머신러닝 실습노트

Case study

iPython 기초

```
In [1]:  print 'Hello World!'
```

```
Hello World!
```

```
In [2]:  i = 4
```

```
In [3]:  type(i)
```

```
Out[3]:  int
```

```
In [4]:  f = 4.1
```

```
In [5]:  type(f)
```

```
Out[5]:  float
```

```
In [17]:  b = True #꼭 대문자이어야 함
```

```
In [7]:  s = "This is a string!"
```

```
In [8]:  print s
```

```
This is a string!
```

```
In [9]:  l = [3,1,2] #리스트
```

```
In [10]:  print l #결과가 입력한 순서대로 출력된다.
```

```
[3, 1, 2]
```

```
In [11]:  d = {'foo':1, 'bar':2.3, 's':'my first dictionary'} #'키 값':'원소'
```

```
In [12]:  print d
```

```
{'s': 'my first dictionary', 'foo': 1, 'bar': 2.3}
```

```
In [13]:  print d['foo']
```

```
1
```

```
In [14]:  n = None #파이썬에선 null타입은 None이다.
```

```
In [15]:  type(n)
```

```
Out[15]:  NoneType
```

```
In [16]:  print "Our float value is %s. Out int value is %s." %(f,i)
```

```
Our float value is 4.1. Out int value is 4.
```

```
In [18]:
if i == 1 and f > 4:
    print "The value of i is 1 and f is greater than 4."
elif i > 4 or f > 4:
    print "i or f are both greater than 4."
else:
    print "both i and f are less than or equal to 4."

    #파이썬에서는 {}를 사용하지 않기 때문에 다음 줄의 공백이나 탭 몇칸으로 입력을 인식한다. 그러므로 elif를 탭하여 쓰거나 하면 안된다.
    #들여쓰기는 중요하다능.

#파이썬에서의 조건문 사용 방법
```

```
i or f are both greater than 4.
```

```
In [19]: print l
```

```
[3, 1, 2]
```

```
In [20]: for e in l:
             print e

             #파이썬에서는 print를 호출하면 개행도 함께 수행한다.

         #반복문을 돌리는 첫 번째 방법
```

```
3
1
2
```

```
In [21]: counter = 6
```

```
In [22]: while counter < 10:
             print counter
             counter += 1

         #반복문을 돌리는 두 번째 방법
```

```
6
7
8
9
```

```
In [23]: def add2(x):
             y = x+2
             return y

         #함수를 정의하는 첫 번째 방법
```

```
In [24]: i = 5
```

```
In [25]: add2(i)
```

```
Out[25]: 7
```

```
In [26]: square = lambda x: x*x

         #함수를 정의하는 두 번째 방법
```

```
In [27]: square(5)
```

```
Out[27]: 25
```

Graphlab 다루기

In [1]:
```
import graphlab
```

In [3]:
```
sf = graphlab.SFrame('people-example.csv')

#디스크에서부터 특정 파일을 읽어온다.
#표 형태의 dataset을 SFrame 형식으로 읽어들인다.
#SFrame이란?
#많은 양의 데이터에 대응할 수 있는 확장가능한 자료구조.
#HDD 등의 디스크에 직접 접근하기 때문에 메모리 용량에 상관없이 수백만개의 행의 data를 처리할 수 있다.
```

```
Finished parsing file C:\Users\dato\ML_practice\people-example.csv

Parsing completed. Parsed 7 lines in 0.037098 secs.

Finished parsing file C:\Users\dato\ML_practice\people-example.csv

Parsing completed. Parsed 7 lines in 0.01504 secs.

------------------------------------------------------
Inferred types from first line of file as
column_type_hints=[str,str,str,long]
If parsing fails due to incorrect types, you can correct
the inferred type list above and pass it to read_csv in
the column_type_hints argument
------------------------------------------------------
```

In [4]:
```
sf

# sf == sf.head()
# 즉 처음 몇개의 줄 밖에 볼 수 없다.
```

Out[4]:

First Name	Last Name	Country	age
Bob	Smith	United States	24
Alice	Williams	Canada	23
Malcolm	Jone	England	22
Felix	Brown	USA	23
Alex	Cooper	Poland	23
Tod	Campbell	United States	22
Derek	Ward	Switzerland	25

[7 rows x 4 columns]

In [5]:
```
sf.tail()
```

Out[5]:

First Name	Last Name	Country	age
Bob	Smith	United States	24
Alice	Williams	Canada	23
Malcolm	Jone	England	22
Felix	Brown	USA	23
Alex	Cooper	Poland	23
Tod	Campbell	United States	22
Derek	Ward	Switzerland	25

[7 rows x 4 columns]

In [7]:
```
_.show()

#여러 가지 방법으로 시각화하여 보여줌
```

```
Canvas is accessible via web browser at the URL: http://localhost:58035/index.html
Opening Canvas in default web browser.
```

In [8]:
```
sf.show()
```

```
Canvas is accessible via web browser at the URL: http://localhost:58035/index.html
Opening Canvas in default web browser.
```

In [9]:
```
graphlab.canvas.set_target('ipynb')

#또 다른 브라우저창이 뜨기 때문에 불편하다면
#iPython 노트북 이 위에 뜨게 하게 할 수 있다.
#이제부터 모든 시각화는 이 노트 내에서 한다.
```

In [10]:
```
sf['age'].show(view='Categorical')

#특정 열을 선택할 땐 위와 같은 방식을 쓴다.
```

Most frequent items from <*SArray*>

Value	Count	Percent	
23	3	42.857%	🟩🟩🟩
22	2	28.571%	🟩🟩
24	1	14.286%	🟩
25	1	14.286%	🟩

```
In [11]: sf['Country']
```

```
Out[11]: dtype: str
         Rows: 7
         ['United States', 'Canada', 'England', 'USA', 'Poland', 'United States', 'Switzerland']
```

```
In [12]: sf['age']
```

```
Out[12]: dtype: int
         Rows: 7
         [24L, 23L, 22L, 23L, 23L, 22L, 25L]
```

```
In [13]: sf['age'].mean()
```

```
Out[13]: 23.142857142857146
```

```
In [14]: sf['age'].max()
```

```
Out[14]: 25L
```

```
In [15]: sf
```

Out[15]:

First Name	Last Name	Country	age
Bob	Smith	United States	24
Alice	Williams	Canada	23
Malcolm	Jone	England	22
Felix	Brown	USA	23
Alex	Cooper	Poland	23
Tod	Campbell	United States	22
Derek	Ward	Switzerland	25

[7 rows x 4 columns]

```
In [28]: sf['Full name'] = sf['First Name'] + ' ' + sf['Last Name']

         #이와 같이 새로운 열을 생성할 수 있다.
         #잘못 생성한 열을 지우는 법은 무엇인지 모르겠다.
```

```
In [18]: sf
```

Out[18]:

First Name	Last Name	Country	age	Full name
Bob	Smith	United States	24	Bob Smith
Alice	Williams	Canada	23	Alice Williams
Malcolm	Jone	England	22	Malcolm Jone
Felix	Brown	USA	23	Felix Brown
Alex	Cooper	Poland	23	Alex Cooper
Tod	Campbell	United States	22	Tod Campbell
Derek	Ward	Switzerland	25	Derek Ward

[7 rows x 5 columns]

```
In [19]: sf['age'] * sf['age']
```

```
Out[19]: dtype: int
         Rows: 7
         [576L, 529L, 484L, 529L, 529L, 484L, 625L]
```

데이터 변환하기

dataset에 작은 오류가 있어서 수정할 필요가 있는 상황이라고 가정한다.

In [20]:
```python
sf['Country'].show()
```

Most frequent items from <*SArray*>

Value	Count	Percent
United States	2	28.571%
Canada	1	14.286%
England	1	14.286%
USA	1	14.286%
Poland	1	14.286%
Switzerland	1	14.286%

In [21]:
```python
def transform_country(country):
    if country == 'USA':
        return 'United States'
    else:
        return country

#데이터 정규화
```

In [22]:
```python
transform_country('Brazil')
```

Out[22]:
```
'Brazil'
```

In [23]:
```python
transform_country('USA')
```

Out[23]:
```
'United States'
```

In [24]:
```python
sf['Country'].apply(transform_country)

#함수를 모든 행에 적용시킨다.
#for 등을 쓸 필요 없이 apply만 쓰면 된다.
```

Out[24]:
```
dtype: str
Rows: 7
['United States', 'Canada', 'England', 'United States', 'Poland', 'United States', 'Switzerland']
```

In [25]:
```python
sf['Country'] = sf['Country'].apply(transform_country)

#실제 수정
```

In [26]:
```python
sf
```

Out[26]:

First Name	Last Name	Country	age	Full name
Bob	Smith	United States	24	Bob Smith
Alice	Williams	Canada	23	Alice Williams
Malcolm	Jone	England	22	Malcolm Jone
Felix	Brown	United States	23	Felix Brown
Alex	Cooper	Poland	23	Alex Cooper
Tod	Campbell	United States	22	Tod Campbell
Derek	Ward	Switzerland	25	Derek Ward

[7 rows x 5 columns]

Case study 실습

```
In [1]: import graphlab
```

```
In [2]: sales = graphlab.SFrame('home_data.gl/')

        # SFrame은 표 형태의 데이터를 담는 자료구조
        # 이번 실습에서는 주택 정보가 있는 데이터를 읽어오도록 할 것임
```

```
2016-03-28 09:53:33,967 [INFO] graphlab.cython.cy_server, 176: GraphLab Create v1.8.5 started. Logging: C:\Users\dato
\AppData\Local\Temp\graphlab_server_1459126411.log.0
This non-commercial license of GraphLab Create is assigned to dohk@koreatech.ac.kr and will expire on January 17, 201
7. For commercial licensing options, visit https://dato.com/buy/.
```

```
In [3]: sales

        # 데이터가 SFrame 자료구조에 담겨진 모습으로 출력됨
```

Out[3]:

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
7129300520	2014-10-13 00:00:00+00:00	221900	3	1	1180	5650	1	0
6414100192	2014-12-09 00:00:00+00:00	538000	3	2.25	2570	7242	2	0
5631500400	2015-02-25 00:00:00+00:00	180000	2	1	770	10000	1	0
2487200875	2014-12-09 00:00:00+00:00	604000	4	3	1960	5000	1	0
1954400510	2015-02-18 00:00:00+00:00	510000	3	2	1680	8080	1	0
7237550310	2014-05-12 00:00:00+00:00	1225000	4	4.5	5420	101930	1	0
1321400060	2014-06-27 00:00:00+00:00	257500	3	2.25	1715	6819	2	0
2008000270	2015-01-15 00:00:00+00:00	291850	3	1.5	1060	9711	1	0
2414600126	2015-04-15 00:00:00+00:00	229500	3	1	1780	7470	1	0
3793500160	2015-03-12 00:00:00+00:00	323000	3	2.5	1890	6560	2	0

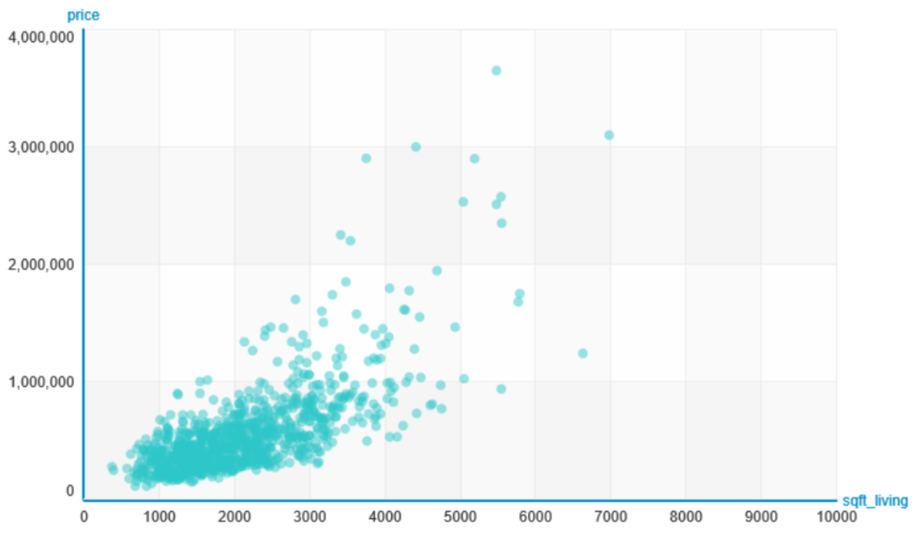
```
In [38]: sales.print_rows(20000,20)

         # 더 많은 행과 열을 출력
```

```
| -122.17234693 |     2910.0     | ... |
| -122.12473029 |     2730.0     | ... |
|  -122.1995542 |     1730.0     | ... |
| -122.36109716 |     1680.0     | ... |
| -122.29898523 |     1390.0     | ... |
| -121.88549484 |     2660.0     | ... |
| -122.29571932 |     1230.0     | ... |
| -122.20521622 |     1910.0     | ... |
| -122.36452948 |     1720.0     | ... |
| -122.01953889 |     3290.0     | ... |
| -122.08317455 |     2250.0     | ... |
| -122.37463454 |     1140.0     | ... |
| -122.08195753 |     2110.0     | ... |
| -122.29293672 |     1610.0     | ... |
| -122.37066399 |     1350.0     | ... |
| -122.34657242 |     2070.0     | ... |
| -122.07224524 |     2440.0     | ... |
| -122.37665187 |     1780.0     | ... |
| -122.34759899 |     2848.0     | ... |
| -122.19927867 |     3060.0     | ... |
```

산점도를 이용하여 데이터의 생김새 관찰하기

In [5]:
```
graphlab.canvas.set_target('ipynb')

# iPython 노트북 위에 결과가 보여지게 한다.
```

In [6]:
```
sales.show(view="Scatter Plot", x="sqft_living", y="price")

# 두 변수의 관계를 산점도를 이용해 표현하게 한다.
```



Simple Linear Regression 모델 만들기

In [7]:
```
train_data,test_data = sales.random_split(.8, seed=0)

# .8 : 80%가 train set
# 나머지 .2 : 20%의 test set
```

In [8]:
```
sqft_model = graphlab.linear_regression.create(train_data,target='price',features=['sqft_living'])

# target : 예측하고 싶은 값
# features : 어떤 특징값을 넣을 것인가
# newton's method가 쓰임
```

```
Linear regression:

--------------------------------------------------------

Number of examples          : 16477

Number of features          : 1

Number of unpacked features : 1

Number of coefficients    : 2

Starting Newton Method

--------------------------------------------------------

+-----------+----------+--------------+--------------------+----------------------+---------------+-----------------+

| Iteration | Passes   | Elapsed Time | Training-max_error | Validation-max_error | Training-rmse | Validation-rmse |

+-----------+----------+--------------+--------------------+----------------------+---------------+-----------------+

| 1         | 2        | 1.027880     | 4367493.014961     | 3272979.225649       | 261977.736742 | 280006.200602   |

+-----------+----------+--------------+--------------------+----------------------+---------------+-----------------+

SUCCESS: Optimal solution found.


PROGRESS: Creating a validation set from 5 percent of training data. This may take a while.
          You can set ``validation_set=None`` to disable validation tracking.
```

In [9]:
```
print test_data['price'].mean()

# 선형회귀모델 평가 단계
# 평균 출력
```

```
543054.042563
```

In [10]:
```
print sqft_model.evaluate(test_data)

# train_set을 통해 훈련된 선형회귀모델을 test_set을 이용해 평가
# 최고오류는 410만, RMSE도 높은 수치
```

```
{'max_error': 4157611.0868680067, 'rmse': 255154.2425276025}
```

