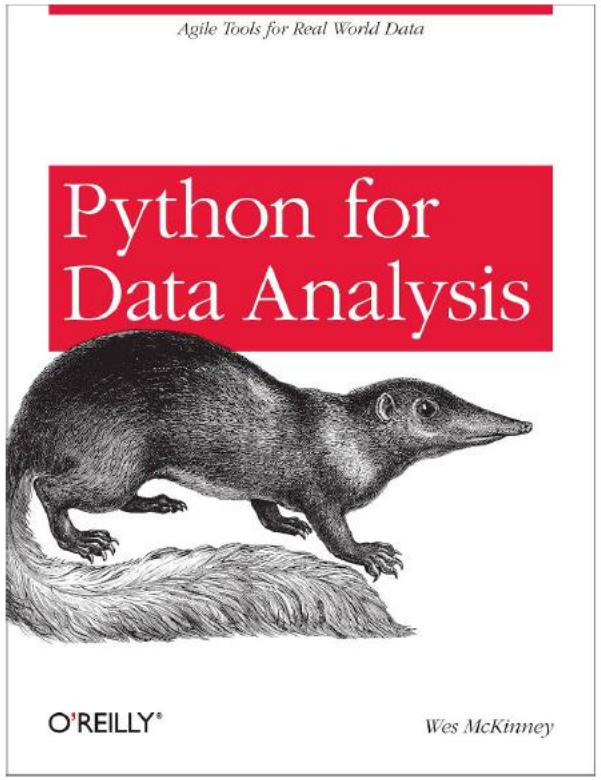
Riphah Institute of System Engineering

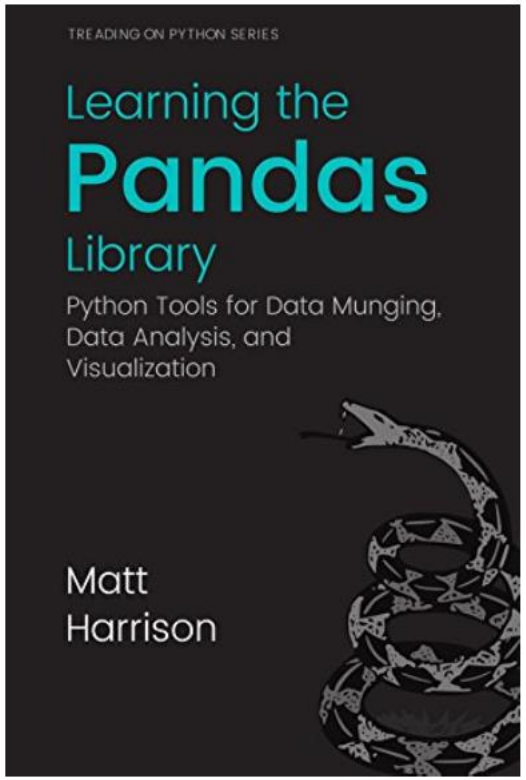
Pandas

- **Created in 2008 by Wes McKinney**
- **Open source New BSD license**
- **100 different contributors**



Pandas Books





What Problems does Pandas Solve?

- Python has long been great for data munging and preparation, but less so for data analysis and modeling

- *Pandas* helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language like R

Library Highlights

- A fast and efficient **DataFrame** object for data manipulation with integrated indexing;

- Tools for **reading and writing data** between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format;

- Intelligent **data alignment** and integrated handling of **missing data**: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form;

Library Highlights

- Flexible **reshaping** and pivoting of data sets;

- Intelligent label-based **slicing**, **fancy indexing**, and **subsetting** of large data sets;

and so on……

Installing Pandas

- After installing the Anaconda package, you should have a conda executable

Running:$ conda install pandas

Installing Pandas

- Will install pandas and any dependencies. To verify that this works, simply try to import the pandas package:

```
$ python
>>> import pandas
>>> pandas.__version_'0.18.0'
```

- If the library successfully imports, you should be good to go.

Data Structures

- ONE OF THE KEYS TO UNDERSTANDING PANDAS IS TO UNDERSTAND THE DATA model. At the core of pandas are three data structures:

Different dimensions of pandas data structures

Data Structure	Dimensionality	Spreadsheet Analog
`Series`	1D	Column
`DataFrame`	2D	Single Sheet
`Panel`	3D	Multiple Sheets

DataFrame

- The most widely used data structures are the Series and the DataFrame that deal with array data and tabular data respectively

- An analogy with the spreadsheet world illustrates the basic differences between these types

- A DataFrame is similar to a sheet with rows and columns

Series

- while a Series is similar to a single column of data.

Panel

- A Panel is a group of sheets. Likewise, in pandas a Panel can have many DataFrames, each which in turn may have multiple Series.

Series

	age
0	15
1	16
2	16
3	15

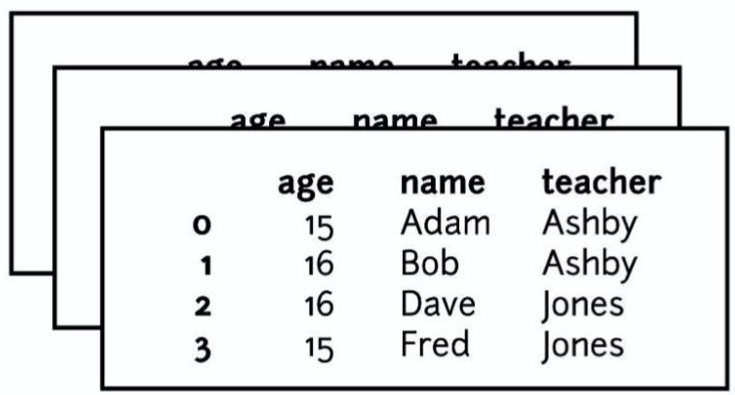
	teacher
0	Ashby
1	Ashby
2	Jones
3	Jones

	name
0	Adam
1	Bob
2	Dave
3	Fred

Data Frame

	age	name	teacher
0	15	Adam	Ashby
1	16	Bob	Ashby
2	16	Dave	Jones
3	15	Fred	Jones

Panel



Series

- A SERIES IS USED TO MODEL ONE DIMENSIONAL DATA, SIMILAR TO A LIST IN Python

- The Series object also has a few more bits of data, including an index and a name

