pandas

$$y_i t = \beta' x_{it} + \mu_i + \epsilon_{it}$$

Advanced Munging
Descriptive Statistics
Basic Visualizations

Presented by @jessecdaniel

Denver Data Science Day 2017

Recoding Data

There are many times when you want to take a code or abbreviation, and change the column or add a new columns based on those values.

```
df = DataFrame({'storecode': ['P', 'P', 'D', 'L', 'P', 'D'],
                'data': range(5)})
df2 = df
df2
```

	data	storecode
0	0	P
1	1	D
2	2	L
3	3	P
4	4	D

One way to do this is with **`np.where()`**.

```
df2['Store'] = np.where(df.storecode=='P', 'Parker',
                np.where(df.storecode=='D','DU',
                'Littleton'))
df2
```

	data	storecode	Store
0	0	P	Parker
1	1	D	DU
2	2	L	Littleton
3	3	P	Parker
4	4	D	DU

Recoding Data

A second way is with pandas **`replace`** function. In the first version, we will just replace the storecode column with new values.

```
df3=df.replace(to_replace={'storecode':{'P':'Parker',
    'D':'DU','L':'Littleton'}})
df3
```

```
   data  storecode
0     0     Parker
1     1         DU
2     2  Littleton
3     3     Parker
4     4         DU
```

In the second version, we will keep the original storecode column and add a new column.

```
df['Store']=df['storecode'].replace(to_replace=['P','D','L'],
                                    value=['Parker','DU','Littleton'])
df
```

```
   data storecode       Store
0     0          P     Parker
1     1          D         DU
2     2          L  Littleton
3     3          P     Parker
4     4          D         DU
```

Binning Continuous Data

Continuous data is often discretized or otherwise separated into 'bins' for analysis. For example, suppose you have ages of customers in your database and you want to group them into discrete age groups or bins.

Suppose we want to divide these into bins of 18 to 25, 26 to 35, 35 to 60, and 60 and older. We can use the **`cut`** function to do this in pandas.

What is returned is a special **Categorical** object that is like an array of strings indicating the bin name for each original value. The **`categories`** method indicates the distinct category names. Note that a parenthesis indicates 'open' and a closed bracket indicates a 'closed' interval.

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
```

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60,
100], (35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, object): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

NOTE: if you just want to view the categories, use `cats.categories`.

Binning Continuous Data

We can use the **`value_counts()`** method to get counts for each category.

```
pd.value_counts(cats)
```

```
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

You can change the closed side of the interval to the left side by passing **`right=False`**.

```
pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

```
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61),
[36, 61), [26, 36)]
Length: 12
Categories (4, object): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

You can also pass your own bin names by passing a list or array to the **`labels`** option.

```
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
cats2=pd.cut(ages, bins, labels=group_names)
pd.value_counts(cats2)
```

```
Youth          5
MiddleAged     3
YoungAdult     3
Senior         1
dtype: int64
```

Binning Continuous Data

If you pass **`cut`** an integer number of bins instead of explicit bin edges, it will compute *equal-length bins* based on the min and max of the values in the data. Consider some uniformly distributed data that we chop in to fourths.

```
data = np.random.rand(20)
cats = pd.cut(data, 4, precision=2)
pd.value_counts(cats)
```

```
(0.76, 0.99]     8
(0.045, 0.28]    7
(0.52, 0.76]     3
(0.28, 0.52]     2
dtype: int64
```

```
data
```

```
array([ 0.95433786,  0.26697001,
0.56281362,
        0.74916291,  0.22456987,
0.04550157,
```

NOTE: precision=2 means the ranges will have 2 significant digits.

Another option is the **`qcut`** function that bins the data into roughly *equally-sized groups*. It does this by using sample quantiles of the data.

```
data = np.random.randn(1000) # Normally distributed
cats = pd.qcut(data, 4) # Cut into quartiles
cats
pd.value_counts(cats)
```

```
(0.696, 2.819]       250
(0.00927, 0.696]     250
(-0.668, 0.00927]    250
[-3.67, -0.668]      250
dtype: int64
```

```
data
```

```
array([ -6.09599405e-02,  -5.61872572e-01,
        -7.74531533e-01,   1.66921411e-01,
        -9.24654486e-02,  -6.38065982e-01,
```

Binning Continuous Data

Finally, you can use **`qcut`** and specify your own quantiles (these should be numbers between 0 and 1, inclusive.)

```
data
```

```
array([ -6.09599405e-02,  -5.61872572e-01,
        -7.74531533e-01,   1.66921411e-01,
        -9.24654486e-02,  -6.38065982e-01,
```

```
cats = pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
pd.value_counts(cats)
```

```
(0.00927, 1.226]      400
(-1.237, 0.00927]     400
(1.226, 2.819]        100
[-3.67, -1.237]       100
dtype: int64
```

Indicator/Dummy/OneHot Encoding

A common way to transform data is to take a categorical variable and convert it into a 'dummy' or 'indicator' matrix. This basically means that each category gets its own column and it will contain 1 if the value is that category and 0 if not.

```
df = DataFrame(dict(type= ['b', 'b', 'a', 'c', 'a', 'b'],
                    data1= range(6)))
df
```

	data1	type
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	b

Indicator/Dummy/OneHot Encoding

pandas has a **get_dummies** function to help us create these dummies automatically.

```
df
```

	data1	type
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	b

```
pd.get_dummies(df.type)
```

	a	b	c
0	0.0	1.0	0.0
1	0.0	1.0	0.0
2	1.0	0.0	0.0
3	0.0	0.0	1.0
4	1.0	0.0	0.0
5	0.0	1.0	0.0

You can use the **prefix** option to indicate a prefix for the new dummies. And we can use the **join** to merge back with the original data1 column.

```
dummies = pd.get_dummies(df.type, prefix='type')
df_with_dummy = DataFrame(df.data1).join(dummies)
df_with_dummy
```

	data1	type_a	type_b	type_c
0	0	0.0	1.0	0.0
1	1	0.0	1.0	0.0
2	2	1.0	0.0	0.0
3	3	0.0	0.0	1.0
4	4	1.0	0.0	0.0
5	5	0.0	1.0	0.0

NOTE: we use DataFrame(df.data1) because df.data1 will return a Series.

Combining/Merging DataFrames

Data contained in pandas objects can be combined together in a number of built-in ways:

`pandas.join` glues columns from DataFrames together

`pandas.concat` glues or stacks together objects along an axis

`pandas.merge` connects rows in DataFrames based on one or more keys matching columns). This is similar to joining in a relational database.

`combine_first` method enables splicing together overlapping data to fill in missing values in one object with values from another

Concatenating DataFrames

```
dfJan = DataFrame({'SalesmanID': [1, 2, 3, 3, 2],
                   'data': range(5)})
dfFeb = DataFrame({'SalesmanID': [2,3,2,4],
                   'data': range(4)})
```

dfJan

	SalesmanID	data
0	1	0
1	2	1
2	3	2
3	3	3
4	2	4

dfFeb

	SalesmanID	data
0	2	0
1	3	1
2	2	2
3	4	3

To concatenate or join two DataFrames on top of each other, use **`pd.concat()`**.

```
dfall = pd.concat([dfJan,dfFeb])
```

	SalesmanID	data
0	1	0
1	2	1
2	3	2
3	3	3
4	2	4
0	2	0
1	3	1
2	2	2
3	4	3

To reset the index to 0, 1, 2, … Use drop=True so won't keep old index.

```
dfall.reset_index(drop=True)
```

