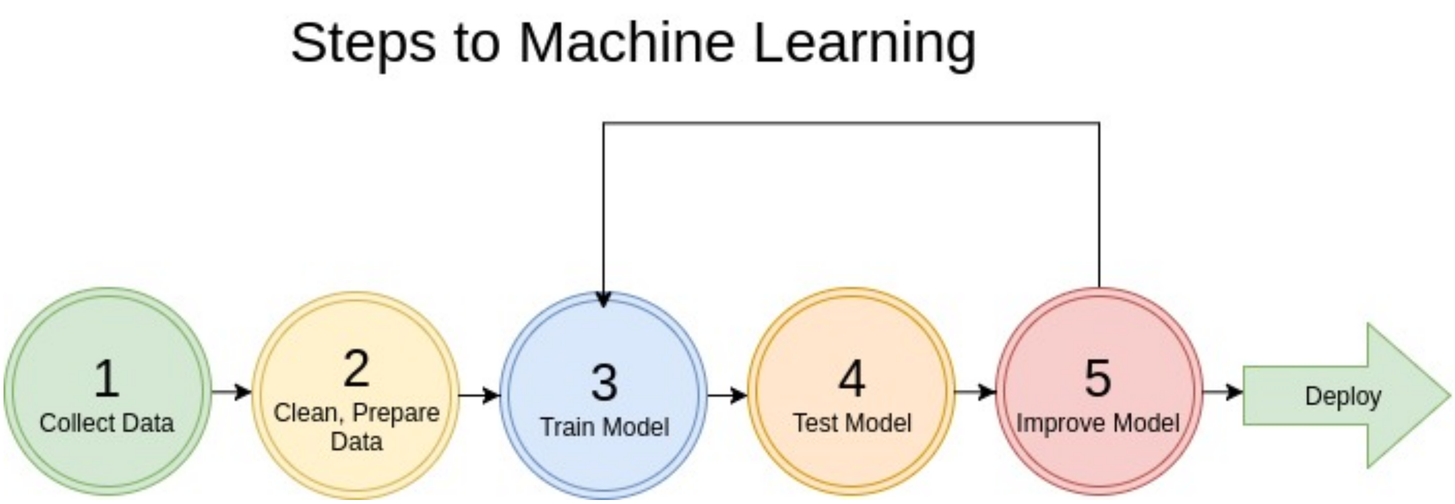
Introduction

Welcome to **CS188 - Data Science Fundamentals!** We plan on having you go through some grueling training so you can start crunching data out there... in today's day and age "data is the new oil" or perhaps "snake oil" nonetheless, there's a lot of it, each with different purity (so pure that perhaps you could feed off it for a life time) or dirty which then at that point you can either decide to dump it or try to weed out something useful (that's where they need you...)

In this project you will work through an example project end to end.

Here are the main steps:

1. Get the data
2. Visualize the data for insights
3. Preprocess the data for your machine learning algorithm
4. Select a model and train
5. Does it meet the requirements? Fine tune the model



Working with Real Data

It is best to experiment with real-data as opposed to aritifical datasets.

There are many different open datasets depending on the type of problems you might be interested in!

Here are a few data repositories you could check out:

- UCI Datasets (http://archive.ics.uci.edu/ml/)
- Kaggle Datasets (kaggle.com)
- AWS Datasets (https://registry.opendata.aws)

Below we will run through an California Housing example collected from the 1990's.

Setup

In [1]:
```
import sys
assert sys.version_info >= (3, 5) # python>=3.5
import sklearn
assert sklearn.__version__ >= "0.20" # sklearn >= 0.20

import numpy as np #numerical package in python
import os
%matplotlib inline
import matplotlib.pyplot as plt #plotting package

# to make this notebook's output identical at every run
np.random.seed(42)

#matplotlib magic for inline figures
%matplotlib inline
import matplotlib # plotting library
import matplotlib.pyplot as plt

# Where to save the figures
ROOT_DIR = "."
IMAGES_PATH = os.path.join(ROOT_DIR, "images")
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_name, tight_layout=True, fig_extension="png", resolut
ion=300):
    '''
        plt.savefig wrapper. refer to
        https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.sav
efig.html
    '''
    path = os.path.join(IMAGES_PATH, fig_name + "." + fig_extension)
    print("Saving figure", fig_name)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

In [2]:
```
import os
import tarfile
import urllib
DATASET_PATH = os.path.join("datasets", "housing")
```

Intro to Data Exploration Using Pandas

In this section we will load the dataset, and visualize different features using different types of plots.

Packages we will use:

- **Pandas (https://pandas.pydata.org):** is a fast, flexibile and expressive data structure widely used for tabular and multidimensional datasets.
- **Matplotlib (https://matplotlib.org)**: is a 2d python plotting library which you can use to create quality figures (you can plot almost anything if you're willing to code it out!)
  - other plotting libraries:seaborn (https://seaborn.pydata.org), ggplot2 (https://ggplot2.tidyverse.org)

In [3]:

```
import pandas as pd

def load_housing_data(housing_path):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

In [4]:

```
housing = load_housing_data(DATASET_PATH) # we load the pandas datafra
me
housing.head() # show the first few elements of the dataframe
               # typically this is the first thing you do
               # to see how the dataframe looks like
```

Out[4]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

A dataset may have different types of features

- real valued
- Discrete (integers)
- categorical (strings)

The two categorical features are essentialy the same as you can always map a categorical string/character to an integer.

In the dataset example, all our features are real valued floats, except ocean proximity which is categorical.