- A common idea through pandas is the notion of an axis. Because a series is one dimensional, it has a single *axis—the index.*

Series

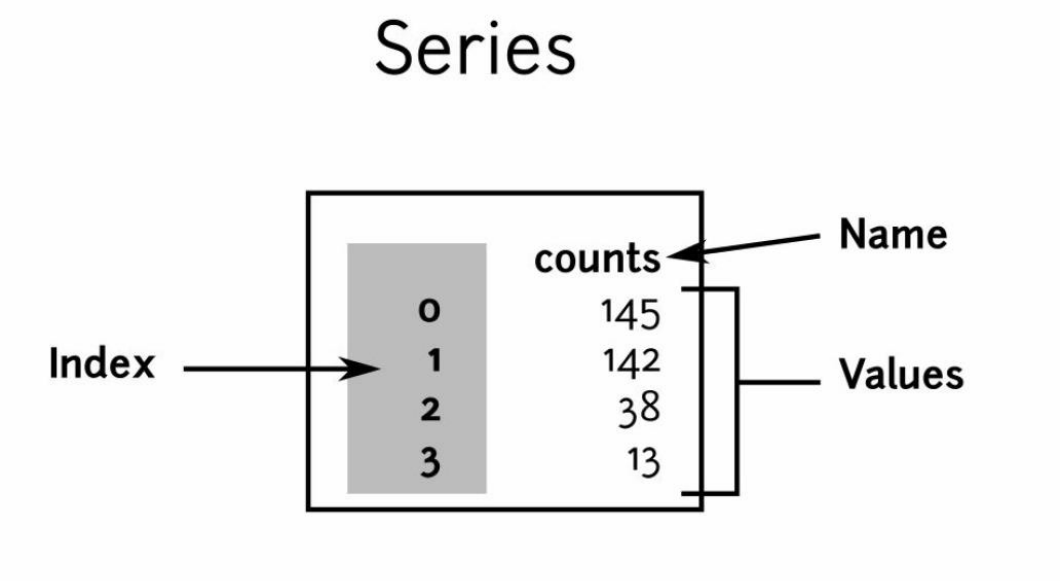
Artist	Data
0	145
1	142
2	38
3	13

Series Example in Python

```
import pandas as pd
ser1 = pd.Series(data=[145,142,38,13], index=['a','b','c','d'], name='counts')
print(ser1)
```

✓ 0.0s

```
a    145
b    142
c     38
d     13
Name: counts, dtype: int64
```



Index of Series

```
ser1.index
```

✓ 0.0s

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
ser1.index
```

✓ 0.0s

```
RangeIndex(start=0, stop=4, step=1)
```

Note

The index can be string based as well, in which case pandas indicates that the datatype for the index is `object` (not string):

```
>>> songs3 = pd.Series([145, 142, 38, 13],
...      name='counts',
...      index=['Paul', 'John', 'George', 'Ringo'])
```

Note that the `dtype` that we see when we print a `Series` is the type of the values, not of the index:

```
>>> songs3
Paul      145
John      142
George     38
Ringo      13
Name: counts, dtype: int64
```

When we inspect the index attribute, we see that the `dtype` is `object`:

```
>>> songs3.index
Index(['Paul', 'John', 'George', 'Ringo'],
dtype='object')
```

Similar to NumPy

- The Series object behaves similarly to a NumPy array. As show below, both types respond to index operations:

```
>>> import numpy as np
>>> numpy_ser = np.array([145, 142, 38, 13])
>>> songs3[1]
142
>>> numpy_ser[1]
142
```

Common to NumPy

```
>>> songs3.mean()
84.5
```

```
>>> mask = songs3 > songs3.median()  # boolean array

>>> mask
Paul       True
John       True
George    False
Ringo     False
Name: counts, dtype: bool
```

```
mask = ser1>ser1.mean()
mask
```

✓ 0.0s

```
0     True
1     True
2    False
3    False
Name: counts, dtype: bool
```

```
mask = ser1 > ser1.median()
mask
```

✓ 0.0s

```
0     True
1     True
2    False
3    False
Name: counts, dtype: bool
```

Series CRUD

- THE PANDAS SERIES DATA STRUCTURE PROVIDES SUPPORT FOR THE BASIC CRUD operations—create, read, update, and delete

- One thing to be aware of is that in general pandas objects tend to behave in an immutable manner

- Although they are mutable, you don’t normally update a series, but rather perform an operation that will return a new Series

Creation

- Here we create a series with the count of songs attributed to George Harrison during the final years of The Beatles and the release of his 1970 album

```
>>> george_dupe = pd.Series([10, 7, 1, 22],
...    index=['1968', '1969', '1970', '1970'],
...    name='George Songs')

>>> george_dupe
1968    10
1969     7
1970     1
1970    22
Name: George Songs, dtype: int64
```

Reading

- To read or select the data from a series, one can simply use an index operation in combination with the index entry:

```
>>> george_dupe['1968']
10
```

```
# may not be a scalar!
>>> george_dupe['1970']
1970     1
1970    22
Name: George Songs, dtype: int64
```

Reading

- We can iterate over data in a series as well. When iterating over a series, we loop over the values of the series:

