Modeling and Data Visualization

Lukas Hager

2024-05-13

Overview

Packages

- `statsmodels` is a package for implementing traditional frequentist statistics models
  - IV, robust standard errors, ANOVA, etc.
- `scikit-learn` is a workhorse package for general statistical modeling (inclusive of ML)
  - Implements plenty of algorithms like k-means clustering and LASSO in one clean API
- `matplotlib` and `seaborn` are packages for visualizing data results
  - Often the most important part of analysis

Learning Objectives

- Use formulas and arrays in existing packages to implement statistical/ML models
- Feel comfortable taking analysis results and visualizing them with `matplotlib`
- Be able to think critically about the best way to make visualizations interpretable

statsmodels

Import

- We will import `statsmodels` the following way:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

- The former is an interface that's array-based
- The latter is an interface that's formula-based (similar to `R`'s `lm`)

A Reproducible Example[1]

```
1  import numpy as np
2
3  rng = np.random.default_rng(seed=12345)
4
5  def dnorm(mean, variance, rng, size=1):
6      if isinstance(size, int):
7          size = size,
8      return mean + np.sqrt(variance) * rng.standard_normal(*si
9
10 N = 100
11 X = np.c_[dnorm(0, 0.4, rng, size=N),
12           dnorm(0, 0.6, rng, size=N),
13           dnorm(0, 0.2, rng, size=N)]
14 eps = dnorm(0, 0.1, rng, size=N)
15 beta = [0.1, 0.3, 0.5]
16
17 y = np.dot(X, beta) + eps
```

What are the dimensions of `X` and `y`?

[1] As always, credit to Wes McKinney's book

Generated Data

Note that this is a linear, homoskedastic model.

```
X[:5]
```

```
array([[-0.90050602, -0.18942958, -1.0278702 ],
       [ 0.79925205, -1.54598388, -0.32739708],
       [-0.55065483, -0.12025429,  0.32935899],
       [-0.16391555,  0.82403985,  0.20827485],
       [-0.04765129, -0.21314698, -0.04824364]])
```

Generated Data

Note that this is a linear, homoskedastic model.

```
X[:5]
```

```
array([[-0.90050602, -0.18942958, -1.0278702 ],
       [ 0.79925205, -1.54598388, -0.32739708],
       [-0.55065483, -0.12025429,  0.32935899],
       [-0.16391555,  0.82403985,  0.20827485],
       [-0.04765129, -0.21314698, -0.04824364]])
```

```
y[:5]
```

```
array([-0.59952668, -0.58845445,  0.18563386, -0.00747657, -0.01
```

Adding Intercept

As in our `numpy` lecture, we also want to fit an intercept – `statsmodels` makes this easy

```
X_model = sm.add_constant(X)
X_model[:5]
```

```
array([[ 1.        , -0.90050602, -0.18942958, -1.0278702 ],
       [ 1.        ,  0.79925205, -1.54598388, -0.32739708],
       [ 1.        , -0.55065483, -0.12025429,  0.32935899],
       [ 1.        , -0.16391555,  0.82403985,  0.20827485],
       [ 1.        , -0.04765129, -0.21314698, -0.04824364]])
```

sm.OLS

We begin by creating a sm.OLS object using our data (this is the array-based method):

```
model = sm.OLS(y, X_model)
```

sm.OLS

We begin by creating a `sm.OLS` object using our data (this is the array-based method):

```
model = sm.OLS(y, X_model)
```

We then need to call the `fit` method on the object we just created:

```
results = model.fit()
```

sm.OLS

We begin by creating a `sm.OLS` object using our data (this is the array-based method):

```
model = sm.OLS(y, X_model)
```

We then need to call the `fit` method on the object we just created:

```
results = model.fit()
```

Finally, we can get parameters:

```
results.params
```

```
array([-0.02079903,  0.06581276,  0.26897046,  0.44941894])
```

summary()

```
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Mon, 13 May 2024   Prob (F-statistic):
Time:                        11:57:51   Log-Likelihood:
No. Observations:                 100   AIC:
Df Residuals:                      96   BIC:
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.0
------------------------------------------------------------------------------
const          -0.0208      0.032     -0.653      0.516      -0.0
x1              0.0658      0.054      1.220      0.226      -0.0
x2              0.2690      0.043      6.312      0.000       0.1
x3              0.4494      0.068      6.567      0.000       0.3
```

Standard Errors

Let's rerun with heteroskedasticity-robust standard errors:

```
results_robust = model.fit(cov_type='HC1')
print(results_robust.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Mon, 13 May 2024   Prob (F-statistic):
Time:                        11:57:51   Log-Likelihood:
No. Observations:                 100   AIC:
Df Residuals:                      96   BIC:
Df Model:                           3
Covariance Type:                  HC1
==============================================================================
                 coef    std err          z      P>|z|      [0.0
------------------------------------------------------------------------------
const          -0.0208      0.032     -0.657      0.511      -0.0
```

Differences?

- Why aren't there large differences here?
- What would we need to change to create a meaningful difference?

Using a pd.DataFrame as Input to statsmodels

```
import pandas as pd

data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])
data['y'] = y
print(data[:5])
```

```
       col0      col1      col2         y
0 -0.900506 -0.189430 -1.027870 -0.599527
1  0.799252 -1.545984 -0.327397 -0.588454
2 -0.550655 -0.120254  0.329359  0.185634
3 -0.163916  0.824040  0.208275 -0.007477
4 -0.047651 -0.213147 -0.048244 -0.015374
```

Formula-Based API

This can now be used much like R's `lm` – we use a formula to access the API:

```
results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()
results.params
```

```
Intercept   -0.020799
col0         0.065813
col1         0.268970
col2         0.449419
dtype: float64
```

Formula-Based API

This can now be used much like R's `lm` – we use a formula to access the API:

```
results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()
results.params
```

```
Intercept   -0.020799
col0         0.065813
col1         0.268970
col2         0.449419
dtype: float64
```

```
results.tvalues
```

```
Intercept   -0.652501
col0         1.219768
col1         6.312369
col2         6.567428
dtype: float64
```

Formula-Based API

This can now be used much like R's `lm` – we use a formula to access the API:

```
results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()
results.params
```

```
Intercept   -0.020799
col0         0.065813
col1         0.268970
col2         0.449419
dtype: float64
```

```
results.tvalues
```

```
Intercept   -0.652501
col0         1.219768
col1         6.312369
col2         6.567428
dtype: float64
```

Modeling Complex Formulae

```
results = smf.ols(
    'y ~ -1 + col0 + col1 + col2 + col0*col1 + np.exp(col1)',
    data=data
).fit()
results.params
```

```
col0             0.069201
col1             0.341687
col2             0.444801
col0:col1       -0.121129
np.exp(col1)    -0.032190
dtype: float64
```

Modeling Complex Formulae

```
results = smf.ols(
    'y ~ -1 + col0 + col1 + col2 + col0*col1 + np.exp(col1)',
    data=data
).fit()
results.params
```

```
col0             0.069201
col1             0.341687
col2             0.444801
col0:col1       -0.121129
np.exp(col1)    -0.032190
dtype: float64
```

If you're doing something in your formula that `patsy` doesn't recognize by default, you can pass it within `I()` in the formula, which tells `patsy` to let python handle it.

Prediction

We can also use the results object to predict on out-of-sample data

```
results.predict(data[:5])
```

```
0   -0.631536
1   -0.475749
2    0.030740
3    0.305839
4   -0.124827
dtype: float64
```

Using patsy

In fact, patsy is quite useful for creating design matrices from formulae (again, very similar to R's model.matrix):

```
import patsy
y, X = patsy.dmatrices('y~col0 + col2 + I(col2**2)', data)
X[:5]
```

```
array([[ 1.        , -0.90050602, -1.0278702 ,  1.05651715],
       [ 1.        ,  0.79925205, -0.32739708,  0.10718885],
       [ 1.        , -0.55065483,  0.32935899,  0.10847735],
       [ 1.        , -0.16391555,  0.20827485,  0.04337841],
       [ 1.        , -0.04765129, -0.04824364,  0.00232745]])
```

Exercise: `statsmodels`

Simulate 1000 observations from

$$y = 2 + 3 \times \sin(x) + \varepsilon$$

where $x \sim \mathcal{N}(0, 1)$ and $\varepsilon \sim \mathcal{N}(0, 2)$. Then fit two statsmodels objects reporting the coefficients from a correctly specified regression (using $\sin(x)$) and from an incorrectly specified one (just using x).

Solution: statsmodels

```
N = 1000
x = np.random.standard_normal(N)
y = 2 + 3*np.sin(x) + np.random.normal(scale=np.sqrt(2), size

y_correct,X_correct = patsy.dmatrices('y~np.sin(x)')
y_incorrect,X_incorrect = patsy.dmatrices('y~x')
```

Solution: Correctly Specified

```
print(sm.OLS(y_correct,X_correct).fit().summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Mon, 13 May 2024   Prob (F-statistic):
Time:                        11:57:51   Log-Likelihood:
No. Observations:                1000   AIC:
Df Residuals:                     998   BIC:
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.0
------------------------------------------------------------------------------
Intercept      2.0459      0.045     45.830      0.000       1.9
np.sin(x)      3.0660      0.067     45.830      0.000       2.9
==============================================================================
Omnibus:                        0.087   Durbin-Watson:
```

Solution: Incorrectly Specified

```
print(sm.OLS(y_incorrect,X_incorrect).fit().summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Mon, 13 May 2024   Prob (F-statistic):
Time:                        11:57:51   Log-Likelihood:
No. Observations:                1000   AIC:
Df Residuals:                     998   BIC:
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.0
------------------------------------------------------------------------------
Intercept      2.0309      0.051     39.791      0.000       1.9
x              1.8869      0.051     37.033      0.000       1.7
==============================================================================
Omnibus:                       14.047   Durbin-Watson:
```

scikit-learn

What is `scikit-learn`?

- Package for machine learning in python
- Built-in methods for supervised and unsupervised methods
- There's a consistent `scikit-learn` API style which is very accessible

Titanic Example[2]

Data is from a Kaggle competition about passenger survival rates on the Titanic

```
train = pd.read_csv('https://raw.githubusercontent.com/wesm/p
test = pd.read_csv('https://raw.githubusercontent.com/wesm/py
```

[2] As before, from Wes McKinney

Data Inspection

```
print(train.head())
```

```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex    A
0                            Braund, Mr. Owen Harris    male   22
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female   38
2                             Heikkinen, Miss. Laina  female   26
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female   35
4                           Allen, Mr. William Henry    male   35

   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
```

Data Inspection

```
print(train.head())
```

```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex    A
0                            Braund, Mr. Owen Harris    male   22
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female   38
2                             Heikkinen, Miss. Laina  female   26
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female   35
4                           Allen, Mr. William Henry    male   35

   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
```

Missing Data

With a few exceptions, you can't feed ML algorithms missing data, so we need see the prevalence of missing data in predictors:

```
train.isna().sum() / len(train)
```

```
PassengerId    0.000000
Survived       0.000000
Pclass         0.000000
Name           0.000000
Sex            0.000000
Age            0.198653
SibSp          0.000000
Parch          0.000000
Ticket         0.000000
Fare           0.000000
Cabin          0.771044
Embarked       0.002245
dtype: float64
```

Median Imputation

```
impute_value = train['Age'].median()
train['Age'] = train['Age'].fillna(impute_value)
test['Age'] = test['Age'].fillna(impute_value)

train.isna().sum() / len(train)
```

```
PassengerId    0.000000
Survived       0.000000
Pclass         0.000000
Name           0.000000
Sex            0.000000
Age            0.000000
SibSp          0.000000
Parch          0.000000
Ticket         0.000000
Fare           0.000000
Cabin          0.771044
Embarked       0.002245
dtype: float64
```

Encoding Sex

```
train['IsFemale'] = (train['Sex'] == 'female').astype(int)
test['IsFemale'] = (test['Sex'] == 'female').astype(int)
train.loc[:5, ['IsFemale', 'Sex']]
```

	IsFemale	Sex
0	0	male
1	1	female
2	1	female
3	1	female
4	0	male
5	0	male

Picking Features and Transforming to Array

We have to pass `np.arrays` to `sklearn`, so we pick features and convert them to `numpy`

```
predictors = ['Pclass', 'IsFemale', 'Age']
X_train = train[predictors].to_numpy()
X_test = test[predictors].to_numpy()
y_train = train['Survived'].to_numpy()
X_train[:5]
```

```
array([[ 3.,  0., 22.],
       [ 1.,  1., 38.],
       [ 3.,  1., 26.],
       [ 1.,  1., 35.],
       [ 3.,  0., 35.]])
```

Fitting a Logistic Regression

We will fit a logistic regression – this means that we have to import the model from the `sklearn` module `linear_model` and create an instance of the model type

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
```

Fitting a Logistic Regression

We will fit a logistic regression – this means that we have to import the model from the `sklearn` module `linear_model` and create an instance of the model type

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
```

We can then fit this model using the data and the `fit()` method, which is consistent across all of `sklearn`'s models:

```
model.fit(X_train, y_train)
```

```
LogisticRegression()
```

Logistic Regression

Recall that the probability of survival in this case is being modeled as

$$\mathbb{P}(\text{Survival}_i) = \frac{1}{1 + \exp(-(\beta^\top x_i))} = \frac{\exp(\beta^\top x_i)}{1 + \exp(\beta^\top x_i)}$$

Are the coefficients interpretable?

Making Predictions

Finally, we can make predictions using the fitted model:

```
model.predict(X_test)[:10]
```

```
array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

Making Predictions

Finally, we can make predictions using the fitted model:

```
model.predict(X_test)[:10]
```

```
array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

We can also assess the fit using the `score` method (here we have no test data outcomes, so we reuse the training data)

```
model.score(X_train, y_train)
```

```
0.7878787878787878
```

This is the percentage of observations for which the model makes the correct prediction.

```
np.sum(model.predict(X_train) == y_train) / len(y_train)
```

```
0.7878787878787878
```

Extracting Coefficients for Parametric Models

Since we're fitting a logistic regression model, we can also retrieve the coefficients:

```
model.coef_
```

```
array([[-1.14157583,  2.51975386, -0.03272378]])
```

Extracting Coefficients for Parametric Models

Since we're fitting a logistic regression model, we can also retrieve the coefficients:

```
model.coef_
```

```
array([[-1.14157583,  2.51975386, -0.03272378]])
```

How should we interpret these coefficients (on ticket class, female, and age, respectively)?