	SalesmanID	data
0	1	0
1	2	1
2	3	2
3	3	3
4	2	4
5	2	0
6	3	1
7	2	2
8	4	3

Merging DataFrames

Let's define df1 and df2.

```
df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                 'data1': range(7)})
df2 = DataFrame({'key': ['a', 'b', 'd'],
                 'data2': range(3)})
```

df1

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	a
6	6	b

df2

	data2	key
0	0	a
1	1	b
2	2	d

Merging DataFrames

df1

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	a
6	6	b

df2

	data2	key
0	0	a
1	1	b
2	2	d

This is a **many-to-one** merge where the data in df1 has multiple rows labeled a and b and where df2 has only one row for each key value.

```
pd.merge(df1, df2, on='key')
pd.merge(df1, df2) # same, but don't use
```

	data1	key	data2
0	0	b	1
1	1	b	1
2	6	b	1
3	2	a	0
4	4	a	0
5	5	a	0

The second version works because it will find the matching column, but better coding practice to actually specify the key

Merging DataFrames

If the column names don't match, you just specify them separately.

```
df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                 'data1': range(7)})
df4 = DataFrame({'rkey': ['a', 'b', 'd'],
                 'data2': range(3)})
```

df3

	data1	lkey
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	a
6	6	b

df4

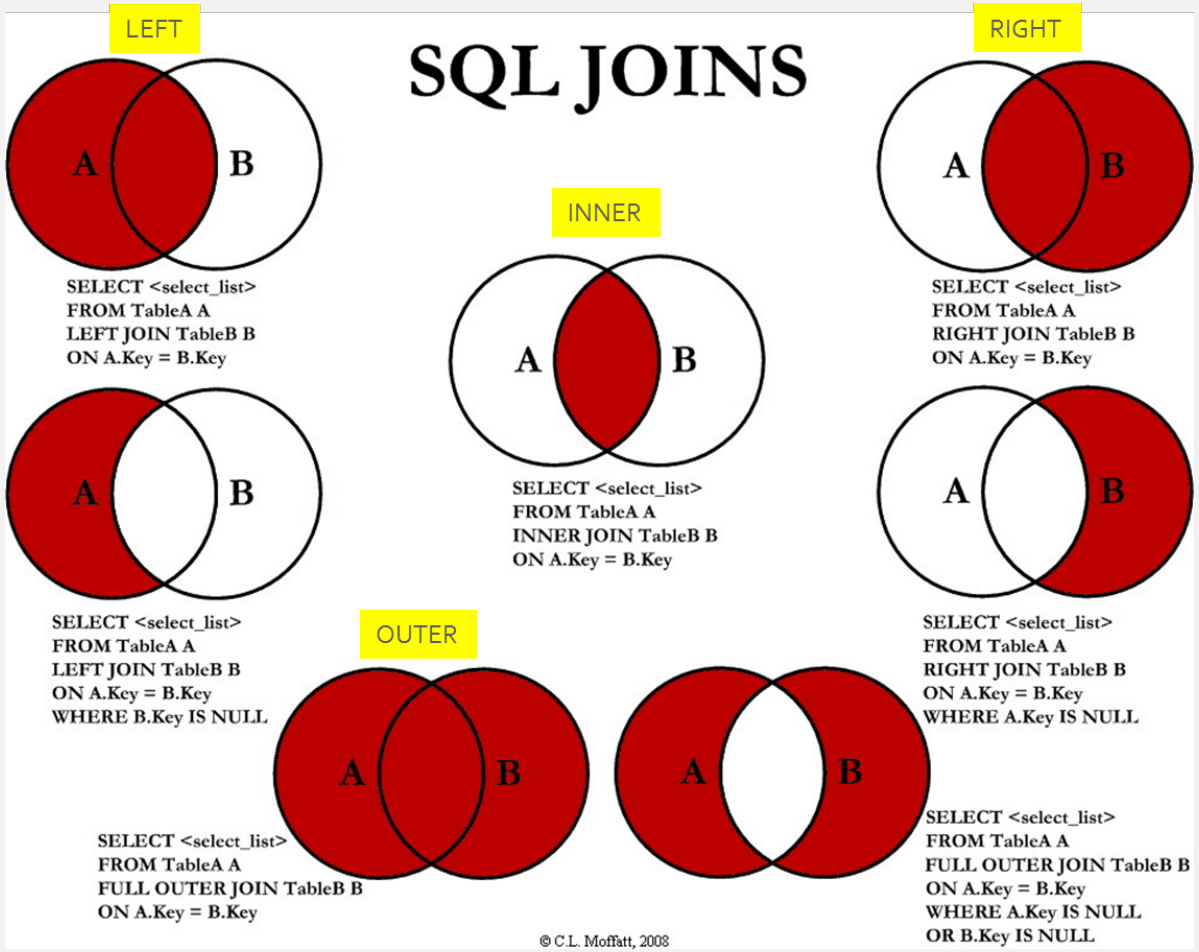
	data2	rkey
0	0	a
1	1	b
2	2	d

```
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

	data1	lkey	data2	rkey
0	0	b	1	b
1	1	b	1	b
2	6	b	1	b
3	2	a	0	a
4	4	a	0	a
5	5	a	0	a

NOTE: No c or d in result. An "INNER JOIN" was performed by default. Could add `,how='inner'` for same result.

Merging DataFrames



The default merge is an INNER join which returns only the intersection.

Other options are LEFT, RIGHT, and OUTER (a Union which is basically both a left and right join.)

Merging DataFrames

Note an OUTER join matches on both the left and right table and puts a missing NaN if there is no match.

df1

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	a
6	6	b

df2

	data2	key
0	0	a
1	1	b
2	2	d

```
pd.merge(df1, df2, on='key', how='outer')
```

	data1	key	data2
0	0.0	b	1.0
1	1.0	b	1.0
2	6.0	b	1.0
3	2.0	a	0.0
4	4.0	a	0.0
5	5.0	a	0.0
6	3.0	c	NaN
7	NaN	d	2.0

NOTE: No c in df2, so returns NaN. Also, no d in df1, so returns NaN.

Merging DataFrames

```
df1
```

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	a
6	6	b

```
df5 = DataFrame(dict(key=['a', 'b', 'a', 'b', 'd'],
                     data2 = range(5)))
df5
```

	data2	key
0	0	a
1	1	b
2	2	a
3	3	b
4	4	d

NOTE: Since there are 3 a values in df1 and 2 a values in df5, the result is 6 a values

```
pd.merge(df1, df5, on='key', how='outer')
```

	data1	key	data2
0	0	b	1.0
1	0	b	3.0
2	1	b	1.0
3	1	b	3.0
4	2	a	0.0
5	2	a	2.0
6	3	c	NaN
7	4	a	0.0
8	4	a	2.0
9	5	a	0.0
10	5	a	2.0
11	6	b	1.0
12	6	b	3.0

```
,how='inner')
```

	data1	key	data2
0	0	b	1
1	0	b	3
2	1	b	1
3	1	b	3
4	6	b	1
5	6	b	3
6	2	a	0
7	2	a	2
8	4	a	0
9	4	a	2
10	5	a	0
11	5	a	2

If we use INNER, only returns matches

Merging DataFrames

Suppose we have these DataFrames:

```
  key1 key2  lval
0  foo  one     1
1  foo  two     2
2  bar  one     3
```

```
  key1 key2  rval
0  foo  one     4
1  foo  one     5
2  bar  one     6
3  bar  two     7
```

To merge with multiple keys, pass a list of column names.