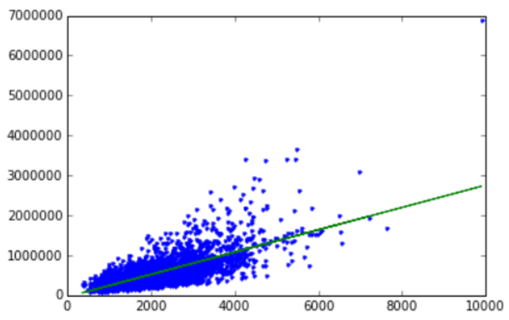
선형회귀모델 시각화하기

In [11]:
```
import matplotlib.pyplot as plt

# pyplot을 이용하여 시각화할 생각
```

In [39]:
```
%matplotlib inline

# iPython 노트북에 직접 출력
```

In [41]:
```
plt.plot(test_data['sqft_living'], test_data['price'], '.',
        test_data['sqft_living'], sqft_model.predict(test_data),'-')

# 위와 같은 방식으로 두 그래프를 겹쳐 그릴 수 있음
# test_data['sqft_living'], test_data['price'], '.' --> test_set
# test_data['sqft_living'], sqft_model.predict(test_data),'-' --> prediction
# '.', ,'-' : 어떤 모양으로 표시할 것인가
```

Out[41]:
```
[<matplotlib.lines.Line2D at 0x19c126a0>,
 <matplotlib.lines.Line2D at 0x19c12898>]
```



In [14]:
```
sqft_model.get('coefficients')

# simple linear regression 모델에 쓰인 계수(coefficient)알아내기
# w_0 : y절편;-43267
# w_1 : 기울기;280(한 평당 가격은 280)
```

Out[14]:

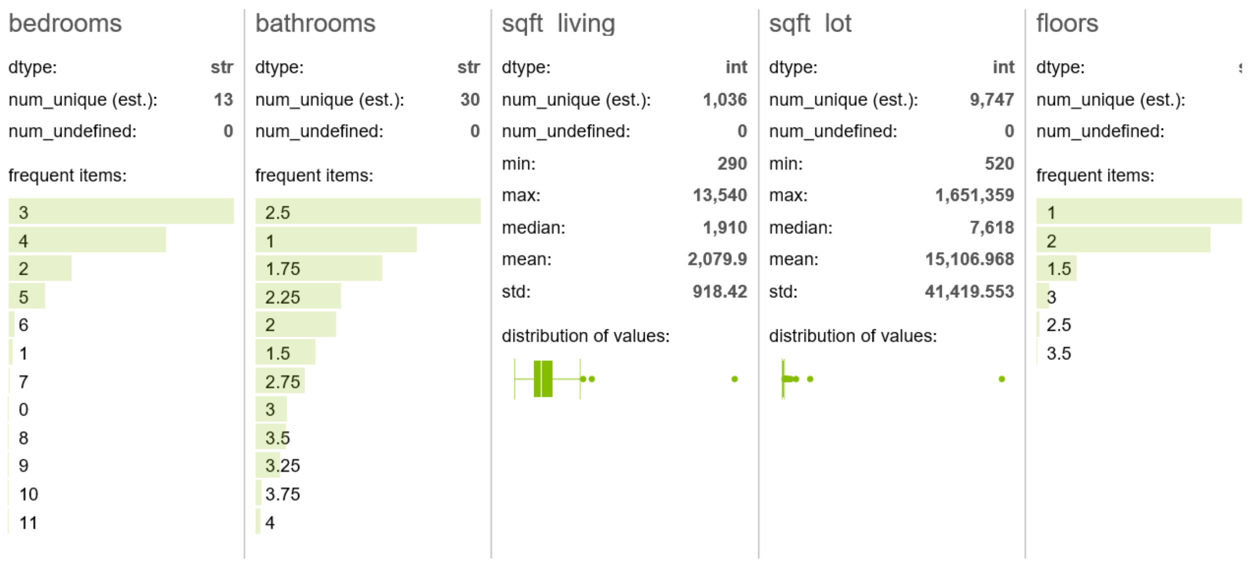
name	index	value	stderr
(intercept)	None	-43267.4438871	5027.93376583
sqft_living	None	280.147255513	2.20764511196

[2 rows x 4 columns]

complex linear regression 모델

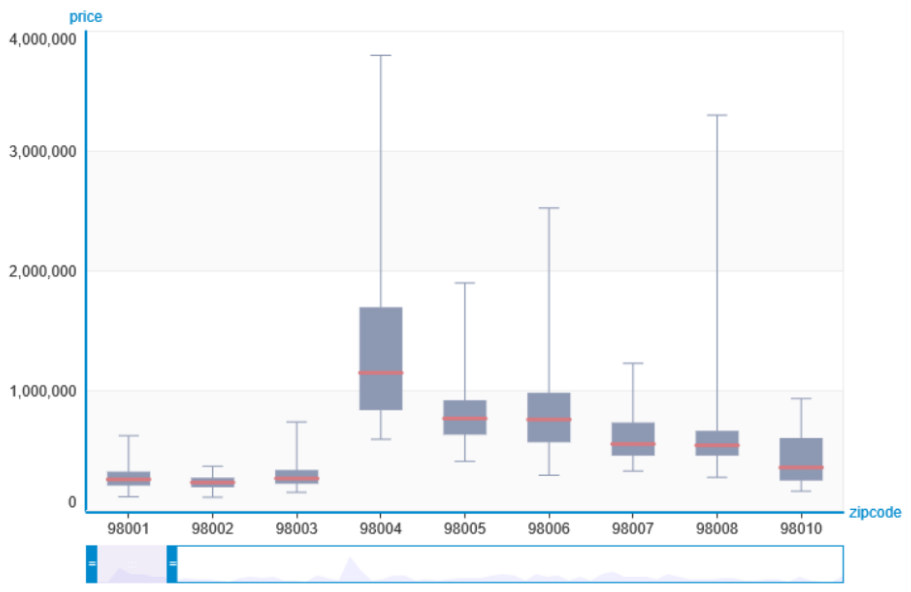
In [15]:
```
my_features=['bedrooms','bathrooms','sqft_living','sqft_lot','floors','zipcode']
# 더 많은 특징 추가
```

In [42]:
```
sales[my_features].show()
# 각각의 특징을 시각화
```



```
bedrooms                      bathrooms                     sqft_living                   sqft_lot                        floors

dtype:                  str   dtype:                  str   dtype:                  int   dtype:                    int   dtype:
num_unique (est.):       13   num_unique (est.):       30   num_unique (est.):    1,036   num_unique (est.):      9,747   num_unique (est.):
num_undefined:            0   num_undefined:            0   num_undefined:            0   num_undefined:              0   num_undefined:

frequent items:               frequent items:               min:                    290   min:                      520   frequent items:
3                             2.5                           max:                 13,540   max:                  1,651,359   1
4                             1                             median:               1,910   median:                   7,618   2
2                             1.75                          mean:               2,079.9   mean:                15,106.968   1.5
5                             2.25                          std:                 918.42   std:                 41,419.553   3
6                             2                                                                                             2.5
1                             1.5                           distribution of values:       distribution of values:         3.5
7                             2.75
0                             3
8                             3.5
9                             3.25
10                            3.75
11                            4
```

In [43]:
```
sales.show(view='BoxWhisker Plot', x='zipcode', y='price')

# 상자 그림으로 표현하기(시각화의 다른 종류)
# 98002의 주택가격은 낮고, 변동성은 크지 않다.
# 98004의 주택가격은 높고, 변동성은 크다.
```



Complex Linear Regression 모델 만들기

In [19]:
```
my_features_model = graphlab.linear_regression.create(train_data,target='price',features=my_features)
```

```
Linear regression:

--------------------------------------------------------

Number of examples          : 16526

Number of features          : 6

Number of unpacked features : 6

Number of coefficients    : 115

Starting Newton Method

--------------------------------------------------------

+-----------+----------+--------------+--------------------+----------------------+---------------+-----------------+

| Iteration | Passes   | Elapsed Time | Training-max_error | Validation-max_error | Training-rmse | Validation-rmse |

+-----------+----------+--------------+--------------------+----------------------+---------------+-----------------+

| 1         | 2        | 0.016016     | 3760103.056712     | 2429501.411101       | 182578.492144 | 171671.595425   |

+-----------+----------+--------------+--------------------+----------------------+---------------+-----------------+

SUCCESS: Optimal solution found.

PROGRESS: Creating a validation set from 5 percent of training data. This may take a while.
          You can set ``validation_set=None`` to disable validation tracking.
```

In [20]:
```
print sqft_model.evaluate(test_data)
```

```
{'max_error': 4157611.0868680067, 'rmse': 255154.2425276025}
```

In [21]:
```
print my_features_model.evaluate(test_data)

# test_set으로 평가하기
# simple 모델에 비해 최대오류와 RMSE가 줄어듦
```

```
{'max_error': 3509109.830486317, 'rmse': 179420.54970826607}
```

특정 집에 대한 가격 예측(using simple model)

In [22]:
```
house1 = sales[sales['id']=='5309101200']

# 특정 집에 대한 정보를 house1에 저장
```

In [23]:
```
house1

# house1이 가진 특징 출력
```

Out[23]:

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
5309101200	2014-06-05 00:00:00+00:00	620000	4	2.25	2400	5350	1.5	0

view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat
0	4	7	1460	940	1929	0	98117	47.67632376

long	sqft_living15	sqft_lot15
-122.37010126	1250.0	4880.0

[? rows x 21 columns]
Note: Only the head of the SFrame is printed. This SFrame is lazily evaluated.
You can use len(sf) to force materialization.

In [24]:
```
print house1['price']

# house1의 가격은? (실제 거래된 값)
```

```
[620000L, ... ]
```

In [25]:
```
print sqft_model.predict(house1)

# simple model을 이용해서 house1의 가격을 예측하면? (예측 값)
```

```
[629085.9693430441]
```

In [26]:
```
print my_features_model.predict(house1)

# complex model을 이용해서 house1의 가격을 예측하면? (예측 값)

# complex model이 항상 좋은 결괄르 내는 것은 아니라는 것을 알 수 있음. 예외는 항상 존재함.
```

```
[721987.2727698934]
```

특정 집에 대한 가격 예측(using complex model)

In [27]:
```
house2 = sales[sales['id']=='1925069082']
```

In [28]:
```
house2
```

Out[28]:

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
1925069082	2015-05-11 00:00:00+00:00	2200000	5	4.25	4640	22703	2	1

view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat
4	5	8	2860	1780	1952	0	98052	47.63925783

long	sqft_living15	sqft_lot15
-122.09722322	3140.0	14200.0

[? rows x 21 columns]
Note: Only the head of the SFrame is printed. This SFrame is lazily evaluated.
You can use len(sf) to force materialization.

In [29]:
```
print house2['price']
```

```
[2200000L, ... ]
```

In [30]:
```
print sqft_model.predict(house2)
```

```
[1256615.821691141]
```

In [31]:
```
print my_features_model.predict(house2)
```

```
[1451779.3947381824]
```

우리 집의 특징을 직접 사전을 이용하여 입력한 후 가격 예측하기

In [32]:
```
bill_gates={'bedrooms':[8],
            'bathrooms':[25],
            'sqft_living':[50000],
            'sqft_lot':[225000],
            'floors':[4],
            'zipcode':['98039'],
            'condition':[10],
            'grade':[10],
            'waterfront':[1],
            'view':[4],
            'sqft_above':[37500],
            'sqft_basement':[12500],
            'yr_built':[1994],
             'yr_renovated':[2010],
            'lat':[47.627606],
            'long':[-122.242054],
            'sqft_living5':[5000],
            'sqft_lot15':[40000]}
```

In [34]:
```
bill_gates_transform = graphlab.SFrame(bill_gates)

# 사전으로 만들어진 우리 집 정보를 SFrame 자료구조에 담아야 함(우리가 세운 모든 모델들이 SFrame 형태의 자료구조만을 인식하기 때)
```

In [35]:
```
bill_gates_transform
```

Out[35]:

bathrooms	bedrooms	condition	floors	grade	lat	long	sqft_above	sqft_basement	sqft_living
25	8	10	4	10	47.627606	-122.242054	37500	12500	50000

sqft_living5	sqft_lot	sqft_lot15	view	waterfront	yr_built	yr_renovated	zipcode
5000	225000	40000	4	1	1994	2010	98039

[1 rows x 18 columns]

In [37]:
```
my_features_model.predict(bill_gates_transform)
```

Out[37]:
```
dtype: float
Rows: 1
[13713549.643862298]
```

Simple linear Regression

실습1

단일 모델을 이용하여 회귀모델분석하기

집값과 범죄율 사이의 관계를 단일회귀모델을 이용하여 세우기(graphlab library 사용)

16.03.29 실습 수행, by 권도형(DOHK)

In [2]:
```
import graphlab
```

In [3]:
```
sales = graphlab.SFrame.read_csv('Philadelphia_Crime_Rate_noNA.csv/')
```

```
2016-03-29 10:57:18,154 [INFO] graphlab.cython.cy_server, 176: GraphLab Create v1.8.5 started. Logging: C:₩Users₩dato₩AppData₩Local₩Temp₩gra
phlab_server_1459216635.log.0
```

```
Finished parsing file C:₩Users₩dato₩ML_practice₩Philadelphia_Crime_Rate_noNA.csv

Parsing completed. Parsed 99 lines in 0.023092 secs.

This non-commercial license of GraphLab Create is assigned to dohk@koreatech.ac.kr and will expire on January 17, 2017. For commercial licen
sing options, visit https://dato.com/buy/.
------------------------------------------------------

Finished parsing file C:₩Users₩dato₩ML_practice₩Philadelphia_Crime_Rate_noNA.csv

Parsing completed. Parsed 99 lines in 0.01802 secs.


Inferred types from first line of file as
column_type_hints=[long,float,float,float,float,str,str]
If parsing fails due to incorrect types, you can correct
the inferred type list above and pass it to read_csv in
the column_type_hints argument
------------------------------------------------------
```

In [4]:
```
sales
```

Out[4]:

HousePrice	HsPrc ($10,000)	CrimeRate	MilesPhila	PopChg	Name	County
140463	14.0463	29.7	10.0	-1.0	Abington	Montgome
113033	11.3033	24.1	18.0	4.0	Ambler	Montgome
124186	12.4186	19.5	25.0	8.0	Aston	Delaware
110490	11.049	49.4	25.0	2.7	Bensalem	Bucks
79124	7.9124	54.1	19.0	3.9	Bristol B.	Bucks
92634	9.2634	48.6	20.0	0.6	Bristol T.	Bucks
89246	8.9246	30.8	15.0	-2.6	Brookhaven	Delaware
195145	19.5145	10.8	20.0	-3.5	Bryn Athyn	Montgome
297342	29.7342	20.2	14.0	0.6	Bryn Mawr	Montgome
264298	26.4298	20.4	26.0	6.0	Buckingham	Bucks