Using `sklearn` API More Fully

- Note how much of the data cleaning we did "manually" in `pandas`
- Also note that we had a data preparation "pipeline"
  1. Convert categorical variables to dummies
  2. Impute missing values
- This is pretty common, and thus, `scikit-learn` has an API for doing this

Creating Dummy Variables

We have two options – using `pandas` (`pd.get_dummies`) or using `sklearn` (`preprocessing.OneHotEncoder`). I'll use the former here, but the latter is also fine

```
X_train = pd.get_dummies(
    train[['Pclass', 'Sex', 'Age']],
    drop_first=True
).to_numpy()

X_train[:5]
```

```
array([[3, 22.0, True],
       [1, 38.0, False],
       [3, 26.0, False],
       [1, 35.0, False],
       [3, 35.0, True]], dtype=object)
```

Pipeline

Let's say that we were comfortable saying that we wanted to use our median imputation on any features that we pass to our model – then we can use a `Pipeline`

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

pipe = Pipeline(
    [
        ('impute_median', SimpleImputer(strategy='median')),
        ('logit', LogisticRegression())
    ]
)

pipe.fit(X_train, y_train)
```

```
Pipeline(steps=[('impute_median', SimpleImputer(strategy='median
                ('logit', LogisticRegression())])
```

Output

To access the output, we now have to specify which step in the pipeline we want to access

```
pipe.named_steps['logit'].coef_
```

```
array([[-1.14120327, -0.03271306, -2.51954002]])
```

Pipelines are Useful

- Standardized and interpretable
  - You know how the data's being cleaned
  - Using libraries instead of relying on our own abilities
- Example: LASSO
  - We want to standardize our regressors when we run LASSO; can do this easily with a pipeline

Cross-Validation in `scikit-learn`

- Cross-Validation refresher
- LASSO refresher
- Let's say we wanted to manually implement k-fold cross-validation for a LASSO model (instead of using `sklearn.linear_model.LassoCV`)

Generate “ideal” LASSO Data

```
X = np.random.standard_normal(size = (200,100))
beta = np.zeros((100,1))
beta[:5,:] = np.arange(1,6).reshape(-1,1)
y = X @ beta + 10 * np.random.standard_normal(size = (200,1))
beta.flatten()
```

```
array([1., 2., 3., 4., 5., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
```

Generate "ideal" LASSO Data

```
X = np.random.standard_normal(size = (200,100))
beta = np.zeros((100,1))
beta[:5,:] = np.arange(1,6).reshape(-1,1)
y = X @ beta + 10 * np.random.standard_normal(size = (200,1))
beta.flatten()
```

```
array([1., 2., 3., 4., 5., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0
```

- Note that `beta` is "sparse"

Fit OLS Model

```
from sklearn.linear_model import LinearRegression, Lasso

ols = LinearRegression()
ols.fit(X, y)
```

```
LinearRegression()
```

Fit OLS Model

```
from sklearn.linear_model import LinearRegression, Lasso

ols = LinearRegression()
ols.fit(X, y)
```

```
LinearRegression()
```

What is the model performance in sample?

```
ols.score(X,y)
```

```
0.660191830956375
```

OLS Model out of Sample

Let's generate unseen data:

```
X_unseen = np.random.standard_normal(size = (200,100))
y_unseen = X_unseen @ beta + 10 * np.random.standard_normal(s
ols.score(X_unseen,y_unseen)
```

```
-0.40907911433195676
```

Cross-Validation for LASSO

Implementation of five-fold CV for a parameter value of $\alpha = 2$

```
from sklearn.model_selection import cross_val_score

lasso = Lasso(alpha = 2.)
cv_scores = cross_val_score(lasso, X, y, cv=5)
cv_scores
```

```
array([0.05229446, 0.110592  , 0.1352647 , 0.06777857, 0.1766083
```

Cross-Validation for LASSO

Implementation of five-fold CV for a parameter value of $\alpha = 2$

```
from sklearn.model_selection import cross_val_score

lasso = Lasso(alpha = 2.)
cv_scores = cross_val_score(lasso, X, y, cv=5)
cv_scores
```

```
array([0.05229446, 0.110592  , 0.1352647 , 0.06777857, 0.1766083
```

What does each value in this array mean?

Grid Search

We want to look for the best parameter over a grid of possibilities. Let's say we restrict ourselves to $\alpha \in \{1, ..., 10\}$:

```
from sklearn.model_selection import GridSearchCV

grid = GridSearchCV(
    lasso,
    param_grid = {'alpha': np.arange(1,11)}
)

grid.fit(X, y)
```

```
GridSearchCV(estimator=Lasso(alpha=2.0),
             param_grid={'alpha': array([ 1,  2,  3,  4,  5,  6,
```

Grid Search Results

```
print(pd.DataFrame(grid.cv_results_))
```

```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time
0       0.000410      0.000071         0.000192        0.000028
1       0.000335      0.000016         0.000171        0.000008
2       0.000317      0.000008         0.000174        0.000012
3       0.000360      0.000048         0.000201        0.000051
4       0.000318      0.000004         0.000171        0.000002
5       0.000313      0.000003         0.000169        0.000002
6       0.000374      0.000071         0.000188        0.000027
7       0.000345      0.000034         0.000181        0.000019
8       0.000312      0.000005         0.000167        0.000002
9       0.000313      0.000007         0.000190        0.000042

          params  split0_test_score  split1_test_score  split2_t
0   {'alpha': 1}           0.055458           0.219426
1   {'alpha': 2}           0.052294           0.110592
2   {'alpha': 3}           0.030939           0.059064
3   {'alpha': 4}          -0.002265           0.034352
4   {'alpha': 5}          -0.031806           0.003537
```

Grid Best Estimator

```
grid.best_estimator_
```

```
Lasso(alpha=1)
```

Grid Best Estimator

```
grid.best_estimator_
```

```
Lasso(alpha=1)
```

Does this estimator perform better on the unseen data?

```
grid.best_estimator_.score(X_unseen, y_unseen)
```

```
0.2395501935632599
```

Cross-Validated Model Coefficients

```
grid.best_estimator_.coef_.flatten()
```

```
array([ 0.18018531,  0.19411377,  2.31230729,  2.60385006,  3.99
        0.        , -0.        , -0.        , -0.        , -0.55
       -0.        ,  0.        ,  0.89657092,  0.        , -0.
        0.        ,  0.        , -0.        , -0.        ,  0.
        0.        ,  0.        ,  0.        , -0.        , -0.27
       -0.        , -0.        , -0.        , -0.        , -0.
       -0.        ,  0.        , -0.        ,  0.        , -0.
       -0.        , -0.        ,  0.41305422, -0.        ,  0.
       -0.        ,  0.        ,  0.        ,  0.        ,  0.
        0.        ,  0.        , -0.        , -0.        ,  0.45
       -0.        ,  0.        , -0.        , -0.        ,  0.
       -0.        , -0.19840069,  0.        , -0.        ,  0.
       -0.        ,  0.        ,  0.        ,  0.        , -0.
        0.        , -0.        , -0.        ,  0.        , -0.
       -0.        , -0.        ,  0.        ,  0.19424094,  0.
       -0.99068853,  0.        , -0.        , -0.20650089, -0.
       -0.        ,  0.        ,  0.        , -0.        ,  0.
        0.        , -0.        , -0.        ,  0.08357251, -0.
```

Exercise: `LassoCV`

Look up the documentation for `sklearn.linear_model.LassoCV` and fit a cross-validated model directly with the API (feel free to just use the package defaults). What is the optimal value of α and what is the score on our unseen data?

Solutions: LassoCV

```
from sklearn.linear_model import LassoCV

cv_lasso = LassoCV()
cv_lasso.fit(X=X,y=y)
print(cv_lasso.alpha_)
print(cv_lasso.score(X_unseen,y_unseen))
```

```
1.216055121159551
0.2352665563051211 7
```

Note: Big Data

- Originally in Economics, data is "big" when $n < k$

Note: Big Data

- Originally in Economics, data is "big" when $n < k$
- Problem: can't run a regression

Note: Big Data

- Originally in Economics, data is "big" when $n < k$
- Problem: can't run a regression
- Solution: you can run a regularized regression

Note: Big Data

- Originally in Economics, data is "big" when $n < k$
- Problem: can't run a regression
- Solution: you can run a regularized regression

```
X = np.random.standard_normal(size = (10,100))
beta = np.zeros((100,1))
beta[:5,:] = np.arange(1,6).reshape(-1,1)
y = X @ beta + 10 * np.random.standard_normal(size = (10,1))
LassoCV().fit(X,y).score(X_unseen,y_unseen)
```

```
0.042056822058645804
```

matplotlib

Import

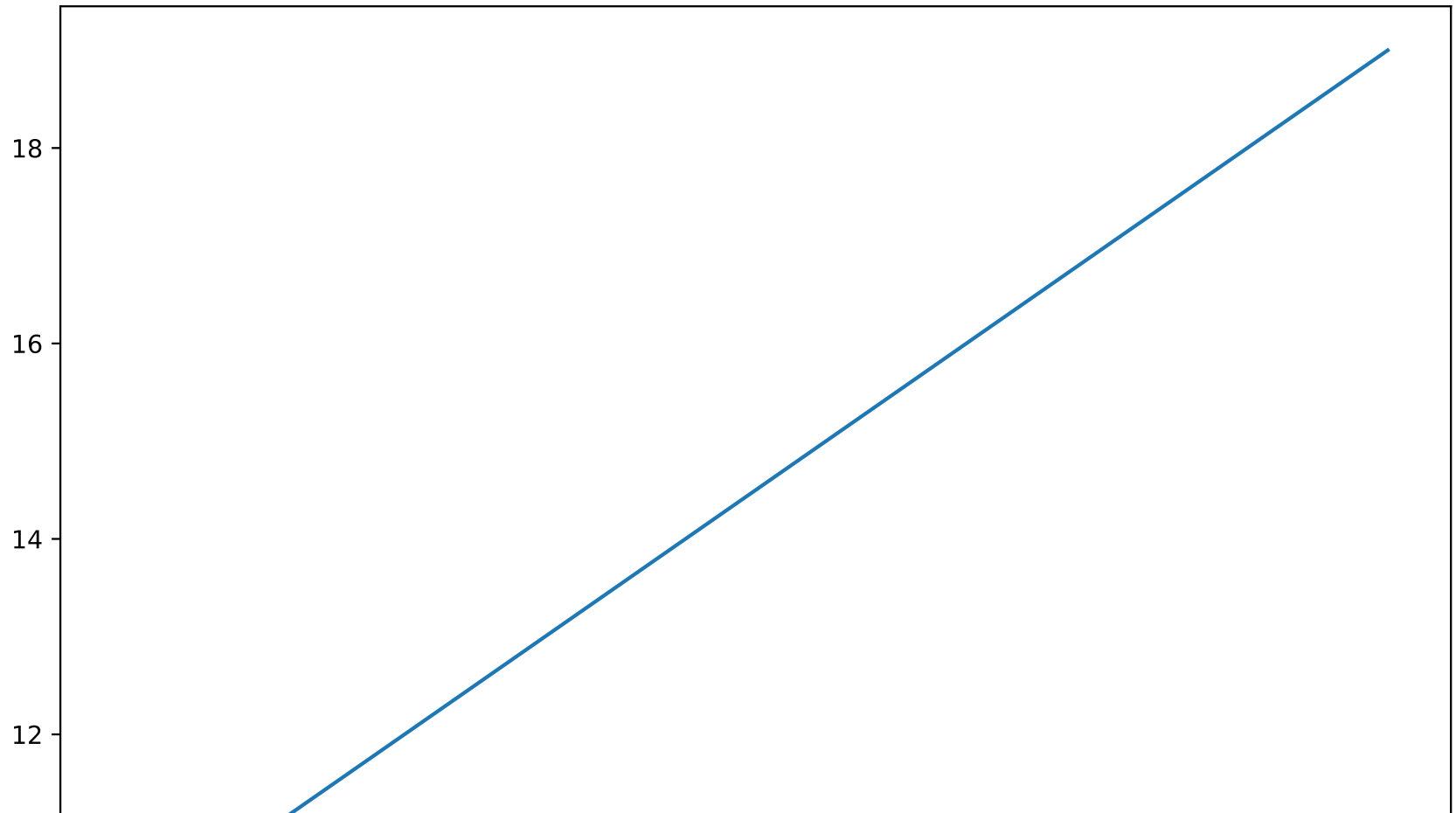
We will always import `matplotlib` like so:

```
import matplotlib.pyplot as plt
```

Basic Line Plots

These are very easy! Let's generate some data and pass it directly to `matplotlib`

```
data = np.arange(10,20)
plt.plot(data)
```



`matplotlib` API Broadly

- Figures: combinations of subplots
- Subplots: have axes and data
- Subplots can share axes with other subplots, are arranged within the figure

Creating Figures[3]

Creating a figure:

```
fig = plt.figure()
```

```
<Figure size 3000x2100 with 0 Axes>
```

[3] As previously, referencing Wes McKinney's book here.

Creating Figures[3]

Creating a figure:

```
fig = plt.figure()
```

```
<Figure size 3000x2100 with 0 Axes>
```

Creating subplots:

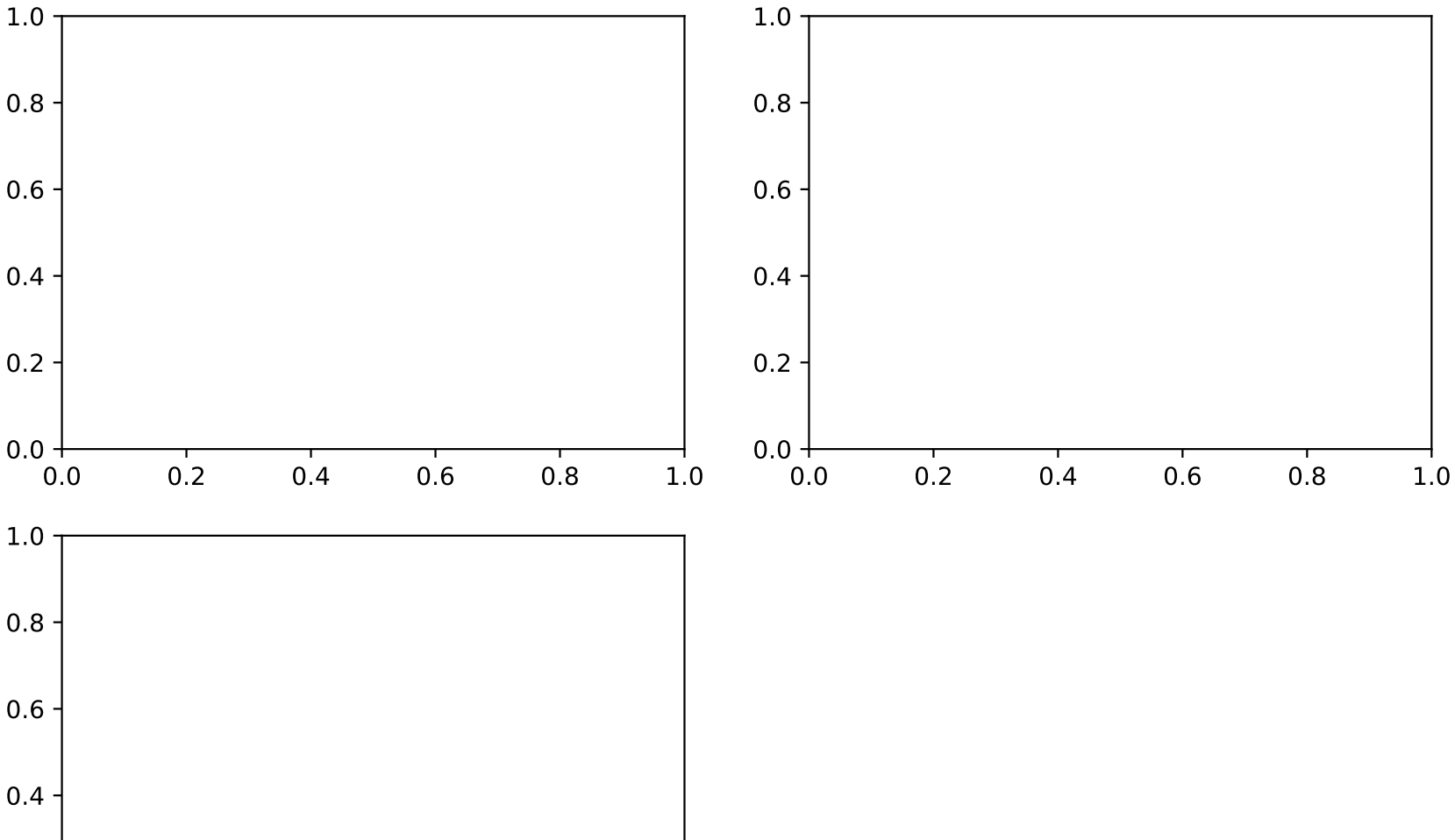
```
ax1 = fig.add_subplot(
    2, # two rows of subplots
    2, # two columns of subplots
    1 # the first of these four subplots
)
```

[3] As previously, referencing Wes McKinney's book here.

Showing A Figure with Subplots

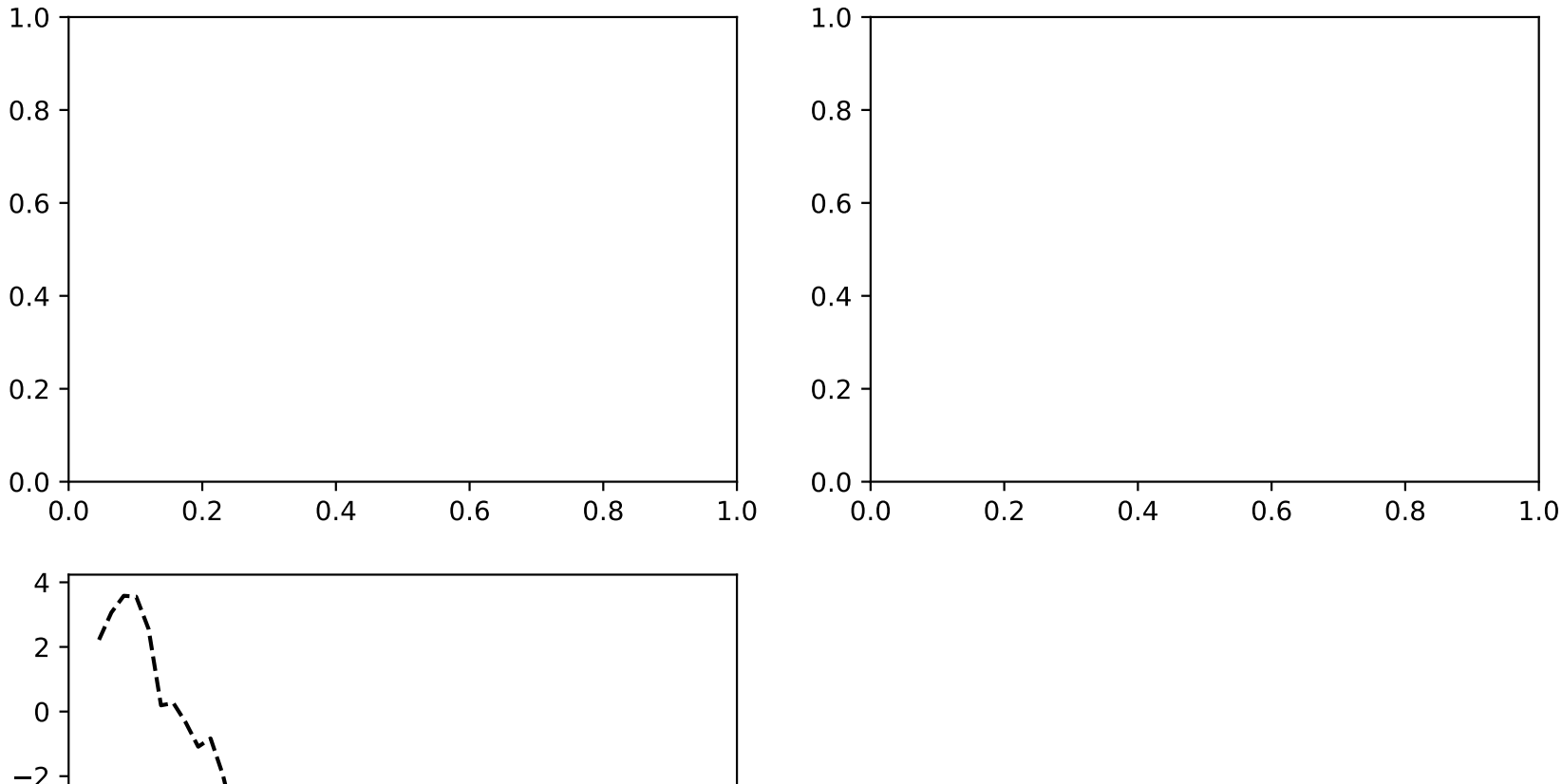
```
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)