In [5]:
```
# to see a concise summary of data types, null values, and counts
# use the info() method on the dataframe
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

In [6]:
```
# you can access individual columns similarly
# to accessing elements in a python dict
housing["ocean_proximity"].head() # added head() to avoid printing man
y columns..
```

Out[6]:
```
0    NEAR BAY
1    NEAR BAY
2    NEAR BAY
3    NEAR BAY
4    NEAR BAY
Name: ocean_proximity, dtype: object
```

In [7]:
```
# to access a particular row we can use iloc
housing.iloc[1]
```

Out[7]:
```
longitude              -122.22
latitude                 37.86
housing_median_age          21
total_rooms               7099
total_bedrooms            1106
population                2401
households                1138
median_income           8.3014
median_house_value      358500
ocean_proximity       NEAR BAY
Name: 1, dtype: object
```

In [8]:
```
# one other function that might be useful is
# value_counts(), which counts the number of occurences
# for categorical features
housing["ocean_proximity"].value_counts()
```

Out[8]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

In [9]:
```
# The describe function compiles your typical statistics for each
# column
housing.describe()
```

Out[9]:

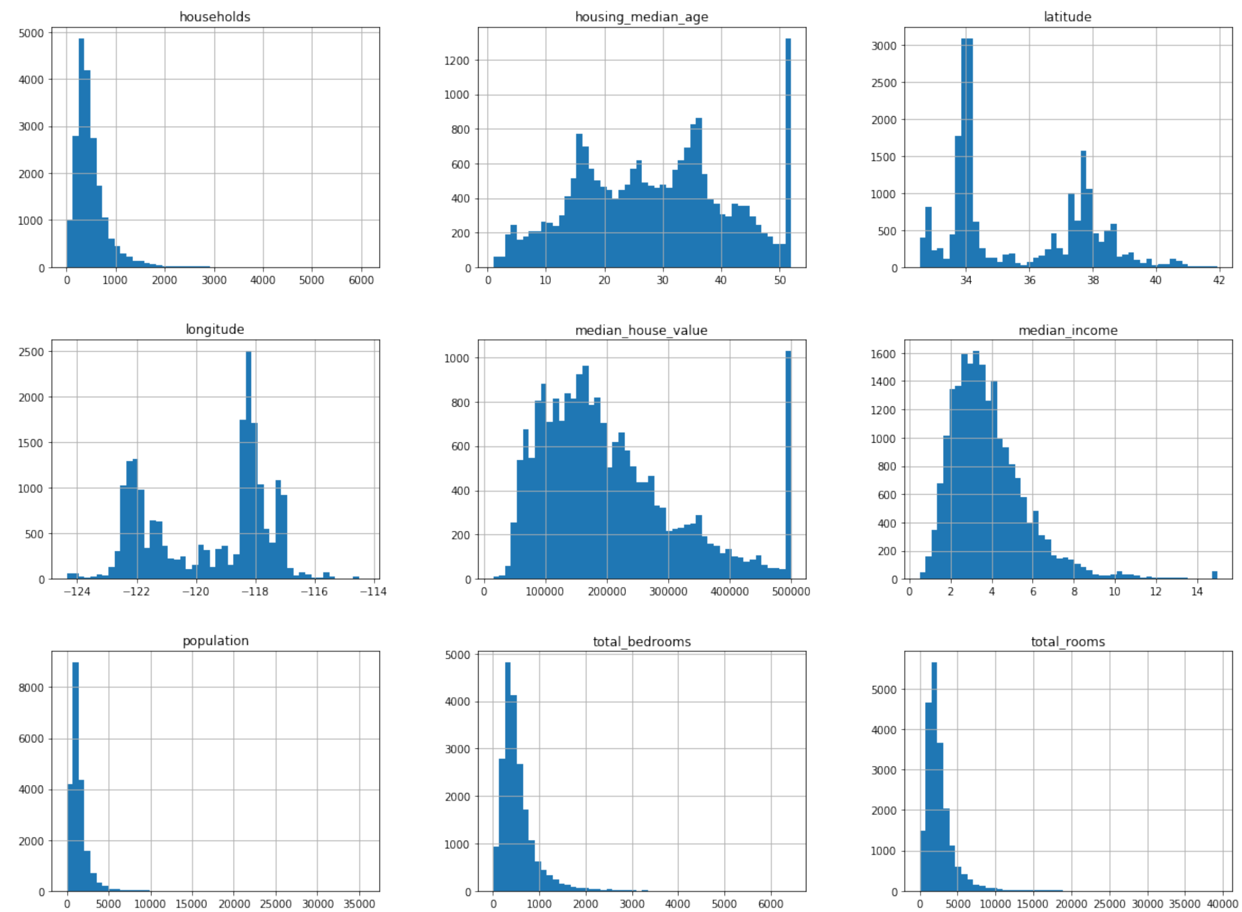
	longitude	latitude	housing_median_age	total_rooms	total_bedroo
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.00000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

If you want to learn about different ways of accessing elements or other functions it's useful to check out the getting started section [here (https://pandas.pydata.org/pandas-docs/stable/getting_started/index.html)](https://pandas.pydata.org/pandas-docs/stable/getting_started/index.html)

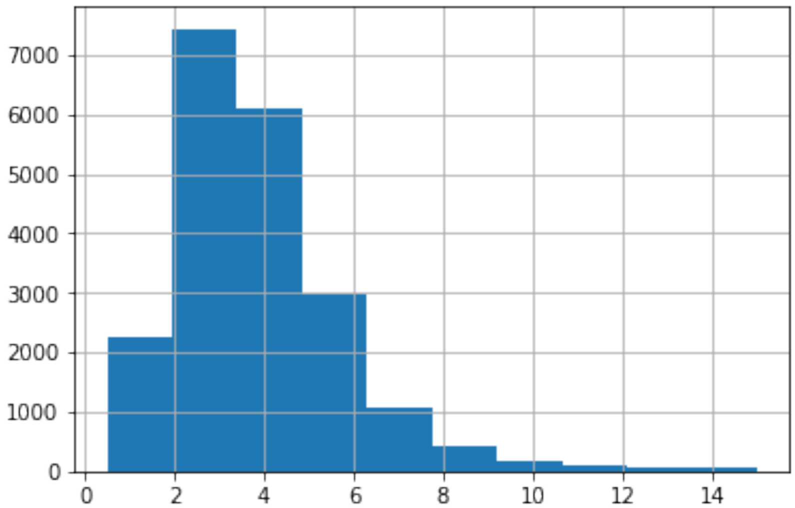
Let's start visualizing the dataset

In [10]:

```
# We can draw a histogram for each of the dataframes features
# using the hist function
housing.hist(bins=50, figsize=(20,15))
# save_fig("attribute_histogram_plots")
plt.show() # pandas internally uses matplotlib, and to display all the
figures
           # the show() function must be called
```



In [11]:
```
# if you want to have a histogram on an individual feature:
housing["median_income"].hist()
plt.show()
```



We can convert a floating point feature to a categorical feature by binning or by defining a set of intervals.

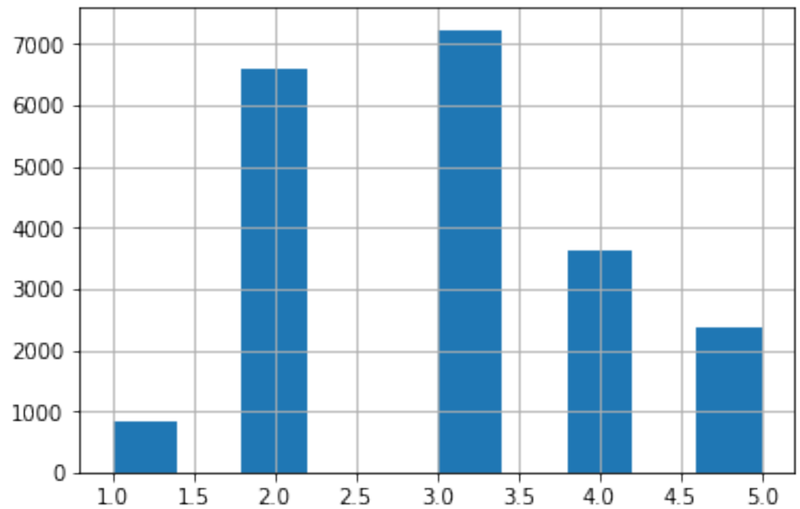
For example, to bin the households based on median_income we can use the pd.cut function

In [12]:
```
# assign each bin a categorical value [1, 2, 3, 4, 5] in this case.
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])