```
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

```
  key1 key2  lval  rval
0  foo  one   1.0   4.0
1  foo  one   1.0   5.0
2  foo  two   2.0   NaN
3  bar  one   3.0   6.0
4  bar  two   NaN   7.0
```

```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

```
  key1 key2_left  lval key2_right  rval
0  foo        one     1         one     4
1  foo        one     1         one     5
2  foo        two     2         one     4
3  foo        two     2         one     5
4  bar        one     3         one     6
5  bar        one     3         two     7
```

NOTE: If you have overlapping column names, the default suffix added is x and y but you can set it with `suffixes=`.

Merging DataFrames

In some cases, the merge key or keys in a DataFrame will be found in its index. In this case you can pass **left_index=True** or **right_index=True** (or both) in indicate the index is the merge key.

Suppose we have these DataFrames:

left1

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

right

	group_val
a	3.5
b	7.0

```
pd.merge(left1, right1, left_on='key', right_index=True)
```

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0

The default is INNER, but you can set to OUTER TO KEEP THE c

```
,how='outer')
```

	key	value	group_val
0	a	0	3.5
2	a	2	3.5
3	a	3	3.5
1	b	1	7.0
4	b	4	7.0
5	c	5	NaN

Merging DataFrames

You may also want to use the index from one DataFrame to be the index for the merged DataFrame. In that case you can set the index using **`set_index`** to be the the index from the DataFrame left2 by pulling its values with **`left2.index`** (this would also work with any list – not just .index)

```
df6 = pd.merge(left2,right2,how="left",left_on='key',right_on='name')
df6
```

	key	quan	ID	name
0	Sam	0	1	Sam
1	Frank	1	2	Frank
2	John	2	3	John
3	John	3	3	John
4	Sam	4	1	Sam
5	John	5	3	John

```
df6=df6.set_index(left2.index)
df6
```

	key	quan	ID	name
A12	Sam	0	1	Sam
B23	Frank	1	2	Frank
A23	John	2	3	John
C23	John	3	3	John
B45	Sam	4	1	Sam
D1	John	5	3	John

```
left2
```

	key	quan
A12	Sam	0
B23	Frank	1
A23	John	2
C23	John	3
B45	Sam	4
D1	John	5

```
right2
```

	ID	name
0	1	Sam
1	2	Frank
2	3	John

Stacking and Unstacking Data

Some data is stored in 'long' or 'stacked' format where the identifier like a date or an ID appear on multiple lines and there is a column that indicates the type of measure and a second column with its value. But there are times when we need to use the data in an unstacked format:

Example Stacked Data

	Date	measure	value
0	9/1/2016	Littleton	340
1	9/1/2016	Parker	321
2	9/1/2016	DU	273
3	9/2/2016	Littleton	290
4	9/2/2016	Parker	392
5	9/2/2016	DU	404

Example Unstacked Data

	Date	Littleton	Parker	DU
0	9/1/2016	340	321	273
1	9/2/2016	290	392	404

Stacking and Unstacking Data

```
stackeddf[:5]
```

	Unnamed: 0	date	item	value
0	0	3/31/1959 0:00	realgdp	2710.349
1	1	3/31/1959 0:00	infl	0.000
2	2	3/31/1959 0:00	unemp	5.800
3	3	6/30/1959 0:00	realgdp	2778.801
4	4	6/30/1959 0:00	infl	2.340

Note when we read in this data that there is a date but it is just stored as a default type 'object' when it is loaded.

We can convert it to a date/time field with **`pd.to_datetime()`**

```
stackeddf.date[:2]
```

```
0    3/31/1959 0:00
1    3/31/1959 0:00
Name: date, dtype: object
```

```
stackeddf.date = pd.to_datetime(stackeddf.date)
stackeddf.date[:2]
```

```
0   1959-03-31
1   1959-03-31
Name: date, dtype: datetime64[ns]
```

Note how the format changed and it is the type: datetime.

Stacking and Unstacking Data

```
stackeddf[:7]
```

	Unnamed: 0	date	item	value
0	0	1959-03-31	realgdp	2710.349
1	1	1959-03-31	infl	0.000
2	2	1959-03-31	unemp	5.800
3	3	1959-06-30	realgdp	2778.801
4	4	1959-06-30	infl	2.340
5	5	1959-06-30	unemp	5.100
6	6	1959-09-30	realgdp	2775.488

To perform the pivot and 'unstack' the data you first pass the columns to be used as the rows and columns and then column with the values for each variable.

```
pivoted = stackeddf.pivot('date', 'item', 'value')
pivoted.head() #returns first 5 values
```

item	infl	realgdp	unemp
date			
1959-03-31	0.00	2710.349	5.8
1959-06-30	2.34	2778.801	5.1
1959-09-30	2.74	2775.488	5.3
1959-12-31	0.27	2785.204	5.6
1960-03-31	2.31	2847.699	5.2

Descriptive Statistics & Histograms

As a motivating example, let's look at some Ice Cream sales data that has the age of the customer, their type (Adult, Child, Teenager), the flavor they bought, and the sales amount.

```
ICData = pd.read_csv('IceCreamData.csv')
ICData = ICData.drop('Customer', axis=1)
ICData.head()
```

	Type	Flavor	Age	Sales
0	Adult	Chocolate	45	4.25
1	Child	Vanilla	5	2.90
2	Teenager	Chocolate	14	3.10
3	Adult	Vanilla	23	3.25
4	Adult	Chocolate	47	4.10

Descriptive Statistics & Histograms

We can use the **`describe`** method to get the basic summary statistics for the quantitative variables.

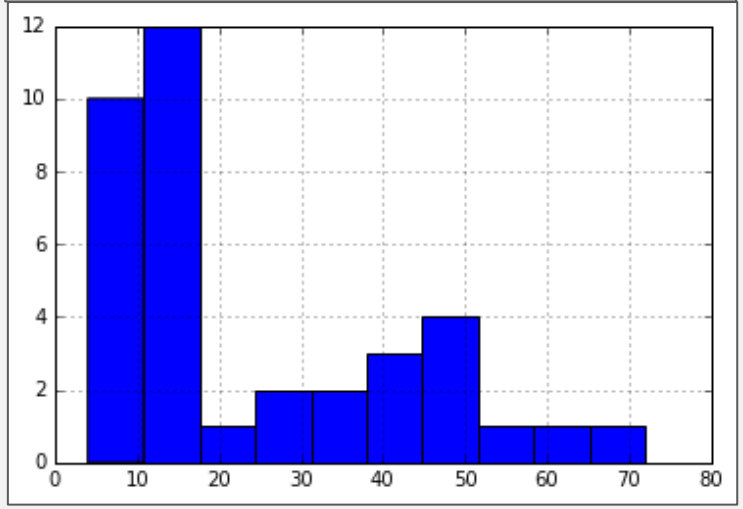
```
ICData[['Age','Sales']].describe()
```

	Age	Sales
count	37.000000	37.000000
mean	23.675676	3.489189
std	18.452723	0.547232
min	4.000000	2.500000
25%	6.000000	3.000000
50%	16.000000	3.500000
75%	40.000000	4.000000
max	72.000000	4.300000

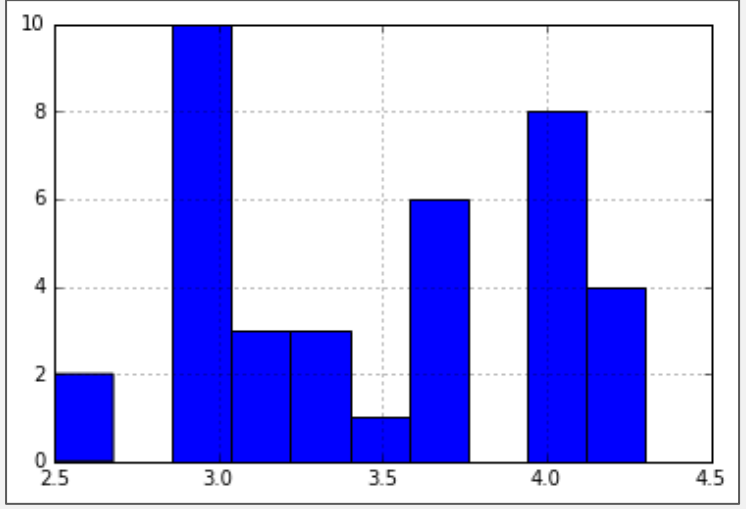
Descriptive Statistics & Histograms

The **`hist`** method from pandas produces the histogram graph.