fig
```



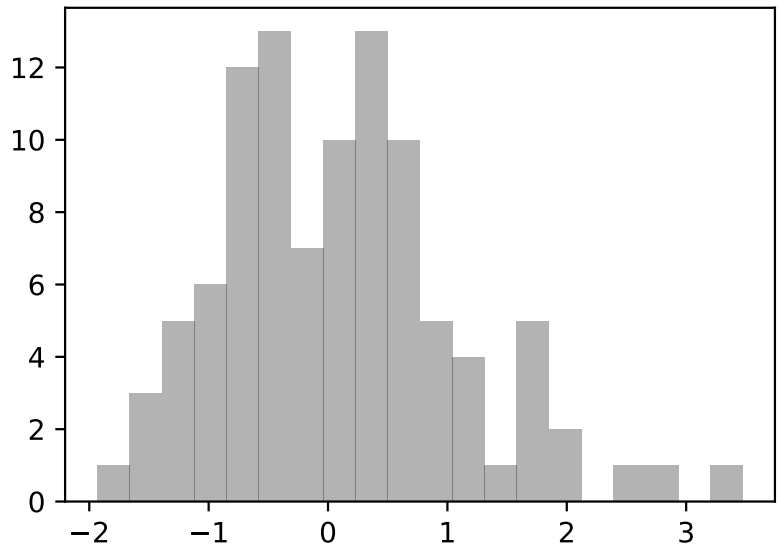
Adding Data to a Subplot

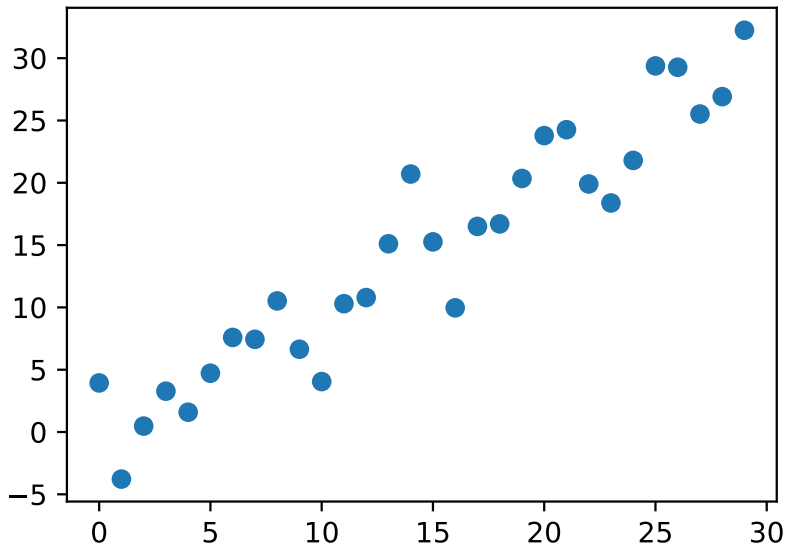
```
ax3.plot(
    np.random.standard_normal(50).cumsum(),
    color="black",
    linestyle="dashed"
)
fig
```

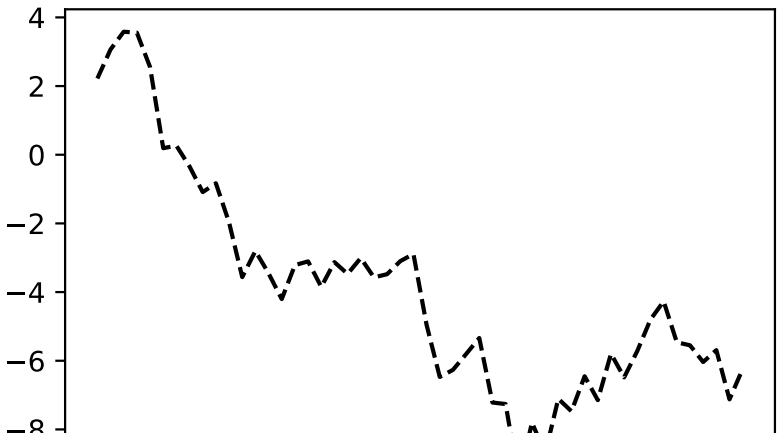


Adding Data to Other Subplots

```
ax1.hist(np.random.standard_normal(100), bins=20, color="blac
ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.stan
fig
```

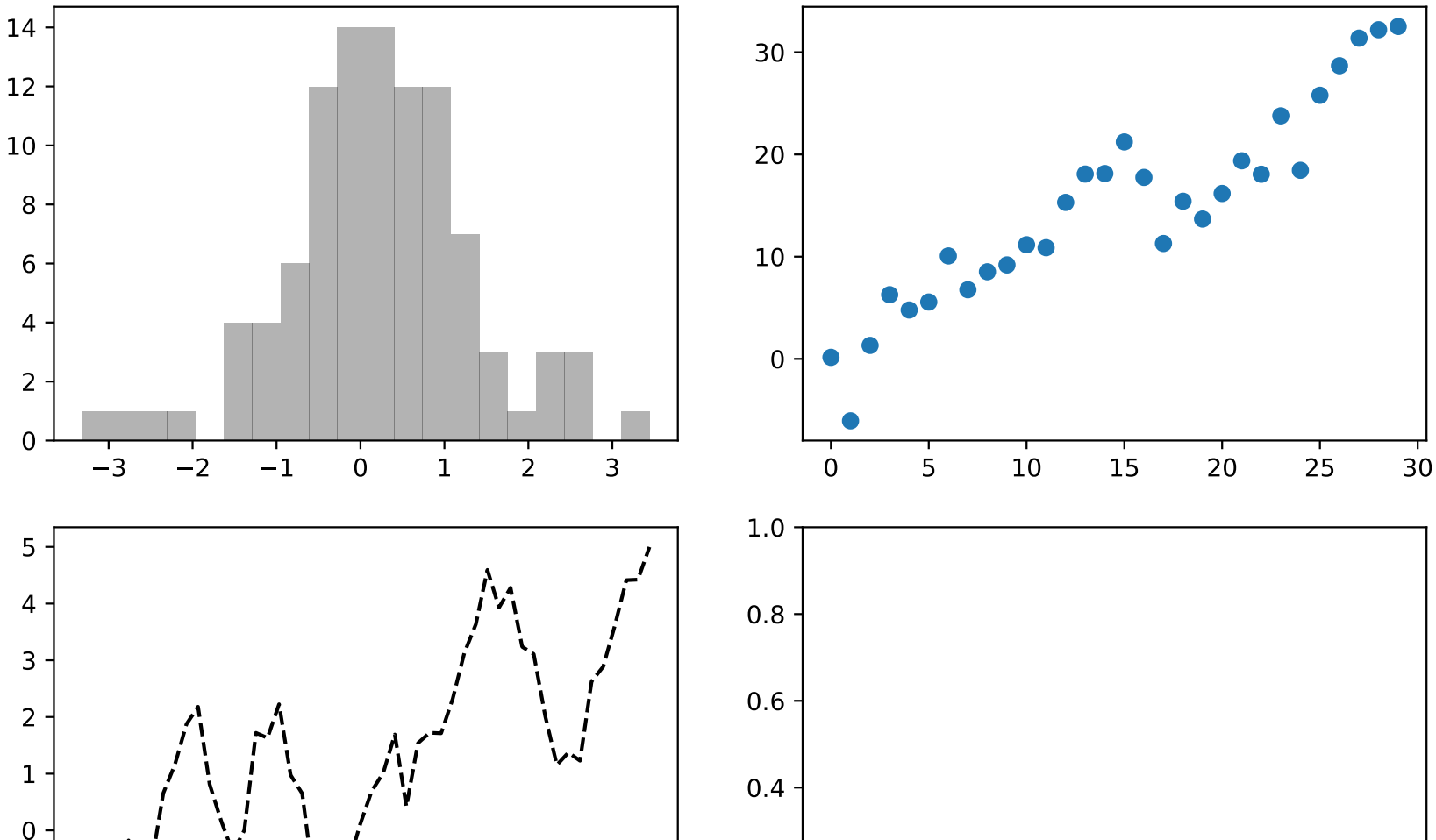






Creating Subplots Easily

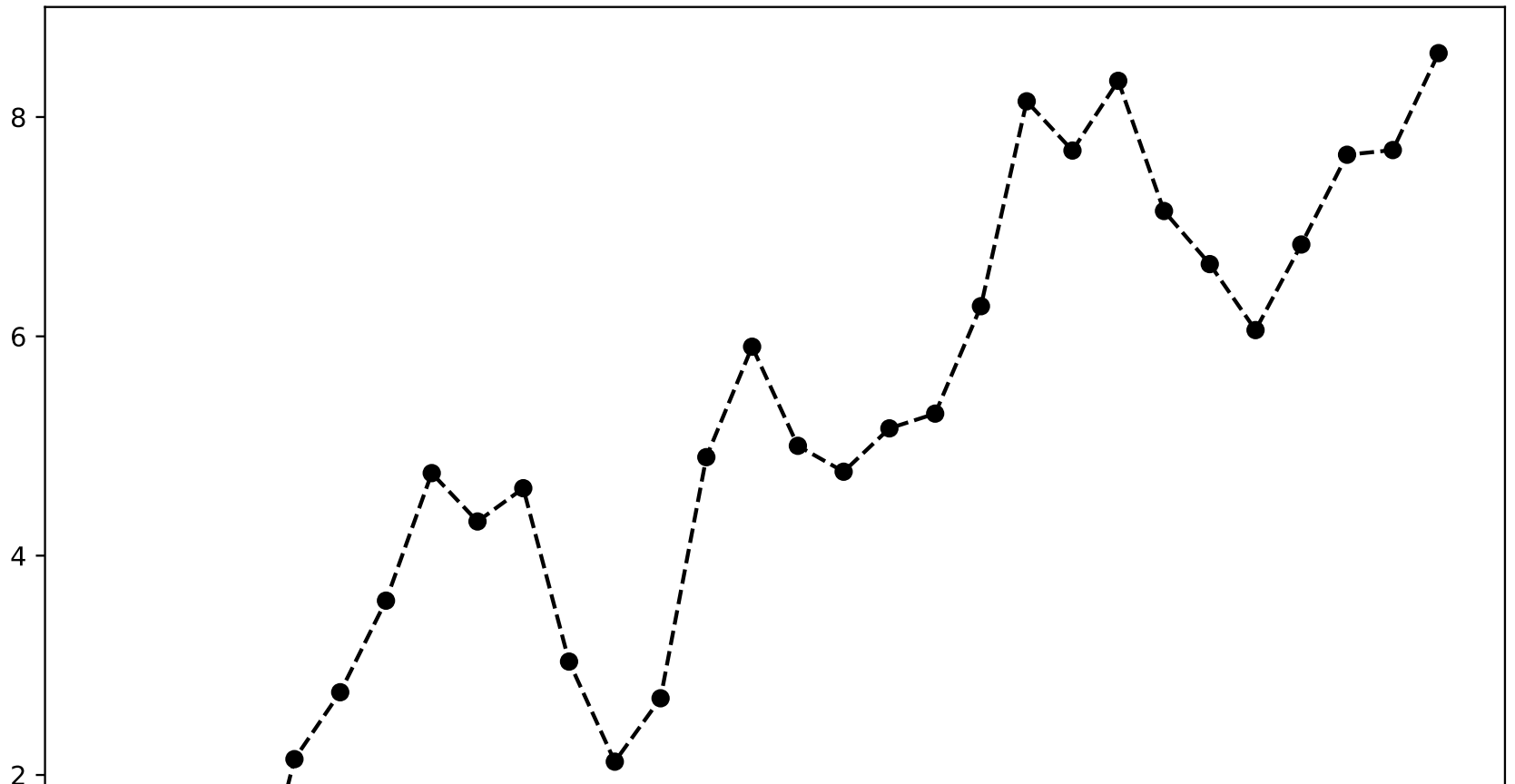
```
fig, axes = plt.subplots(2,2)
axes[0,0].hist(np.random.standard_normal(100), bins=20, color=
axes[0,1].scatter(np.arange(30), np.arange(30) + 3 * np.rando
axes[1,0].plot(np.random.standard_normal(50).cumsum(), color=
```



Lineplot Options

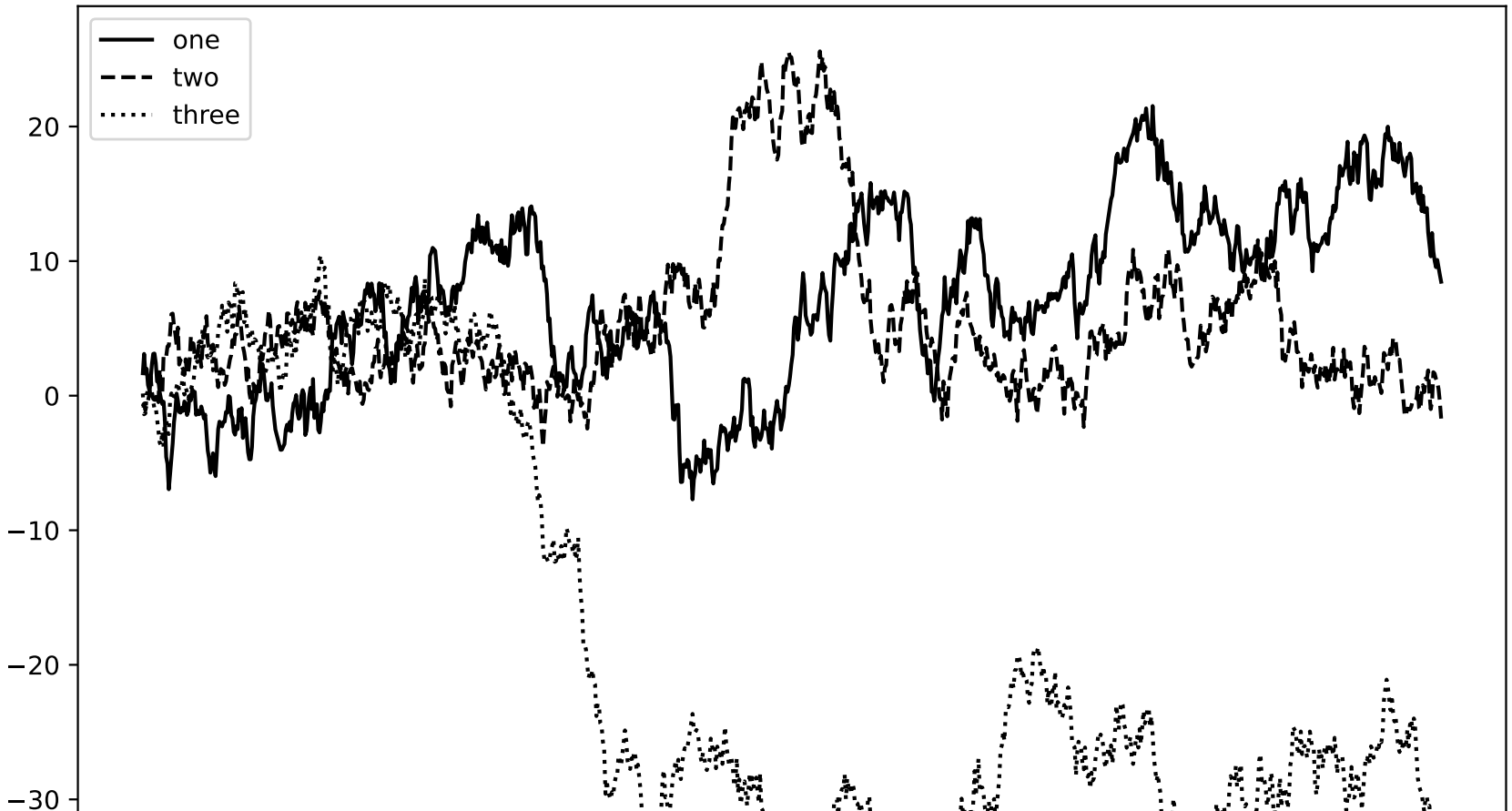
Plenty more in the documentation