```
>>> for item in george_dupe:
...     print(item)
10
7
1
22
```

Updating

- Updating values in a series can be a little tricky as well. To update a value for a given index label, the standard index assignment operation works and performs the update in-place (in effect mutating the series):

```
>>> george_dupe['1969'] = 6
>>> george_dupe['1969']
6
```

```
>>> george_dupe['1973'] = 11
>>> george_dupe
1968    10
1969     6
1970     1
1970    22
1973    11
Name: George Songs, dtype: int64
```

Deletion

- Deletion is not common in the pandas world. It is more common to use filters or masks to create a new series that has only the items that you want
- However, if you really want to remove entries, you can delete based on index entries

```
>>> del george_dupe['1973']

>>> george_dupe
1968    10
1969     6
1970     2
1970    22
1974     9
Name: George Songs, dtype: int64
```

Series as one-dimensional array: Data

A	1
B	2
C	3
D	4
E	5
F	6

Series Indexing

- AS ILLUSTRATED WITH OUR example series, the index does not have to be whole numbers. Here we use strings for the index:

```
>>> george = pd.Series([10, 7],
...    index=['1968', '1969'],
...    name='George Songs')

>>> george
1968    10
1969     7
Name: George Songs, dtype: int64
```

george's index type is `object` (pandas indicates that strings index entries are objects), note the `dtype` of the index attribute:

```
>>> george.index
Index(['1968', '1969'], dtype='object')
```

Series Indexing

- We have previously seen that indexes do not have to be unique. To determine whether an index has duplicates, simply inspect the .is_unique attribute on the index:

```
>>> dupe = pd.Series([10, 2, 7],
...    index=['1968', '1968', '1969'],
...    name='George Songs')

>>> dupe.index.is_unique
False

>>> george.index.is_unique
True
```

Series Indexing

- Much like numpy arrays, a Series object can be both indexed and sliced along the axis. *Indexing pulls out either a scalar or multiple values* (if there are non-unique index labels):

```
>>> george
1968    10
1969     7
Name: George Songs, dtype: int64
```

```
>>> george[0]
10
```

.loc, .iloc

- While standard Python / Numpy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, .at, .iat, .loc, .iloc and .ix.

```
>>> george.iloc[0]
10

>>> george.iloc[-1]
7
```

.loc, .iloc

```
>>> george.iloc[0:3]  # slice
1968    10
1969     7
Name: George Songs, dtype: int64
```

```
>>> george.iloc[[0,1]]  # list
1968    10
1969     7
Name: George Songs, dtype: int64
```

.loc, .iloc

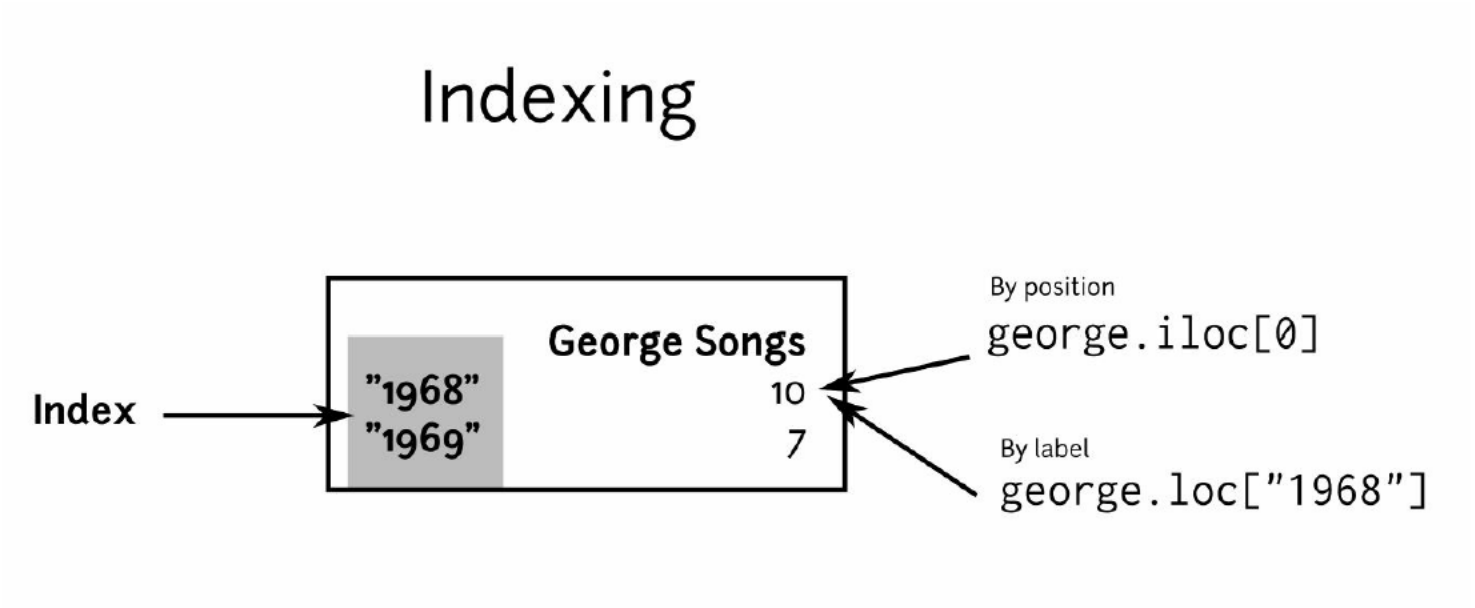


Figure showing how `iloc` and `loc` behave.

.at and .iat

- The .at and .iat index accessors are analogous to .loc and .iloc. The difference being that they will return a numpy.ndarray when pulling out a duplicate value, whereas .loc and .iloc return a Series:

```
>>> george_dupe = pd.Series([10, 7, 1, 22],
...       index=['1968', '1969', '1970', '1970'],
...       name='George Songs')

>>> george_dupe.at['1970']
array([ 1, 22])

>>> george_dupe.loc['1970']
1970     1
1970    22
Name: George Songs, dtype: int64
```

Series Methods

- A SERIES OBJECT HAS MANY ATTRIBUTES AND METHODS THAT ARE USEFUL FOR DATA analysis

```
>>> songs_66 = pd.Series([3, None , 11, 9],
...     index=['George', 'Ringo', 'John', 'Paul'],
...     name='Counts')

>>> songs_69 = pd.Series([18, 22, 7, 5],
...     index=[ 'John', 'Paul', 'George', 'Ringo'],
...     name='Counts')
```

Iteration

- Itera

```
>>> for value in songs_66:
...     print(value)
3.0
nan
11.0
9.0
```

There is an `.iteritems` method to loop over the index, va

```
>>> for idx, value in songs_66.iteritems():
...     print(idx, value)
George 3.0
Ringo nan
John 11.0
Paul 9.0
```

.unique and .nunique

- To get the unique values or the count of non-NaN items use the .unique and .nunique methods respectively. Note that .unique includes the nan value, but .nunique does not count it:

```
>>> scores2.unique()
array([  67.3,  100. ,   96.7,    nan])

>>> scores2.nunique()
3
```

.drop_duplicates

- Dealing with duplicate values is another feature of pandas. To drop duplicate values use the .drop_duplicates method. Since Billy has the same score as Paul, he will get dropped:

```
>>> scores2.drop_duplicates()
Ringo      67.3
Paul      100.0
George     96.7
Peter       NaN
Name: test2, dtype: float64
```

.duplicated

- To retrieve a series with boolean values indicating whether its value was repeated, use the .duplicated method:

```
>>> scores2.duplicated()
Ringo     False
Paul      False
George    False
Peter     False
Billy      True
Name: test2, dtype: bool
```

.groupby method

- To drop duplicate index entries requires a little more effort. Lets create a series, scores3, that has 'Paul' in the index twice

- If we use the .groupby method, and group by the index, we can then take the first or last item from the values for each index label:

```
>>> scores3 = pd.Series([67.3, 100, 96.7, None, 100, 79],
...     index=['Ringo', 'Paul', 'George', 'Peter', 'Billy',
...            'Paul'])

>>> scores3.groupby(scores3.index).first()
Billy     100.0
George     96.7
Paul      100.0
Peter       NaN
Ringo      67.3
dtype: float64
```

```
>>> scores3.groupby(scores3.index).last()
Billy     100.0
George     96.7
Paul       79.0
Peter       NaN
Ringo      67.3
dtype: float64
```

Statistics

- There are many basic statistical measures in a series object's methods. We will look at a few of them in this section

- One of the most basic measurements is the sum of the values in a series:

```
>>> songs_66.sum()
23.0
```

Mean and Median

- Calculating the *mean (the "expected value" or average) and the median* (the "middle" value at 50% that separates the lower values from the upper values) is simple
- As discussed, both of these methods ignore NaN (unless skipna is set to False):

```
>>> songs_66.mean()
7.666666666666667

>>> songs_66.median()
9.0
```

NumPy vs Pandas

NumPy	Pandas
Low level Data Structure	High level Data Structure
Support for multi-dimensional arrays	Handling of tabular data, time series data
Large Mathematical Operations	Data alignment and handling missing data
Pandas runs on top of NumPy	Can be used with NumPy, once data is structured through Pandas

Example

- How to combine many series to form a dataframe?

  - import numpy as np
  - ser1 = pd.Series(list('abcedfghijklmnopqrstuvwxyz'))
  - ser2 = pd.Series(np.arange(26))

Solutions

```
# Solution 1
df = pd.concat([ser1, ser2], axis=1)
# Solution 2
df = pd.DataFrame({'col1': ser1, 'col2': ser2})
print(df.head())
```

Output

	col1	col2
0	a	0
1	b	1
2	c	2
3	e	3
4	d	4

DataFrames

- THE TWO-DIMENSIONAL COUNTERPART TO THE ONE-DIMENSIONAL SERIES IS THE DataFrame

- A DataFrame, is often used for analytical purposes and is better understood when thought of as column oriented, where each column is a Series

DataFrames

```
>>> import pandas as pd
>>> df = pd.DataFrame({
...    'growth':[.5, .7, 1.2],
...    'Name':['Paul', 'George', 'Ringo'] })