```
ICData.Age.hist()
```



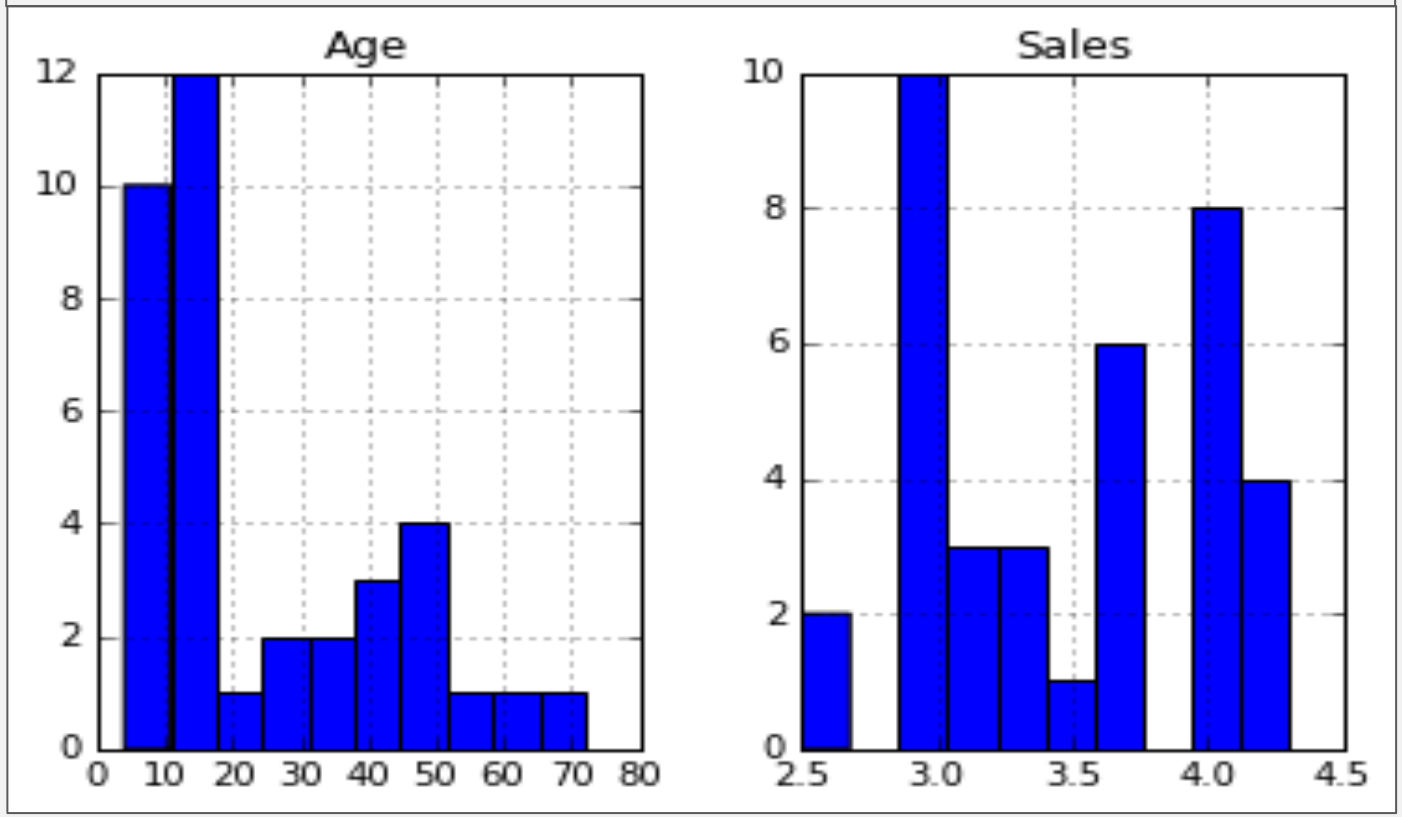

```
ICData.Sales.hist()
```



Descriptive Statistics & Histograms

If you give it two datasets, it will create side-by-side plots. (We will create more advanced histograms with matplotlib later.)

```
ICData[['Sales','Age']].hist()
```



Unique Values and Counts

Another class of related methods extracts information about the values in a series. These are a great way to examine qualitative data. If we have qualitative data, the **`describe`** function returns count, unique, top (like mode), and freq (the count of the top category).

```
ICData[['Type','Flavor']].describe()
```

	Type	Flavor
count	37	37
unique	3	2
top	Adult	Chocolate
freq	15	19

Unique Values and Counts

The **unique** function returns an array of the different values.

```
ICData.Type.unique()
```

```
array(['Adult', 'Child', 'Teenager'], dtype=object)
```

```
ICData.Flavor.unique()
```

```
array(['Chocolate', 'Vanilla'], dtype=object)
```

The **value_counts** function returns all the values with their associated counts or frequencies.

```
ICData.Type.value_counts()
```

```
Adult        15
Teenager     12
Child        10
Name: Type, dtype: int64
```

CrossTabs

We can use Pandas built in crosstab function to create counts for two qualitative variables.

```
FlavorType = pd.crosstab(ICData.Flavor,ICData.Type)
FlavorType
```

```
Type        Adult   Child   Teenager
Flavor
Chocolate       9       3          7
Vanilla         6       7          5
```

```
FlavorType.ix['Chocolate']
```

```
Type
Adult        9
Child        3
Teenager     7
Name: Chocolate, dtype: int64
```

```
FlavorType.ix['Chocolate','Adult']
```

```
9
```

Counts and BarPlots

We can create a **DataFrame** from the **value_counts** output to create a bar plot with the **.plot(kind='bar')** where we specify kind='bar' - note the default is a line plot.

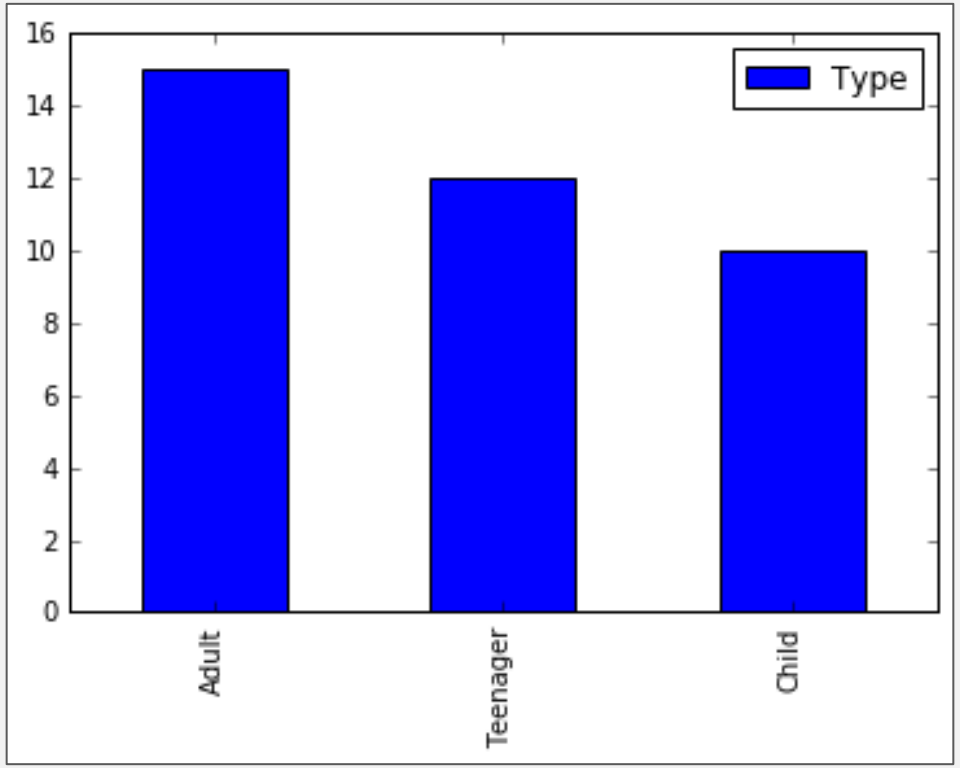
```
TypeCnts=DataFrame(ICData.Type.value_counts())
TypeCnts
```