```
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(np.random.standard_normal(30).cumsum(), color="black"
        linestyle="dashed", marker="o")
```



Legends

```
fig, ax = plt.subplots()
ax.plot(np.random.randn(1000).cumsum(), color="black", label=
ax.plot(np.random.randn(1000).cumsum(), color="black", linest
ax.plot(np.random.randn(1000).cumsum(), color="black", linest
ax.legend()
```



Exercise: Plotting Predictions

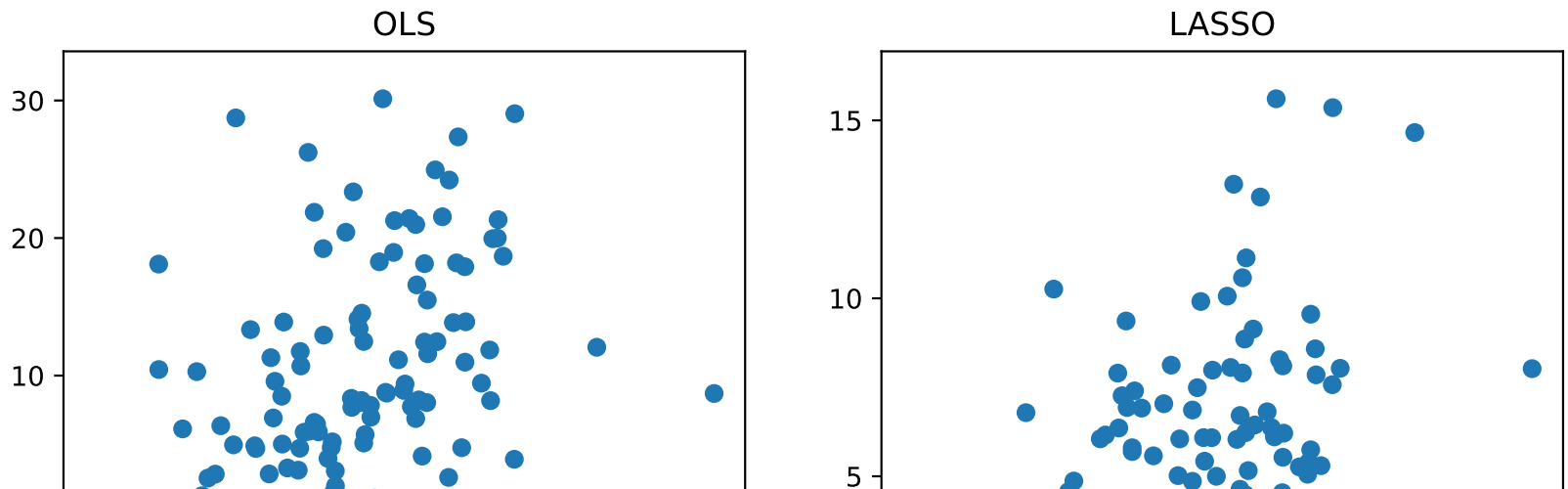
Using our OLS model and LASSO model from before, plot actual vs. predictions on unseen data from both models.

1. Plot side-by-side.
2. Plot together in different colors with a legend.
3. If you have time, try to figure out how to add a title and axis labels.

Exercise: Solutions (1)

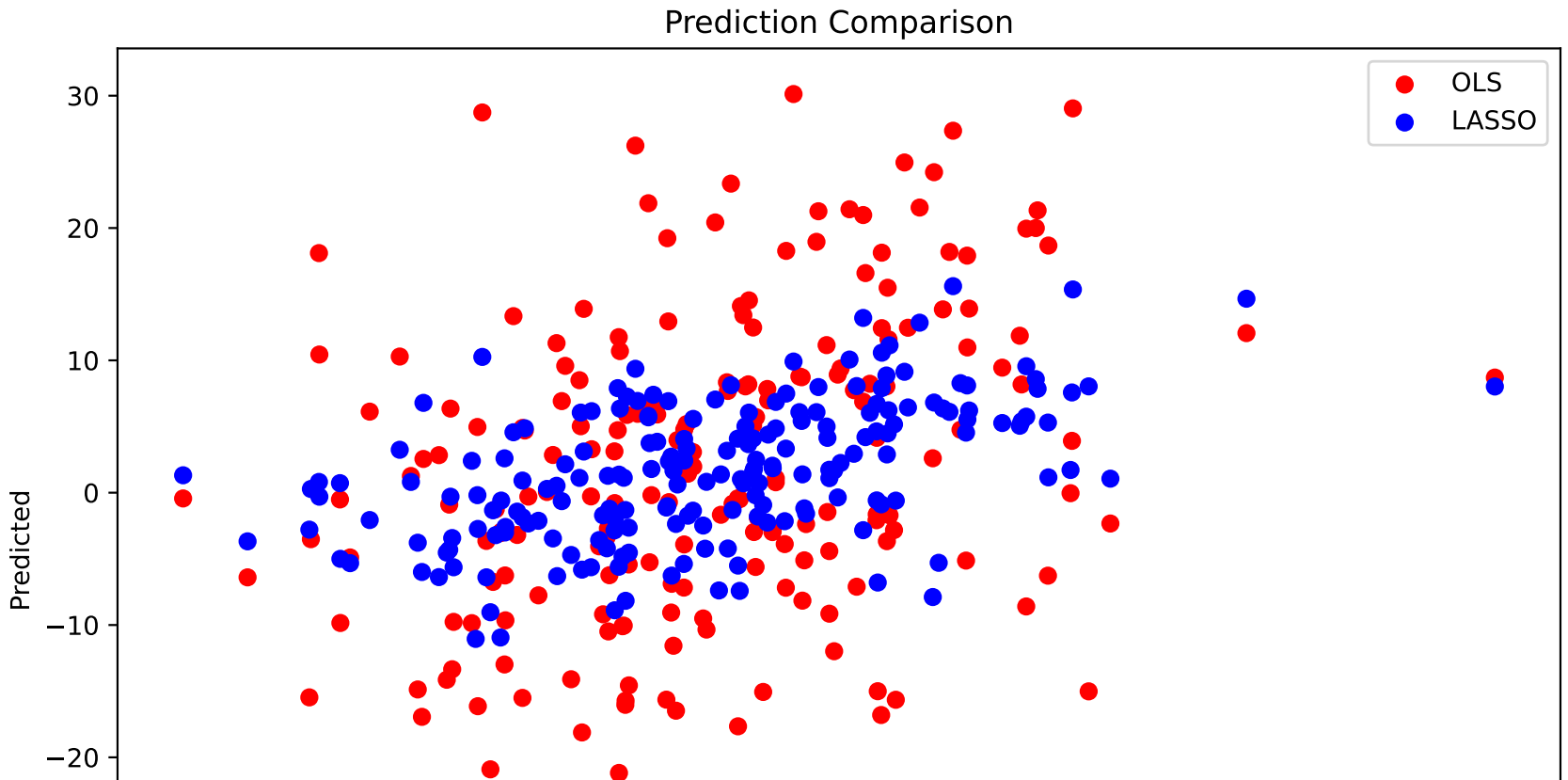
```
preds_lasso = grid.best_estimator_.predict(X_unseen)
preds_ols = ols.predict(X_unseen)

fig, axes = plt.subplots(1,2)
axes[0].scatter(y_unseen, preds_ols)
axes[1].scatter(y_unseen, preds_lasso)
axes[0].set_title('OLS')
axes[1].set_title('LASSO')
for i in range(2):
    axes[i].set_xlabel('Actual')
    axes[i].set_ylabel('Predicted')
```



Exercise: Solutions (2)

```
plt.scatter(y_unseen, preds_ols, color = 'red', label = 'OLS')
plt.scatter(y_unseen, preds_lasso, color = 'blue', label = 'L
plt.legend()
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Prediction Comparison');
```



Annotations – Import Data

To illustrate how to annotate, we'll create a plot of the closing S&P 500 price over the course of the financial crisis, with important dates called out. First, let's get the data

```
data = pd.read_csv(
    "https://raw.githubusercontent.com/wesm/pydata-book/3rd-e
    index_col=0,
    parse_dates=True
)
print(data.head())
```

```
                SPX
Date
1990-02-01  328.79
1990-02-02  330.92
1990-02-05  331.85
1990-02-06  329.66
1990-02-07  333.75
```

Annotations – Import Data

To illustrate how to annotate, we'll create a plot of the closing S&P 500 price over the course of the financial crisis, with important dates called out. First, let's get the data

```
data = pd.read_csv(
    "https://raw.githubusercontent.com/wesm/pydata-book/3rd-e
    index_col=0,
    parse_dates=True
)
print(data.head())
```

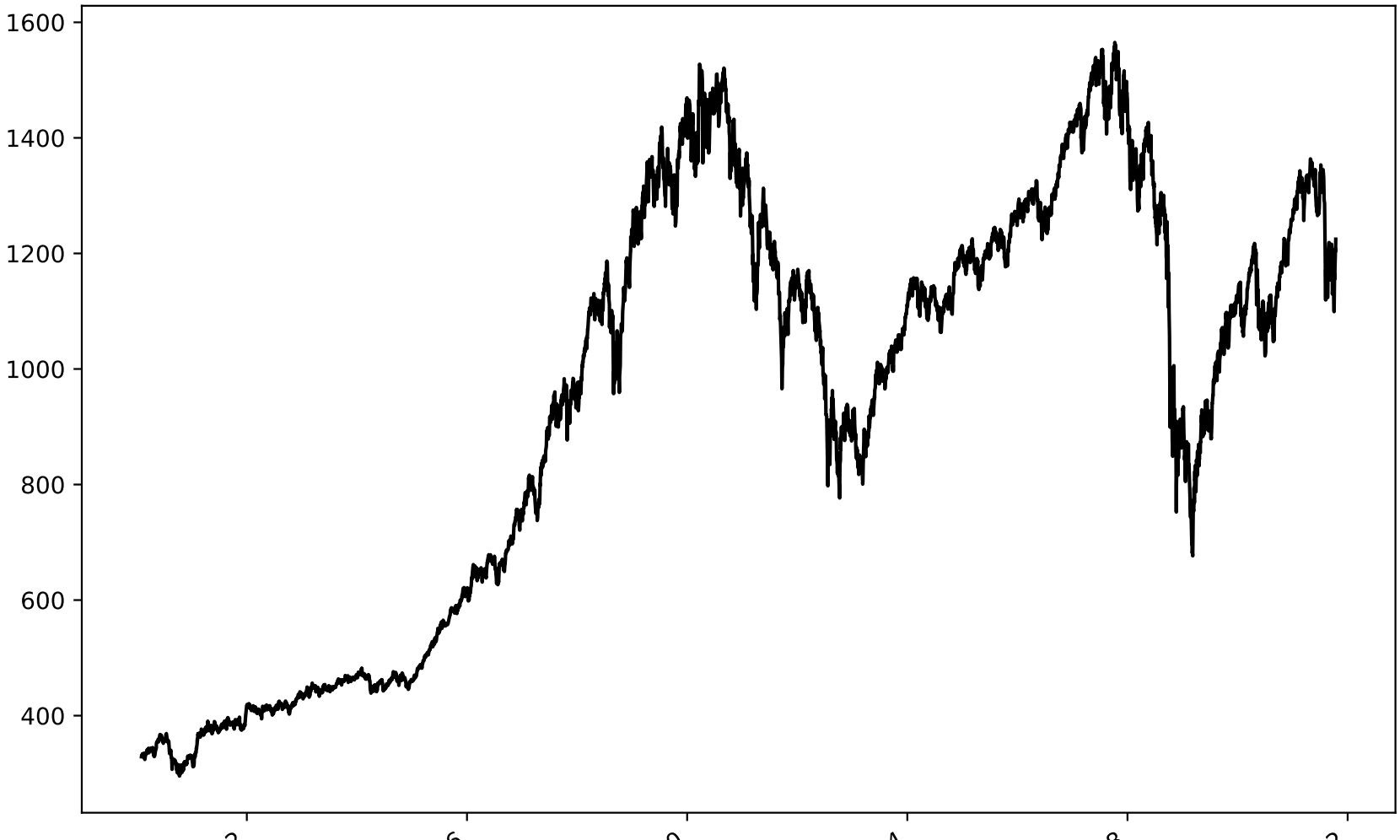
```
                SPX
Date
1990-02-01  328.79
1990-02-02  330.92
1990-02-05  331.85
1990-02-06  329.66
1990-02-07  333.75
```