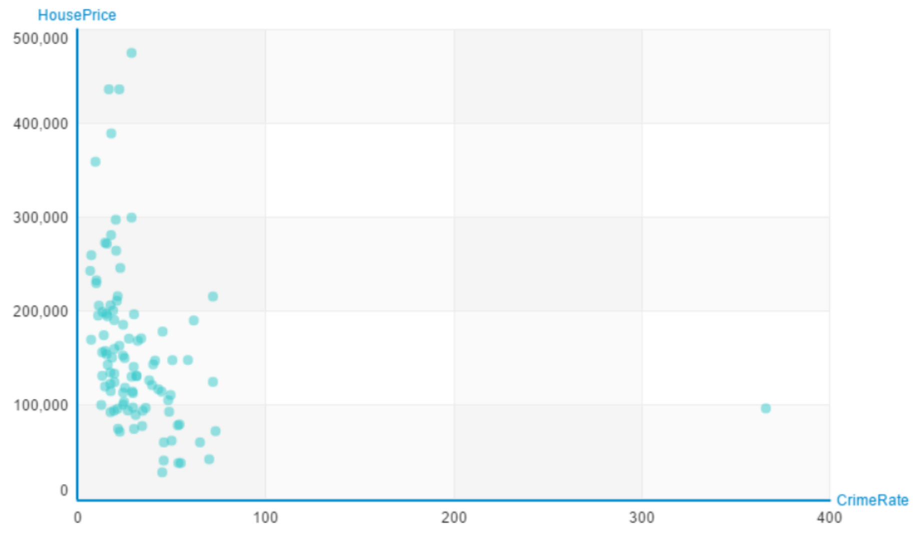
[99 rows x 7 columns]
Note: Only the head of the SFrame is printed.
You can use print_rows(num_rows=m, num_columns=n) to print more rows and columns.

```
In [12]: sales.print_rows(99,7)
```

```
|   99843    |     9.9843     |    12.5    |    12.0    |    -3.7    |
|   60000    |      6.0       |    45.8    |    18.0    |    -1.4    |
|   28000    |      2.8       |    44.9    |    5.5     |    -8.4    |
|   60000    |      6.0       |    65.0    |    9.0     |    -4.9    |
|   61800    |      6.18      |    49.9    |    9.0     |    -6.4    |
|   38000    |      3.8       |    54.8    |    4.5     |    -5.1    |
|   38000    |      3.8       |    53.5    |    2.0     |    -9.2    |
|   42000    |      4.2       |    69.9    |    4.0     |    -5.7    |
|   96200    |      9.62      |   366.1    |    0.0     |    4.8     |
|   103087   |    10.3087     |    24.6    |    24.0    |    3.9     |
|   147720   |     14.772     |    58.6    |    25.0    |    1.5     |
|   78175    |     7.8175     |    53.2    |    41.0    |    2.2     |
|   92215    |     9.2215     |    17.4    |    14.0    |    7.8     |
|   271804   |    27.1804     |    15.5    |    17.0    |    1.2     |
|   119566   |    11.9566     |    14.5    |    12.0    |    -2.9    |
|   100231   |    10.0231     |    24.1    |    15.0    |    1.9     |
|   95831    |     9.5831     |    21.2    |    32.0    |    3.2     |
|   229711   |    22.9711     |    9.8     |    22.0    |    5.3     |
|   74308    |     7.4308     |    29.9    |    7.0     |    1.8     |
|   259506   |    25.9506     |    7.2     |    40.0    |    17.4    |
```

```
In [5]: graphlab.canvas.set_target('ipynb')
```

```
In [6]: sales.show(view="Scatter Plot", x="CrimeRate", y="HousePrice")
```

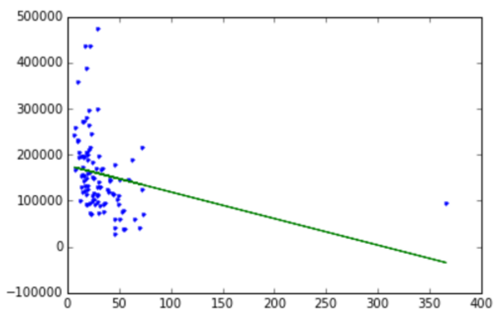


```
In [8]: crime_model = graphlab.linear_regression.create(sales, target='HousePrice', features=['CrimeRate'],
                                                        validation_set=None, verbose=False)

        # sales : dataset
        # target : 예측의 대상
        # features : 고려할 특징. 이번 실습은 단일 모델임
```

```
In [9]: import matplotlib.pyplot as plt
```

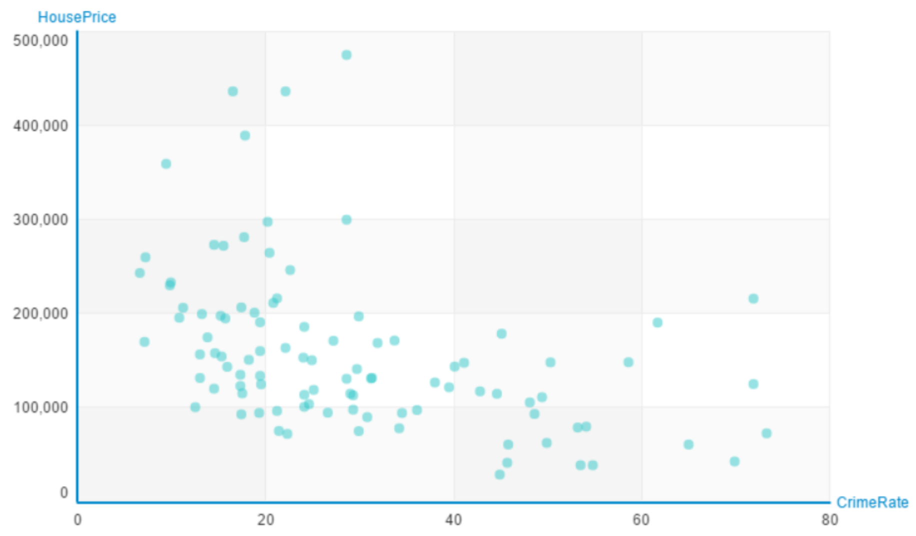
```
In [10]: %matplotlib inline
```

```
In [11]: plt.plot(sales['CrimeRate'],sales['HousePrice'],'.',
                 sales['CrimeRate'],crime_model.predict(sales),'-')

Out[11]: [<matplotlib.lines.Line2D at 0x34d06a58>,
          <matplotlib.lines.Line2D at 0x34d06c88>]
```



In [13]:
```
sales_noCC = sales[sales['MilesPhila'] != 0.0]

# MilesPhila 즉 필라델피아의 도심과의 거리가 0이 아닌 곳들의 정보만을 추출한다.
# 도심은 범죄율이 높음에도 불구하고 집값이 높기 때문이다.
# 도심은 MilesPhila == 0 이다.
```

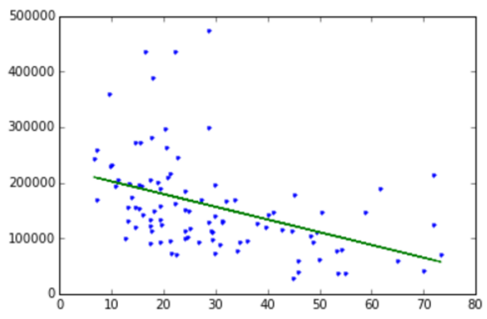
In [14]:
```
sales_noCC.show(view="Scatter Plot", x="CrimeRate", y="HousePrice")

# 도심을 제외한 dataset을 이용해 다시 산점도를 그려보면 아래와 같다.
```



In [15]:
```
# 이제 plt.plot을 이용해서 선형회귀모델을 세운 뒤 그래프를 통해 결과를 보고 싶다.
# 무작정 plt.plot을 쓰려고 하지 말아라. 그건 그냥 보여주는 역할일 뿐 모델을 세우는 것이 우선이다.

crime_model_noCC = graphlab.linear_regression.create(sales_noCC, target="HousePrice", features=['CrimeRate'],
                                                      validation_set=None,verbose=False)

# sales_noCC : dataset
# target : 예측의 대상
# features : 고려할 특징. 이번 실습은 단일 모델임
```

In [17]:
```
plt.plot(sales_noCC['CrimeRate'],sales_noCC['HousePrice'],'.',
         sales_noCC['CrimeRate'],crime_model_noCC.predict(sales_noCC),'-')

# 첫 번째 줄과 두 번째 줄의 그래프를 각각 그린 후 겹치게 한다.
# x축과 y축을 각각 설정한다.
# 특히 두 번째 줄의 y축으로 predicted value가 들어가야 하기 때문에 regression model이 적용된 것을 써야 한다.
# 바로 위의 테스트 코드를 보면 crime_model_noCC가 linear_regression을 통해 생성되었다는 걸 알 수 있다.
# predict는 linear_regression의 메소드이기 때문에 이 관계를 항상 잘 생각해야 한다.
```

Out[17]:
```
[<matplotlib.lines.Line2D at 0x34fc7fd0>,
 <matplotlib.lines.Line2D at 0x34cb0588>]
```



계수 구하기 : high-leverage point의 유무

In [18]:
```
crime_model.get('coefficients')

# 아래의 결과를 해석해보자
# 범죄 단위가 하나 증가하면, 집 값은 -576으로 증가한다는 것을 알 수 있다.
# 하지만 이것이 과연 정확한 결과일까?
# 참고로 crime_model은 high-leverage point가 포함된 dataset으로 세운 linear regression model이다.
```

Out[18]:

name	index	value	stderr
(intercept)	None	176626.046881	11245.5882194
CrimeRate	None	-576.804949058	226.90225951

[2 rows x 4 columns]

In [20]:
```
crime_model_noCC.get('coefficients')

# 아래의 결과를 해석해보자
# 범죄 단위가 하나 증가하면, 집 값은 -2287로 증가한다는 것을 알 수 있다.
# 이것은 바로 위의 코드와 어떤 차이점이 있을까? --> high-leverage point가 제외된 dataset.
# high-leverage point를 제외하고 안하고의 영향이 무척 크다는 것을 관찰할 수 있다.
```

Out[20]:

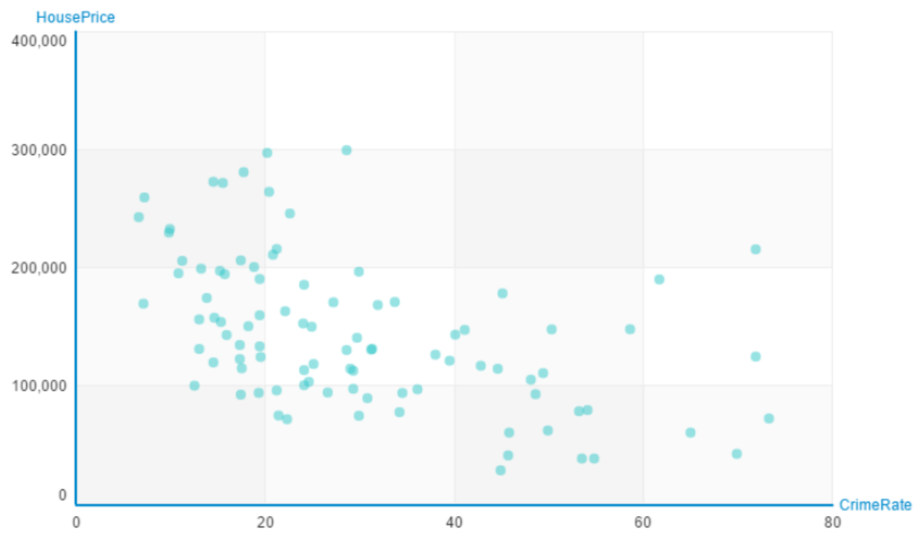
name	index	value	stderr
(intercept)	None	225204.604303	16404.0247514
CrimeRate	None	-2287.69717443	491.537478123

[2 rows x 4 columns]

계수 구하기 : high-value outlier의 유무

In [21]:
```
# plt.plot으로 그려본 저 그래프를 보자. 350000값보다 큰 5개의 점이 있다. 이를 high-value outlier라고 한다.
# 전체에 이상을 줄 정도의 이상점인지 아닌지, 그리고 high-leverage point에 비해서 얼마나 영향을 주는지 관찰해보자.

sales_nohighend = sales_noCC[sales_noCC['HousePrice'] < 350000]
# slaes_noCC는 high-leverage point가 제외된 dataset이다.
# high-leverage point가 제외된 dataset에서 이젠 high-value outlier마저 제외시켜 보겠다는 생각이다.
```

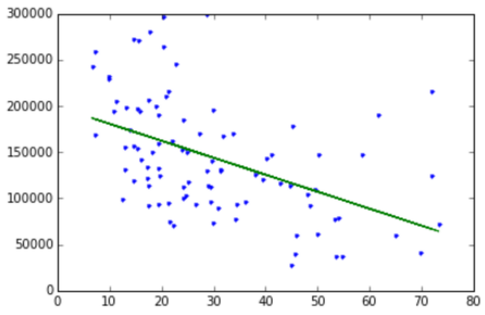
In [22]:
```
sales_nohighend.show(view="Scatter Plot", x="CrimeRate", y="HousePrice")
```



In [24]:
```
# 모델을 세워야 한다.

model_sales_nohighend = graphlab.linear_regression.create(sales_nohighend, target='HousePrice', features=['CrimeRate'],
                                                          validation_set=None, verbose=False)
```

In [25]:
```
plt.plot(sales_nohighend['CrimeRate'],sales_nohighend['HousePrice'],'.',
        sales_nohighend['CrimeRate'],model_sales_nohighend.predict(sales_nohighend),'-')
```

Out[25]:
```
[<matplotlib.lines.Line2D at 0x36447940>,
 <matplotlib.lines.Line2D at 0x36447b38>]
```



In [26]:
```
crime_model_noCC.get('coefficients')

# 계수는 항상 모델에서부터 얻어내는 것이구나.
```

Out[26]:

name	index	value	stderr
(intercept)	None	225204.604303	16404.0247514
CrimeRate	None	-2287.69717443	491.537478123

[2 rows x 4 columns]

In [27]:
```
model_sales_nohighend.get('coefficients')

# 결과를 해석해보자.
# high-value outlier는 high-leverage point에 비해 큰 차이를 일으키지는 않지만
# 비교적 어느 정도의 영향은 준다고 할 수 있다.
```

Out[27]:

name	index	value	stderr
(intercept)	None	199073.589615	11932.5101105
CrimeRate	None	-1837.71280989	351.519609333

[2 rows x 4 columns]

실습2(설명)

Regression Week 1: Simple Linear Regression Assignment

Predicting House Prices (One feature)

In this notebook we will use data on house sales in King County, where Seattle is located, to predict house prices using simple (one feature) linear regression. You will:

- Use SArray and SFrame functions to compute important summary statistics
- Write a function to compute the Simple Linear Regression weights using the closed form solution
- Write a function to make predictions of the output given the input feature
- Turn the regression around to predict the input/feature given the output
- Compare two different models for predicting house prices

If you are doing the assignment with IPython Notebook

An IPython Notebook has been provided below to you for this assignment. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

What you need to download

If you are using GraphLab Create:

- Download the King County House Sales data In SFrame format: kc_house_data.gl.zip
- Download the companion IPython Notebook: week-1-simple-regression-quiz-blank.ipynb
- Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

If you are not using GraphLab Create:

- Download the King County House Sales data csv file: kc_house_data.csv
- Download the King County House Sales training data csv file: kc_house_train_data.csv
- Download the King County House Sales testing data csv file: kc_house_test_data.csv
- IMPORTANT: use the following types for columns when importing the csv files. Otherwise, they may not be imported correctly: [str, str, float, float, float, float, int, str, int, int, int, int, int, int, int, str, float, float, float, float]. If your tool of choice requires a dictionary of types for importing csv files (e.g. Pandas), use:

```
dtype_dict = {'bathrooms':float, 'waterfront':int, 'sqft_above':int, 'sqft_living15':float, 'grade':int, 'yr_renovated':int, 'price':float, 'bedrooms':float, 'zipcode':str, 'long':float, 'sqft_lot15':float, 'sqft_living':float, 'floors':str, 'condition':int, 'lat':float, 'date':str, 'sqft_basement':int, 'yr_built':int, 'id':str, 'sqft_lot':int, 'view':int}
```

Useful resources

You may need to install the software tools or use the free Amazon EC2 machine. Instructions for both options are provided in the reading for Module 1.

If instead you are using other tools to do your homework

You are welcome, however, to write your own code and use any other libraries, like Pandas or R, to help you in the process. If you would like to take this path, follow the instructions below.

1. If you are using SFrame, import graphlab and load in the house data, otherwise you can also download the csv. (Note that we will be using the training and testing csv files provided). e.g in python with SFrames:

```
sales = graphlab.SFrame('kc_house_data.gl/')
```

2. Split data into 80% training and 20% test data. Using SFrame, use this command to set the same seed for everyone. e.g. in python with SFrames:

```
train_data,test_data = sales.random_split(.8,seed=0)
```

For those students not using graphlab please download the training and testing data csv files.

From now on we will train the models using train_data. It will be important that we use the same split here to ensure the results are the same.