```
            Type
Adult         15
Teenager      12
Child         10
```

```
TypeCnts.plot(kind='bar')
```

We don't really need the Legend with 'Type' – but we could make it say Counts to be more representative.

(Alternatively, we could create a y axis label. We will look at using matplotlib.pyplot shortly with more graphics options.)



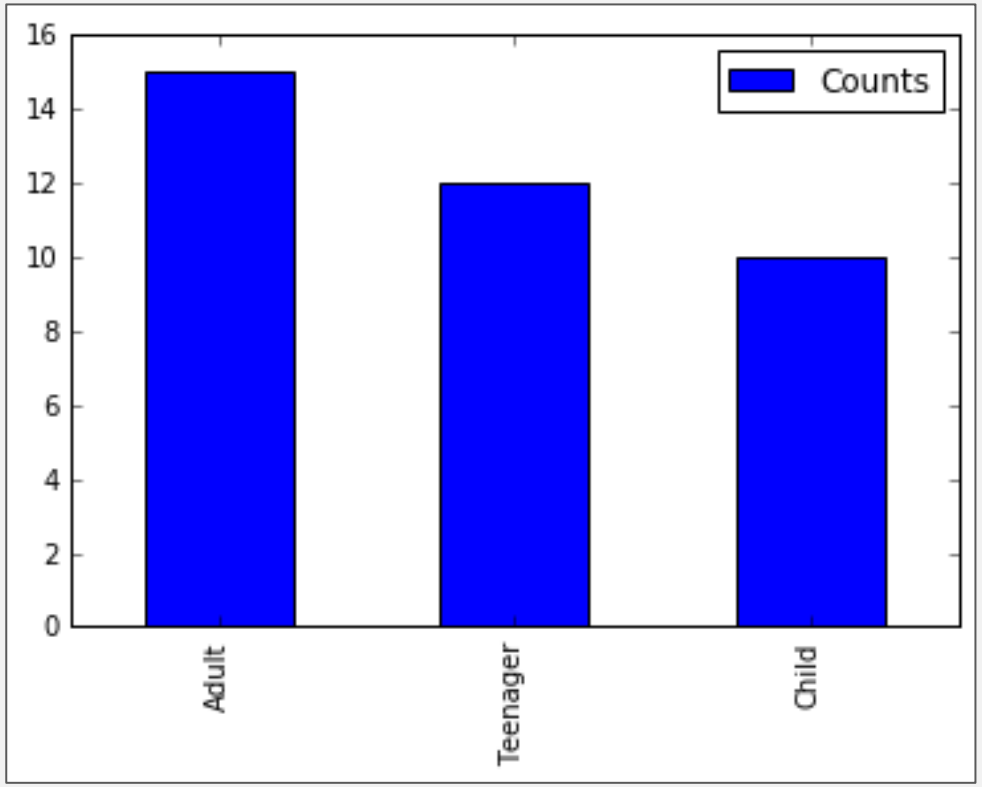
Counts and BarPlots

If you have appropriate column names and an index on the **`DataFrame`**, it will correctly label the bar plot.

```
TypeCnts.columns = ['Counts']
TypeCnts
```

	Counts
Adult	15
Teenager	12
Child	10

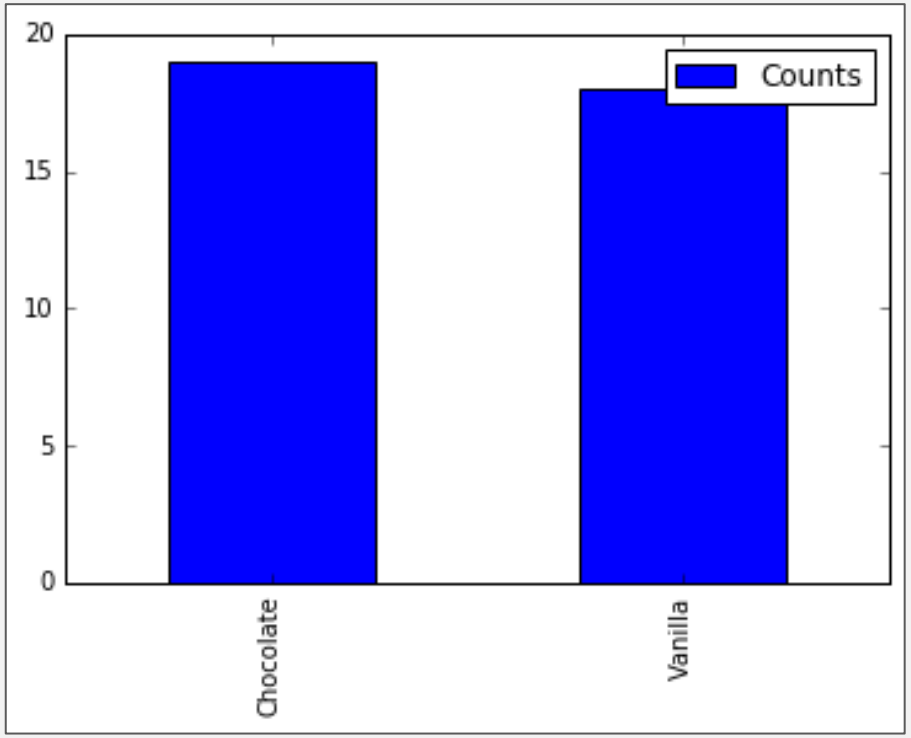
```
TypeCnts.plot(kind='bar')
```



Counts and BarPlots

We can do the same with Flavor.

```
FlavorCnts = DataFrame(ICData.Flavor.value_counts())
FlavorCnts.columns = ['Counts']
FlavorCnts.plot(kind='bar')
```



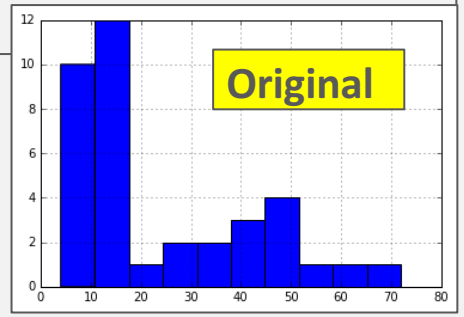
Pandas and Matplotlib

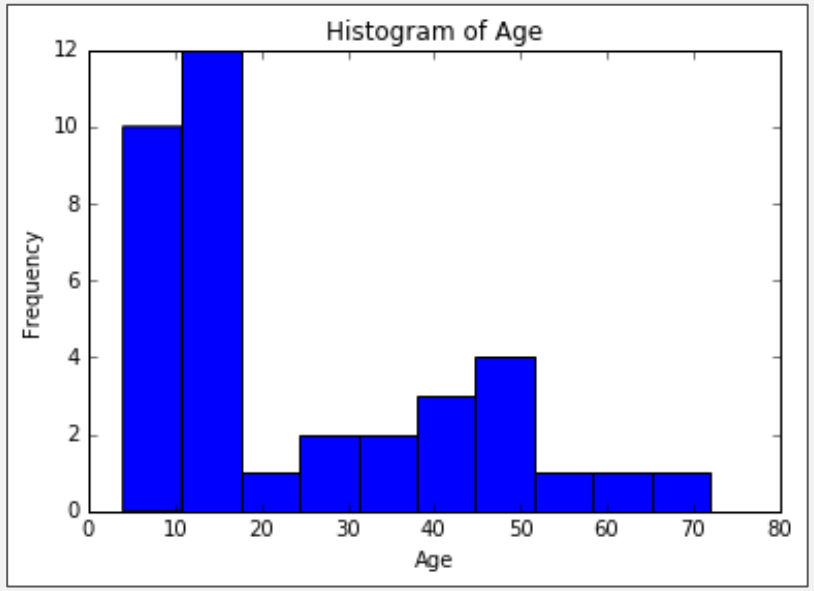
`matplotlib.pyplot` is a collection of command-style functions that make **`matplotlib`** work like MATLAB. To simplify it often is aliased as **`plt`**. Each **`pyplot`** function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels and titles, etc.

Here is an example creating a histogram.