```
spx = data["SPX"]
```

Annotations – Basic Plot

This is enough to create a basic plot of the S&P 500 closing price

```
fig, ax = plt.subplots()
spx.plot(ax=ax, color="black")
```



Annotations – Crisis Data

We need to know what happened and when:

```
from datetime import datetime

crisis_data = [
    (datetime(2007, 10, 11), "Peak of bull market"),
    (datetime(2008, 3, 12), "Bear Stearns Fails"),
    (datetime(2008, 9, 15), "Lehman Bankruptcy")
]
type(crisis_data)
```

```
list
```

Annotations – Crisis Data

We need to know what happened and when:

```
from datetime import datetime

crisis_data = [
    (datetime(2007, 10, 11), "Peak of bull market"),
    (datetime(2008, 3, 12), "Bear Stearns Fails"),
    (datetime(2008, 9, 15), "Lehman Bankruptcy")
]
type(crisis_data)
```

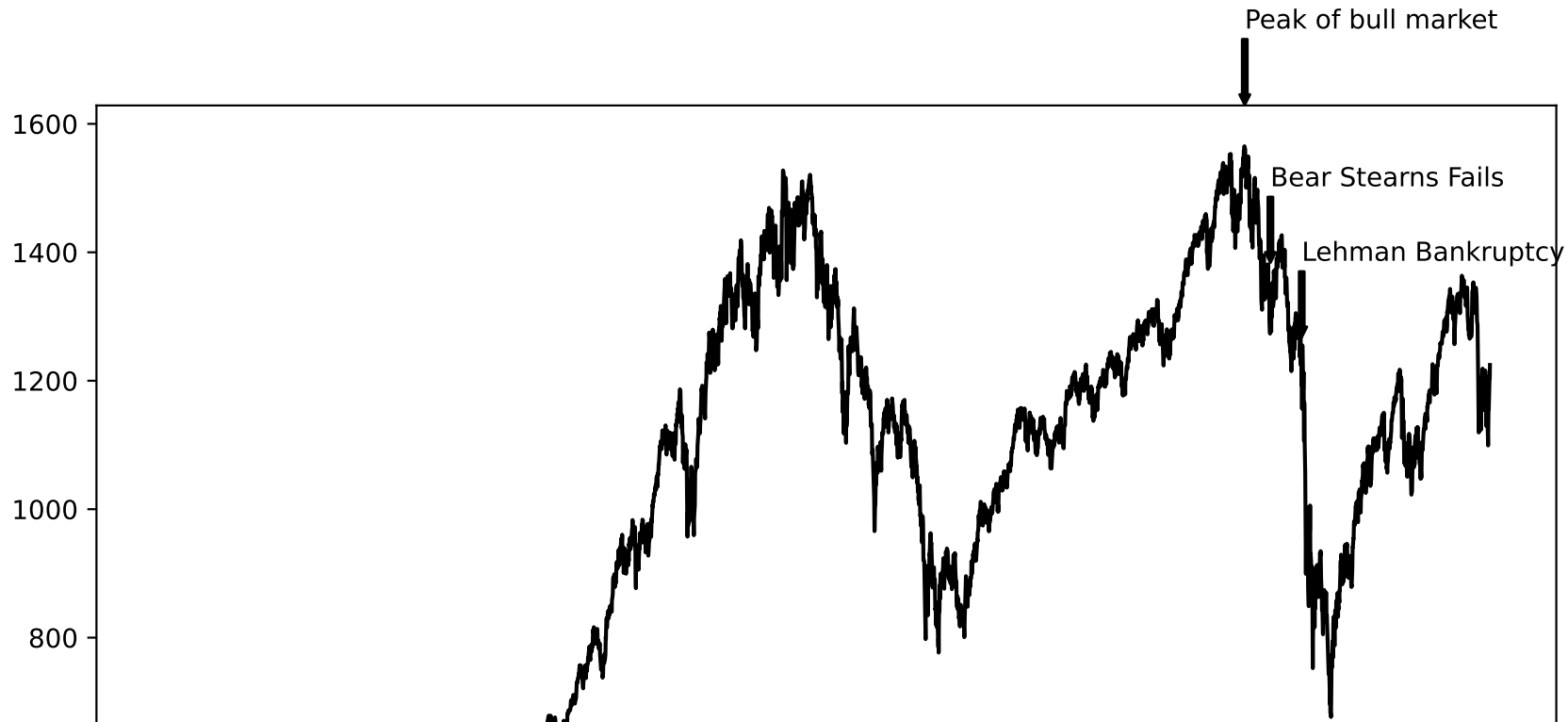
```
list
```

```
type(crisis_data[0])
```

```
tuple
```

Annotations – Annotate

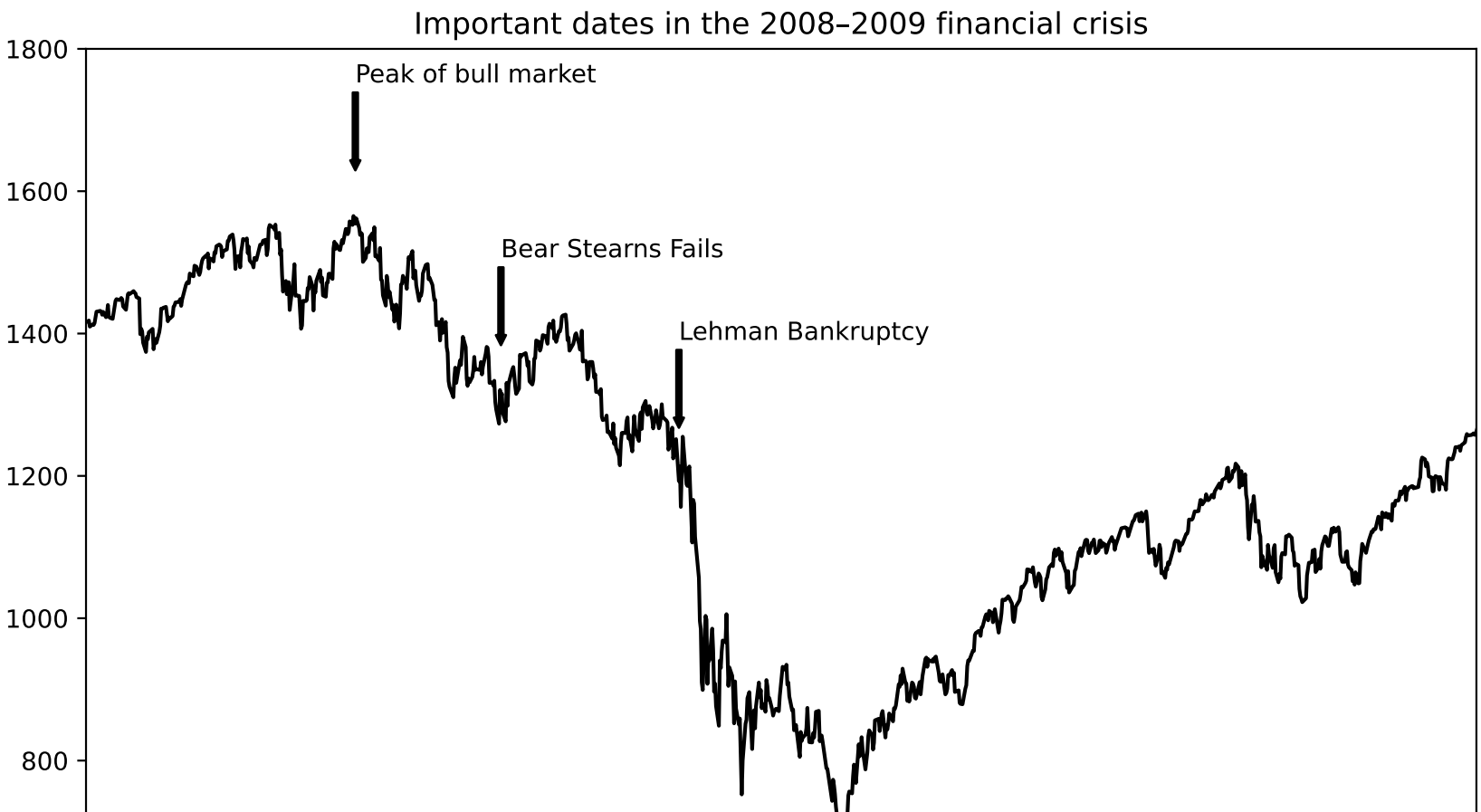
```
for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor="black", headwidth=
                                headlength=4),
                horizontalalignment="left", verticalalignment=
fig
```



Annotations – Date Range and Title

```
ax.set_xlim(["1/1/2007", "1/1/2011"])
ax.set_ylim([600, 1800])

ax.set_title("Important dates in the 2008-2009 financial crisi
fig
```



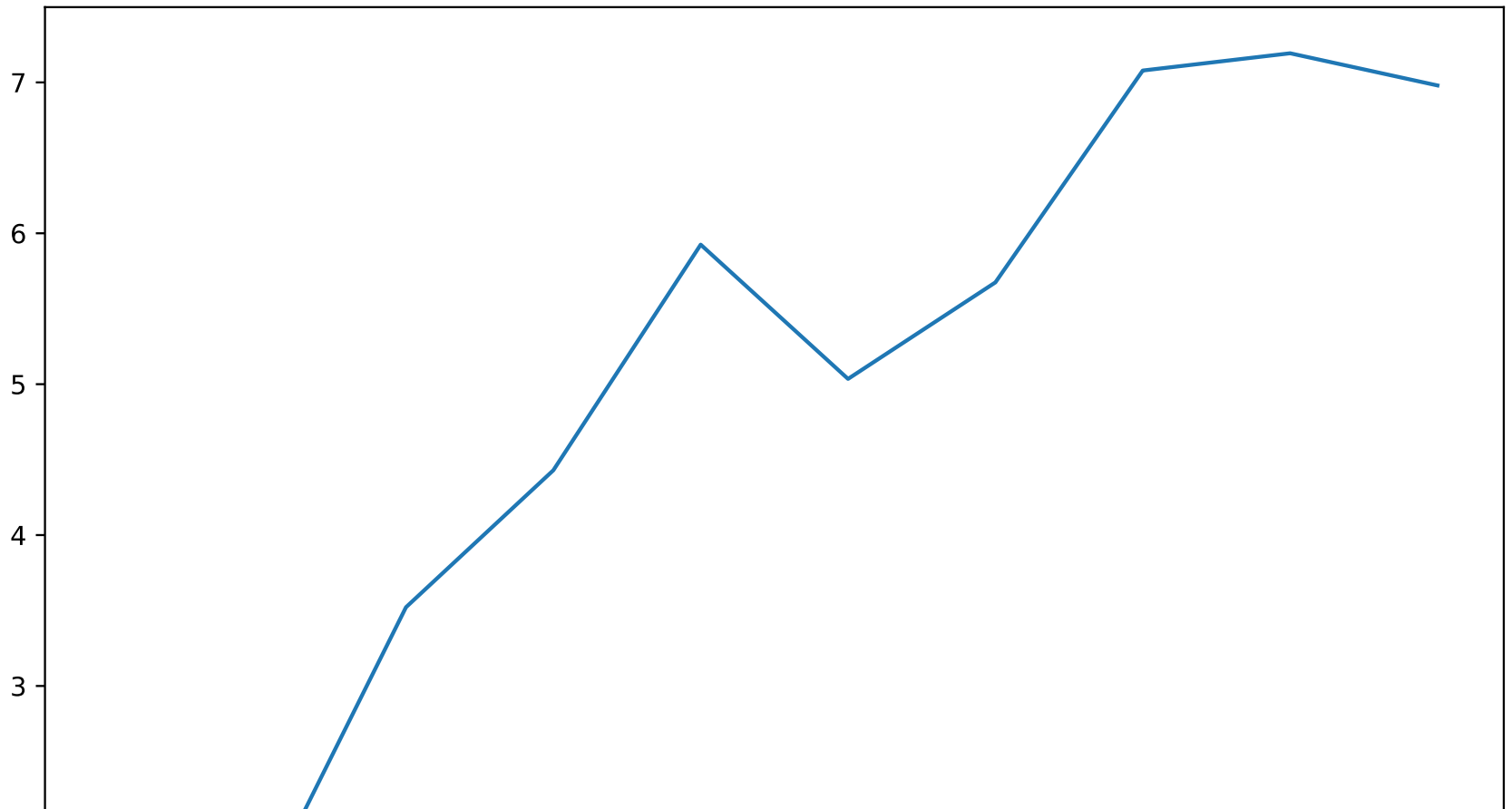
seaborn

`matplotlib` API can be Inconvenient

- If we have a dataframe, we may want to leverage many features of the data
  - For example, a scatterplot with different shapes for different groups
- We can do this in `matplotlib`, but it's not ideal
  - Need to references each variable on its own
- `seaborn` works directly with `pandas` to leverage `DataFrame` and `Series` objects

Series have a plot Method

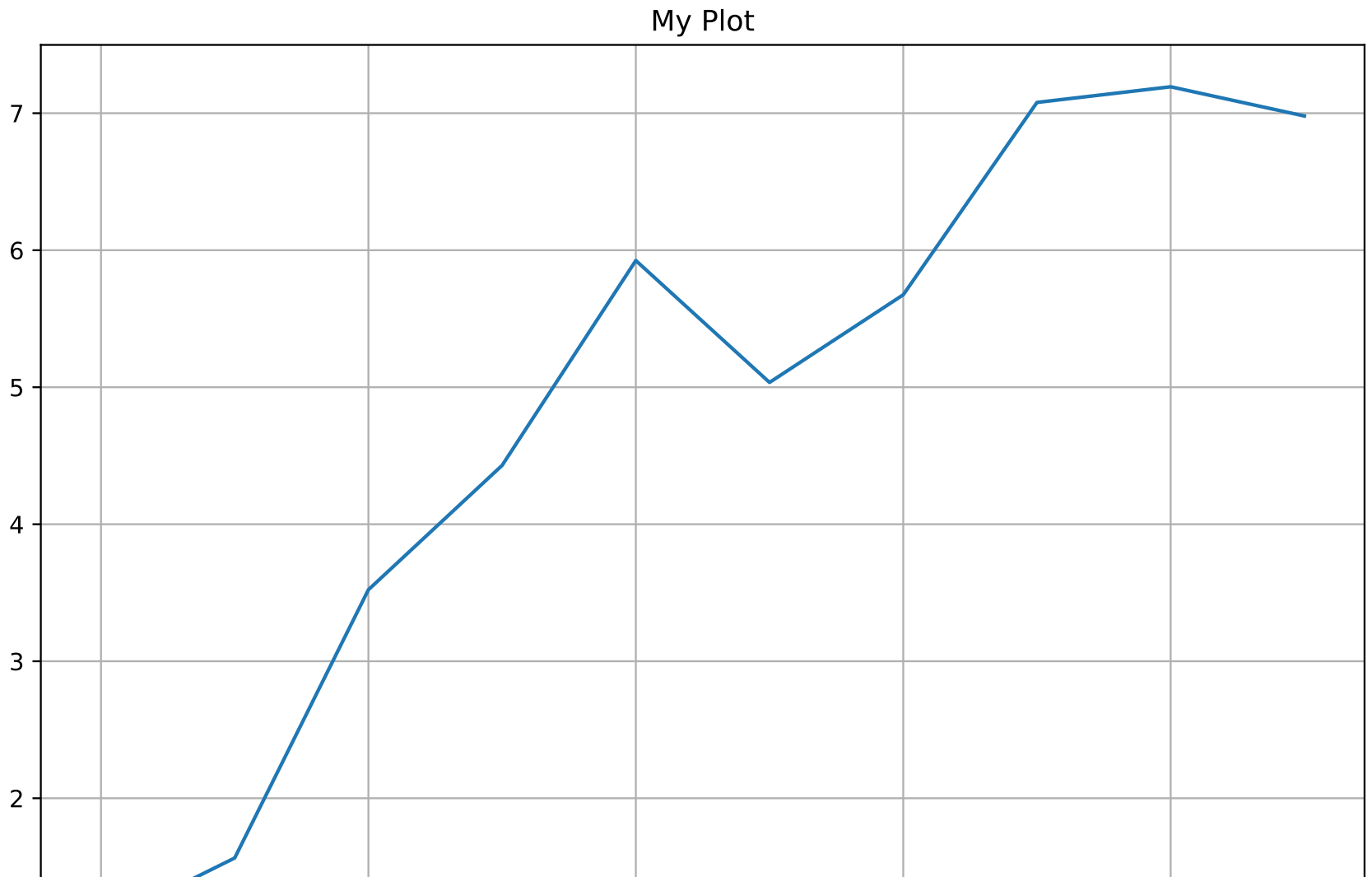
```
s = pd.Series(
    np.random.standard_normal(10).cumsum(),
    index=np.arange(0, 100, 10)
)
s.plot()
```



`plot` Has Formatting Options

Fuller list here