>>> df
     Name  growth
0    Paul     0.5
1  George     0.7
2   Ringo     1.2
```

DataFrames

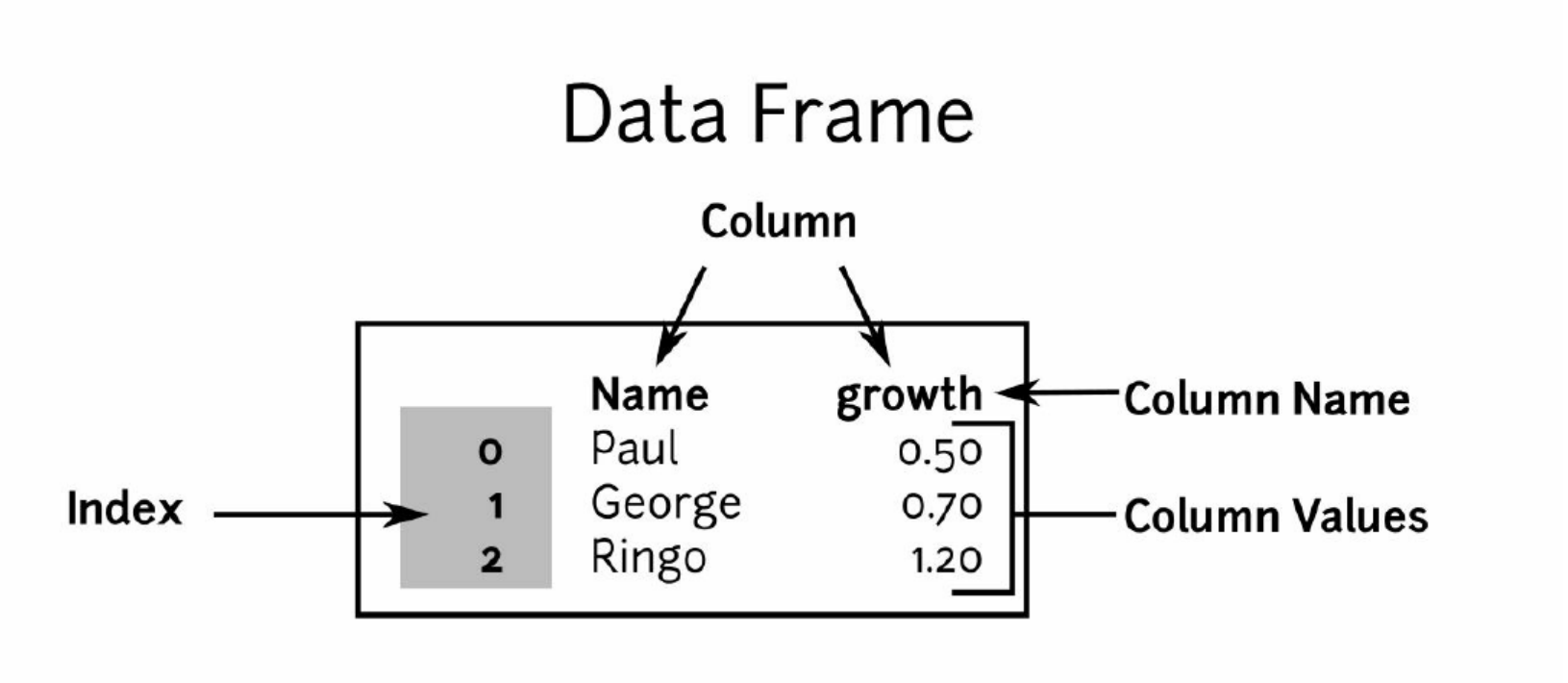


Figure showing column oriented nature of Data Frame. (Note that a column can be pulled off as a `Series`)

Accessing Rows

To access a row by location, index off of the `.iloc` attribute:

```
>>> df.iloc[2]
Name      Ringo
growth      1.2
Name: 2, dtype: object
```

Accessing Columns

Columns are accessible via indexing the column name off of the object:

```
>>> df['Name']
0      Paul
1    George
2     Ringo
Name: Name, dtype: object
```

Columns is equal to Series

- Note the type of column is a pandas Series instance. Any operation that can be done to a series can be applied to a column:

```
>>> type(df['Name'])
<class 'pandas.core.series.Series'>

>>> df['Name'].str.lower()
0      paul
1    george
2     ringo
Name: Name, dtype: object
```

Construction

- Data frames can be created from many types of input:
  - columns (dicts of lists)
  - rows (list of dicts)
  - CSV file (pd.read_csv)
  - from NumPy ndarray
  - And more, SQL, HDF5, etc

Creating a dataframe from rows

- The previous creation of **df** illustrated making a data frame from columns. Below is an example of creating a data frame from rows:

```
>>> pd.DataFrame([
...   {'growth':.5, 'Name':'Paul'},
...   {'growth':.7, 'Name':'George'},
...   {'growth':1.2, 'Name':'Ringo'}])
     Name  growth
0    Paul     0.5
1  George     0.7
2   Ringo     1.2
```

Reading a csv file

```
>>> csv_file = StringIO("""growth,Name
... .5,Paul
... .7,George
... 1.2,Ringo""")