```
import matplotlib.pyplot as plt
plt.hist(ICData.Age, 10)
plt.title('Histogram of Age')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()
```

The `plt.show()` is a signal that you are finished with this plot.





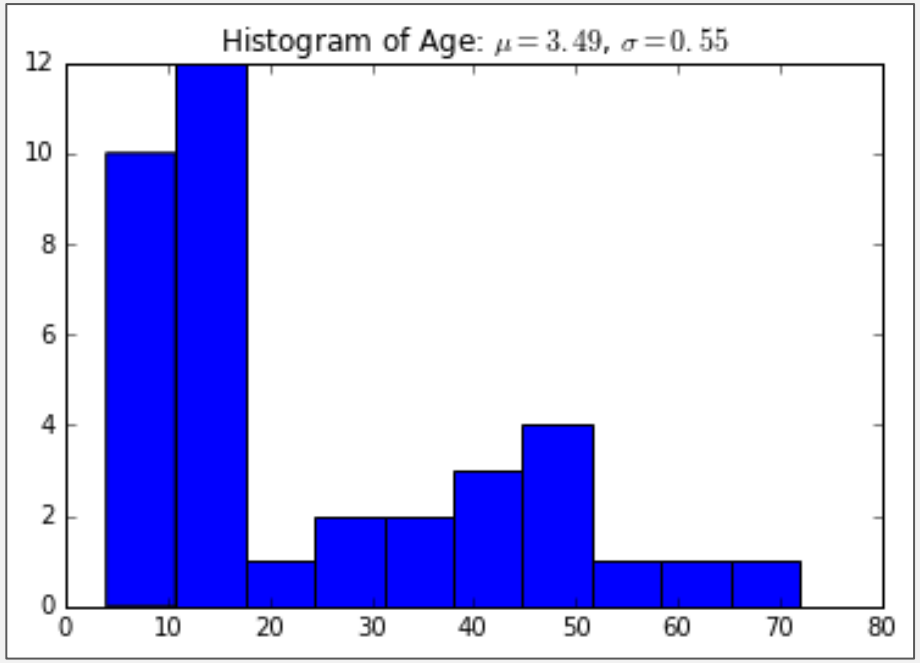
Pandas and Matplotlib

A histogram is a graphical representation of the distribution of set of numerical data. The syntax is:

plt.hist(y:data series, number of bins, [options])

```
plt.hist(ICData.Age, 10)
mn = str(round(ICData.Sales.mean(),2))
sd = str(round(ICData.Sales.std(),2))
plt.title('Histogram of Age: $\mu=' + mn +
          '$, $\sigma=' + sd + '$')
plt.savefig('AgeHist.png')
plt.show()
```

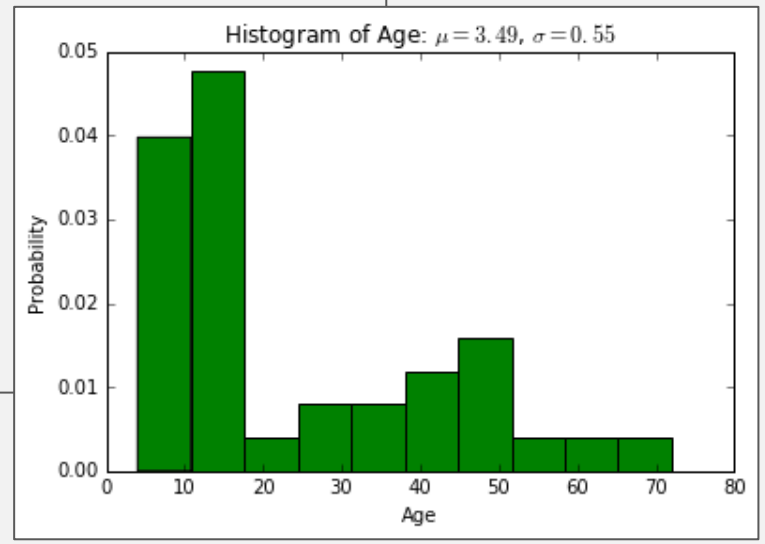
The syntax $\sigma= .55$ returns italics with the Greek sigma for display of mathematical notation.



Pandas and Matplotlib

Instead of showing the frequency at each level, the function can be modified to show the probability at each level by setting the **normed** attribute to 1. To change the colors of the vertical bars in the histogram, set the **color** to the desired color.

```
plt.hist(ICData.Age, 10, color='green', normed=1)
plt.xlabel('Age')
plt.ylabel('Probability')
mn = str(round(ICData.Sales.mean(),2))
sd = str(round(ICData.Sales.std(),2))
plt.title('Histogram of Age: $\mu=' +
   mn + '$, $\sigma=' + sd + '$')
plt.show()
```



Pandas and Matplotlib

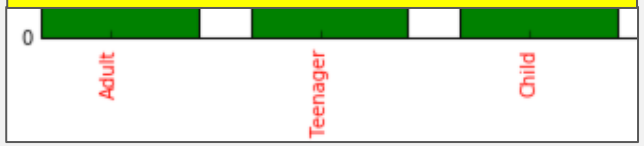
A bar/column chart shows bars/columns with lengths proportional to the values that they represent.
The syntax is: **`plt.bar(x,y,[width],[options])`**

x is the left edge of the bar along the x axis. y is the height.

`plt.xticks(x, labels, rotation='vertical')`

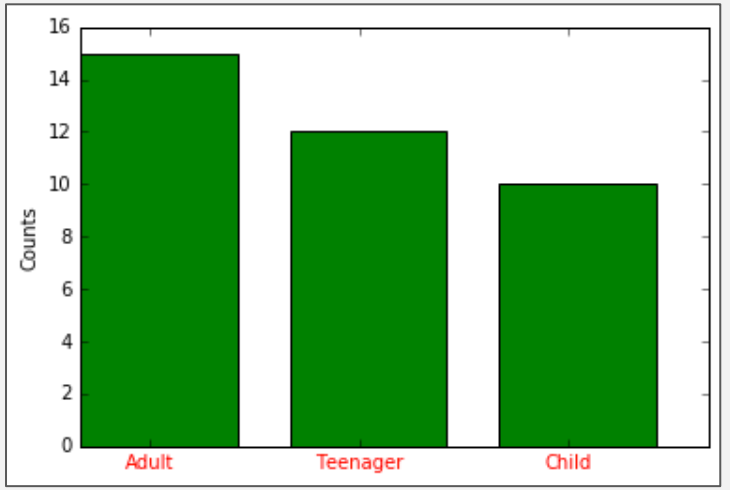
```
y=TypeCnts.Counts
n=len(y)
x=np.arange(n)
plt.bar(x,y, width=.75,color='green')
plt.ylabel('Counts')
plt.xticks(x + 1/n,TypeCnts.index,color='red')
plt.show()
```

Using rotation='vertical' in `plt.xticks` will rotate the x labels



`TypeCnts`

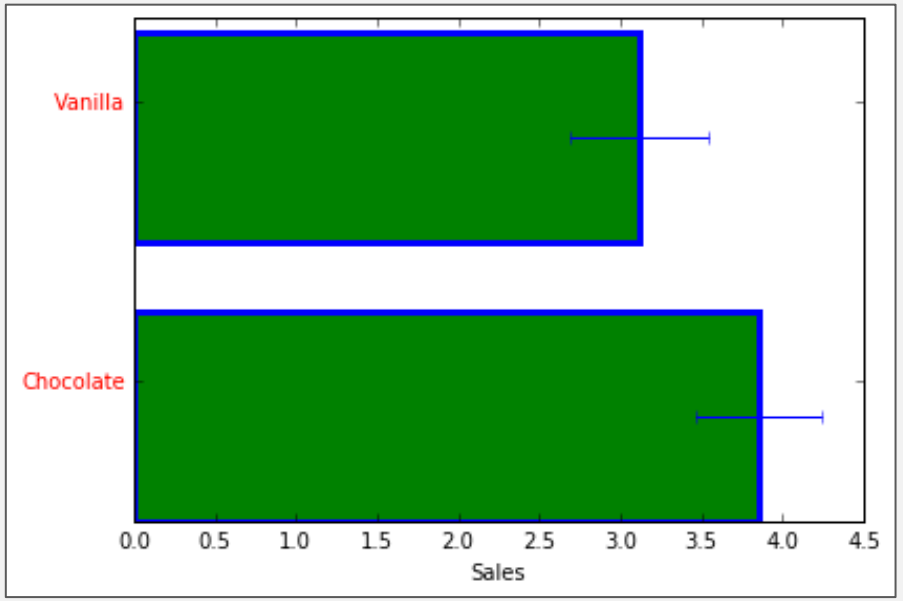
```
Counts
Adult
15
Teenager
12
Child
10
```



Pandas and Matplotlib

Some additional options: **`edgecolor`** (bar outline), **`linewidth`** (outline width), **`yerr/xerr`** (y or x error bars) and you can use **`plt.barh`** and **`height`** instead of **`width`** for a horizontal chart.