```
s.plot(title = 'My Plot', grid = True)
```



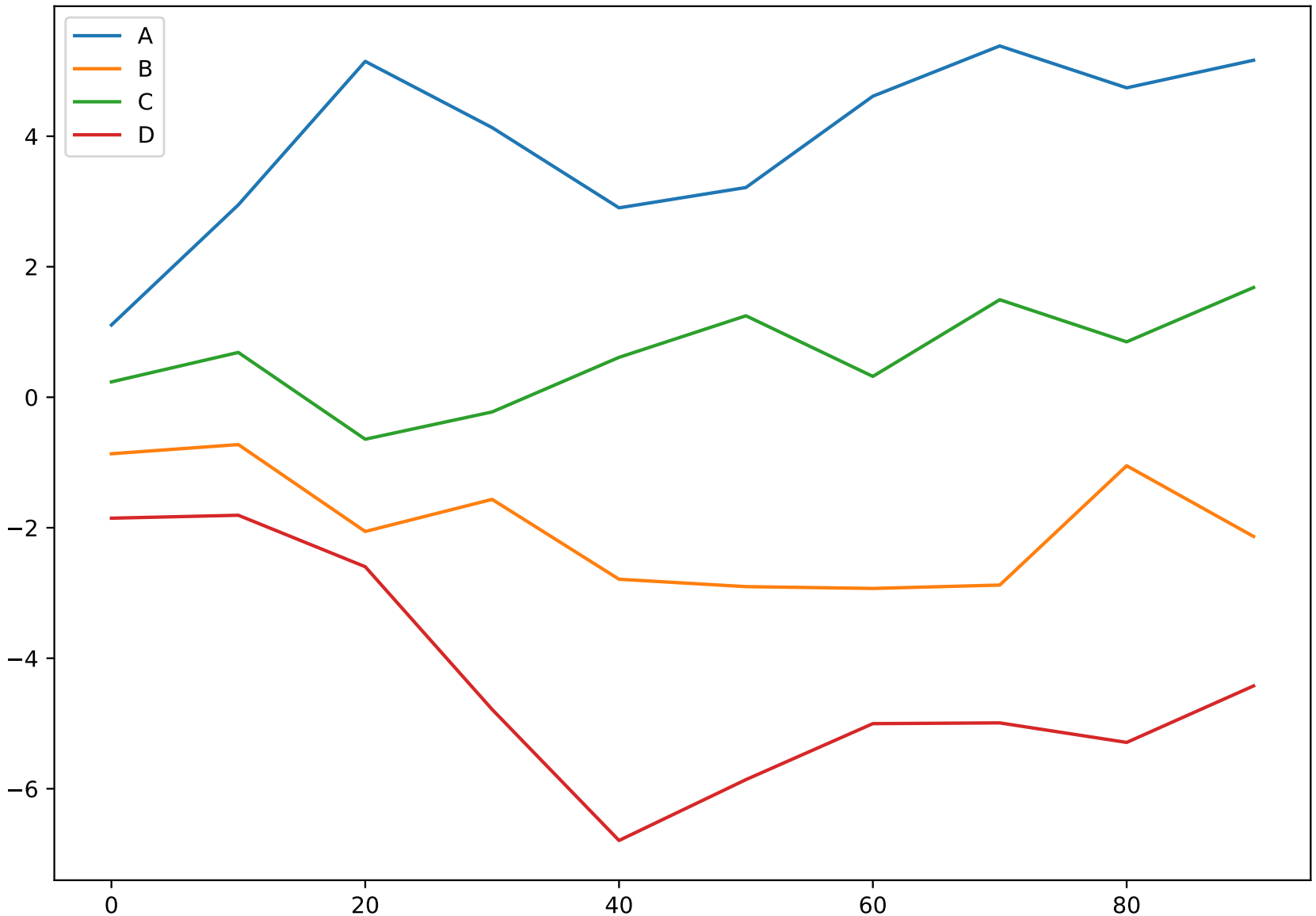
Since a DataFrame is made up of Series...

```
df = pd.DataFrame(np.random.standard_normal((10, 4)).cumsum(0
                  columns=["A", "B", "C", "D"],
                  index=np.arange(0, 100, 10))
print(df)
```

```
           A         B         C         D
0   1.106728 -0.866214  0.235206 -1.853799
10  2.948470 -0.725233  0.686533 -1.808754
20  5.147404 -2.057531 -0.643525 -2.597403
30  4.132320 -1.565383 -0.225219 -4.784608
40  2.902674 -2.789219  0.611306 -6.792956
50  3.214318 -2.902681  1.248224 -5.860315
60  4.613268 -2.930443  0.319327 -5.002226
70  5.384007 -2.878670  1.494778 -4.990756
80  4.741819 -1.049899  0.849258 -5.290131
90  5.164763 -2.133666  1.681957 -4.424057
```

...we can call `plot` on a `DataFrame`

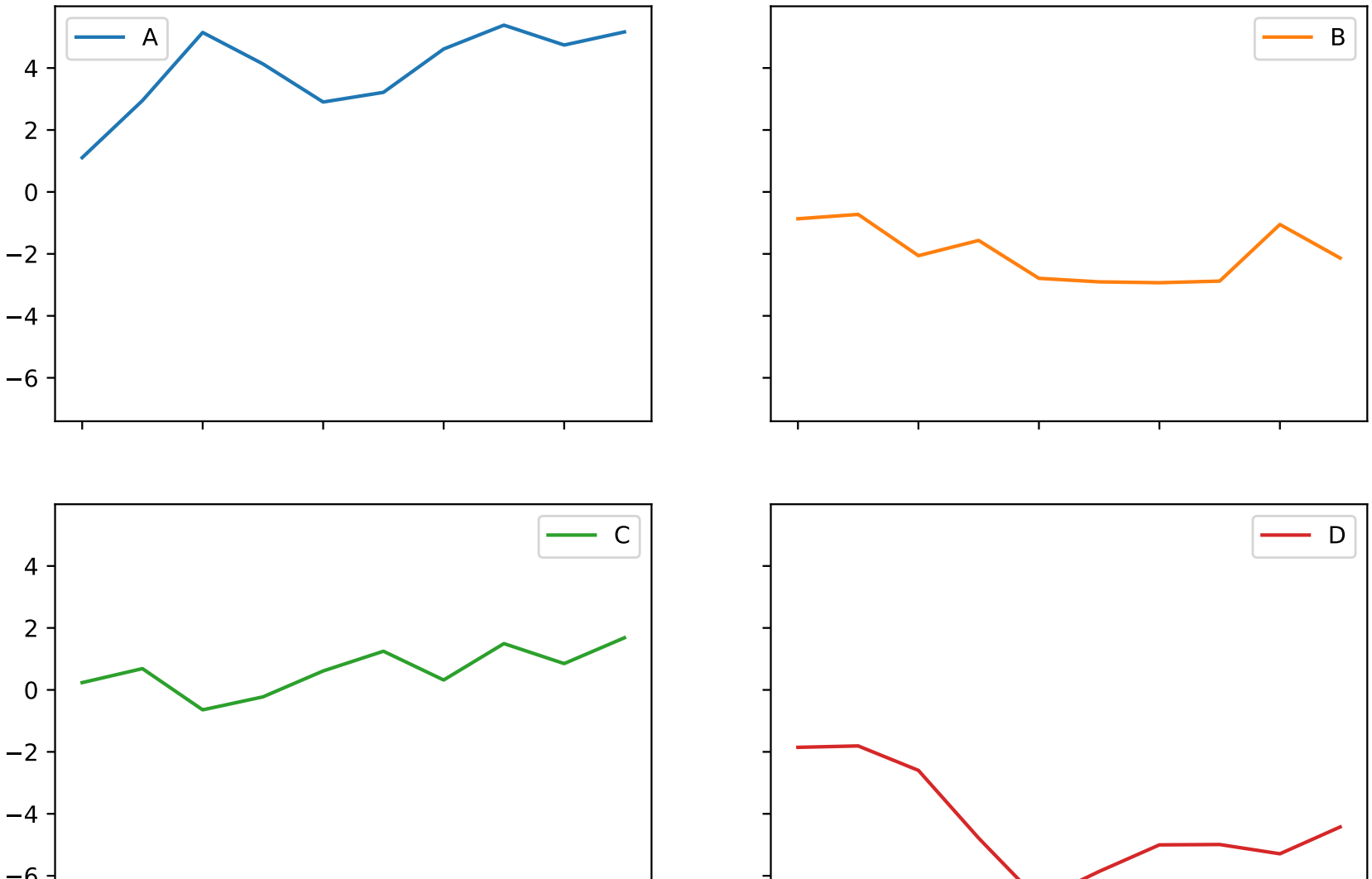
```
df.plot()
```



Individual Plots

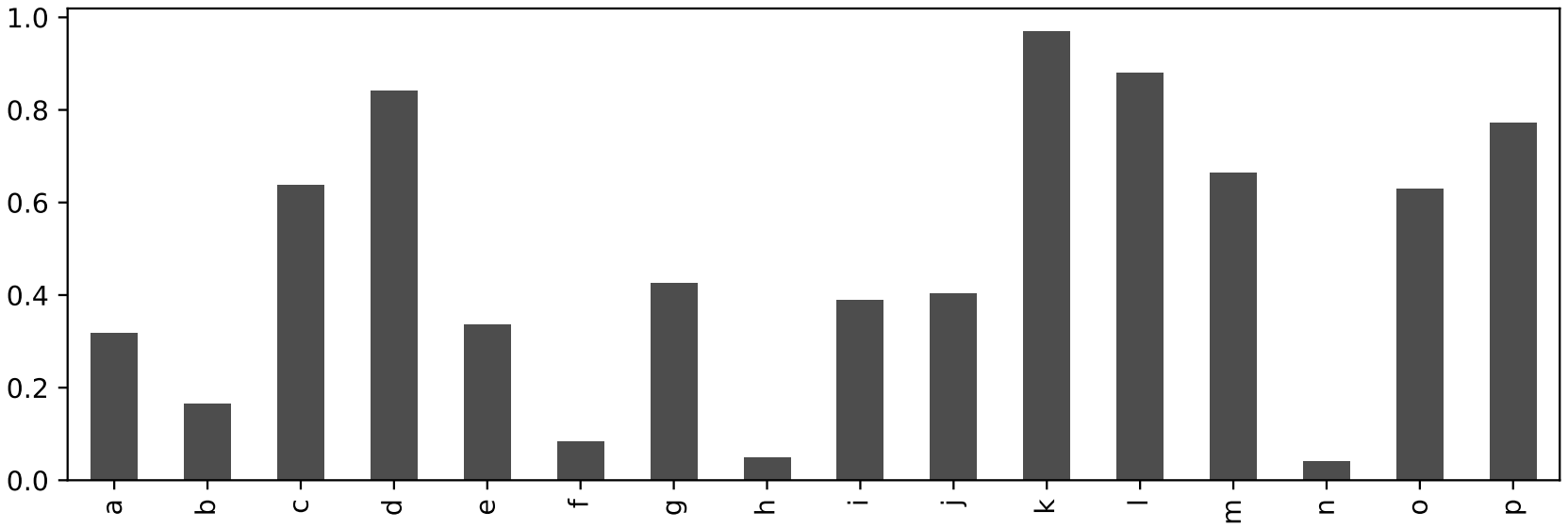
Fuller list of arguments here

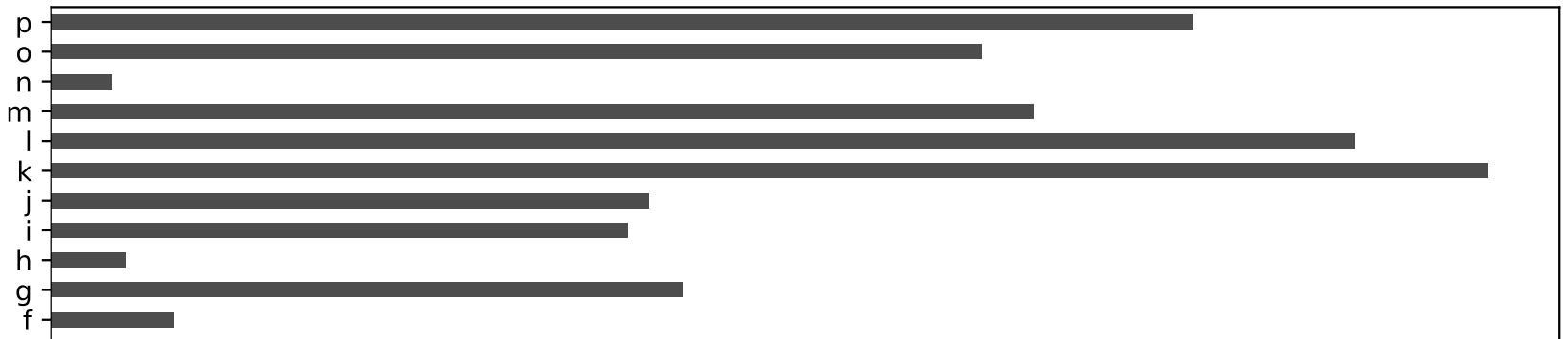
```
df.plot(subplots=True, layout = (2,2), sharex=True, sharey=Tru
```



Bar Charts

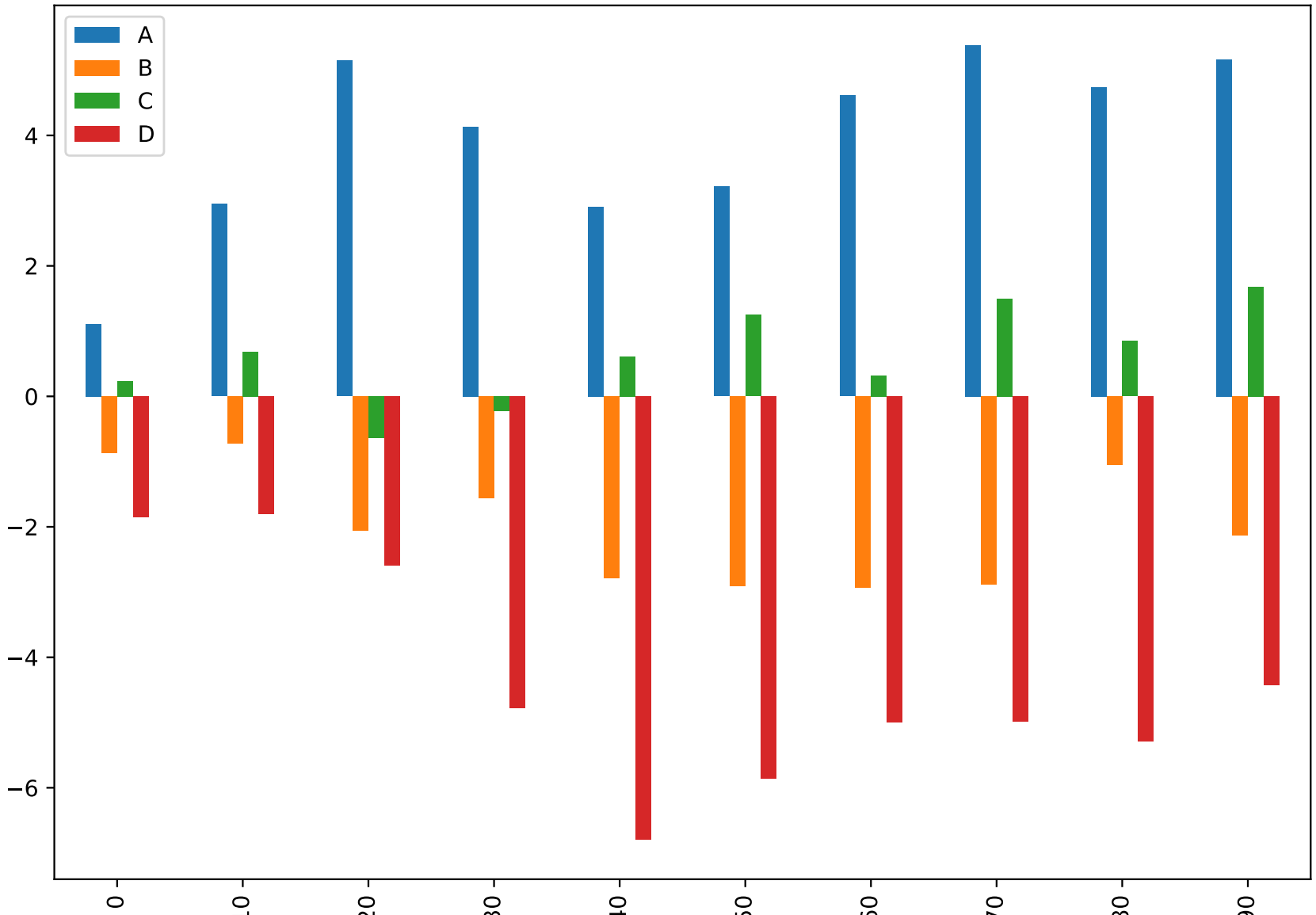
```
fig, axes = plt.subplots(2, 1)
data = pd.Series(np.random.uniform(size=16), index=list("abcd
data.plot.bar(ax=axes[0], color="black", alpha=0.7)
data.plot.barh(ax=axes[1], color="black", alpha=0.7);
```





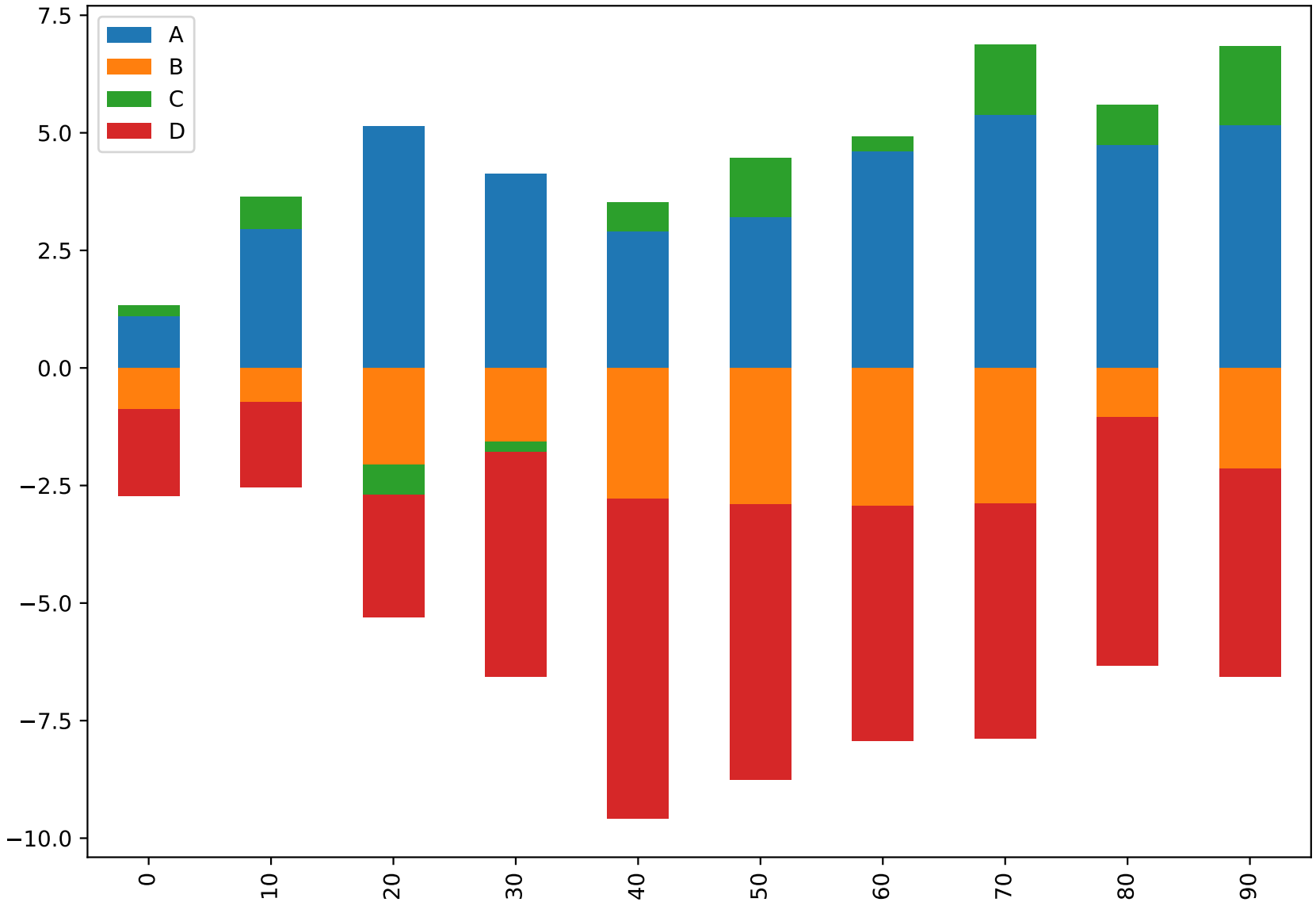
DataFrame Bar Chart