3. Write a generic function that accepts a column of data (e.g, an SArray) ‘input_feature’ and another column ‘output’ and returns the Simple Linear Regression parameters ‘intercept’ and ‘slope’. Use the closed form solution from lecture to calculate the slope and intercept. e.g. in python:

```
def simple_linear_regression(input_feature, output):

    [your code here]

return(intercept, slope)
```

4. Use your function to calculate the estimated slope and intercept on the training data to predict ‘price’ given ‘sqft_living’. e.g. in python with SFrames using:

```
input_feature = train_data[‘sqft_living’]
```

```
output = train_data['price']
```

save the value of the slope and intercept for later (you might want to call them e.g. squarfeet_slope, and squarefeet_intercept)

5. Write a function that accepts a column of data 'input_feature', the 'slope', and the 'intercept' you learned, and returns an a column of predictions 'predicted_output' for each entry in the input column. e.g. in python:

```
def get_regression_predictions(input_feature, intercept, slope)
    [your code here]
return(predicted_output)
```

6. Quiz Question: Using your Slope and Intercept from (4), What is the predicted price for a house with 2650 sqft?

7. Write a function that accepts column of data: 'input_feature', and 'output' and the regression parameters 'slope' and 'intercept' and outputs the Residual Sum of Squares (RSS). e.g. in python:

```
def get_residual_sum_of_squares(input_feature, output, intercept,slope):
    [your code here]
return(RSS)
```

Recall that the RSS is the sum of the squares of the prediction errors (difference between output and prediction).

8. Quiz Question: According to this function and the slope and intercept from (4) What is the RSS for the simple linear regression using squarefeet to predict prices on TRAINING data?

9. Note that although we estimated the regression slope and intercept in order to predict the output from the input, since this is a simple linear relationship with only two variables we can invert the linear function to estimate the input given the output!

Write a function that accept a column of data:'output' and the regression parameters 'slope' and 'intercept' and outputs the column of data: 'estimated_input'. Do this by solving the linear function output = intercept + slope*input for the 'input' variable (i.e. 'input' should be on one side of the equals sign by itself). e.g. in python:

```
def inverse_regression_predictions(output, intercept, slope):
    [your code here]
return(estimated_input)
```

10. Quiz Question: According to this function and the regression slope and intercept from (3) what is the estimated square-feet for a house costing $800,000?

11. Instead of using ‘sqft_living’ to estimate prices we could use ‘bedrooms’ (a count of the number of bedrooms in the house) to estimate prices. Using your function from (3) calculate the Simple Linear Regression slope and intercept for estimating price based on bedrooms. Save this slope and intercept for later (you might want to call them e.g. bedroom_slope, bedroom_intercept).

12. Now that we have 2 different models compute the RSS from BOTH models on TEST data.

13. Quiz Question: Which model (square feet or bedrooms) has lowest RSS on TEST data? Think about why this might be the case.

실습2

단일 모델을 이용하여 회귀모델분석하기 2

집값과 범죄율 사이의 관계를 단일회귀모델을 이용하여 세우기

16.03.29 실습 수행, by 권도형(DOHK)

데이터셋 불러오기

In [2]:
```
import graphlab
```

In [4]:
```
sales = graphlab.SFrame('kc_house_data.gl/')

# graphlab을 쓰지 않는다고 할 때, dataset 전체에 해당하는 파일 1개,
# test set을 위한 파일 1개, train set을 위한 파일 1개 이렇게 세 개가 각각 필요하게 된다.

# --------------- #
# SFrame자료구조와 같은 자료구조를 내가 직접 만들어야 할까?
# --------------- #
```

In [5]:
```
train_set, test_set = sales.random_split(.8, seed=0)

# --------------- #
# seed가 0이라는 것은 무슨 의미일까?
# seed는 한마디로 초기값을 의미한다. seed(씨앗, 초기값)에서부터 난수가 생성되기 시작한다.
# 실제 구현시엔 train_set과 test_set 나누기를 어떻게 해야할까?
# --------------- #
```

In [6]:
```
prices = sales['price']

# 가격에 대한 정보만을 따로 저장하게 한다. sales는 SFrame, 즉 table형태의 자료구조이다.
# 반면 prices는 SFrame으로부터 떨어져나온 SArray다.

# --------------- #
# 실제 구현시엔 어떻게 특정 자료구조의 내용만을 따로 저장하는 것을 가능하게 할까?
# --------------- #
```

기본적인 내용에 대하여

SArray를 이용한 sum, mean 연산

In [9]:
```
sum_prices = prices.sum()
sum_prices
```

Out[9]:
```
11672925011.0
```

In [11]:
```
num_houses = prices.size()
num_houses
```

Out[11]:
```
21613
```

In [12]:
```
avg_prices_method1 = sum_prices/num_houses
avg_prices_method1
```

Out[12]:
```
540088.1419053348
```

In [13]:
```
avg_prices_method2 = prices.mean()
avg_prices_method2
```

Out[13]:
```
540088.1419053345
```

SArray를 이용한 scala-mutiply 연산

```
In [15]: prices
```

```
Out[15]: dtype: float
Rows: 21613
[221900.0, 538000.0, 180000.0, 604000.0, 510000.0, 1225000.0, 257500.0, 291850.0, 229500.0, 323000.0, 662500.0, 468000.0, 310000.0, 400000.
0, 530000.0, 650000.0, 395000.0, 485000.0, 189000.0, 230000.0, 385000.0, 2000000.0, 285000.0, 252700.0, 329000.0, 233000.0, 937000.0, 66700
0.0, 438000.0, 719000.0, 580500.0, 280000.0, 687500.0, 535000.0, 322500.0, 696000.0, 550000.0, 640000.0, 240000.0, 605000.0, 625000.0, 77500
0.0, 861990.0, 685000.0, 309000.0, 488000.0, 210490.0, 785000.0, 450000.0, 1350000.0, 228000.0, 345000.0, 600000.0, 585000.0, 920000.0, 8850
00.0, 292500.0, 301000.0, 951000.0, 430000.0, 650000.0, 289000.0, 505000.0, 549000.0, 425000.0, 317625.0, 975000.0, 287000.0, 204000.0, 1325
000.0, 1040000.0, 325000.0, 571000.0, 360000.0, 349000.0, 832500.0, 380000.0, 480000.0, 410000.0, 720000.0, 390000.0, 360000.0, 355000.0, 35
6000.0, 315000.0, 940000.0, 305000.0, 461000.0, 215000.0, 335000.0, 243500.0, 1099880.0, 153000.0, 430000.0, 700000.0, 905000.0, 247500.0, 1
99000.0, 314000.0, 437500.0, ... ]
```

```
In [14]: multiplying_with_constant = 0.5*prices
multiplying_with_constant
```

```
Out[14]: dtype: float
Rows: 21613
[110950.0, 269000.0, 90000.0, 302000.0, 255000.0, 612500.0, 128750.0, 145925.0, 114750.0, 161500.0, 331250.0, 234000.0, 155000.0, 200000.0,
265000.0, 325000.0, 197500.0, 242500.0, 94500.0, 115000.0, 192500.0, 1000000.0, 142500.0, 126350.0, 164500.0, 116500.0, 468500.0, 333500.0,
219000.0, 359500.0, 290250.0, 140000.0, 343750.0, 267500.0, 161250.0, 348000.0, 275000.0, 320000.0, 120000.0, 302500.0, 312500.0, 387500.0,
430995.0, 342500.0, 154500.0, 244000.0, 105245.0, 392500.0, 225000.0, 675000.0, 114000.0, 172500.0, 300000.0, 292500.0, 460000.0, 442500.0,
146250.0, 150500.0, 475500.0, 215000.0, 325000.0, 144500.0, 252500.0, 274500.0, 212500.0, 158812.5, 487500.0, 143500.0, 102000.0, 662500.0,
520000.0, 162500.0, 285500.0, 180000.0, 174500.0, 416250.0, 190000.0, 240000.0, 205000.0, 360000.0, 195000.0, 180000.0, 177500.0, 178000.0,
157500.0, 470000.0, 152500.0, 230500.0, 107500.0, 167500.0, 121750.0, 549940.0, 76500.0, 215000.0, 350000.0, 452500.0, 123750.0, 99500.0, 15
7000.0, 218750.0, ... ]
```

SArray를 이용한 SArray-multiply 연산

```
In [17]: prices_squared = prices*prices
prices_squared
```

```
Out[17]: dtype: float
Rows: 21613
[49239610000.0, 289444000000.0, 32400000000.0, 364816000000.0, 260100000000.0, 1500625000000.0, 66306250000.0, 85176422500.0, 52670250000.0,
104329000000.0, 438906250000.0, 219024000000.0, 96100000000.0, 160000000000.0, 280900000000.0, 422500000000.0, 156025000000.0, 235225000000.
0, 35721000000.0, 52900000000.0, 148225000000.0, 4000000000000.0, 81225000000.0, 63857290000.0, 108241000000.0, 54289000000.0, 877969000000.
0, 444889000000.0, 191844000000.0, 516961000000.0, 336980250000.0, 78400000000.0, 472656250000.0, 286225000000.0, 104006250000.0, 4844160000
00.0, 302500000000.0, 409600000000.0, 57600000000.0, 366025000000.0, 390625000000.0, 600625000000.0, 743026760100.0, 469225000000.0, 9548100
0000.0, 238144000000.0, 44306040100.0, 616225000000.0, 202500000000.0, 1822500000000.0, 51984000000.0, 119025000000.0, 360000000000.0, 34222
5000000.0, 846400000000.0, 783225000000.0, 85556250000.0, 90601000000.0, 904401000000.0, 184900000000.0, 422500000000.0, 83521000000.0, 2550
25000000.0, 301401000000.0, 180625000000.0, 100885640625.0, 950625000000.0, 82369000000.0, 41616000000.0, 1755625000000.0, 1081600000000.0,
105625000000.0, 326041000000.0, 129600000000.0, 121801000000.0, 693056250000.0, 144400000000.0, 230400000000.0, 168100000000.0, 51840000000
0.0, 152100000000.0, 129600000000.0, 126025000000.0, 126736000000.0, 99225000000.0, 883600000000.0, 93025000000.0, 212521000000.0, 462250000
00.0, 112225000000.0, 59292250000.0, 1209736014400.0, 23409000000.0, 184900000000.0, 490000000000.0, 819025000000.0, 61256250000.0, 39601000
000.0, 98596000000.0, 191406250000.0, ... ]
```

SArray를 이용한 SS(sum of squares:제곱합) 연산

```
In [19]: sum_prices_squares = prices_squared.sum()
sum_prices_squares
```

```
Out[19]: 9217325133550736.0
```

gradient가 적용된 simple linear regression 모델 만들기

In [130]:
```
def simple_linear_regression(input_feature, output):
# xi의 시그마, yi의 시그마 구하기
    N = input_feature.size()
    sum_output = output.sum()                                 # sum of Y
    sum_input_feature = input_feature.sum()                   # sum of X
    product_input_output = input_feature*output
    squared_input_feature = input_feature*input_feature
    sum_product_input_output = product_input_output.sum()     # sum of X*Y
    sum_squared_input_feature = squared_input_feature.sum()   # sum of X^2
    squared_sum_input_feature = sum_input_feature*sum_input_feature  # (sum of X) * (sum of X)

    print "length of feature : " + str(N)

    numerator = (sum_product_input_output)-(1/N)*((sum_input_feature)*(sum_output))
    denominator = (sum_squared_input_feature) - (1/N)*((sum_input_feature)*(sum_input_feature))
    slope = numerator/denominator

    mean_sum_output = sum_output/N
    mean_sum_input_feature = sum_input_feature/N

    intercept = mean_sum_output - ((slope)*(mean_sum_input_feature))


#    intercept = ((1/N)*(sum_output)-((sum_product_input_output
#                         -((1/N)*sum_product_input_output)/((sum_squared_input_feature
#                                     -(1/N)*squared_sum_input_feature)))*(1/N)*(sum_input_feature))

#    print "test - " + str(sum_input_feature)
#    print "test - " + str(sum_output)
#    print "test - " + str(squared_sum_input_feature)
#    print "test - " + str(product_input_output)
#    print "test - " + str(sum_product_input_output)
#    print "test - " + str(squared_input_feature)
#    print "test - " + str(sum_squared_input_feature)

    return (intercept, slope)
```

In [131]:
```
# 간단한 모수 추정, 성공했는지 확인
test_feature = graphlab.SArray(range(5))
test_output = graphlab.SArray(1 + 1*test_feature)
(test_intercept,test_slope) = simple_linear_regression(test_feature, test_output)
print "Intercept : " + str(test_intercept)
print "Slope : " + str(test_slope)
```

```
length of feature : 5
Intercept : 1
Slope : 1
```

In [132]:
```
# 80%의 훈련집합으로 모델을 훈련시켜 모수를 구함
sqft_intercept, sqft_slope = simple_linear_regression(train_set['sqft_living'], train_set['price'])

print "Intercept :" + str(sqft_intercept)
print "Slope : " + str(sqft_slope)
```

```
length of feature : 17384
Intercept :-7731.6858275
Slope : 263.024303956
```

In [133]:
```
def get_regression_predictions(input_feature, intercept, slope):
    product_slope_feature = slope*(input_feature)
    predicted_values = intercept + product_slope_feature # 예측치 구하기 y=w0+w1*x

    return predicted_values
```

In [140]:
```
my_house_sqft = 2650
estimated_price = get_regression_predictions(my_house_sqft,sqft_intercept, sqft_slope)
print "%d평방피트인 집값의 예측치는 $%.2f" % (my_house_sqft, estimated_price)
```

```
2650평방피트인 집값의 예측치는 $689282.72
```

만든 모델에 대해 RSS를 이용하여 평가하기

In [141]:
```
def get_residual_sum_of_squares(input_feature, output, intercept, slope):
    product_slope_feature = slope*(input_feature)
    predicted_values = intercept + product_slope_feature # 예측치

    residuals = output-predicted_values # "잔차"구하기;빼는 순서는 상관없다. 어자피 square되기 때문에.

    squared_residuals = residuals*residuals # "잔차제곱"구하기
    sum_squared_residuals = squared_residuals.sum() # "잔차제곱합(RSS)"구하기

    RSS = sum_squared_residuals

    return(RSS)
```

In [142]:
```
print get_residual_sum_of_squares(test_feature,test_output,test_intercept,test_slope)
# 정확히 회귀선 위에 놓인 데이터로 RSS를 구하는 중이기 때문에 0이 나와야 정상이다.
```
```
0
```

In [144]:
```
rss_prices_on_sqft = get_residual_sum_of_squares(train_set['sqft_living'],
                                                 train_set['price'],
                                                 sqft_intercept,
                                                 sqft_slope)
print '면적에 대한 가격을 예측해본 결과에 대한 RSS는 ' + str(rss_prices_on_sqft)
```
```
면적에 대한 가격을 예측해본 결과에 대한 RSS는 1.2072119188e+15
```

가격이 주어졌을 때 feature를 예측하기

In [148]:
```
# 함수관계; y = a + b*x가 주어진 상태이므로 x도 예측 가능하다.
# 기울기:b, 절편:a, 가격:y
# x = ?

def inverse_regression_predictions(output, intercept, slope):
    estimated_feature = (output-intercept)*(1/slope)

    return estimated_feature
```

In [151]:
```
my_house_price = 800000
estimated_squarefeet = inverse_regression_predictions(my_house_price, sqft_intercept, sqft_slope)
print "가격이 %.2f인 집의 평방피트는 %d피트로 예상됨" % (my_house_price, estimated_squarefeet)
```
```
가격이 800000.00인 집의 평방피트는 3070피트로 예상됨
```