```
y=[ICData.Sales[ICData.Flavor=='Chocolate'].mean(),
   ICData.Sales[ICData.Flavor=='Vanilla'].mean()]
stdevs = [ICData.Sales[ICData.Flavor=='Chocolate'].std(),
   ICData.Sales[ICData.Flavor=='Vanilla'].std()]
n=len(y)
x=np.arange(n)
plt.barh(x,y, height=.75,color='green`,
    edgecolor='blue',linewidth=3,xerr=stdevs)
plt.xlabel('Sales')
plt.yticks(x + 1/n,
    ['Chocolate','Vanilla'],
    color='red')
plt.show()
```



Pandas and Matplotlib

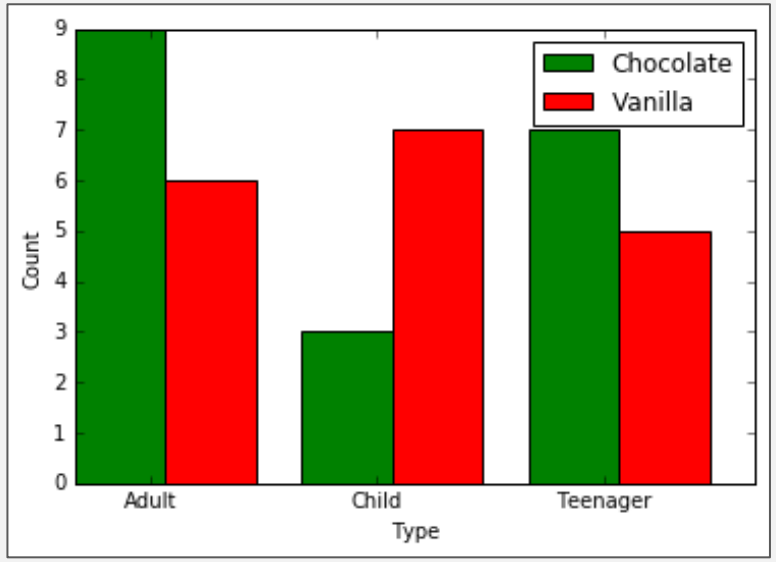
Adding multiple series to a column chart can be tricky.

First, change the **`width`** attribute of the original series so that the column's width is less than 50% of the x.

Second, do the same with new series, but also change the series' x to the original x plus the column width. This will place the second series just to the right of the first series along the x axis.

```
width=.4
#Plot Chocolate
y=FlavorType.ix['Chocolate']
n=len(y)
x=np.arange(n)
plt.bar(x,y, width,color='green',
        label="Chocolate")
#Plot Vanilla
y=FlavorType.ix['Vanilla']
plt.bar(x+width,y, width,color='red',
        label="Vanilla")
#Plot Options
plt.xticks(x + 1/n,FlavorType.columns)
plt.xlabel('Type')
plt.ylabel('Count')
plt.legend()
```

Type	Adult	Child	Teenager
Flavor			
Chocolate	9	3	7
Vanilla	6	7	5



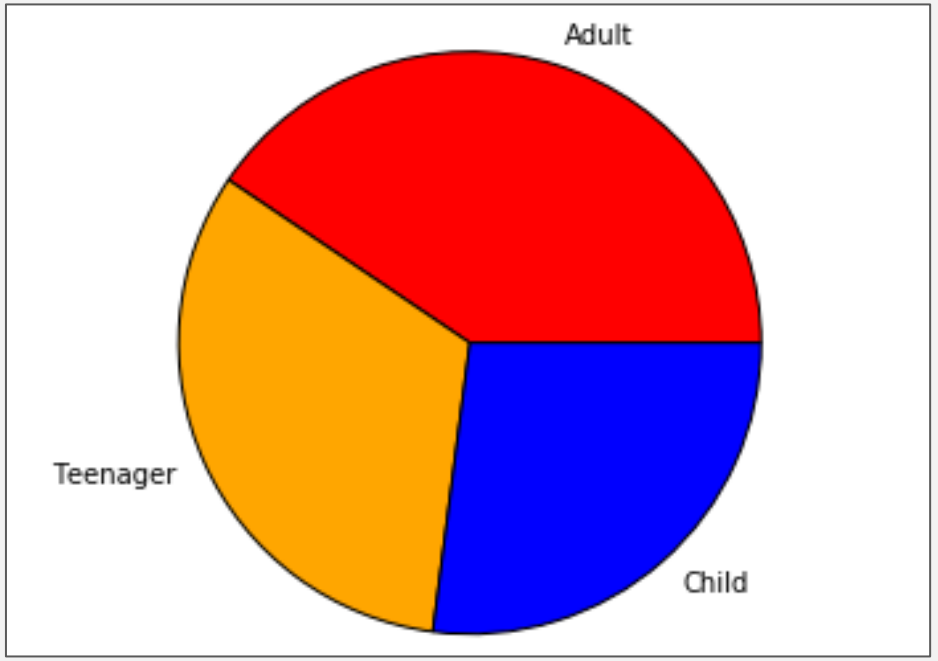
NOTE: for Vanilla x and n are same as for Chocolate

Pandas and Matplotlib

A pie chart is a circular statistical graphic that is divided into slices to illustrate numerical proportion.

The syntax is: **`plt.pie(data series,labels,colors)`**

```
plt.pie(TypeCnts.Counts, labels=TypeCnts.index,
        colors=['red','orange','blue'])
plt.axis('equal') #force circle with equal aspect ratio
plt.show()
```

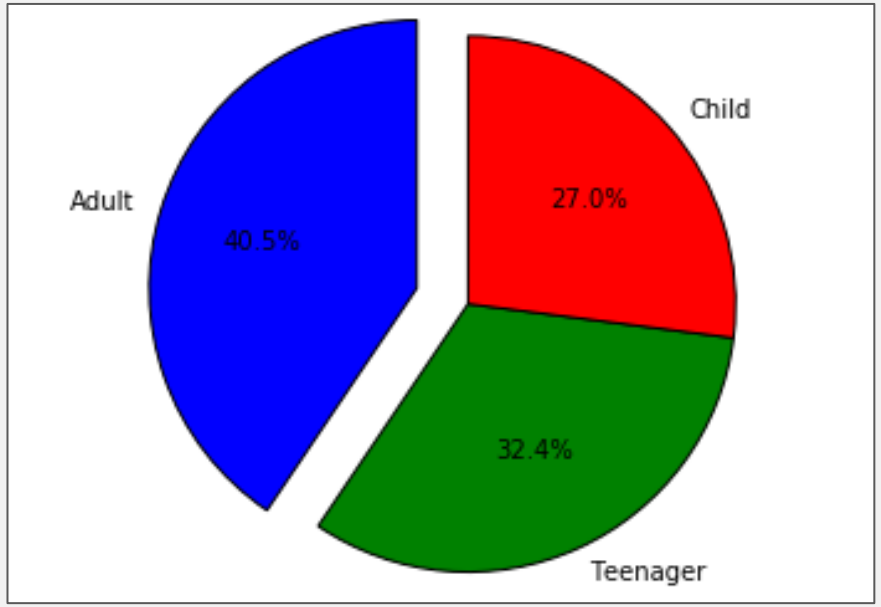


Pandas and Matplotlib

To add percentage labels to each slice, use the **`autopct`** attribute with the format surrounded by %.

To modify the charts starting angle, change the **`startangle`** attribute. To explode the slices use the **`explode`** attribute with the percentage to explode (for example, .20 is 20%).