```
df.plot.bar()
```



DataFrame Stacked Bar Chart

```
df.plot.bar(stacked=True)
```



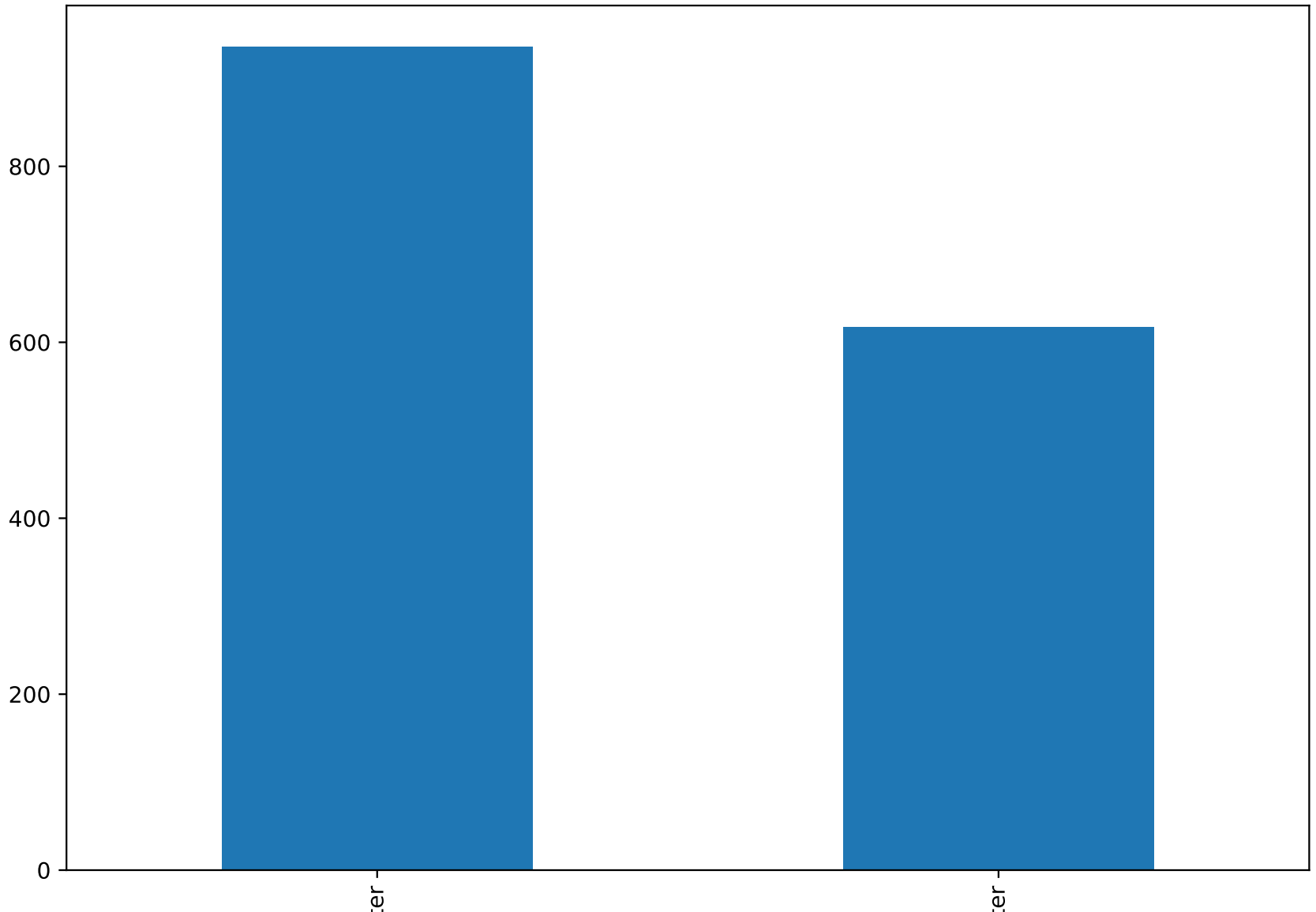
Application: NBA Shots

```
shots = pd.read_csv('https://lukashager.netlify.app/econ-481/
shots.head(3)
```

	Unnamed: 0.2	Unnamed: 0.1	Unnamed: 0	match_id	shotX
0	0	0	0	202203210BRK	14.8
1	1	1	1	202203210BRK	24.8
2	2	2	2	202203210BRK	20.0

Plot Types of Shots

```
shots['shot_type'].value_counts().plot.bar()
```



Count Types of Shots by Make

```
counts = shots[['shot_type', 'made']].value_counts()
counts
```

```
shot_type  made
2-pointer  True     526
           False    410
3-pointer  False    400
           True     217
Name: count, dtype: int64
```

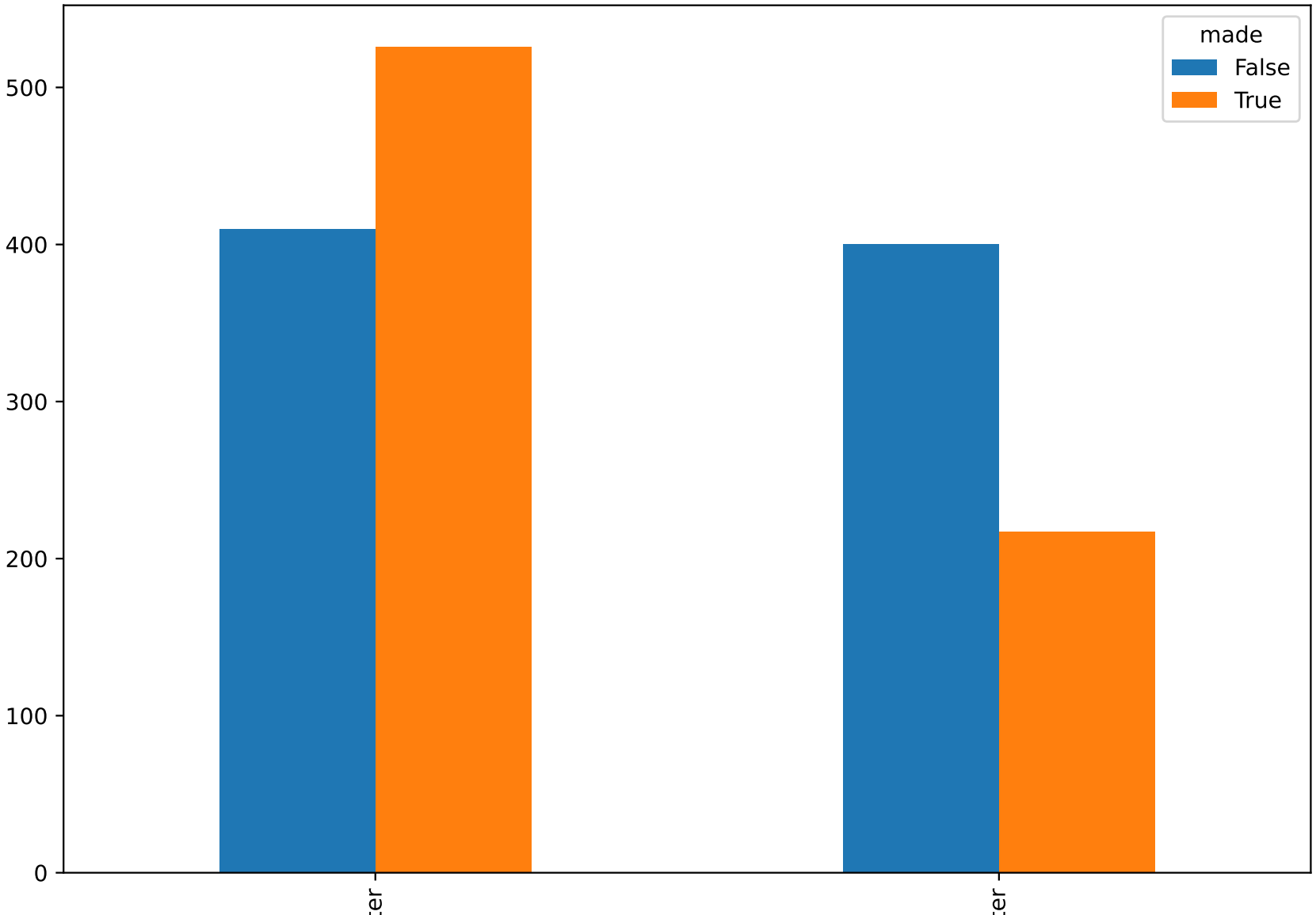
Create a DataFrame

```
counts_df = pd.DataFrame(counts)
df = counts_df.reset_index(level = 1)\
    .pivot(columns='made', values=counts_df.columns[0])
df
```

made shot_type	False	True
2-pointer	410	526
3-pointer	400	217

Plot

```
df.plot.bar()
```



Better Route: `unstack`

```
unstacked = counts.unstack()
unstacked
```

made shot_type	False	True
2-pointer	410	526
3-pointer	400	217

Better Route: `unstack`

```
unstacked = counts.unstack()
unstacked
```

made shot_type	False	True
2-pointer	410	526
3-pointer	400	217