Random_split 이용

```
object samples
{
	import org.apache.spark.sql.SQLContext
	def simple_linear_regression(input_feature:List[Double],output:List[Double]):(Double,Double)=
	{
	val N = input_feature.size
	val sum_output = output.sum
	val sum_input_feature = input_feature.sum
	val product_input_output= (input_feature,output).zipped.map(_*_)
	val squared_input_feature= (input_feature,output).zipped.map(_*_)
	val sum_product_input_output= product_input_output.sum
	val sum_squared_input_feature= squared_input_feature.sum
	println("Length of feature :" + N)
	val numerator = sum_product_input_output - (1/N) * (sum_input_feature * sum_output)
	val denominator = sum_squared_input_feature - (1/N) * (sum_input_feature * sum_input_feature)
	val slope = numerator / denominator
	val mean_sum_output = sum_output / N
	val mean_sum_input_feature = sum_input_feature / N
	val intercept = mean_sum_output - slope * mean_sum_input_feature
	return (intercept,slope)
	}

	def get_residual_sum_of_squares
	(input_feature:List[Double],output:List[Double],intercept:Double,slope:Double):(Double)=
	{
	val product_slope_feature = input_feature.map(x=> x * slope)
	val predicted_values = product_slope_feature.map(x=>x+intercept)
	val residuals = (output,predicted_values).zipped.map(_-_)
	val squared_residuals= (residuals, residuals).zipped.map(_*_)
	val sum_squared_residuals = squared_residuals.sum
	return sum_squared_residuals
	}

	def get_regression_predictions(input_feature:Double, intercept:Double, slope:Double):(Double)=
	{
		val product_slope_feature = slope * input_feature
		return (intercept + product_slope_feature)
	}

	def inverse_regression_predictions(output:Double,intercept:Double,slope:Double):(Double)=
	{
	return output - intercept * (1/slope)
	}
	def main2()
	{
	val sqlContext = new SQLContext(sc)
	val df = sqlContext.read.format("com.databricks.spark.csv").option("header","true")
	.option("inferSchema","true").load("./kc_house_data.csv")

	val splits = df.randomSplit(Array(.8,.2), seed = 0)
	val train_set = splits(0).cache()
	val test_set = splits(1)

	val train_set_df = train_set.toDF()
	val test_set_df = test_set.toDF()
	train_set_df.registerTempTable("train_set")


	val trset_sqft_living = sqlContext.sql("SELECT sqft_living FROM train_set")
	val trset_temp1 = trset_sqft_living.map(t=>t(0)).collect()
	val trset_List1 = trset_temp1.toList
	val trset_living_Double = trset_List1.map(_.toString.toDouble)
	val trset_price = sqlContext.sql("SELECT price FROM train_set")
	val trset_temp2 = trset_price.map(t=>t(0)).collect()
	val trset_List2 = trset_temp2.toList
	val trset_price_Double = trset_List2.map(_.toString.toDouble)


	val (sqft_intercept,sqft_slope) = simple_linear_regression(trset_living_Double,trset_price_Double)
	println("Intercept : "+sqft_intercept+"\nSlope : "+sqft_slope)

	val rss_prices_on_sqft = get_residual_sum_of_squares
			(trset_living_Double,trset_price_Double,sqft_intercept,sqft_slope)
```

```
        println("RSS : "+rss_prices_on_sqft)

        val my_house_sqft=2650.0
        val estimated_price = get_regression_predictions(my_house_sqft,sqft_intercept,sqft_slope)
        println(my_house_sqft+" : $"+estimated_price)

        val my_house_price= 800000.0
        val estimated_squarefeet = inverse_regression_predictions(my_house_price,sqft_intercept,sqft_slope)
        println("$"+my_house_price+" : "+estimated_squarefeet)
        }
}
```

Test_set.csv, train_set.csv 파일 이용

```
object samples
{
	import org.apache.spark.sql.SQLContext
	def simple_linear_regression(input_feature:List[Double],output:List[Double]):(Double,Double)=
	{
	val N = input_feature.size
	val sum_output = output.sum
	val sum_input_feature = input_feature.sum
	val product_input_output= (input_feature,output).zipped.map(_*_)
	val squared_input_feature= (input_feature,input_feature).zipped.map(_*_)
	val sum_product_input_output= product_input_output.sum
	val sum_squared_input_feature= squared_input_feature.sum
	println("Length of feature :" + N)
	val numerator = sum_product_input_output - (1/N) * (sum_input_feature * sum_output)
	val denominator = sum_squared_input_feature - (1/N) * (sum_input_feature * sum_input_feature)
	val slope = numerator / denominator
	val mean_sum_output = sum_output / N
	val mean_sum_input_feature = sum_input_feature / N
	val intercept = mean_sum_output - (slope * mean_sum_input_feature)
	return (intercept,slope)
	}

	def
get_residual_sum_of_squares(input_feature:List[Double],output:List[Double],intercept:Double,slope:Double):(Double)=
	{
	val product_slope_feature = input_feature.map(x=> x * slope)
	val predicted_values = product_slope_feature.map(x=>x+intercept)
	val residuals = (output,predicted_values).zipped.map(_-_)
	val squared_residuals= (residuals, residuals).zipped.map(_*_)
	val sum_squared_residuals = squared_residuals.sum
	return sum_squared_residuals
	}

	def get_regression_predictions(input_feature:Double, intercept:Double, slope:Double):(Double)=
	{
		val product_slope_feature = slope * input_feature
		return (intercept + product_slope_feature)
	}

	def inverse_regression_predictions(output:Double,intercept:Double,slope:Double):(Double)=
	{
	return ((output - intercept) * (1/slope))
	}

	def main1()
	{
	val sqlContext = new SQLContext(sc)
	val                                              train_set_df                                           =
sqlContext.read.format("com.databricks.spark.csv").option("header","true").option("inferSchema","true").load("./kc_hou
se_train.csv")
	/*
	val splits = df.randomSplit(Array(.8,.2), seed = 0)
	val train_set = splits(0).cache()
	val test_set = splits(1)
	val train_set_df = train_set.toDF()
	val test_set_df = test_set.toDF()
	*/
	train_set_df.registerTempTable("train_set")
	val trset_sqft_living = sqlContext.sql("SELECT sqft_living FROM train_set")
	val trset_temp1 = trset_sqft_living.map(t=>t(0)).collect()
	val trset_List1 = trset_temp1.toList
	val trset_living_Double = trset_List1.map(_.toString.toDouble)
	val trset_price = sqlContext.sql("SELECT price FROM train_set")
	val trset_temp2 = trset_price.map(t=>t(0)).collect()
	val trset_List2 = trset_temp2.toList
	val trset_price_Double = trset_List2.map(_.toString.toDouble.toDouble)
	val (sqft_intercept,sqft_slope) = simple_linear_regression(trset_living_Double,trset_price_Double)
	println("Intercept : "+sqft_intercept+"\nSlope : "+sqft_slope)
	val                                           rss_prices_on_sqft                                         =
get_residual_sum_of_squares(trset_living_Double,trset_price_Double,sqft_intercept,sqft_slope)
	println("RSS : "+rss_prices_on_sqft)
	val my_house_sqft=2650.0
	val estimated_price = get_regression_predictions(my_house_sqft,sqft_intercept,sqft_slope)
```

```
        println(my_house_sqft+" : $"+estimated_price)
        val my_house_price= 800000.0
        val estimated_squarefeet = inverse_regression_predictions(my_house_price,sqft_intercept,sqft_slope)
        println("$"+my_house_price+" : "+estimated_squarefeet)
        }
}
```

Multiple Linear Regression

실습설명

Regression Week 2: Multiple Linear Regression Assignment 1

Predicting House Prices (Multiple Variables)

In this notebook you will use data on house sales in King County to predict prices using multiple regression. The first assignment will be about exploring multiple regression in particular exploring the impact of adding features to a regression and measuring error. In the second assignment you will implement a gradient descent algorithm. In this assignment you will:

- Use SFrames to do some feature engineering
- Use built-in GraphLab Create (or otherwise) functions to compute the regression weights (coefficients)
- Given the regression weights, predictors and outcome write a function to compute the Residual Sum of Squares
- Look at coefficients and interpret their meanings
- Evaluate multiple models via RSS

If you are doing the assignment with IPython Notebook

An IPython Notebook has been provided below to you for this assignment. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

What you need to download

If you are using GraphLab Create:

- Download the King County House Sales data In SFrame format: kc_house_data.gl.zip
- Download the companion IPython Notebook: week-2-multiple-regression-assignment-1-blank.ipynb
- Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

If you are not using GraphLab Create:

- Download the King County House Sales data csv file: kc_house_data.csv
- Download the King County House Sales training data csv file: kc_house_train_data.csv
- Download the King County House Sales testing data csv file: kc_house_test_data.csv

- IMPORTANT: use the following types for columns when importing the csv files. Otherwise, they may not be imported correctly: [str, str, float, float, float, float, int, str, int, int, int, int, int, int, int, int, str, float, float, float, float]. If your tool of choice requires a dictionary of types for importing csv files (e.g. Pandas), use:

```
dtype_dict = {'bathrooms':float, 'waterfront':int, 'sqft_above':int, 'sqft_living15':float, 'grade':int, 'yr_renovated':int, 'price':float, 'bedrooms':float, 'zipcode':str, 'long':float, 'sqft_lot15':float, 'sqft_living':float, 'floors':str, 'condition':int, 'lat':float, 'date':str, 'sqft_basement':int, 'yr_built':int, 'id':str, 'sqft_lot':int, 'view':int}
```

Useful resources

You may need to install the software tools or use the free Amazon EC2 machine. Instructions for both options are provided in the reading for Module 1.

If instead you are using other tools to do your homework

You are welcome, however, to write your own code and use any other libraries, like Pandas or R, to help you in the process. If you would like to take this path, follow the instructions below.

1. If you are using SFrame, import graphlab and load in the house data, otherwise you can also download the csv. (Note that we will be using the training and testing csv files provided) e.g in python with SFrames:

```
sales = graphlab.SFrame('kc_house_data.gl/')
```

2. Split into training and test data Use this command to set the same seed for everyone: e.g. in python with SFrames:

```
train_data,test_data = sales.random_split(.8,seed=0)
```

For those students not using SFrames please download the training and testing data csv files.

From now on we will train the models using train_data. It will be important that we use the same split here to ensure the results are the same.

3. Although we often think of multiple regression as including multiple different features (e.g. # of bedrooms, square feet, and # of bathrooms) but we can also consider transformations of existing variables e.g. the log of the square feet or even "interaction" variables such as the product of bedrooms and bathrooms. Add 4 new variables in both your train_data and test_data.

- ‘bedrooms_squared’ = ‘bedrooms’*‘bedrooms’
- ‘bed_bath_rooms’ = ‘bedrooms’*‘bathrooms’
- ‘log_sqft_living’ = log(‘sqft_living’)
- ‘lat_plus_long’ = ‘lat’ + ‘long’

Before we continue let's explain these new variables:

- Squaring bedrooms will increase the separation between not many bedrooms (e.g. 1) and lots of bedrooms (e.g. 4) since 1^2 = 1 but 4^2 = 16. Consequently this variable will mostly affect houses with many bedrooms.
- Bedrooms times bathrooms is what's called an "interaction" variable. It is large when both of them are large.
- Taking the log of square feet has the effect of bringing large values closer together and spreading out small values.
- Adding latitude to longitude is non-sensical but we will do it anyway (you'll see why)

For those students not using SFrames you should first download and import the training and testing data sets provided and then add the four new variables each to both data sets (training and testing)

4. Quiz Question: what are the mean (arithmetic average) values of your 4 new variables on TEST data? (round to 2 digits)

5. Use graphlab.linear_regression.create (or any other regression library/function) to estimate the regression coefficients/weights for predicting 'price' for the following three models:(In all 3 models include an intercept -- most software does this by default).

- Model 1: 'sqft_living', 'bedrooms', 'bathrooms', 'lat', and 'long'
- Model 2: 'sqft_living', 'bedrooms', 'bathrooms', 'lat','long', and 'bed_bath_rooms'
- Model 3: 'sqft_living', 'bedrooms', 'bathrooms', 'lat','long', 'bed_bath_rooms', 'bedrooms_squared', 'log_sqft_living', and 'lat_plus_long'

You'll note that the three models here are "nested" in that all of the features of the Model 1 are in Model 2 and all of the features of Model 2 are in Model 3.

If you use graphlab.linear_regression.create() to estimate these models please ensure that you set validation_set = None. This way you will get the same answer every time you run the code.

Learn all three models on the TRAINING data set. Save your model results for quiz questions later.

6. Quiz Question: What is the sign (positive or negative) for the coefficient/weight for 'bathrooms' in Model 1?

7. Quiz Question: What is the sign (positive or negative) for the coefficient/weight for 'bathrooms' in Model 2?

8. Is the sign for the coefficient the same in both models? Think about why this might be the case.

9. Now using your three estimated models compute the RSS (Residual Sum of Squares) on the Training data.

10. Quiz Question: Which model (1, 2 or 3) had the lowest RSS on TRAINING data?

11. Now using your three estimated models compute the RSS on the Testing data

12. Quiz Question: Which model (1, 2, or 3) had the lowest RSS on TESTING data?

13. Did you get the same answer for 9 and 11? Think about why this might be the case.

실습1

```
In [ ]:
```

```
In [1]: import graphlab
```

```
In [2]: sales = graphlab.SFrame('kc_house_data.gl/')
```

```
2016-04-04 13:11:33,792 [INFO] graphlab.cython.cy_server, 176: GraphLab Create v1.8.5 started. Logging: C:₩Users₩dato₩AppData₩Local₩Temp₩graphlab_server_1459743091.log.0
```

```
This non-commercial license of GraphLab Create is assigned to dohk@koreatech.ac.kr and will expire on January 17, 2017. For commercial licensing options, visit https://dato.com/buy/.
```

```
In [3]: train_data, test_data = sales.random_split(.8,seed=0)
```

```
In [4]: example_features = ['sqft_living','bedrooms','bathrooms']
```

Learning a multiple regression model

```
In [5]: example_model = graphlab.linear_regression.create(train_data,target='price',features=example_features,validation_set=None)
        # 다변수(여러 개의 feature들)로 이루어진 회귀 모델을 fitting하는 단계.
```

```
Linear regression:

--------------------------------------------------------

Number of examples          : 17384

Number of features          : 3

Number of unpacked features : 3

Number of coefficients    : 4

Starting Newton Method

--------------------------------------------------------

+-----------+----------+--------------+--------------------+---------------+

| Iteration | Passes   | Elapsed Time | Training-max_error | Training-rmse |

+-----------+----------+--------------+--------------------+---------------+

| 1         | 2        | 1.044522     | 4146407.600631     | 258679.804477 |

+-----------+----------+--------------+--------------------+---------------+

SUCCESS: Optimal solution found.
```

```
In [6]: # data를 모델에 성공적으로 적용했으므로, 회귀 가중치(regression weights;coefficients)를 뽑아낼 수 있다.
        example_weight_summary = example_model.get("coefficients")
```

In [7]:
```
print example_weight_summary
```
```
+-------------+-------+----------------+---------------+
|     name    | index |     value      |     stderr    |
+-------------+-------+----------------+---------------+
| (intercept) |  None | 87910.0724924  |  7873.3381434 |
| sqft_living |  None | 315.403440552  | 3.45570032585 |
|   bedrooms  |  None | -65080.2155528 | 2717.45685442 |
|  bathrooms  |  None | 6944.02019265  | 3923.11493144 |
+-------------+-------+----------------+---------------+
[4 rows x 4 columns]
```

In [ ]:

In [8]:
```
example_predictions = example_model.predict(train_data)
```

In [9]:
```
print example_predictions[0]
```
```
271789.505878
```

In [ ]:

In [46]:
```
def get_residual_sum_of_squares(model, data, outcome):
    # First get the predictions
    predicted_data = model.predict(data)
    sum_of_predicted_data = predicted_data.sum()
    print sum_of_predicted_data

    # Then compute the residuals/errors
    rss = outcome.sum() - sum_of_predicted_data
    print "original data.sum() is : " + str(outcome.sum())
    print "residual : " + str(rss)

    # Then square and add them up
    squared_rss = rss*rss
    RSS = squared_rss
    print "rss is " +str(squared_rss)

    return RSS
```

In [11]:
```
rss_example_train = get_residual_sum_of_squares(example_model, test_data, test_data['price'])
```
```
2277972837.72
original data.sum() is : 2296575546.0
residual : 18602708.2762
3.46060755208e+14
```