```
plt.pie(TypeCnts.Counts, labels=TypeCnts.index,
        autopct='%1.1f%%', startangle=90,explode=[0.20,0,0])
#force circle with equal aspect ratio
plt.axis('equal')
plt.show()
```



Pandas and Matplotlib

We can capture some closing stock prices and columns for Apple from Yahoo! Finance and put them in a DataFrame using **`pandas-datareader`**.

```
import pandas_datareader.data as web
AAPL = web.DataReader('AAPL','yahoo')
AAPL.head(2)
AAPL.tail(2)
```

```
                 Open        High         Low       Close     Volume  \
Date                                                                    2010-01-04
213.429998  214.499996  212.380001  214.009998  123432400   2010-01-05  214.599998
215.589994  213.249994  214.379993  150476200
            Adj Close
Date
2010-01-04  27.990226
2010-01-05  28.038618
                 Open        High         Low       Close    Volume  \
Date                                                                   2016-10-06
113.699997  114.339996  113.129997  113.889999  28779300   2016-10-07  114.309998
114.559998  113.510002  114.059998  24358400
             Adj Close
Date
2016-10-06  113.889999
2016-10-07  114.059998
```

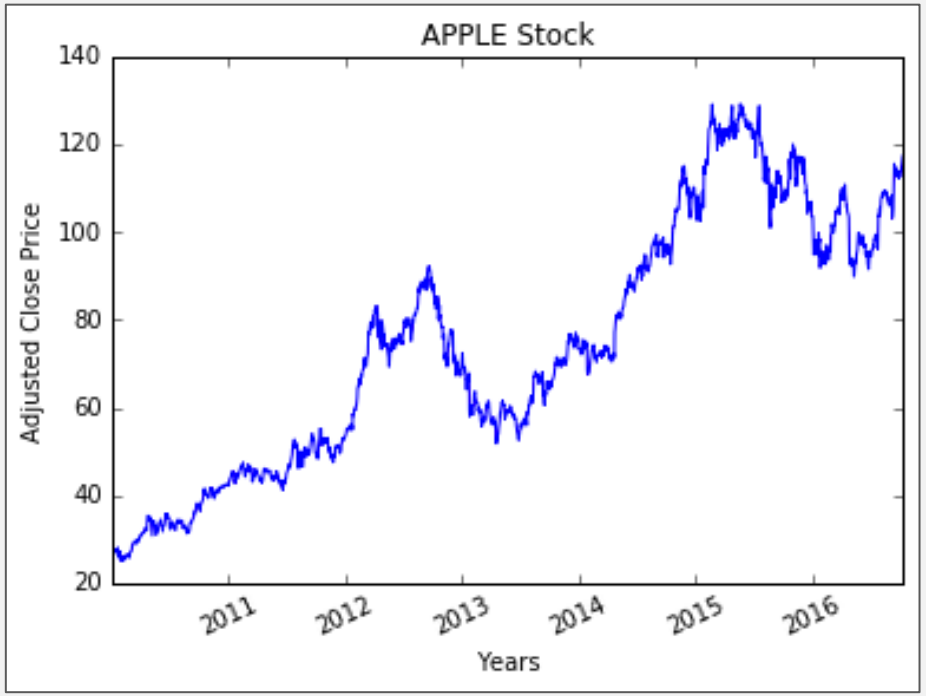
Pandas and Matplotlib

A line chart displays a series of data points connected by straight line segments. It is often used to visualize a trend in data over intervals of time.

The syntax is **`plt.plot(x:time, y)`**.

```
plt.plot(AAPL['Adj Close'].index, AAPL['Adj Close'])
plt.title('APPLE Stock')
plt.xlabel('Years')
plt.ylabel('Adjusted Close Price')
plt.xticks(rotation=25)
plt.show()
```

before used
`plt.xticks(rotation='vertical')`
here using 25 degrees

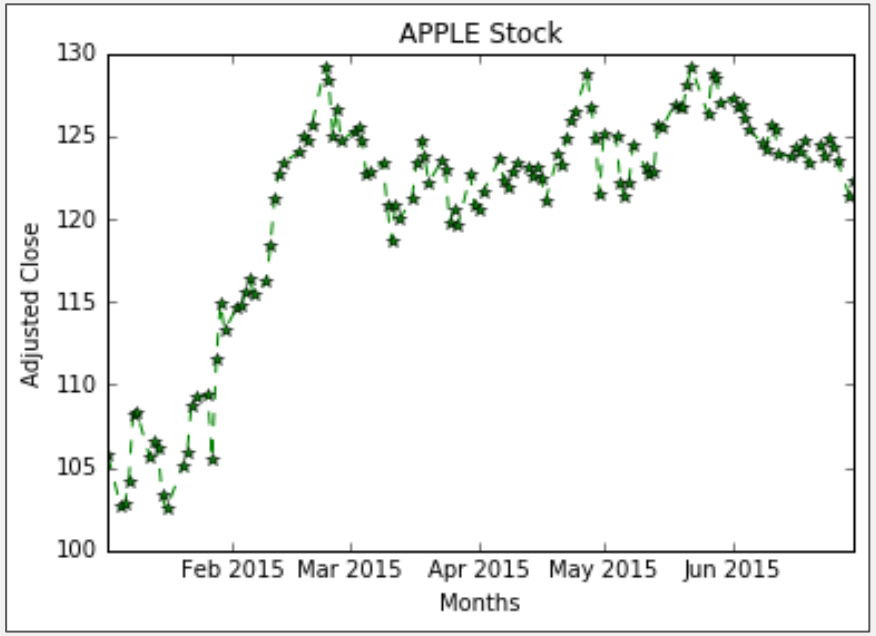


Pandas and Matplotlib

```
#Subset first 6 months of 2015
start = pd.datetime(2015, 1, 1)
end = pd.datetime(2015, 6, 30)
AAPL15 = web.DataReader('AAPL','yahoo',start,end)
AAPL15.head(2)
AAPL15.tail(2)
plt.plot(AAPL15['Adj Close'].index, AAPL15['Adj Close'],'g*--')
plt.title('APPLE Stock')
plt.xlabel('Months')
plt.ylabel('Adjusted Close')
plt.show()
```

```
                Open        High         Low       Close     Volume  \
Date
2015-01-02  111.389999  111.440002  107.349998  109.330002  53204600
2015-01-05  108.290001  108.650002  105.410004  106.250000  64285500
            Adj Close
Date
2015-01-02  105.69862
2015-01-05  102.72092

                Open        High         Low       Close     Volume  \
Date
2015-06-29  125.459999  126.470001  124.480003  124.529999  49161400
2015-06-30  125.570000  126.120003  124.860001  125.430000  44370700
            Adj Close
Date
2015-06-29  121.373768
2015-06-30  122.250959
```



More Information

Pandas: http://pandas.pydata.org

Thank You!

Please connect with me on LinkedIn @ linkedin.com/in/jcdaniel91

Or follow me on Twitter @jessecdaniel