housing["income_cat"].value_counts()
```

Out[12]:
```
3    7236
2    6581
4    3639
5    2362
1     822
Name: income_cat, dtype: int64
```

In [13]:
```
housing["income_cat"].hist()
```

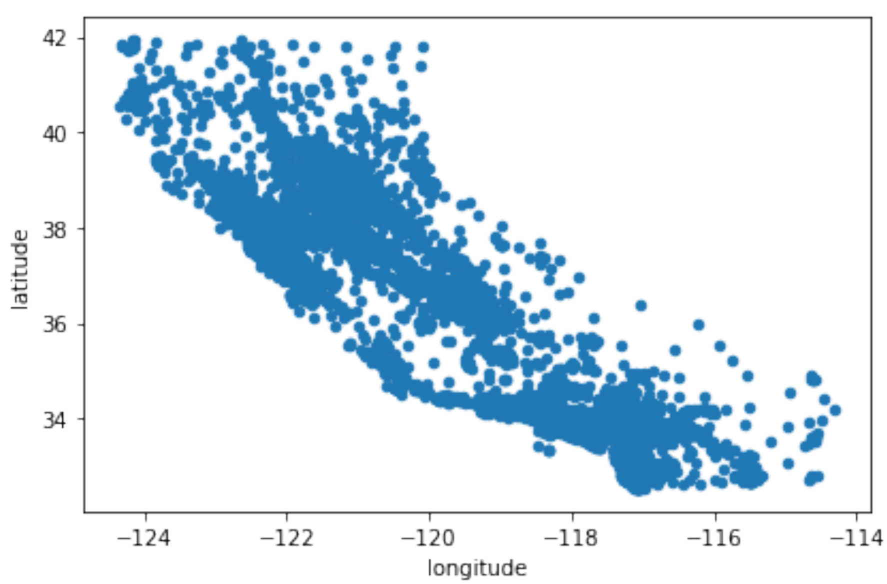
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x11b345dd8>



Next let's visualize the household incomes based on latitude & longitude coordinates

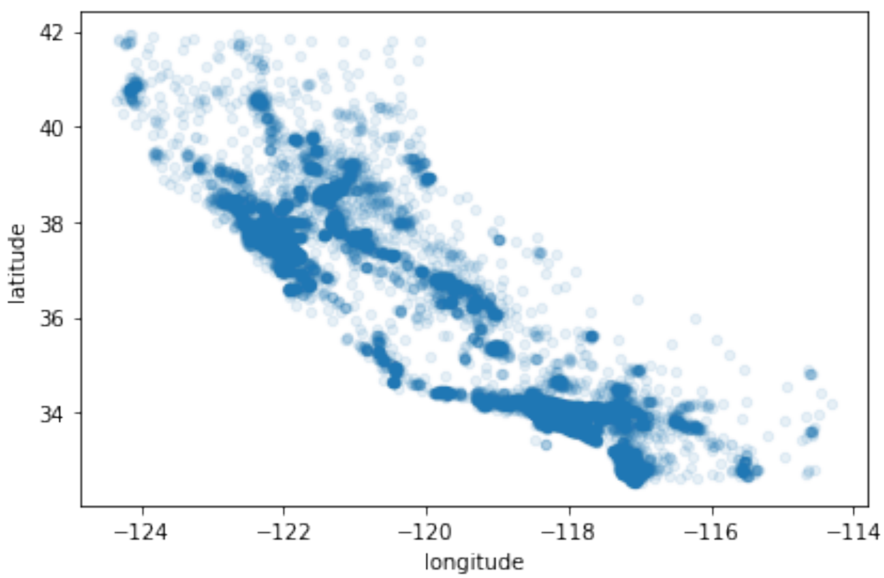
In [14]:
```
## here's a not so interestting way plotting it
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

Saving figure bad_visualization_plot



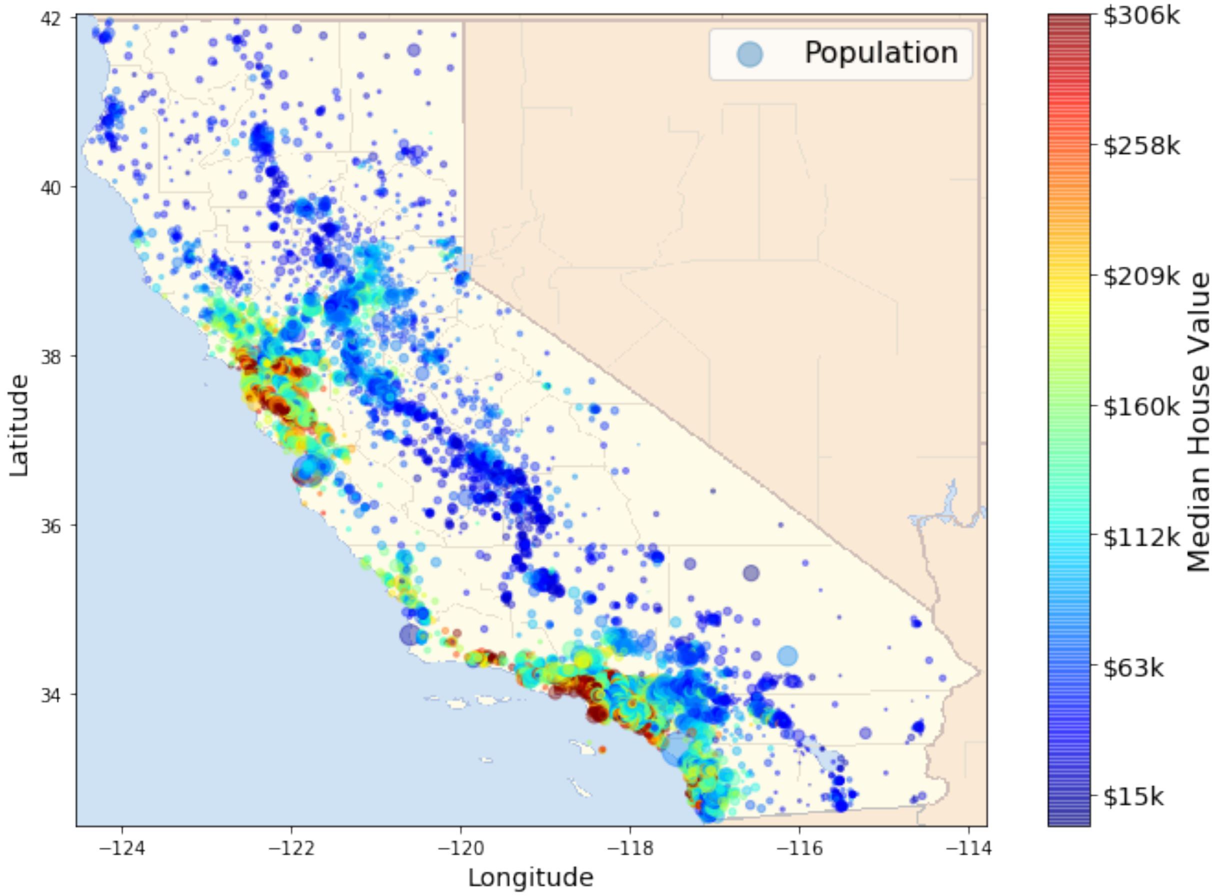
In [15]:
```
# we can make it look a bit nicer by using the alpha parameter,
# it simply plots less dense areas lighter.
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
save_fig("better_visualization_plot")
```

```
Saving figure better_visualization_plot
```



In [16]:
```
# A more interesting plot is to color code (heatmap) the dots
# based on income. The code below achieves this

# load an image of california
images_path = os.path.join('./', "images")
os.makedirs(images_path, exist_ok=True)
filename = "california.png"

import matplotlib.image as mpimg
california_img=mpimg.imread(os.path.join(images_path, filename))
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize
=(10,7),
                       s=housing['population']/100, label="Population"
,
                       c="median_house_value", cmap=plt.get_cmap("jet"
),
                       colorbar=False, alpha=0.4,
                      )
# overlay the califronia map on the plotted scatter plot
# note: plt.imshow still refers to the most recent figure
# that hasn't been plotted yet.
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], al
pha=0.5,
           cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

# setting up heatmap colors based on median_house_value feature
prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cb = plt.colorbar()
cb.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values], f
ontsize=14)
cb.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
save_fig("california_housing_prices_plot")
plt.show()
```

```
Saving figure california_housing_prices_plot
```



Not suprisingly, the most expensive houses are concentrated around the San Francisco/Los Angeles areas.