In [12]:
```
print rss_example_train

#                나의 결과          |      나왔어야 할 값
#RSS       |  3.46060755208e+14     |      2.7376153833e+14
#residual  |  1.86027082762e+7     |      1.6545740791e+7
#predicted |  2.27797283772e+9     |      2.27797283772e+9
#무시해도 괜찮은 수준인가..
```
```
3.46060755208e+14
```

Create some new features

In [13]:
```
from math import log
```

In [21]:
```
# interaction feature를 이용하기 위해 이런식의 조작을 한다.

# bedrooms * bedrooms
train_data['bedrooms_squared'] = train_data['bedrooms'].apply(lambda x: x**2)
test_data['bedrooms_squared'] = test_data['bedrooms'].apply(lambda x: x**2)

# bedrooms * bathrooms
train_data['bed_bath_rooms'] = train_data['bedrooms']*train_data['bathrooms']
test_data['bed_bath_rooms'] = test_data['bedrooms']*test_data['bathrooms']

# log of squarefeet
train_data['log_sqft_living'] = train_data['sqft_living'].apply(lambda x: log(x))
test_data['log_sqft_living'] = test_data['sqft_living'].apply(lambda x: log(x))

# latitude
train_data['lat_plus_long'] = train_data['lat']+train_data['long']
test_data['lat_plus_long'] = test_data['lat']+test_data['long']
```

In [25]:
```
mean1 = test_data['bedrooms_squared'].mean()
mean2 = test_data['bed_bath_rooms'].mean()
mean3 = test_data['log_sqft_living'].mean()
mean4 = test_data['lat_plus_long'].mean()

print "each of the test_data's mean is : "
print  str(mean1)
print  str(mean2)
print  str(mean3)
print  str(mean4)
```

```
each of the test_data's mean is :
12.4466777016
7.50390163159
7.55027467965
-74.6533349722
```

Learning multiple models ¶

[1] feature 설정

In [40]:
```
# 이제, 다변수 집 값 예측을 위하여 아래와 같은 과정을 거친다.

# model #1은 세 모델 중 가장 적은 개수의 feature를 갖게 한다. model1이 갖는 feature는 다음과 같다.
#  --- squarefeet, # bedrooms, #bathrooms, latitude, longtitude

# model #2는 첫 번째 모델에서 feature 한 개가 더 추가 된 모델이다. ; bedrooms*bathrooms

# model #3는 두 번째 모델에서 여러 개의 feature가 더 추가 된 모델이다. 다음과 같은 feature들을 갖는다.
# --- log squarefeet, bedrooms squared, latitude+longtitude

model_1_feature = ['sqft_living', 'bedrooms', 'bathrooms', 'lat', 'long']
model_2_feature = model_1_feature+['bed_bath_rooms']
model_3_feature = model_2_feature+['log_sqft_living', 'bedrooms_squared', 'lat_plus_long']
```

[2] 모델 생성

In [41]:
```
model_1 = graphlab.linear_regression.create(train_data, target='price', features=model_1_feature, validation_set=None)
```

```
Linear regression:

--------------------------------------------------------

Number of examples          : 17384

Number of features          : 5

Number of unpacked features : 5

Number of coefficients    : 6

Starting Newton Method

--------------------------------------------------------

+-----------+----------+--------------+--------------------+---------------+

| Iteration | Passes   | Elapsed Time | Training-max_error | Training-rmse |

+-----------+----------+--------------+--------------------+---------------+

| 1         | 2        | 0.023062     | 4074878.213096     | 236378.596455 |

+-----------+----------+--------------+--------------------+---------------+

SUCCESS: Optimal solution found.
```

In [42]:
```
model_2 = graphlab.linear_regression.create(train_data, target='price', features=model_2_feature, validation_set=None)
```

```
Linear regression:

--------------------------------------------------------

Number of examples          : 17384

Number of features          : 6

Number of unpacked features : 6

Number of coefficients    : 7

Starting Newton Method

--------------------------------------------------------

+-----------+----------+--------------+--------------------+---------------+

| Iteration | Passes   | Elapsed Time | Training-max_error | Training-rmse |

+-----------+----------+--------------+--------------------+---------------+

| 1         | 2        | 0.029076     | 4014170.932928     | 235190.935428 |

+-----------+----------+--------------+--------------------+---------------+

SUCCESS: Optimal solution found.
```

In [43]:
```
model_3 = graphlab.linear_regression.create(train_data, target='price', features=model_3_feature, validation_set=None)
```

```
Linear regression:

--------------------------------------------------------

Number of examples          : 17384

Number of features          : 9

Number of unpacked features : 9

Number of coefficients    : 10

Starting Newton Method

--------------------------------------------------------

+-----------+----------+--------------+--------------------+---------------+

| Iteration | Passes   | Elapsed Time | Training-max_error | Training-rmse |

+-----------+----------+--------------+--------------------+---------------+

| 1         | 2        | 0.011029     | 3193229.177908     | 228200.043155 |

+-----------+----------+--------------+--------------------+---------------+

SUCCESS: Optimal solution found.
```

[3] weights(coefficients) 추출

In [44]:
```
model_1_coefficients = model_1.get('coefficients')
model_2_coefficients = model_2.get('coefficients')
model_3_coefficients = model_3.get('coefficients')

print "coefficients of model 1 is following"
print model_1_coefficients

print "coefficients of model 2 is following"
print model_2_coefficients

print "coefficients of model 3 is following"
print model_3_coefficients
```

```
coefficients of model 1 is following
+-------------+-------+----------------+---------------+
|     name    | index |     value      |     stderr    |
+-------------+-------+----------------+---------------+
| (intercept) |  None | -56140675.7444 | 1649985.42028 |
| sqft_living |  None | 310.263325778  | 3.18882960408 |
|   bedrooms  |  None | -59577.1160683 | 2487.27977322 |
|  bathrooms  |  None | 13811.8405419  | 3593.54213297 |
|     lat     |  None | 629865.789485  | 13120.7100323 |
|     long    |  None | -214790.285186 | 13284.2851607 |
+-------------+-------+----------------+---------------+
[6 rows x 4 columns]

coefficients of model 2 is following
+----------------+-------+----------------+---------------+
|      name      | index |     value      |     stderr    |
+----------------+-------+----------------+---------------+
|  (intercept)   |  None | -54410676.1152 | 1650405.16541 |
|  sqft_living   |  None | 304.449298056  | 3.20217535637 |
|    bedrooms    |  None | -116366.04323  | 4805.54966546 |
|   bathrooms    |  None | -77972.3305131 | 7565.05991091 |
|      lat       |  None | 625433.834953  | 13058.3530972 |
|      long      |  None | -203958.602959 | 13268.1283711 |
| bed_bath_rooms |  None | 26961.6249091  | 1956.36561555 |
+----------------+-------+----------------+---------------+
[7 rows x 4 columns]


coefficients of model 3 is following
+------------------+-------+----------------+---------------+
|       name       | index |     value      |     stderr    |
+------------------+-------+----------------+---------------+
|   (intercept)    |  None | -52974974.0598 | 1615194.94383 |
|   sqft_living    |  None | 529.196420561  | 7.69913498509 |
|     bedrooms     |  None | 28948.5277291  | 9395.72889104 |
|    bathrooms     |  None | 65661.2072295  | 10795.3380703 |
|       lat        |  None | 704762.148378  |      nan      |
|       long       |  None | -137780.019966 |      nan      |
|  bed_bath_rooms  |  None | -8478.36410481 | 2858.95391257 |
|  log_sqft_living |  None | -563467.78426  | 17567.8230813 |
| bedrooms_squared |  None | -6072.38466052 | 1494.97042777 |
|  lat_plus_long   |  None | -83217.197896  |      nan      |
+------------------+-------+----------------+---------------+
[10 rows x 4 columns]
```

Comparing multiple models

evaluate which model is best

[1] RSS

on Testing data

In [47]:
```
rss_example_model_1 = get_residual_sum_of_squares(model_1, test_data, test_data['price'])
```
```
2291043421.74
original data.sum() is : 2296575546.0
residual : 5532124.25967
rss is 3.06043988244e+13
```

In [48]:
```
rss_example_model_2 = get_residual_sum_of_squares(model_2, test_data, test_data['price'])
```
```
2290440224.87
original data.sum() is : 2296575546.0
residual : 6135321.13281
rss is 3.76421654027e+13
```

In [49]:
```
rss_example_model_3 = get_residual_sum_of_squares(model_3, test_data, test_data['price'])
```
```
2285802201.22
original data.sum() is : 2296575546.0
residual : 10773344.7835
rss is 1.16064957824e+14
```

on Training data

In [50]:
```
rss_example_model_1 = get_residual_sum_of_squares(model_1, train_data, train_data['price'])
```
```
9376349465.0
original data.sum() is : 9376349465.0
residual : 0.000225067138672
rss is 5.06552169099e-08
```

In [51]:
```
get_residual_sum_of_squares(model_2, train_data, train_data['price'])
```
```
9376349465.0
original data.sum() is : 9376349465.0
residual : 0.000263214111328
rss is 6.92816684023e-08
```

Out[51]:
```
6.928166840225458e-08
```

In [52]:
```
get_residual_sum_of_squares(model_3, train_data, train_data['price'])
```
```
9376349465.0
original data.sum() is : 9376349465.0
residual : -0.000200271606445
rss is 4.01087163482e-08
```

Out[52]:
```
4.0108716348186135e-08
```

Exploring different multiple regression models for house price prediction

1. What is the mean value (arithmetic average) of the 'bedrooms_squared' feature on TEST data? (round to 2 decimal places)

12.45

2. What is the mean value (arithmetic average) of the 'bed_bath_rooms' feature on TEST data? (round to 2 decimal places)

7.50

3. What is the mean value (arithmetic average) of the 'log_sqft_living' feature on TEST data? (round to 2 decimal places)

7.55

4. What is the mean value (arithmetic average) of the 'lat_plus_long' feature on TEST data? (round to 2 decimal places)

-74.65

5. **What is the sign (positive or negative) for the coefficient/weight for 'bathrooms' in model 1?**

[✓ Positive (+)] Negative (-)

6. **What is the sign (positive or negative) for the coefficient/weight for 'bathrooms' in model 2?**

Positive (+) [✓ Negative (-)]

7. **Which model (1, 2 or 3) has lowest RSS on TRAINING Data?**

Model 1 Model 2 [✓ Model 3]

8. **Which model (1, 2 or 3) has lowest RSS on TESTING Data?**

Model 1 [✓ Model 2] Model 3

About Numpy. Matrix operations are a significant part of the inner loop of the algorithms we will implement in this module and subsequent modules. To speed up your code, it can be important to use a specialized matrix operations library. There are many such libraries out there. In Python, we recommend Numpy, a popular open-source package for this task.

Numpy tutorial. To help you get started with Numpy, we've created a simple tutorial that goes over all the Numpy operations you will need for this course: numpy-tutorial.ipynb. We strongly recommend you go over this tutorial, before going any further.

More information on Numpy, beyond this tutorial, can be found in the Numpy getting started guide.

Installing Numpy. If you installed the Dato Launcher at the beginning of the course, you will automatically get Numpy. Otherwise, please follow these instructions to install it.(http://www.scipy.org/scipylib/download.html)

Numpy Tutorial

Numpy is a computational library for Python that is optimized for operations on multi-dimensional arrays. In this notebook we will use numpy to work with 1-d arrays (often called vectors) and 2-d arrays (often called matrices).

For a the full user guide and reference for numpy see: http://docs.scipy.org/doc/numpy/

In [ ]:
```
import numpy as np # importing this way allows us to refer to numpy as np
```

Creating Numpy Arrays

New arrays can be made in several ways. We can take an existing list and convert it to a numpy array:

In [ ]:
```
mylist = [1., 2., 3., 4.]
mynparray = np.array(mylist)
mynparray
```

You can initialize an array (of any dimension) of all ones or all zeroes with the ones() and zeros() functions:

In [ ]:
```
one_vector = np.ones(4)
print one_vector # using print removes the array() portion
```

In [ ]:
```
one2Darray = np.ones((2, 4)) # an 2D array with 2 "rows" and 4 "columns"
print one2Darray
```

In [ ]:
```
zero_vector = np.zeros(4)
print zero_vector
```

You can also initialize an empty array which will be filled with values. This is the fastest way to initialize a fixed-size numpy array however you must ensure that you replace all of the values.

In [ ]:
```
empty_vector = np.empty(5)
print empty_vector
```

Accessing array elements

Accessing an array is straight forward. For vectors you access the index by referring to it inside square brackets. Recall that indices in Python start with 0.

In [ ]:
```
mynparray[2]
```

2D arrays are accessed similarly by referring to the row and column index separated by a comma:

In [ ]:
```
my_matrix = np.array([[1, 2, 3], [4, 5, 6]])
print my_matrix
```

In [ ]:
```
print my_matrix[1, 2]
```

Sequences of indices can be accessed using ':' for example

In [ ]:
```
print my_matrix[0:2, 2] # recall 0:2 = [0, 1]
```

In [ ]:
```
print my_matrix[0, 0:3]
```

You can also pass a list of indices.

In [ ]:
```
fib_indices = np.array([1, 1, 2, 3])
random_vector = np.random.random(10) # 10 random numbers between 0 and 1
print random_vector
```

In [ ]:
```
print random_vector[fib_indices]
```

You can also use true/false values to select values

In [ ]:
```
my_vector = np.array([1, 2, 3, 4])
select_index = np.array([True, False, True, False])
print my_vector[select_index]
```

For 2D arrays you can select specific columns and specific rows. Passing ':' selects all rows/columns

In [ ]:
```
select_cols = np.array([True, False, True]) # 1st and 3rd column
select_rows = np.array([False, True]) # 2nd row
```

In [ ]:
```
print my_matrix[select_rows, :] # just 2nd row but all columns
```

In [ ]:
```
print my_matrix[:, select_cols] # all rows and just the 1st and 3rd column
```

Operations on Arrays

You can use the operations '*', '**', '\', '+' and '-' on numpy arrays and they operate elementwise.

In [ ]:
```
my_array = np.array([1., 2., 3., 4.])
print my_array*my_array
```

In [ ]:
```
print my_array**2
```

In [ ]:
```
print my_array - np.ones(4)
```

In [ ]:
```
print my_array + np.ones(4)
```

In [ ]:
```
print my_array / 3
```

In [ ]:
```
print my_array / np.array([2., 3., 4., 5.]) # = [1.0/2.0, 2.0/3.0, 3.0/4.0, 4.0/5.0]
```

You can compute the sum with np.sum() and the average with np.average()

In [ ]:
```
print np.sum(my_array)
```

In [ ]:
```
print np.average(my_array)
```

In [ ]:
```
print np.sum(my_array)/len(my_array)
```

The dot product ¶

An important mathematical operation in linear algebra is the dot product.

When we compute the dot product between two vectors we are simply multiplying them elementwise and adding them up. In numpy you can do this with np.dot()

In [ ]:
```
array1 = np.array([1., 2., 3., 4.])
array2 = np.array([2., 3., 4., 5.])
print np.dot(array1, array2)
```

In [ ]:
```
print np.sum(array1*array2)
```

Recall that the Euclidean length (or magnitude) of a vector is the squareroot of the sum of the squares of the components. This is just the squareroot of the dot product of the vector with itself:

In [ ]:
```
array1_mag = np.sqrt(np.dot(array1, array1))
print array1_mag
```

In [ ]:
```
print np.sqrt(np.sum(array1*array1))
```

We can also use the dot product when we have a 2D array (or matrix). When you have an vector with the same number of elements as the matrix (2D array) has columns you can right-multiply the matrix by the vector to get another vector with the same number of elements as the matrix has rows. For example this is how you compute the predicted values given a matrix of features and an array of weights.

In [ ]:
```
my_features = np.array([[1., 2.], [3., 4.], [5., 6.], [7., 8.]])
print my_features
```

In [ ]:
```
my_weights = np.array([0.4, 0.5])
print my_weights
```

In [ ]:
```
my_predictions = np.dot(my_features, my_weights) # note that the weights are on the right
print my_predictions # which has 4 elements since my_features has 4 rows
```

Similarly if you have a vector with the same number of elements as the matrix has *rows* you can left multiply them.