>>> pd.read_csv(csv_file)
   growth    Name

 0     0.5    Paul
 1     0.7  George
 2     1.2   Ringo
```

Instantiated from a NumPy array

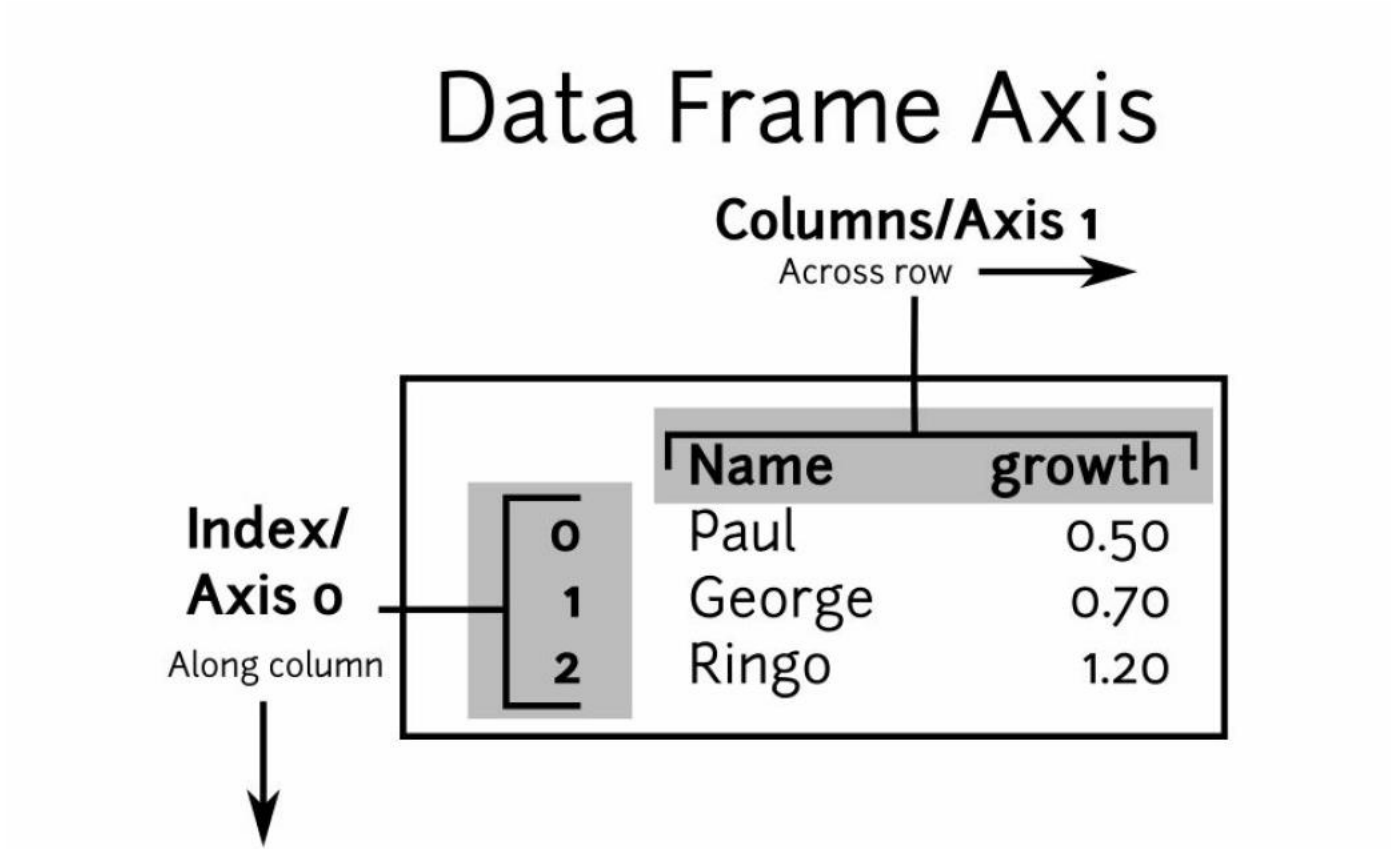
- A data frame can be instantiated from a NumPy array as well. The column names will need to be specified:

```
>>> pd.DataFrame(np.random.randn(10,3), columns=['a', 'b', 'c'])
          a         b         c
0  0.926178  1.909417 -1.398568
1  0.562969 -0.650643 -0.487125
2 -0.592394 -0.863991  0.048522
3 -0.830950  0.270457 -0.050238
4 -0.238948 -0.907564 -0.576771
5  0.755391  0.500917 -0.977555
6  0.099332  0.751387 -1.669405
7  0.543360 -0.662624  0.570599
8 -0.763259 -1.804882 -1.627542
9  0.048085  0.259723 -0.904317
```

Data Frame Axis

- Unlike a series, which has one axis, there are two axes for a data frame

- They are commonly referred to as axis 0 and 1, or the row/index axis and the columns axis respectively:


Data Frame Axis
Columns/Axis 1
Across row
Name
growth
Index/
Axis 0
Along column
0
1
2
Paul
0.50
George
0.70
Ringo
1.20

Lets Run through Code…

Data Frame Example

- BEFORE DISCUSSING DATA FRAMES IN DETAIL, LET'S COVER WORKING WITH A small data set. Below is some data from a portion of trail data of the Wasatch 100 trail race

- Source: https://wasatch100.com/?option=com_content&view=article&id=132&Itemid=10

Data Looks like this…

LOCATION	MILES	ELEVATION	CUMUL	% CUMUL GAIN
Big Mountain Pass Aid Station	39.07	7432	11579	43.8%
Mules Ear Meadow	40.75	7478	12008	45.4%
Bald Mountain	42.46	7869	12593	47.6%
Pence Point	43.99	7521	12813	48.4%
Alexander Ridge Aid Station	46.9	6160	13169	49.8%
Alexander Springs	47.97	5956	13319	50.3%
Rogers Trail junction	49.52	6698	13967	52.8%
Rogers Saddle	49.77	6790	14073	53.2%
Railroad Bed	50.15	6520		
Lambs Canyon Underpass Aid Station	52.48	6111	14329	54.2%
Lambs Trail	54.14	6628	14805	56.0%

Reading in CSV file

- We'll load this data into a data frame and use it data to show basic CRUD operations and plotting

- Reading in CSV files is straightforward in pandas

```
>>> df = pd.read_csv(data)
```

Now that the data is loaded, it can easily be examined:

```
>>> df
                              LOCATION  MILES  ELEVATION    CUMUL % CUMUL
GAIN
0       Big Mountain Pass Aid Station  39.07       7432  11579.0
43.8%
1                    Mules Ear Meadow  40.75       7478  12008.0
45.4%
2                       Bald Mountain  42.46       7869  12593.0
47.6%
3                         Pence Point  43.99       7521  12813.0
48.4%
4          Alexander Ridge Aid Station  46.90       6160  13169.0
49.8%
5                    Alexander Springs  47.97       5956  13319.0
50.3%
6                Rogers Trail junction  49.52       6698  13967.0
52.8%
7                       Rogers Saddle  49.77       6790  14073.0
53.2%
8                        Railroad Bed  50.15       6520       NaN
NaN
9  Lambs Canyon Underpass Aid Station  52.48       6111  14329.0
54.2%
```

Line Wraping