Up until now we have only visualized feature histograms and basic statistics.

When developing machine learning models the predictiveness of a feature for a particular target of intrest is what's important.

It may be that only a few features are useful for the target at hand, or features may need to be augmented by applying certain transfomrations.

None the less we can explore this using correlation matrices.

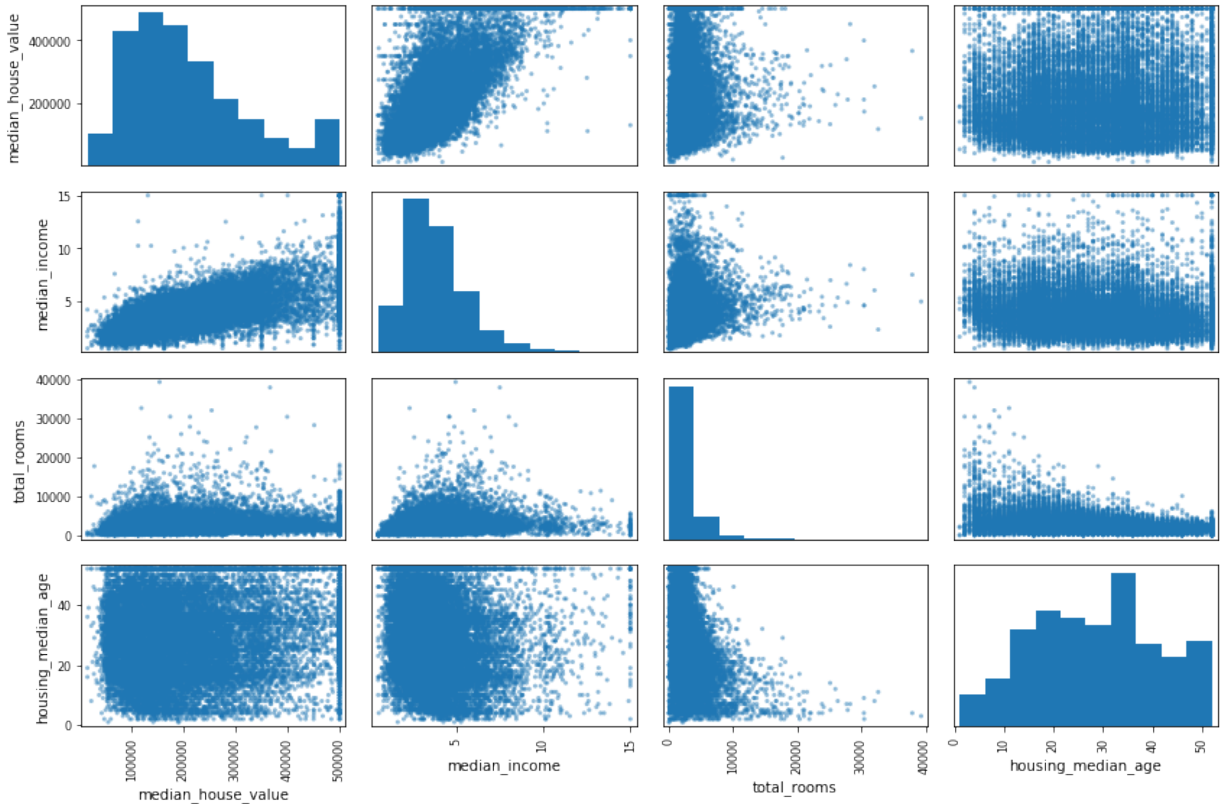
In [17]:
```
corr_matrix = housing.corr()
```

In [18]:
```
# for example if the target is "median_house_value", most correlated f
eatures can be sorted
# which happens to be "median_income". This also intuitively makes sen
se.
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[18]:
```
median_house_value    1.000000
median_income         0.688075
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049686
population           -0.024650
longitude            -0.045967
latitude             -0.144160
Name: median_house_value, dtype: float64
```

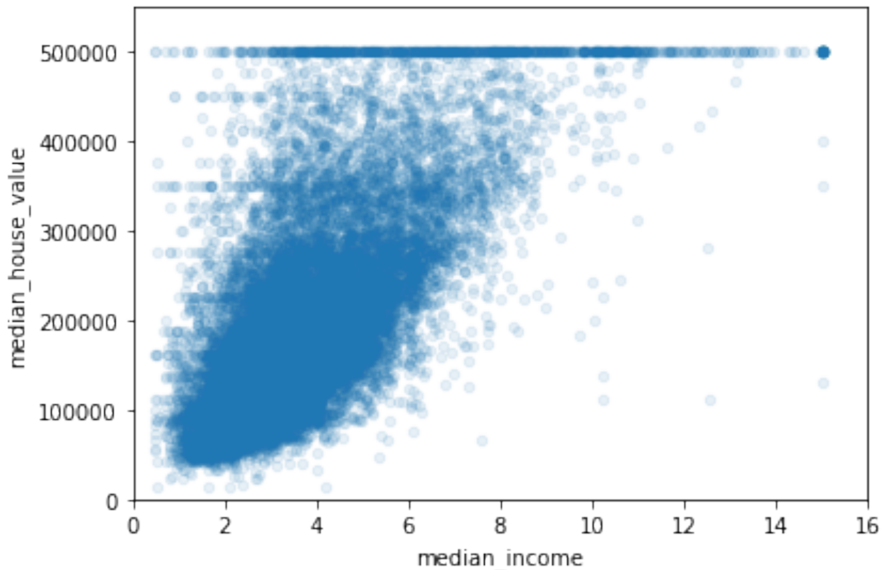
In [19]:
```
# the correlation matrix for different attributes/features can also be
plotted
# some features may show a positive correlation/negative correlation o
r
# it may turn out to be completely random!
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")
```

```
Saving figure scatter_matrix_plot
```



In [20]:
```
# median income vs median house vlue plot plot 2 in the first row of t
op figure
housing.plot(kind="scatter", x="median_income", y="median_house_value"
,
             alpha=0.1)
plt.axis([0, 16, 0, 550000])
save_fig("income_vs_house_value_scatterplot")
```

```
Saving figure income_vs_house_value_scatterplot
```



Augmenting Features

New features can be created by combining different columns from our data set.

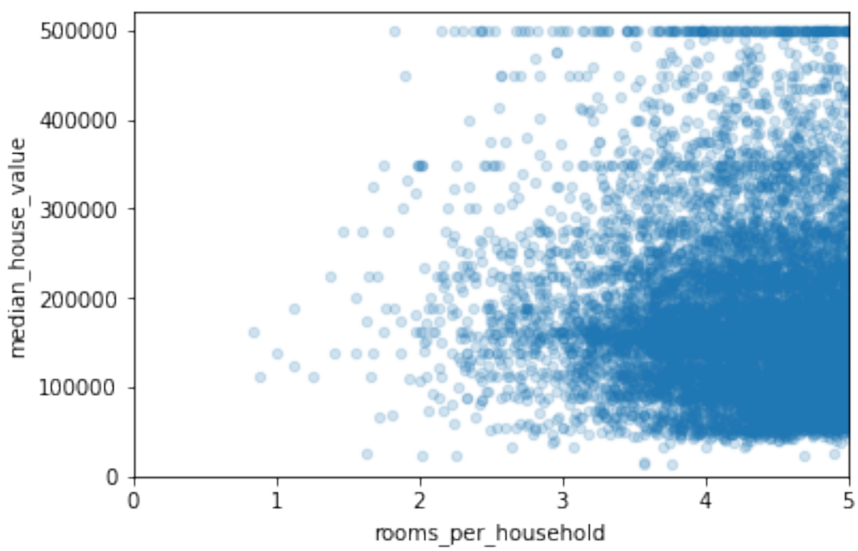
- rooms_per_household = total_rooms / households
- bedrooms_per_room = total_bedrooms / total_rooms
- etc.

In [21]:
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["house
holds"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["tota
l_rooms"]
housing["population_per_household"]=housing["population"]/housing["hou
seholds"]
```