Similarly if you have a vector with the same number of elements as the matrix has *rows* you can left multiply them.

```
In [ ]: my_matrix = my_features
        my_array = np.array([0.3, 0.4, 0.5, 0.6])
```

```
In [ ]: print np.dot(my_array, my_matrix) # which has 2 elements because my_matrix has 2 columns
```

Multiplying Matrices

If we have two 2D arrays (matrices) matrix_1 and matrix_2 where the number of columns of matrix_1 is the same as the number of rows of matrix_2 then we can use np.dot() to perform matrix multiplication.

```
In [ ]: matrix_1 = np.array([[1., 2., 3.],[4., 5., 6.]])
        print matrix_1
```

```
In [ ]: matrix_2 = np.array([[1., 2.], [3., 4.], [5., 6.]])
        print matrix_2
```

```
In [ ]: print np.dot(matrix_1, matrix_2)
```

Quickstart tutorial¶

Prerequisites

Before reading this tutorial you should know a bit of Python. If you would like to refresh your memory, take a look at the Python tutorial.

If you wish to work the examples in this tutorial, you must also have some software installed on your computer. Please see http://scipy.org/install.html for instructions.

The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy dimensions are called *axes*. The number of axes is *rank*.

For example, the coordinates of a point in 3D space `[1, 2, 1]` is an array of rank 1, because it has one axis. That axis has a length of 3. In example pictured below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

Numpy's array class is called `ndarray`. It is also known by the alias `array`. Note that `numpy.array` is not the same as the Standard Python Library class `array.array`, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an `ndarray` object are:

ndarray.ndim
the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as *rank*.

ndarray.shape
the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, `shape` will be `(n,m)`. The length of the `shape` tuple is therefore the rank, or number of dimensions, `ndim`.

ndarray.size
the total number of elements of the array. This is equal to the product of the elements of `shape`.

ndarray.dtype
an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsize
the size in bytes of each element of the array. For example, an array of elements of type `float64` has `itemsize` 8 (=64/8), while one of type `complex32` has `itemsize` 4 (=32/8). It is equivalent to `ndarray.dtype.itemsize`.

ndarray.data
the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

An example

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

Array Creation

There are several ways to create arrays.

For example, you can create an array from a regular Python list or tuple using the `array` function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

A frequent error consists in calling `array` with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```

`array` transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function `zeros` creates an array full of zeros, the function `ones` creates an array full of ones, and the function `empty` creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is `float64`.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be specified
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                                 # uninitialized, output may vary
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

To create sequences of numbers, NumPy provides a function analogous to `range` that returns arrays instead of lists

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When `arange` is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function `linspace` that receives as an argument the number of elements that we want, instead of the step:

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots of points
>>> f = np.sin(x)
```

See also:
array, zeros, zeros_like, ones, ones_like, empty, empty_like, arange, linspace, numpy.random.rand, numpy.random.randn, fromfunction, fromfile

Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

See *below* to get more details on `reshape`.

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>> print(np.arange(10000))
[   0    1    2 ..., 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...,   97   98   99]
 [ 100  101  102 ...,  197  198  199]
 [ 200  201  202 ...,  297  298  299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using `set_printoptions`.

```
>>> np.set_printoptions(threshold='nan')
```

Basic Operations¶

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True, True, False, False], dtype=bool)
```

Unlike in many matrix languages, the product operator `*` operates elementwise in NumPy arrays. The matrix product can be performed using the `dot` function or method:

```
>>> A = np.array( [[1,1],
...             [0,1]] )
>>> B = np.array( [[2,0],
...             [3,4]] )
>>> A*B                         # elementwise product
array([[2, 0],
       [0, 4]])
>>> A.dot(B)                    # matrix product
array([[5, 4],
       [3, 4]])
>>> np.dot(A, B)                # another matrix product
array([[5, 4],
       [3, 4]])
```

Some operations, such as `+=` and `*=`, act in place to modify an existing array rather than create a new one.

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a += 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
>>> a += b                  # b is not automatically converted to integer type
Traceback (most recent call last):
  ...
TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with casting rule 'same_kind'
Traceback (most recent call last):
  ...
TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int64') with casting rule 'same_kind'
```

When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([ 1.        ,  2.57079633,  4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the `ndarray` class.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
>>> a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the `axis` parameter you can apply an operation along the specified axis of an array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Universal Functions

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called "universal functions"(`ufunc`). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([ 1.        ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
array([ 0.        ,  1.        ,  1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```

See also:
all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, *inv*, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where

Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, exclusive, set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
>>> a[ : :-1]                                 # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a:
...     print(i**(1/3.))
...
nan
1.0
nan
3.0
nan
5.0
6.0
7.0
8.0
9.0
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                       # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices`:`

```
>>> b[-1]                                  # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

The expression within brackets in `b[i]` is treated as an `i` followed by as many instances of `:` as needed to represent the remaining axes. NumPy also allows you to write this using dots as `b[i,...]`.

The **dots** (`...`) represent as many colons as needed to produce a complete indexing tuple. For example, if `x` is a rank 5 array (i.e., it has 5 axes), then

- `x[1,2,...]` is equivalent to `x[1,2,:,:,:]`,
- `x[...,3]` to `x[:,:,:,:,3]` and
- `x[4,...,5,:]` to `x[4,:,:,5,:]`.

```
>>> c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D arrays)
...                 [ 10, 12, 13]],
...                [[100,101,102],
...                 [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                                   # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                                   # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the `flat` attribute which is an iterator over all the elements of the array:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
11
12
13
20
21
22
23
30
31
32
33
40
41
42
43
```

See also:
Indexing, *Indexing* (reference), newaxis, ndenumerate, indices

Shape Manipulation

Changing the shape of an array

An array has a shape given by the number of elements along each axis:

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
```

The shape of an array can be changed with various commands:

```
>>> a.ravel() # flatten the array
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,  6.])
>>> a.shape = (6, 2)
>>> a.T
array([[ 2.,  0.,  4.,  1.,  8.,  3.],
       [ 8.,  6.,  5.,  1.,  9.,  6.]])
```

The order of the elements in the array resulting from ravel() is normally "C-style", that is, the rightmost index "changes the fastest", so the element after a[0,0] is a[0,1]. If the array is reshaped to some other shape, again the array is treated as "C-style". Numpy normally creates arrays stored in this order, so ravel() will usually not need to copy its argument, but if the array was made by taking slices of another array or created with unusual options, it may need to be copied. The functions ravel() and reshape() can also be instructed, using an optional argument, to use FORTRAN-style arrays, in which the leftmost index changes the fastest.

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

```
>>> a
array([[ 2.,  8.],
       [ 0.,  6.],
       [ 4.,  5.],
       [ 1.,  1.],
       [ 8.,  9.],
       [ 3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

```
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```

See also:
ndarray.shape, reshape, resize, ravel

Stacking together different arrays

Several arrays can be stacked together along different axes:

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

The function column_stack stacks 1D arrays as columns into a 2D array. It is equivalent to vstack only for 1D arrays:

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))   # With 2D arrays
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
>>> a = np.array([4.,2.])
>>> b = np.array([2.,8.])
>>> a[:,newaxis]  # This allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  2.],
       [ 2.,  8.]])
>>> np.vstack((a[:,newaxis],b[:,newaxis])) # The behavior of vstack is different
array([[ 4.],
       [ 2.],
       [ 2.],
       [ 8.]])
```

For arrays of with more than two dimensions, hstack stacks along their second axes, vstack stacks along their first axes, and concatenate allows for an optional arguments giving the number of the axis along which the concatenation should happen.

Note

In complex cases, r_ and c_ are useful for creating arrays by stacking numbers along one axis. They allow the use of range literals (":") :

```
>>> np.r_[1:4,0,4]
array([1, 2, 3, 0, 4])
```

When used with arrays as arguments, r_ and c_ are similar to vstack and hstack in their default behavior, but allow for an optional argument giving the number of the axis along which to concatenate.

See also:
hstack, vstack, column_stack, concatenate, c_, r_

Splitting one array into several smaller ones

Using hsplit, you can split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur:

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> a
array([[ 9.,  5.,  6.,  3.,  6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 1.,  4.,  9.,  2.,  2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])
>>> np.hsplit(a,3)   # Split a into 3
[array([[ 9.,  5.,  6.,  3.],
       [ 1.,  4.,  9.,  2.]]), array([[ 6.,  8.,  0.,  7.],
       [ 2.,  1.,  0.,  6.]]), array([[ 9.,  7.,  2.,  7.],
       [ 2.,  2.,  4.,  0.]])]
>>> np.hsplit(a,(3,4))   # Split a after the third and the fourth column
[array([[ 9.,  5.,  6.],
       [ 1.,  4.,  9.]]), array([[ 3.],
       [ 2.]]), array([[ 6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])]
```

vsplit splits along the vertical axis, and array_split allows one to specify along which axis to split.

Copies and Views

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not. This is often a source of confusion for beginners. There are three cases:

No Copy at All

Simple assignments make no copy of array objects or of their data.

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray object
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

Python passes mutable objects as references, so function calls make no copy.

```
>>> def f(x):
...     print(id(x))
...
>>> id(a)                           # id is a unique identifier of an object
148293216
>>> f(a)
148293216
```

View or Shallow Copy

Different array objects can share the same data. The `view` method creates a new array object that looks at the same data.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6                      # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0,4] = 1234                      # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

Slicing an array returns a view of it:

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Deep Copy¶

The `copy` method makes a complete copy of the array and its data.

```
>>> d = a.copy()                          # a new array object with new data is created
>>> d is a
False
>>> d.base is a                           # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Functions and Methods Overview

Here is a list of some useful NumPy functions and methods names ordered in categories. See *Routines* for the full list.

Array Creation

arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, *r*, zeros, zeros_like

Conversions

ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat

Manipulations

array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack

Questions

all, any, nonzero, where

Ordering

argmax, argmin, argsort, max, min, ptp, searchsorted, sort

Operations

choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum

Basic Statistics

cov, mean, std, var

Basic Linear Algebra

cross, dot, outer, linalg.svd, vdot

Less Basic

Broadcasting rules

Broadcasting allows universal functions to deal in a meaningful way with inputs that do not have exactly the same shape.

The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.

The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array.

After application of the broadcasting rules, the sizes of all arrays must match. More details can be found in *Broadcasting*.

Fancy indexing and index tricks

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can be indexed by arrays of integers and arrays of booleans.

Indexing with Arrays of Indices

```
>>> a = np.arange(12)**2                       # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )              # an array of indices
>>> a[i]                                       # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
>>>
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )      # a bidimensional array of indices
>>> a[j]                                       # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

When the indexed array a is multidimensional, a single array of indices refers to the first dimension of a. The following example shows this behavior by converting an image of labels into a color image using a palette.

```
>>> palette = np.array( [ [0,0,0],                # black
...                       [255,0,0],              # red
...                       [0,255,0],              # green
...                       [0,0,255],              # blue
...                       [255,255,255] ] )       # white
>>> image = np.array( [ [ 0, 1, 2, 0 ],           # each value corresponds to a color in the palette
...                     [ 0, 3, 4, 0 ]  ] )
>>> palette[image]                            # the (2,4,3) color image
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],
       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array( [ [0,1],                        # indices for the first dim of a
...                 [1,2] ] )
>>> j = np.array( [ [2,1],                        # indices for the second dim
...                 [3,3] ] )
>>>
>>> a[i,j]                                    # i and j must have equal shape
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i,2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:,j]                                    # i.e., a[ : , j]
array([[[ 2,  1],
        [ 3,  3]],
       [[ 6,  5],
        [ 7,  7]],
       [[10,  9],
        [11, 11]]])
```

Naturally, we can put i and j in a sequence (say a list) and then do the indexing with the list.

```
>>> l = [i,j]
>>> a[l]                                      # equivalent to a[i,j]
array([[ 2,  5],
       [ 7, 11]])
```

However, we can not do this by putting `i` and `j` into an array, because this array will be interpreted as indexing the first dimension of a.

```
>>> s = np.array( [i,j] )
>>> a[s]                                     # not what we want
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index (3) out of range (0<=index<=2) in dimension 0
>>>
>>> a[tuple(s)]                              # same as a[i,j]
array([[ 2,  5],
       [ 7, 11]])
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: index (3) out of range (0<=index<=2) in dimension 0
```

Another common use of indexing with arrays is the search of the maximum value of time-dependent series :

```
>>> time = np.linspace(20, 145, 5)                 # time scale
>>> data = np.sin(np.arange(20)).reshape(5,4)      # 4 time-dependent series
>>> time
array([  20.  ,   51.25,   82.5 ,  113.75,  145.  ])
>>> data
array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])
>>>
>>> ind = data.argmax(axis=0)                  # index of the maxima for each series
>>> ind
array([2, 0, 3, 1])
>>>
>>> time_max = time[ ind]                      # times corresponding to the maxima
>>>
>>> data_max = data[ind, xrange(data.shape[1])] # => data[ind[0],0], data[ind[1],1]...
>>>
>>> time_max
array([  82.5 ,   20.  ,  113.75,   51.25])
>>> data_max
array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])
>>>
>>> np.all(data_max == data.max(axis=0))
True
```

You can also use indexing with arrays as a target to assign to:

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[[1,3,4]] = 0
>>> a
array([0, 0, 2, 0, 0])
```

However, when the list of indices contains repetitions, the assignment is done several times, leaving behind the last value:

```
>>> a = np.arange(5)
>>> a[[0,0,2]]=[1,2,3]
>>> a
array([2, 1, 3, 3, 4])
```

This is reasonable enough, but watch out if you want to use Python's `+=` construct, as it may not do what you expect:

```
>>> a = np.arange(5)
>>> a[[0,0,2]]+=1
>>> a
array([1, 1, 3, 3, 4])
```

Even though 0 occurs twice in the list of indices, the 0th element is only incremented once. This is because Python requires "a+=1" to be equivalent to "a=a+1".

Indexing with Boolean Arrays

When we index arrays with arrays of (integer) indices we are providing the list of indices to pick. With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don't.

The most natural way one can think of for boolean indexing is to use boolean arrays that have *the same shape* as the original array:

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                        # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
>>> a[b]                                     # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

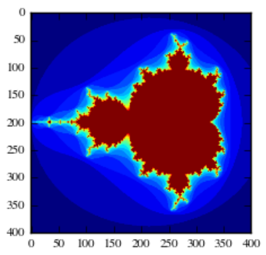
This property can be very useful in assignments:

```
>>> a[b] = 0                                 # All elements of 'a' higher than 4 become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

You can look at the following example to see how to use boolean indexing to generate an image of the Mandelbrot set:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def mandelbrot( h,w, maxit=20 ):
...     """Returns an image of the Mandelbrot fractal of size (h,w)."""
...     y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
...     c = x+y*1j
...     z = c
...     divtime = maxit + np.zeros(z.shape, dtype=int)
...
...     for i in range(maxit):
...         z = z**2 + c
...         diverge = z*np.conj(z) > 2**2            # who is diverging
...         div_now = diverge & (divtime==maxit)  # who is diverging now
...         divtime[div_now] = i                  # note when
...         z[diverge] = 2                        # avoid diverging too much
...
...     return divtime
>>> plt.imshow(mandelbrot(400,400))
>>> plt.show()
```

(Source code, png, pdf)



The second way of indexing with booleans is more similar to integer indexing; for each dimension of the array we give a 1D boolean array selecting the slices we want.

```
>>> a = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True])             # first dim selection
>>> b2 = np.array([True,False,True,False])       # second dim selection
>>>
>>> a[b1,:]                                   # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                                     # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[:,b2]                                   # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                                  # a weird thing to do
array([ 4, 10])
```

Note that the length of the 1D boolean array must coincide with the length of the dimension (or axis) you want to slice. In the previous example, b1 is a 1-rank array with length 3 (the number of *rows* in a), and b2 (of length 4) is suitable to index the 2nd rank (columns) of a.