```
>>> print(df.to_string(line_width=60))
                            LOCATION  MILES  ELEVATION  \
0       Big Mountain Pass Aid Station  39.07       7432
1                   Mules Ear Meadow  40.75       7478
2                      Bald Mountain  42.46       7869
3                        Pence Point  43.99       7521
4         Alexander Ridge Aid Station  46.90       6160
5                   Alexander Springs  47.97       5956
6              Rogers Trail junction  49.52       6698
7                      Rogers Saddle  49.77       6790
8                       Railroad Bed  50.15       6520
9  Lambs Canyon Underpass Aid Station  52.48       6111
```

Looking at the data

- In addition to just looking at the string representation of a data frame, the .describe method provides summary statistics of the numeric data

- It returns the count of items, the average value, the standard deviation, and the range and quantile data for every column that is a float or and integer

Looking at the data

```
>>> df.describe()
            MILES    ELEVATION          CUMUL
count   10.000000    10.000000       9.000000
mean    46.306000  6853.500000  13094.444444
std      4.493574   681.391428    942.511686
min     39.070000  5956.000000  11579.000000
25%     42.842500  6250.000000  12593.000000
50%     47.435000  6744.000000  13169.000000
75%     49.707500  7466.500000  13967.000000
max     52.480000  7869.000000  14329.000000
```

Adding rows

- If we wanted to combine the data with other portions of the trail, it requires using the .concat function or the .append method

```
>>> df2 = pd.DataFrame([('Lambs Trail',54.14,6628,14805,
...     '56.0%')], columns=['LOCATION','MILES','ELEVATION',
...     'CUMUL','% CUMUL GAIN'])
>>> print(pd.concat([df, df2]).to_string(line_width=60))
                          LOCATION  MILES  ELEVATION  \
0       Big Mountain Pass Aid Station  39.07       7432
1                   Mules Ear Meadow  40.75       7478
2                      Bald Mountain  42.46       7869
3                        Pence Point  43.99       7521
4          Alexander Ridge Aid Station  46.90       6160
5                    Alexander Springs  47.97       5956
6                 Rogers Trail junction  49.52       6698
7                        Rogers Saddle  49.77       6790
8                         Railroad Bed  50.15       6520
9  Lambs Canyon Underpass Aid Station  52.48       6111
0                          Lambs Trail  54.14

     CUMUL % CUMUL GAIN
0  11579.0        43.8%
1  12008.0        45.4%
2  12593.0        47.6%
3  12813.0        48.4%
4  13169.0        49.8%
5  13319.0        50.3%
6  13967.0        52.8%
7  14073.0        53.2%
8      NaN          NaN
9  14329.0        54.2%
0  14805.0        56.0%
```

Deleting Rows

- The pandas data frame has a .drop method that takes a sequence of index values. It returns a new data frame without those index entries. To remove the items found in index 5 and 9 use the following:

```
>>> df.drop([5, 9])
                          LOCATION  MILES  ELEVATION  CUMUL % CUMUL GAIN
STATION
0   Big Mountain Pass Aid Station  39.07       7432  11579        43.8%
True
1               Mules Ear Meadow  40.75       7478  12008        45.4%
False
2                  Bald Mountain  42.46       7869  12593        47.6%
False
3                    Pence Point  43.99       7521  12813        48.4%
False
4     Alexander Ridge Aid Station  46.90       6160  13169        49.8%
True
6           Rogers Trail junction  49.52       6698  13967        52.8%
False
7                  Rogers Saddle  49.77       6790  14073        53.2%
False
8                   Railroad Bed  50.15       6520    NaN          NaN
False
10                   Lambs Trail  54.14       6628  14805        56.0%
False
```

Deleting Columns

- To delete columns, use the .pop method, the .drop method with axis=1, or the del statement

```
>>> bogus = df.pop('bogus')
```

The bogus object is now a series holding the column removed from the data frame:

```
>>> bogus
0      0
1      1
2      2
3      3
4      4
5      5
6      6
7      7
8      8
9      9
10    10
Name: bogus, dtype: int64
```

Data Frame Methods

Iteration

- Data frames include a variety of methods to iterate over the values. By default, iteration occurs over the column names:

```
>>> for column in sales:
...     print(column)
UPC
Units
Sales
Date
```

The `.iteritems` method returns pairs of column names and the individual column (as a `Series`):

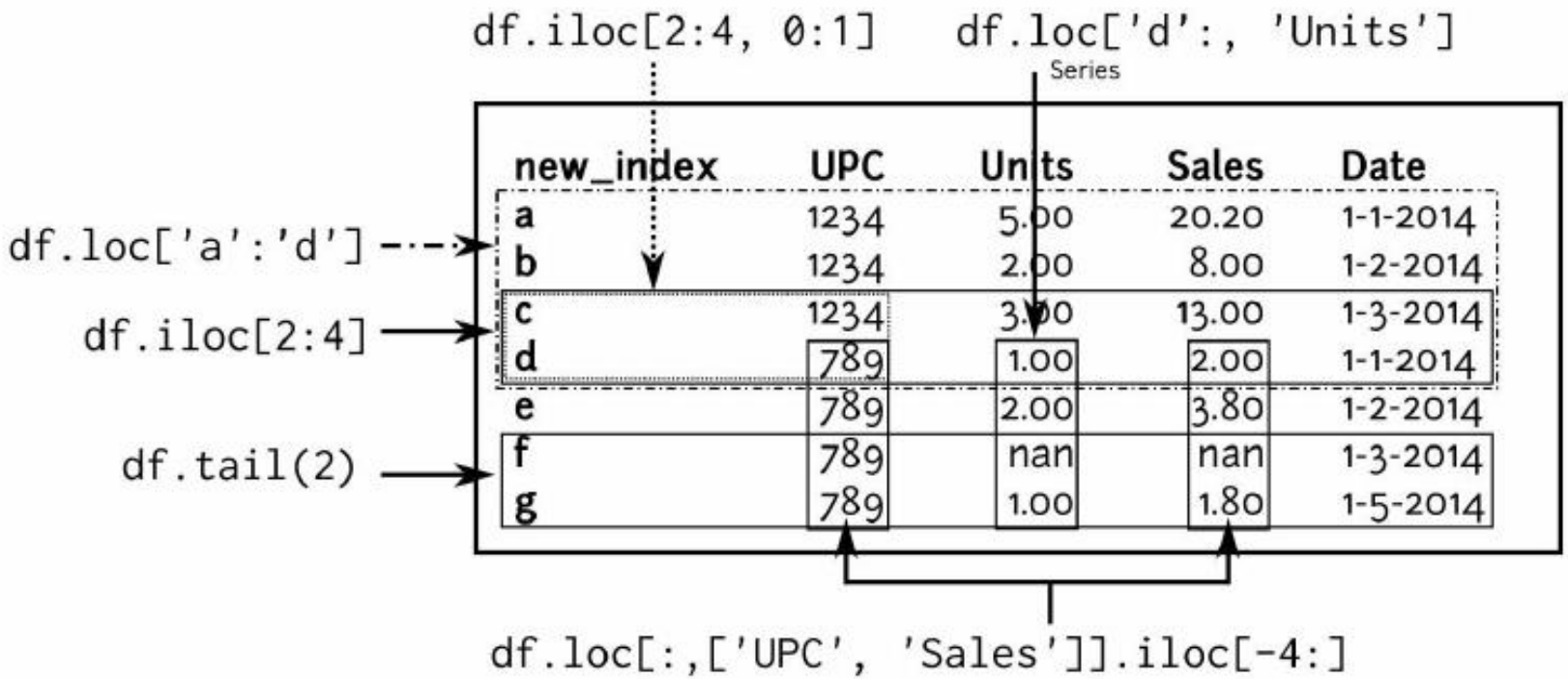
```
>>> for col, ser in sales.iteritems():
...     print(col, ser)
UPC 0    1234
1    1234
2    1234
3     789
4     789
5     789
6     789
Name: UPC, dtype: int64
Units 0    5.0
1    2.0
2    3.0
3    1.0
4    2.0
5    NaN
6    1.0
Name: Units, dtype: float64
Sales 0    20.2
1     8.0
2    13.0
3     2.0
4     3.8
5     NaN
6     1.8
Name: Sales, dtype: float64
Date 0    1-1-2014
1    1-2-2014
2    1-3-2014
```

Matrix Operations

- The data frame can be treated as a matrix. There is support for transposing a matrix:

```
>>> sales.transpose()  # sales.T is a shortcut
              0         1         2         3         4         5         6
UPC        1234      1234      1234       789       789       789       789
Units         5         2         3         1         2       NaN         1
Sales      20.2         8        13         2       3.8       NaN       1.8
Date   1-1-2014  1-2-2014  1-3-2014  1-1-2014  1-2-2014  1-3-2014  1-5-2014
```

Data Frame Slicing Examples



Sorting

Here is the sales data frame:

```
>>> sales
    UPC Category  Units  Sales      Date
0  1234     Food    5.0   20.2  1-1-2014
1  1234     Food    2.0    8.0  1-2-2014
2  1234     Food    3.0   13.0  1-3-2014
3   789     Food    1.0    2.0  1-1-2014
4   789     Food    2.0    3.8  1-2-2014
5   789     Food    NaN    NaN  1-3-2014
6   789     Food    1.0  789.0  1-5-2014
```

To sort by column, use `.sort_values`. Let's sort the `UPC` column:

```
>>> sales.sort_values('UPC')
    UPC Category  Units  Sales      Date
3   789     Food    1.0    2.0  1-1-2014
4   789     Food    2.0    3.8  1-2-2014
5   789     Food    NaN    NaN  1-3-2014
6   789     Food    1.0  789.0  1-5-2014
0  1234     Food    5.0   20.2  1-1-2014
1  1234     Food    2.0    8.0  1-2-2014
2  1234     Food    3.0   13.0  1-3-2014
```

Pandas **dataframe.idxmax()**

- Pandas **dataframe.idxmax()** function returns index of first occurrence of maximum over requested axis

- While finding the index of the maximum value across any index, all NA/null values are excluded

Pandas **dataframe.idxmax()**

```
# importing pandas as pd
import pandas as pd

# Creating the dataframe
df = pd.DataFrame({"A":[4, 5, 2, 6],
                   "B":[11, 2, 5, 8],
                   "C":[1, 8, 66, 4]})

# Print the dataframe
df
# applying idxmax() function.
df.idxmax(axis = 0)
```

	A	B	C
0	4	11	1
1	5	2	8
2	2	5	66
3	6	8	4

```
A    3
B    0
C    2
dtype: int64
```

Pandas Cheat Sheet

- We cannot cover all the basic and advanced methods for pandas

- But here is a cheat sheet for your assistance

Planet Python

- **http://planetpython.org/**
- **Excellent blog aggregator for python related news**
- **Significant number of data science and python tutorials are posted**
- **Great blend of applied beginner and higher level python postings**

Data Skeptic

- **http://dataskeptic.com/**
- **Kyle Polich, created in 2014**
- **Covers data science more generally, including:**
  - ***Mini educational lessons***
  - ***Interviews***
  - ***Trends***
  - ***Shared community project***
  - ***(OpenHouse)***

Thank you…

- Questions please……