In [22]:
```
# obtain new correlations
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[22]:
```
median_house_value          1.000000
median_income               0.688075
rooms_per_household         0.151948
total_rooms                 0.134153
housing_median_age          0.105623
households                  0.065843
total_bedrooms              0.049686
population_per_household   -0.023737
population                 -0.024650
longitude                  -0.045967
latitude                   -0.144160
bedrooms_per_room          -0.255880
Name: median_house_value, dtype: float64
```

In [23]:
```
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_
value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



In [24]:
```
housing.describe()
```

Out[24]:

	longitude	latitude	housing_median_age	total_rooms	total_bedroo
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.00000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

Preparing Dastaset for ML

Once we've visualized the data, and have a certain understanding of how the data looks like. It's time to clean!

Most of your time will be spent on this step, although the datasets used in this project are relatively nice and clean... it could get real dirty.

After having cleaned your dataset you're aiming for:

- train set
- test set

In some cases you might also have a validation set as well for tuning hyperparameters (don't worry if you're not familiar with this term yet..)

In supervised learning setting your train set and test set should contain (**feature**, **target**) tuples.

- **feature**: is the input to your model
- **target**: is the ground truth label
  - when target is categorical the task is a classification task
  - when target is floating point the task is a regression task

We will make use of **scikit-learn (https://scikit-learn.org/stable/)** python package for preprocessing.

Scikit learn is pretty well documented and if you get confused at any point simply look up the function/object!

In [25]:
```
from sklearn.model_selection import StratifiedShuffleSplit
# let's first start by creating our train and test sets
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state
=42)
for train_index, test_index in split.split(housing, housing["income_ca
t"]):
    train_set = housing.loc[train_index]
    test_set = housing.loc[test_index]
```

In [26]:
```
housing = train_set.drop("median_house_value", axis=1) # drop labels f
or training set features
                                                       # the input to
the model should not contain the true label
housing_labels = train_set["median_house_value"].copy()
```

Dealing With Incomplete Data

In [27]:

```
# have you noticed when looking at the dataframe summary certain rows
# contained null values? we can't just leave them as nulls and expect
our
# model to handle them for us...
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

Out[27]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populat
4629	-118.30	34.07	18.0	3759.0	NaN	3296.0
6068	-117.86	34.01	16.0	4632.0	NaN	3038.0
17923	-121.97	37.35	30.0	1955.0	NaN	999.0
13656	-117.30	34.05	6.0	2155.0	NaN	1039.0
19252	-122.79	38.48	7.0	6837.0	NaN	3468.0

In [28]:

```
sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
: simply drop rows that have null values
```

Out[28]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	h

In [29]:

```
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
: drop the complete feature
```

Out[29]:

	longitude	latitude	housing_median_age	total_rooms	population	households
4629	-118.30	34.07	18.0	3759.0	3296.0	1462.0
6068	-117.86	34.01	16.0	4632.0	3038.0	727.0
17923	-121.97	37.35	30.0	1955.0	999.0	386.0
13656	-117.30	34.05	6.0	2155.0	1039.0	391.0
19252	-122.79	38.48	7.0	6837.0	3468.0	1405.0

In [30]:
```
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True)
# option 3: replace na values with median values
sample_incomplete_rows
```

Out[30]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populat
4629	-118.30	34.07	18.0	3759.0	433.0	3296.0
6068	-117.86	34.01	16.0	4632.0	433.0	3038.0
17923	-121.97	37.35	30.0	1955.0	433.0	999.0
13656	-117.30	34.05	6.0	2155.0	433.0	1039.0
19252	-122.79	38.48	7.0	6837.0	433.0	3468.0

Could you think of another plausible imputation for this dataset? (Not graded)

Prepare Data

In [31]:
```
# This cell implements the complete pipeline for preparing the data
# using sklearns TransformerMixins
# Earlier we mentioned different types of features: categorical, and f
loats.
# In the case of floats we might want to convert them to categories.
# On the other hand categories in which are not already represented as
integers must be mapped to integers before
# feeding to the model.

# Additionally, categorical values could either be represented as one-
hot vectors or simple as normalized/unnormalized integers.
# Here we encode them using one hot vectors.

from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder

from sklearn.base import BaseEstimator, TransformerMixin


imputer = SimpleImputer(strategy="median") # use median imputation for
```

```
missing values
housing_num = housing.drop("ocean_proximity", axis=1) # remove the cat
egorical feature
# column index
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

#
class AugmentFeatures(BaseEstimator, TransformerMixin):
    '''
    implements the previous features we had defined
    housing["rooms_per_household"] = housing["total_rooms"]/housing["h
ouseholds"]
    housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["
total_rooms"]
    housing["population_per_household"]=housing["population"]/housing[
"households"]
    '''
    def __init__(self, add_bedrooms_per_room = True):
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, househol
ds_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_househ
old,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_househ
old]

attr_adder = AugmentFeatures(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', AugmentFeatures()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
numerical_features = list(housing_num)
categorical_features = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, numerical_features),
        ("cat", OneHotEncoder(), categorical_features),
```

```
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

Select a model and train

Once we have prepared the dataset it's time to choose a model.

As our task is to predict the median_house_value (a floating value), regression is well suited for this.

In [32]:
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

# let's try the full preprocessing pipeline on a few training instance
s
data = test_set.iloc[:5]
labels = housing_labels.iloc[:5]
data_prepared = full_pipeline.transform(data)

print("Predictions:", lin_reg.predict(data_prepared))
print("Actual labels:", list(labels))
```

```
Predictions: [425672.    267608.75  227325.75  199614.375 161432.125
]
Actual labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

```
/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site
-packages/sklearn/linear_model/_base.py:533: RuntimeWarning: interna
l gelsd driver lwork query error, required iwork dimension not retur
ned. This is likely the result of LAPACK bug 0038, fixed in LAPACK 3
.2.2 (released July 21, 2010). Falling back to 'gelss' driver.
  linalg.lstsq(X, y)
/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site
-packages/sklearn/compose/_column_transformer.py:430: FutureWarning:
Given feature/column names or counts do not match the ones for the d
ata given during fit. This will fail from v0.24.
  FutureWarning)
```

We can evaluate our model using certain metrics, a fitting metric for regresison is the mean-squared-loss

$$L(\hat{Y}, Y) = \sum_{i}^{N} (\hat{y_i} - y_i)^2$$

where $\hat{y}$ is the predicted value, and y is the ground truth label.

In [33]:
```
from sklearn.metrics import mean_squared_error

preds = lin_reg.predict(housing_prepared)
mse = mean_squared_error(housing_labels, preds)
rmse = np.sqrt(mse)
rmse
```

Out[33]:
```
67784.54201357064
```

TODO: Applying the end-end ML steps to a different dataset.

We will apply what we've learnt to another dataset (airbnb dataset). We will predict airbnb price based on other features.