The ix_() function

The ix_ function can be used to combine different vectors so as to obtain the result for each n-uplet. For example, if you want to compute all the a+b*c for all the triplets taken from each of the vectors a, b and c:

```
>>> a = np.array([2,3,4,5])
>>> b = np.array([8,5,4])
>>> c = np.array([5,4,6,8,3])
>>> ax,bx,cx = np.ix_(a,b,c)
>>> ax
array([[[2]],
       [[3]],
       [[4]],
       [[5]]])
>>> bx
array([[[8],
        [5],
        [4]]])
>>> cx
array([[[5, 4, 6, 8, 3]]])
>>> ax.shape, bx.shape, cx.shape
((4, 1, 1), (1, 3, 1), (1, 1, 5))
>>> result = ax+bx*cx
>>> result
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],
       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],
       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],
       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
>>> result[3,2,4]
17
>>> a[3]+b[2]*c[4]
17
```

You could also implement the reduce as follows:

```
>>> def ufunc_reduce(ufct, *vectors):
...    vs = np.ix_(*vectors)
...    r = ufct.identity
...    for v in vs:
...        r = ufct(r,v)
...    return r
```

and then use it as:

```
>>> ufunc_reduce(np.add,a,b,c)
array([[[15, 14, 16, 18, 13],
        [12, 11, 13, 15, 10],
        [11, 10, 12, 14,  9]],
       [[16, 15, 17, 19, 14],
        [13, 12, 14, 16, 11],
        [12, 11, 13, 15, 10]],
       [[17, 16, 18, 20, 15],
        [14, 13, 15, 17, 12],
        [13, 12, 14, 16, 11]],
       [[18, 17, 19, 21, 16],
        [15, 14, 16, 18, 13],
        [14, 13, 15, 17, 12]]])
```

The advantage of this version of reduce compared to the normal ufunc.reduce is that it makes use of the Broadcasting Rules in order to avoid creating an argument array the size of the output times the number of vectors.

Indexing with strings

See RecordArrays.

Linear Algebra¶

Work in progress. Basic linear algebra to be included here.

Simple Array Operations

See linalg.py in numpy folder for more.

```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[ 1.  2.]
 [ 3.  4.]]
```

```
>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])
```

```
>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

```
>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])
```

```
>>> np.dot (j, j) # matrix product
array([[-1.,  0.],
       [ 0., -1.]])
```

```
>>> np.trace(u)  # trace
2.0
```

```
>>> y = np.array([[5.], [7.]])
>>> np.linalg.solve(a, y)
array([[-3.],
       [ 4.]])
```

```
>>> np.linalg.eig(j)
(array([ 0.+1.j,  0.-1.j]), array([[ 0.70710678+0.j        ,  0.70710678-0.j        ],
       [ 0.00000000-0.70710678j,  0.00000000+0.70710678j]]))
```

```
Parameters:
    square matrix
Returns
    The eigenvalues, each repeated according to its multiplicity.
    The normalized (unit "length") eigenvectors, such that the
    column ``v[:,i]`` is the eigenvector corresponding to the
    eigenvalue ``w[i]`` .
```

Tricks and Tips

Here we give a list of short and useful tips.

"Automatic" Reshaping

To change the dimensions of an array, you can omit one of the sizes which will then be deduced automatically:

```
>>> a = np.arange(30)
>>> a.shape = 2,-1,3  # -1 means "whatever is needed"
>>> a.shape
(2, 5, 3)
>>> a
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11],
        [12, 13, 14]],
       [[15, 16, 17],
        [18, 19, 20],
        [21, 22, 23],
        [24, 25, 26],
        [27, 28, 29]]])
```

Vector Stacking¶

How do we construct a 2D array from a list of equally-sized row vectors? In MATLAB this is quite easy: if `x` and `y` are two vectors of the same length you only need do `m=[x;y]`. In NumPy this works via the functions `column_stack`, `dstack`, `hstack` and `vstack`, depending on the dimension in which the stacking is to be done. For example:

```
x = np.arange(0,10,2)                     # x=([0,2,4,6,8])
y = np.arange(5)                          # y=([0,1,2,3,4])
m = np.vstack([x,y])                      # m=([[0,2,4,6,8],
                                          #      [0,1,2,3,4]])
xy = np.hstack([x,y])                     # xy =([0,2,4,6,8,0,1,2,3,4])
```

The logic behind those functions in more than two dimensions can be strange.

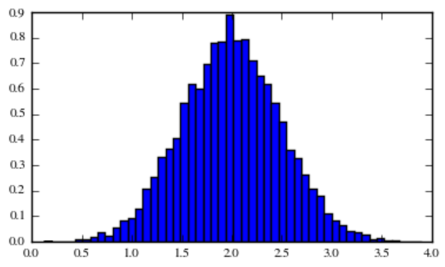
See also:
Numpy for Matlab users

Histograms

The NumPy `histogram` function applied to an array returns a pair of vectors: the histogram of the array and the vector of bins. Beware: `matplotlib` also has a function to build histograms (called `hist`, as in Matlab) that differs from the one in NumPy. The main difference is that `pylab.hist` plots the histogram automatically, while `numpy.histogram` only generates the data.

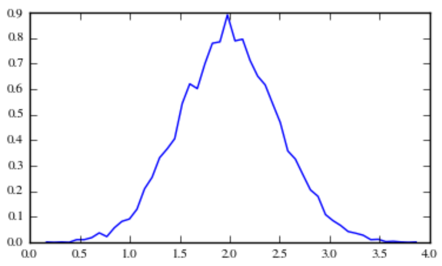
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> # Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
>>> mu, sigma = 2, 0.5
>>> v = np.random.normal(mu,sigma,10000)
>>> # Plot a normalized histogram with 50 bins
>>> plt.hist(v, bins=50, normed=1)       # matplotlib version (plot)
>>> plt.show()
```

(Source code, png, pdf)



```
>>> # Compute the histogram with numpy and then plot it
>>> (n, bins) = np.histogram(v, bins=50, normed=True)  # NumPy version (no plot)
>>> plt.plot(.5*(bins[1:]+bins[:-1]), n)
>>> plt.show()
```

(png, pdf)



https://docs.scipy.org/doc/numpy-dev/user/quickstart.html#automatic-reshaping

Regression Week 2: Multiple Linear Regression Quiz 2

Estimating Multiple Regression Coefficients (Gradient Descent)

In the first notebook we explored multiple regression using GraphLab Create. Now we will use SFrames along with numpy to solve for the regression weights with gradient descent.

In this notebook we will cover estimating multiple regression weights via gradient descent. You will:

- Add a constant column of 1's to a SFrame (or otherwise) to account for the intercept
- Convert an SFrame into a numpy array
- Write a predict_output() function using numpy
- Write a numpy function to compute the derivative of the regression weights with respect to a single feature
- Write gradient descent function to compute the regression weights given an initial weight vector, step size and tolerance.
- Use the gradient descent function to estimate regression weights for multiple features

If you are doing the assignment with IPython Notebook

An IPython Notebook has been provided below to you for this quiz. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

What you need to download

If you are using GraphLab Create:

- Download the King County House Sales data In SFrame format: kc_house_data.gl.zip
- Download the companion IPython Notebook: week-2-multiple-regression-assignment-2-blank.ipynb
- Save both of these files in the same directory (where you are calling IPython notebook from) and unzip the data file.

Upgrade GraphLab Create. If you are using GraphLab Create and already have it installed, please make sure you upgrade to the latest version before doing this assignment. The simplest way to do this is to:

1. Open the Dato Launcher.
2. Click on 'TERMINAL'.
3. On the terminal window, type:

```
pip install --upgrade graphlab-create
```

If you are not using GraphLab Create:

- Download the King County House Sales data csv file: kc_house_data.csv
- Download the King County House Sales training data csv file: kc_house_train_data.csv
- Download the King County House Sales testing data csv file: kc_house_test_data.csv
- IMPORTANT: use the following types for columns when importing the csv files. Otherwise, they may not be imported correctly: [str, str, float, float, float, float, int, str, int, int, int, int, int, int, int, int, str, float, float, float, float]. If your tool of choice requires a dictionary of types for importing csv files (e.g. Pandas), use:

```
dtype_dict = {'bathrooms':float, 'waterfront':int, 'sqft_above':int, 'sqft_living15':float, 'grade':int, 'yr_renovated':int, 'price':float, 'bedrooms':float, 'zipcode':str, 'long':float, 'sqft_lot15':float, 'sqft_living':float, 'floors':str, 'condition':int, 'lat':float, 'date':str, 'sqft_basement':int, 'yr_built':int, 'id':str, 'sqft_lot':int, 'view':int}
```

Useful resources

You may need to install the software tools or use the free Amazon EC2 machine. Instructions for both options are provided in the reading for Module 1.

If you are following the IPython Notebook and/or are new to numpy then you might find the following tutorial helpful: numpy-tutorial.ipynb

If instead you are using other tools to do your homework

You are welcome, however, to write your own code and use any other libraries, like Pandas or R, to help you in the process. If you would like to take this path, follow the instructions below.

1. If you're using SFrames, import graphlab and load in the house data (this is the graphlab command you can also download the csv). e.g. in python with SFrames:

```
sales = graphlab.SFrame('kc_house_data.gl/')
```

2. If you're using python: to do the matrix operations required to perform a gradient descent we will be using the popular python library 'numpy' which is a computational library specialized for operations on arrays. For students unfamiliar with numpy we have created a numpy tutorial (see useful resources). It is common to import numpy under the name 'np' for short, to do this execute:

```
import numpy as np
```

3. Next write a function that takes a data set, a list of features (e.g. [‘sqft_living’, ‘bedrooms’]), to be used as inputs, and a name of the output (e.g. ‘price’). This function should return a features_matrix (2D array) consisting of first a column of ones followed by columns containing the values of the input features in the data set in the same order as the input list. It should also return an output_array which is an array of the values of the output in the data set (e.g. ‘price’). e.g. if you’re using SFrames and numpy you can complete the following function:

```
def get_numpy_data(data_sframe, features, output):

    data_sframe['constant'] = 1 # add a constant column to an SFrame

    # prepend variable 'constant' to the features list

    features = ['constant'] + features

    # select the columns of data_SFrame given by the ‘features’ list into the SFrame ‘features_sframe’



    # this will convert the features_sframe into a numpy matrix with GraphLab Create >= 1.7!!

    features_matrix = features_sframe.to_numpy()

    # assign the column of data_sframe associated with the target to the variable ‘output_sarray’



    # this will convert the SArray into a numpy array:

    output_array = output_sarray.to_numpy() # GraphLab Create>= 1.7!!

    return(features_matrix, output_array)
```

In order for the .to_numpy() command to work ensure that you have GraphLab Create version 1.7 or greater.

4. If the features matrix (including a column of 1s for the constant) is stored as a 2D array (or matrix) and the regression weights are stored as a 1D array then the predicted output is just the dot product between the features matrix and the weights (with the weights on the right). Write a function ‘predict_output’ which accepts a 2D array ‘feature_matrix’ and a 1D array ‘weights’ and returns a 1D array ‘predictions’. e.g. in python:

```
def predict_outcome(feature_matrix, weights):

    [your code here]

    return(predictions)
```

5. If we have a the values of a single input feature in an array ‘feature’ and the prediction ‘errors’ (predictions - output) then the derivative of the regression cost function with respect to the weight of ‘feature’ is just twice the dot product between ‘feature’ and ‘errors’. Write a function that accepts a ‘feature’ array and ‘error’ array and returns the ‘derivative’ (a single number). e.g. in python:

```
def feature_derivative(errors, feature):
    [your code here]

    return(derivative)
```

6. Now we will use our predict_output and feature_derivative to write a gradient descent function. Although we can compute the derivative for all the features simultaneously (the gradient) we will explicitly loop over the features individually for simplicity. Write a gradient descent function that does the following:

- Accepts a numpy feature_matrix 2D array, a 1D output array, an array of initial weights, a step size and a convergence tolerance.
- While not converged updates each feature weight by subtracting the step size times the derivative for that feature given the current weights
- At each step computes the magnitude/length of the gradient (square root of the sum of squared components)
- When the magnitude of the gradient is smaller than the input tolerance returns the final weight vector.

e.g. if you're using SFrames and numpy you can complete the following function:

```
def regression_gradient_descent(feature_matrix, output, initial_weights, step_size, tolerance):

    converged = False

    weights = np.array(initial_weights)

    while not converged:

        # compute the predictions based on feature_matrix and weights:

        # compute the errors as predictions - output:


        gradient_sum_squares = 0 # initialize the gradient

        # while not converged, update each weight individually:

        for i in range(len(weights)):

            # Recall that feature_matrix[:, i] is the feature column associated with weights[i]

            # compute the derivative for weight[i]:


            # add the squared derivative to the gradient magnitude


            # update the weight based on step size and derivative:


        gradient_magnitude = sqrt(gradient_sum_squares)

        if gradient_magnitude < tolerance:

            converged = True
        return(weights)
```

7. Now split the sales data into training and test data. Like previous notebooks it's important to use the same seed.

```
train_data,test_data = sales.random_split(.8,seed=0)
```

For those students not using SFrames please download the training and testing data csv files.

8. Now we will run the regression_gradient_descent function on some actual data. In particular we will use the gradient descent to estimate the model from Week 1 using just an intercept and slope. Use the following parameters:

- features: 'sqft_living'
- output: 'price'
- initial weights: -47000, 1 (intercept, sqft_living respectively)
- step_size = 7e-12
- tolerance = 2.5e7

e.g. in python with numpy and SFrames:

```
simple_features = ['sqft_living']

my_output= 'price'

(simple_feature_matrix, output) = get_numpy_data(train_data, simple_features, my_output)

initial_weights = np.array([-47000., 1.])

step_size = 7e-12

tolerance = 2.5e7
```

Use these parameters to estimate the slope and intercept for predicting prices based only on 'sqft_living'.

e.g. using python:

```
simple_weights = regression_gradient_descent(simple_feature_matrix, output,initial_weights, step_size,
tolerance)
```

9. Quiz Question: What is the value of the weight for sqft_living -- the second element of 'simple_weights' (rounded to 1 decimal place)?

10. Now build a corresponding 'test_simple_feature_matrix' and 'test_output' using test_data. Using 'test_simple_feature_matrix' and 'simple_weights' compute the predicted house prices on all the test data.

11. Quiz Question: What is the predicted price for the 1st house in the Test data set for model 1 (round to nearest dollar)?

12. Now compute RSS on all test data for this model. Record the value and store it for later

13. Now we will use the gradient descent to fit a model with more than 1 predictor variable (and an intercept). Use the following parameters:

- model features = ‘sqft_living’, ‘sqft_living_15’
- output = ‘price’
- initial weights = [-100000, 1, 1] (intercept, sqft_living, and sqft_living_15 respectively)
- step size = 4e-12
- tolerance = 1e9

e.g. in python with numpy and SFrames:

```
model_features = ['sqft_living', 'sqft_living15']

my_output = 'price'

(feature_matrix, output) = get_numpy_data(train_data, model_features,my_output)

initial_weights = np.array([-100000., 1., 1.])

step_size = 4e-12

tolerance = 1e9
```

Note that sqft_living_15 is the average square feet of the nearest 15 neighbouring houses.

Run gradient descent on a model with ‘sqft_living’ and ‘sqft_living_15’ as well as an intercept with the above parameters. Save the resulting regression weights.

14. Use the regression weights from this second model (using sqft_living and sqft_living_15) and predict the outcome of all the house prices on the TEST data.

15. Quiz Question: What is the predicted price for the 1st house in the TEST data set for model 2 (round to nearest dollar)?

16. What is the actual price for the 1st house in the Test data set?

17. Quiz Question: Which estimate was closer to the true price for the 1st house on the TEST data set, model 1 or model 2?

18. Now compute RSS on all test data for the second model. Record the value and store it for later.

19. Quiz Question: Which model (1 or 2) has lowest RSS on all of the TEST data?