[25 pts] Visualizing Data

[5 pts] Load the data + statistics

- load the dataset
- display the first few rows of the data
- drop the following columns: name, host_id, host_name, last_review
- display a summary of the statistics of the loaded data
- plot histograms for 3 features of your choice

In [34]:
```
DATASET_PATH = os.path.join("datasets", "airbnb")
def load_airbnb_data(airbnb_path):
    csv_path = os.path.join(airbnb_path, "AB_NYC_2019.csv")
    return pd.read_csv(csv_path)
airbnb = load_airbnb_data(DATASET_PATH)
```

In [302]:
```
airbnb.head().iloc[:, : 5] # first few rows
```

Out[302]:

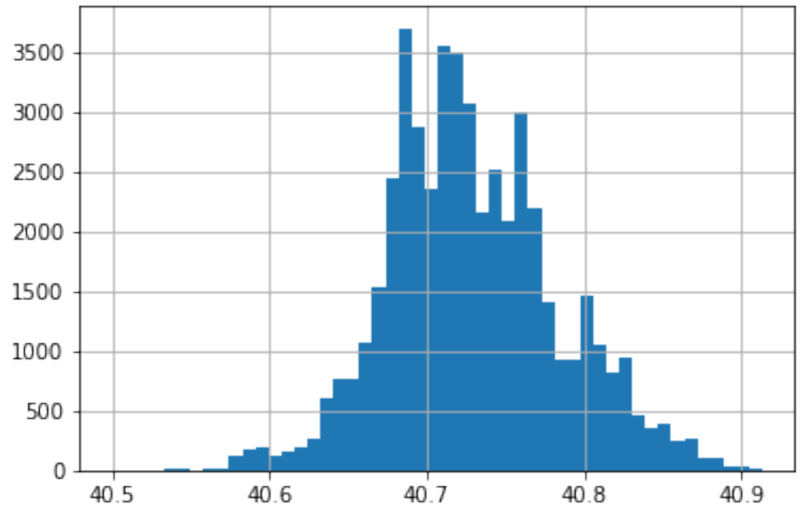
	id	neighbourhood_group	neighbourhood	latitude	longitude
0	2539	Brooklyn	Kensington	40.64749	-73.97237
1	2595	Manhattan	Midtown	40.75362	-73.98377
2	3647	Manhattan	Harlem	40.80902	-73.94190
3	3831	Brooklyn	Clinton Hill	40.68514	-73.95976
4	5022	Manhattan	East Harlem	40.79851	-73.94399

In [36]:
```
airbnb = airbnb.drop(["name", "host_id", "host_name", "last_review"],
axis=1)
```

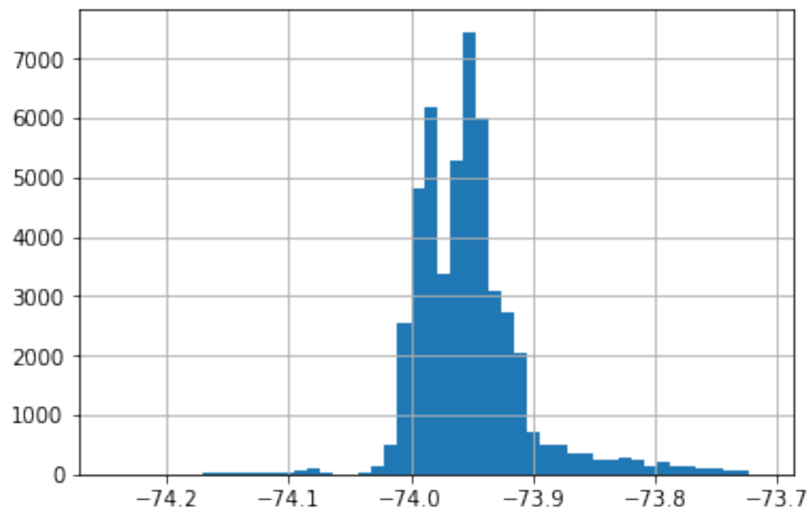
In [37]:
```
airbnb.info() # summary
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48895 entries, 0 to 48894
Data columns (total 12 columns):
id                                48895 non-null int64
neighbourhood_group               48895 non-null object
neighbourhood                     48895 non-null object
latitude                          48895 non-null float64
longitude                         48895 non-null float64
room_type                         48895 non-null object
price                             48895 non-null int64
minimum_nights                    48895 non-null int64
number_of_reviews                 48895 non-null int64
reviews_per_month                 38843 non-null float64
calculated_host_listings_count    48895 non-null int64
availability_365                  48895 non-null int64
dtypes: float64(3), int64(6), object(3)
memory usage: 4.5+ MB
```

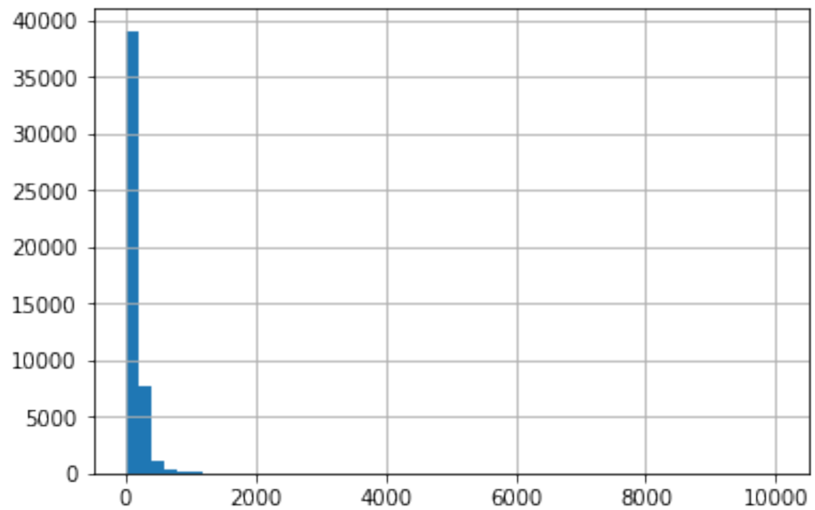
In [38]:
```
airbnb["latitude"].hist(bins=50)
plt.show()
```



In [39]:
```
airbnb["longitude"].hist(bins=50)
plt.show()
```

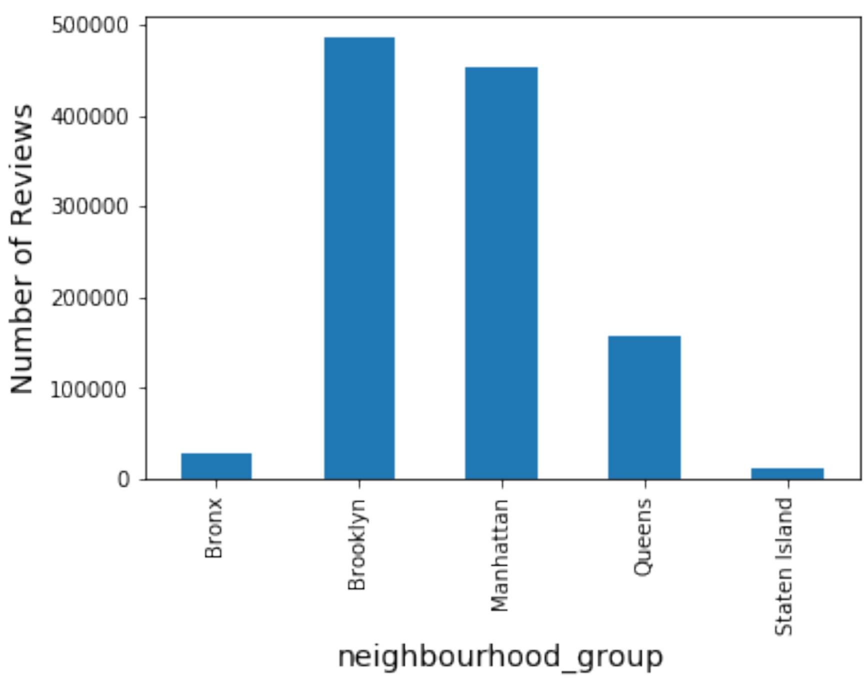


In [40]:
```
airbnb["price"].hist(bins=50)
plt.show()
```



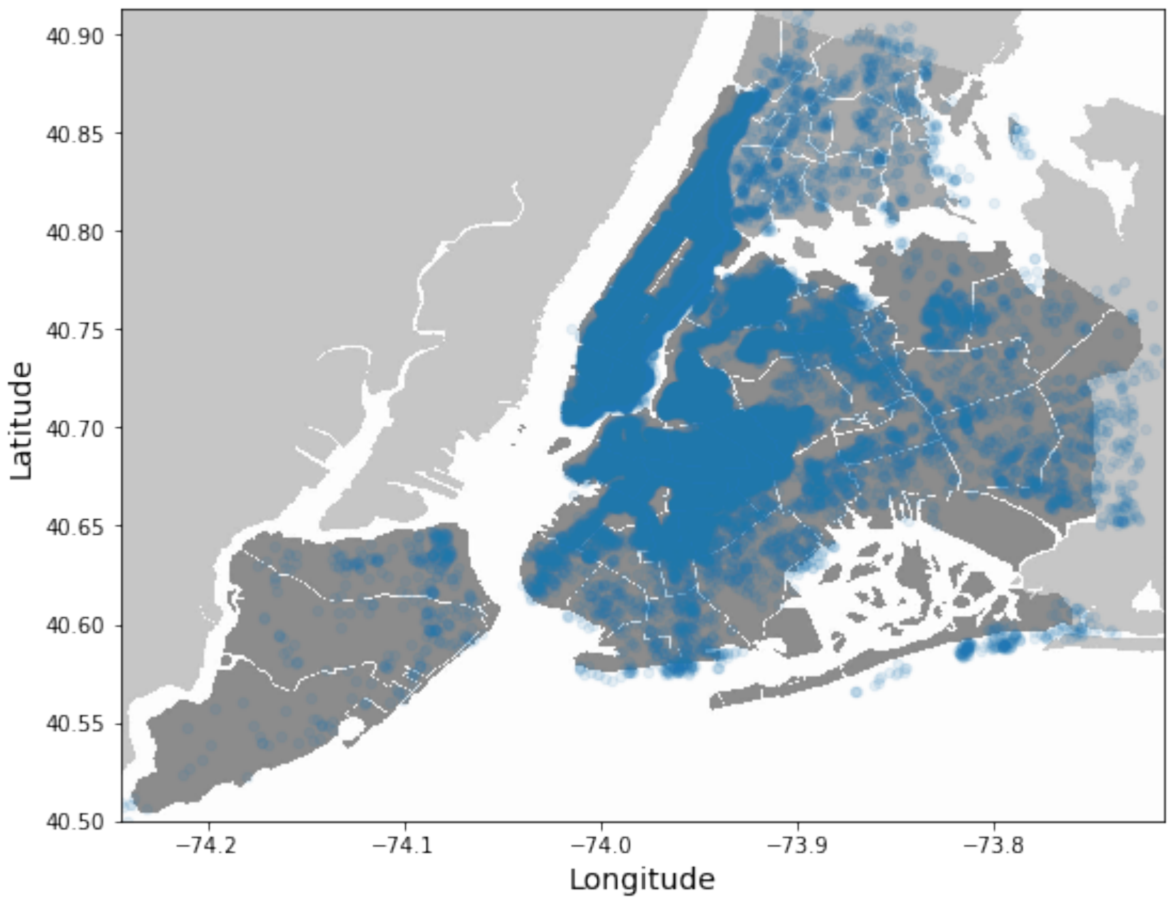
[5 pts] Plot total number_of_reviews per neighbourhood_group

In [41]:
```
reviews = airbnb.groupby("neighbourhood_group")["number_of_reviews"].s
um()
plt.ylabel("Number of Reviews", fontsize=14)
plt.xlabel("Neighbourhood Group", fontsize=14)
reviews.plot(kind="bar")
plt.show()
```



[5 pts] Plot map of airbnbs throughout New York (if it gets too crowded take a subset of the data, and try to make it look nice if you can :)).

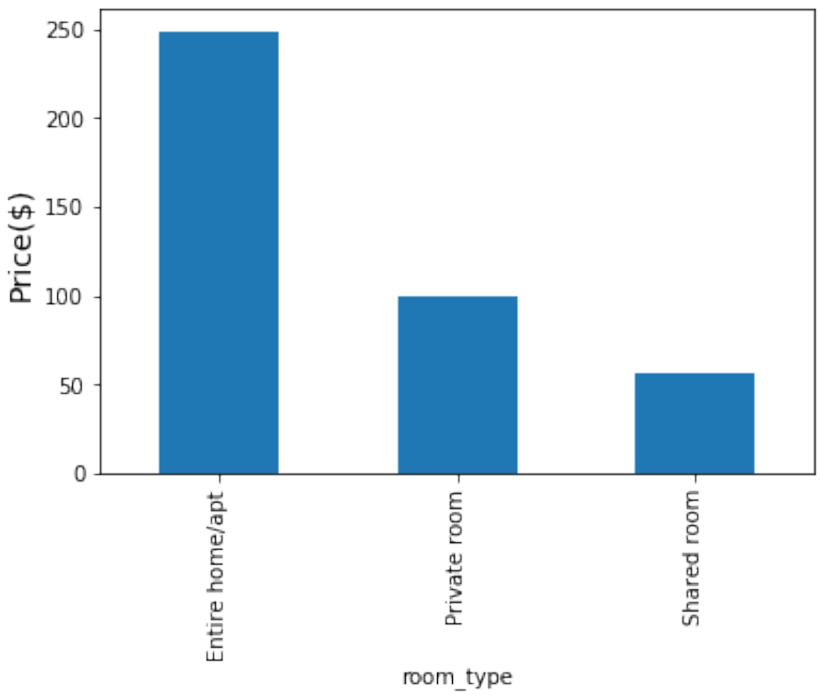
In [131]:
```
images_path = os.path.join('./', "images")
os.makedirs(images_path, exist_ok=True)
filename = "New_York_City_.png"

ny_img = mpimg.imread(os.path.join(images_path, filename), 0)
ax = airbnb.plot(kind="scatter",
                 x="longitude",
                 y="latitude",
                 figsize=(10,7),
                 cmap=plt.get_cmap("jet"),
                 colorbar=False,
                 alpha=0.1)
plt.imshow(ny_img,
           extent=[-74.24440,
                   -73.71290,
                   40.499790,
                   40.913060],
           alpha=0.8,
           cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)
plt.show()
```



[5 pts] Plot average price of room types who have availability greater than 180 days.

In [43]:
```
airbnb_180 = airbnb.loc[airbnb["availability_365"] > 180]
plt.ylabel("Price($)", fontsize=14)
airbnb_180.groupby("room_type")["price"].mean().plot(kind="bar")
```

Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x11f986e10>



[5 pts] Plot correlation matrix

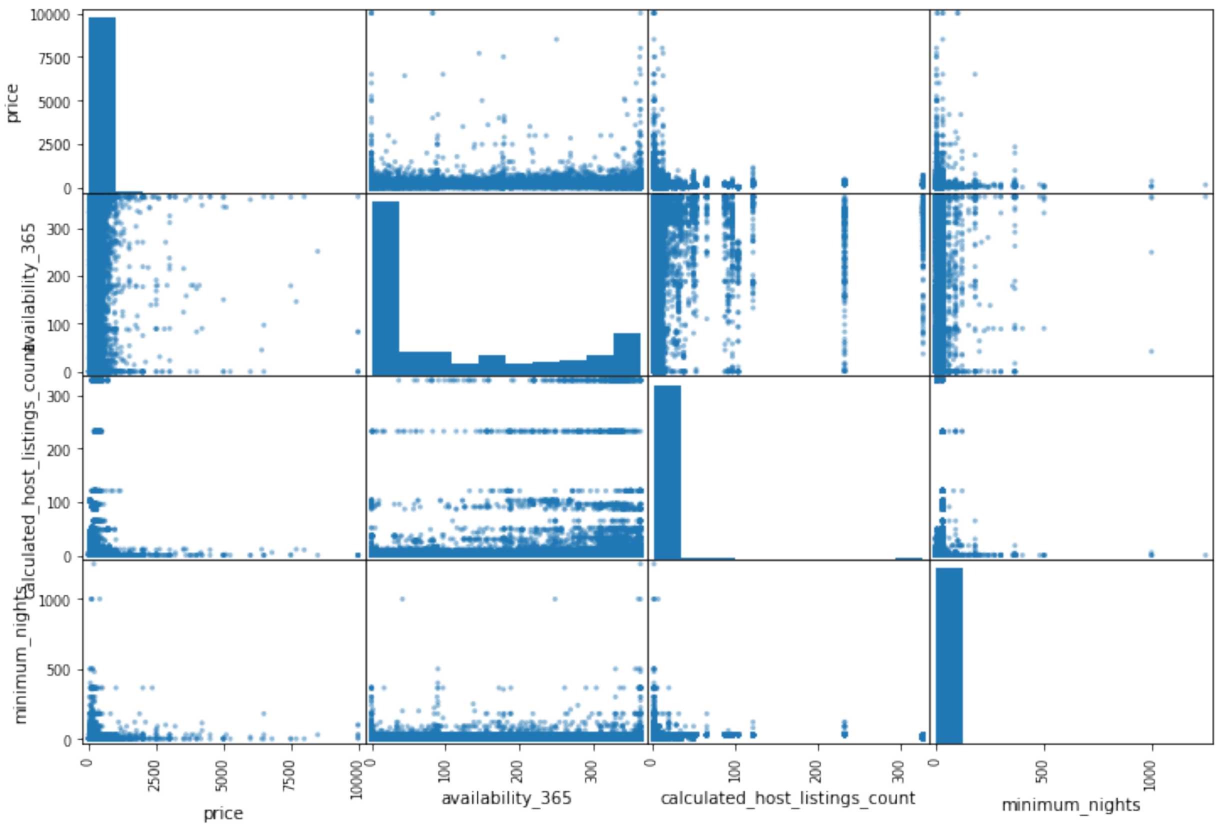
- which features have positive correlation?
- which features have negative correlation?

In [44]:
```
corr_matrix_airbnb = airbnb.corr()
corr_matrix_airbnb["price"].sort_values(ascending=False)
```

Out[44]:
```
price                             1.000000
availability_365                  0.081829
calculated_host_listings_count    0.057472
minimum_nights                    0.042799
latitude                          0.033939
id                                0.010619
reviews_per_month                -0.030608
number_of_reviews                -0.047954
longitude                        -0.150019
Name: price, dtype: float64
```

In [45]:
```
attributes = ["price", "availability_365", "calculated_host_listings_c
ount", "minimum_nights"]
scatter_matrix(airbnb[attributes], figsize=(12, 8))
```

Out[45]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1200fa8d0
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x120193358
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x122e32358
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x122e562e8
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x120f53748
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x120f6ae48
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x12112ceb8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x121153390
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1211f8eb8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x121139320
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x122f42908
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x122f65dd8
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x122f8add8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x122fabd68
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x122fcd3c8
>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x122ff2908
>]],
      dtype=object)
```



Positive correlation

- price 1.000000
- availability_365 0.081829
- calculated_host_listings_count 0.057472
- minimum_nights 0.042799
- latitude 0.033939
- id 0.010619

Negative correlation

- reviews_per_month -0.030608
- number_of_reviews -0.047954
- longitude -0.150019

[25 pts] Prepare the Data

[5 pts] Impute any missing feature with a method of your choice, and briefly discuss why you chose this imputation method

In [51]:
```
imputer = SimpleImputer(strategy="median")
airbnb_ = airbnb.drop(["price", "id"], axis=1)
airbnb_labels = airbnb["price"].copy()
```

The feature with missing values is reviews_per_month. The incomplete data comes from the apartments without a review. We cannot simply drop these two features because we speculate that the price is related to the number of reviews per month. We cannot replace the N/A values with mean values either because mean is easily affected by outliers and thus would affect the relationship between price and number of reviews. Furthermore, if we fill the values with 0, it not reasonable because lacking a review might be due to the fact that it is newly listed. Therefore, filling the missing values with median is the most reasonable choice.

[5 pts] Augment the dataframe with two other features which you think would be useful

In [52]:
```
airbnb_["listing_months"] = airbnb_["number_of_reviews"] / airbnb_["re
views_per_month"]
airbnb_["occupied_365"] = 365 - airbnb_["availability_365"]
```

[10 pts] Code complete data pipeline using sklearn mixins

In [87]:
```
categorical_features = ["neighbourhood_group", "neighbourhood", "room_
type"]
numerical_features = list(airbnb_.drop(categorical_features, axis=1))
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    #('std_scaler', StandardScaler()),
])

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, numerical_features),
    ("cat", OneHotEncoder(), categorical_features),
])
airbnb_prepared = full_pipeline.fit_transform(airbnb_)
airbnb_prepared.shape
```

Out[87]:
```
(48895, 238)
```

I choose to clean up the dataframe before splitting. Augmenting the dataframe before or after the splitting should not make a difference since we do not split the data based on the augmented features. Imputing the data before splitting removes the potential inconsistencies.

[5 pts] Set aside 20% of the data as test test (80% train, 20% test).

In [275]:

```
airbnb_array = airbnb_prepared.toarray()
```

In [294]:

```
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2)
for train_index, test_index in split.split(airbnb, airbnb["room_type"]
):
    X_train = airbnb.loc[train_index]
    X_train_prepared = airbnb_prepared[train_index, :]
    y_train = airbnb_labels.loc[train_index]
    X_test = airbnb.loc[test_index]
    X_test_prepared = airbnb_prepared[test_index, :]
    y_test = airbnb_labels.loc[test_index]
```

[15 pts] Fit a model of your choice

The task is to predict the price, you could refer to the housing example on how to train and evaluate your model using MSE. Provide both test and train set MSE values.

In [295]:

```
# Train the model using the training set
lin_reg = LinearRegression()
lin_reg.fit(X_train_prepared, y_train)

# Training error
preds_train = lin_reg.predict(X_train_prepared)
mse_train = mean_squared_error(y_train, preds_train)

# Testing error
preds_test = lin_reg.predict(X_test_prepared)
mse_test = mean_squared_error(y_test, preds_test)
```

In [296]:

```
print("Training MSE:", mse_train)
print("Testing MSE:", mse_test)
```

```
Training MSE: 47308.50533351442
Testing MSE: 65676.20598785546
```