```
unstacked.plot.bar()
```

Generate Expected Points

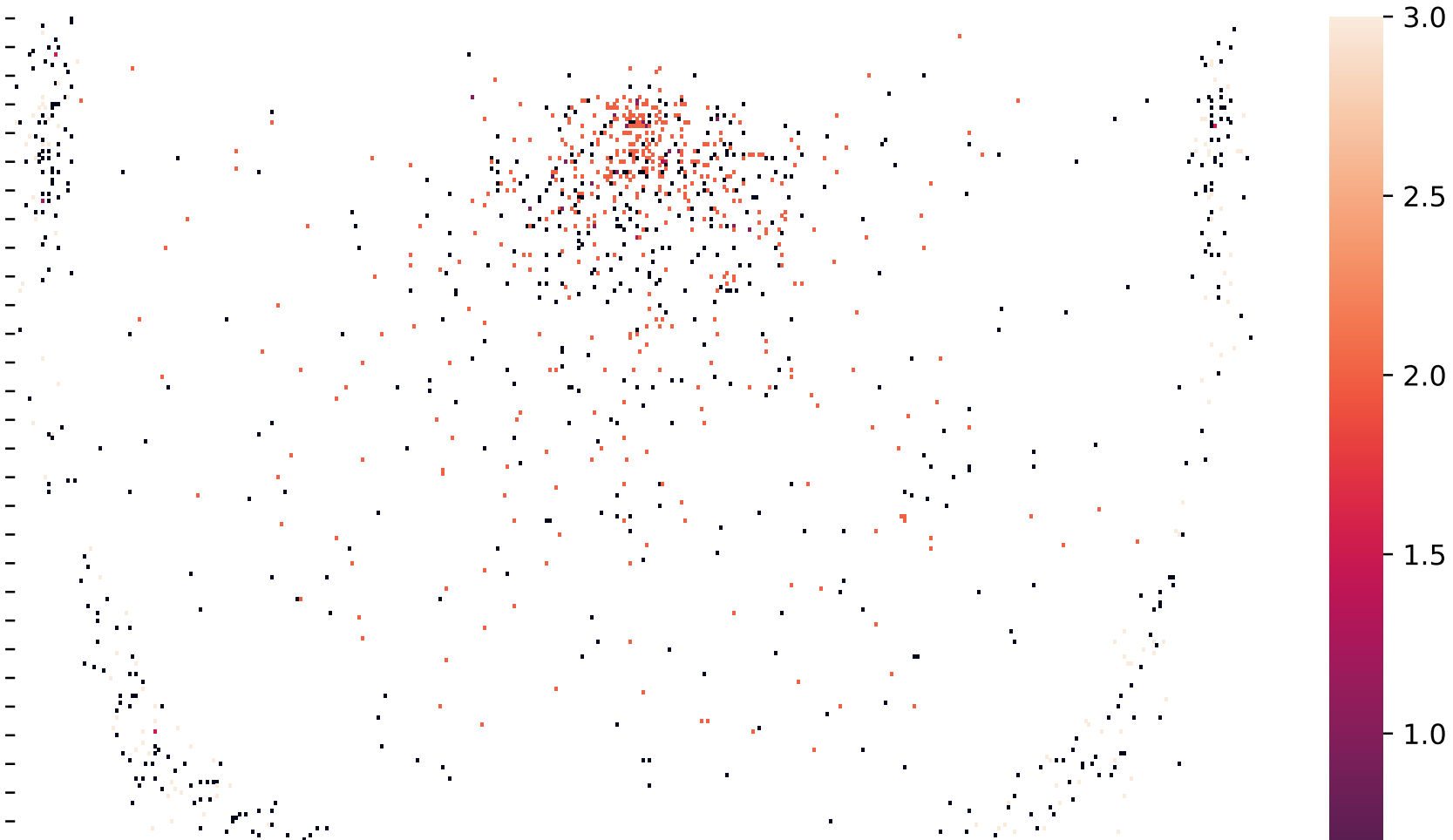
Let's create a heatmap of expected points by location on the floor:

```
pts = shots
pts['points'] = pts['made'] * pts['shot_type'].str.get(0).asty
exp_pts_df = pts[['shotX', 'shotY', 'points']].groupby(['shotY
    .agg('mean').unstack()
exp_pts_df.head(3)
```

	points									
shotX	-0.3	-0.1	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
shotY										
-0.3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
0.3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
0.5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0	NaN

Heatmap

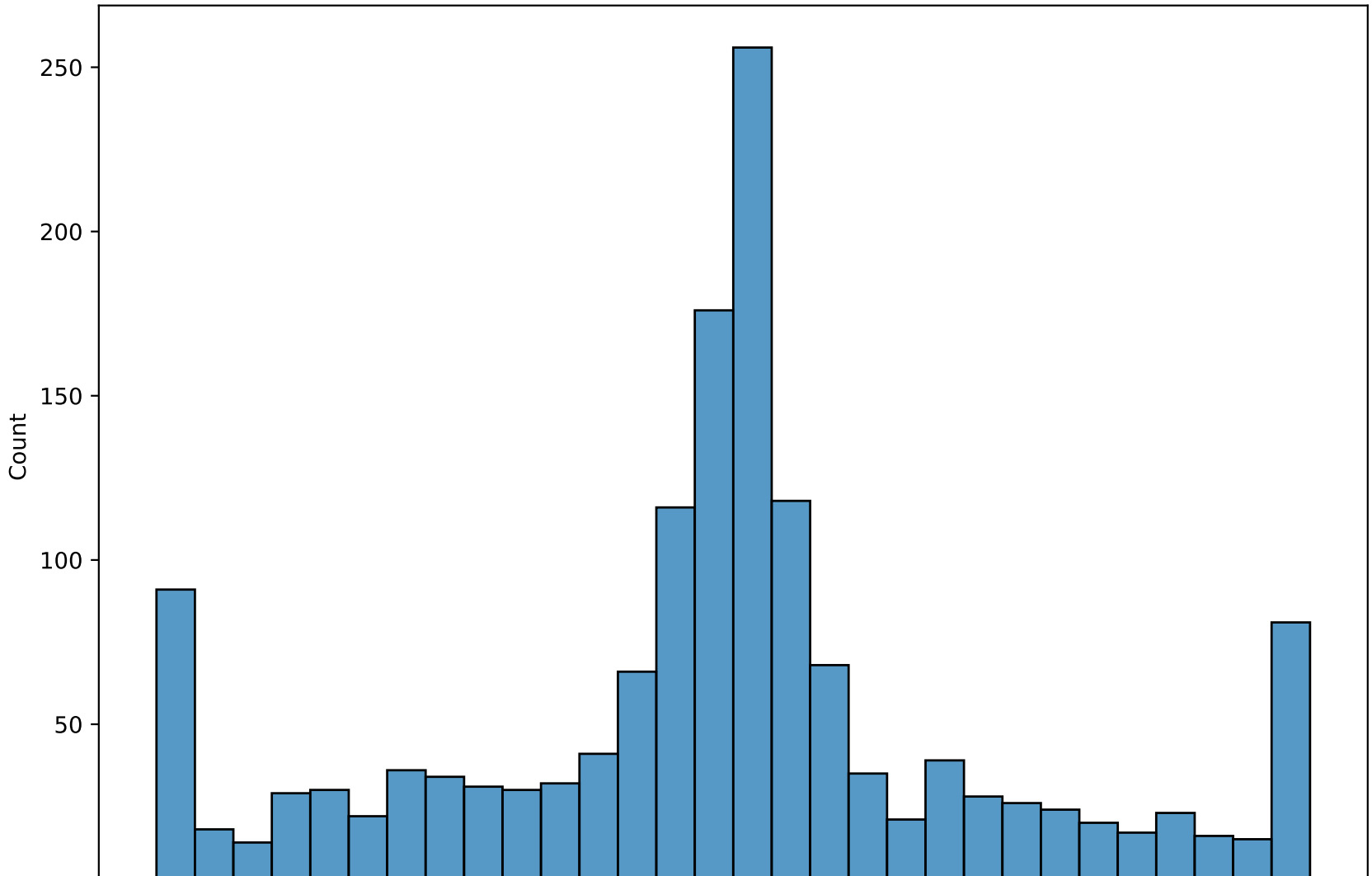
```
import seaborn as sns
sns.heatmap(exp_pts_df).set(
    xticklabels=[],yticklabels=[],xlabel=None,ylabel=None
);
```



Histogram

Note that `pd.Series` objects have a `plot.hist()` method

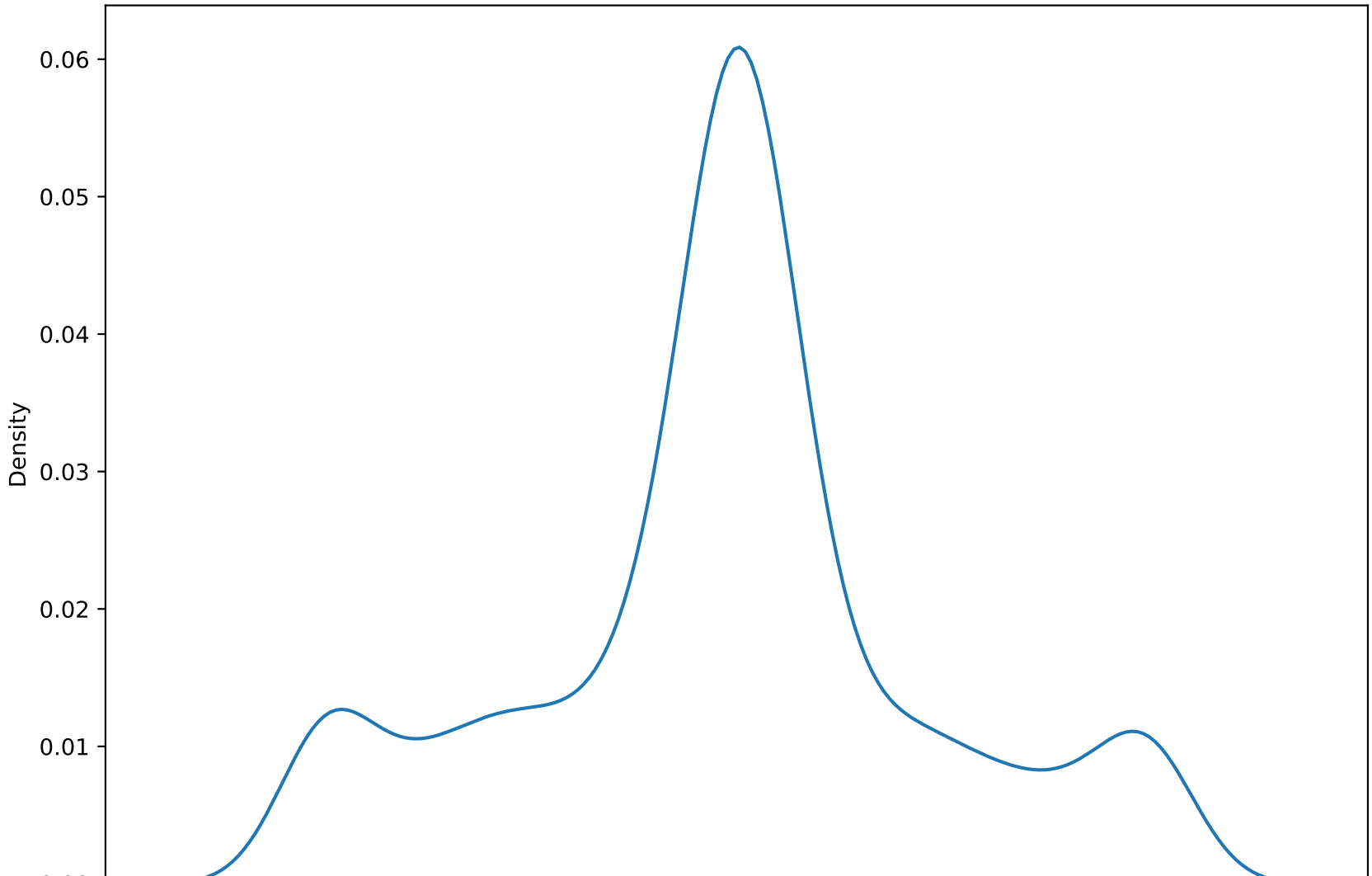
```
sns.histplot(shots['shotX'])
```



Density

Note that `pd.Series` objects have a `plot.density()` method

```
sns.kdeplot(shots['shotX'])
```



Appendix: Cross-Validation

Machine Learning Model Production

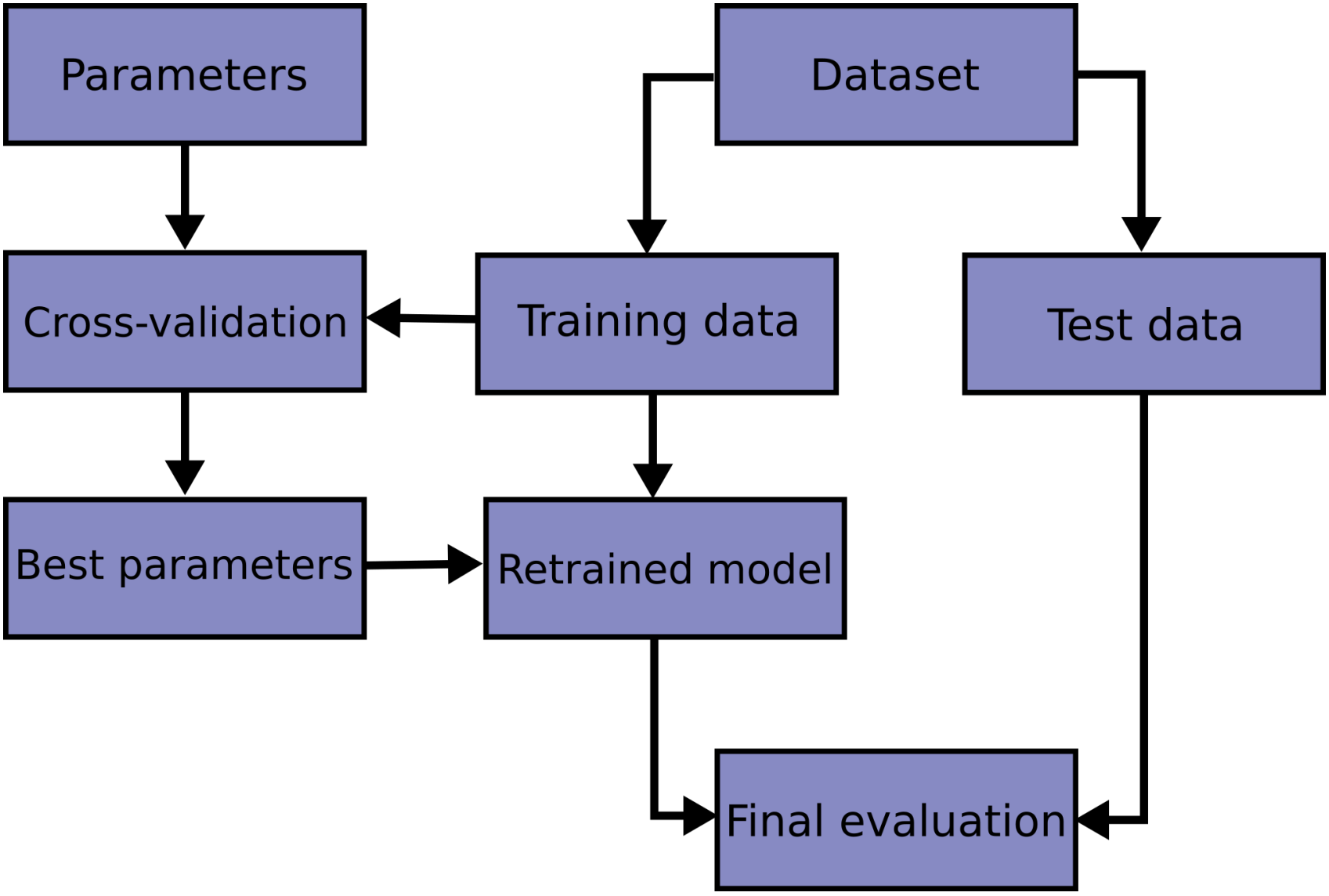


Figure 1: Courtesy of the `scikit-learn` documentation

K-Fold Cross-Validation

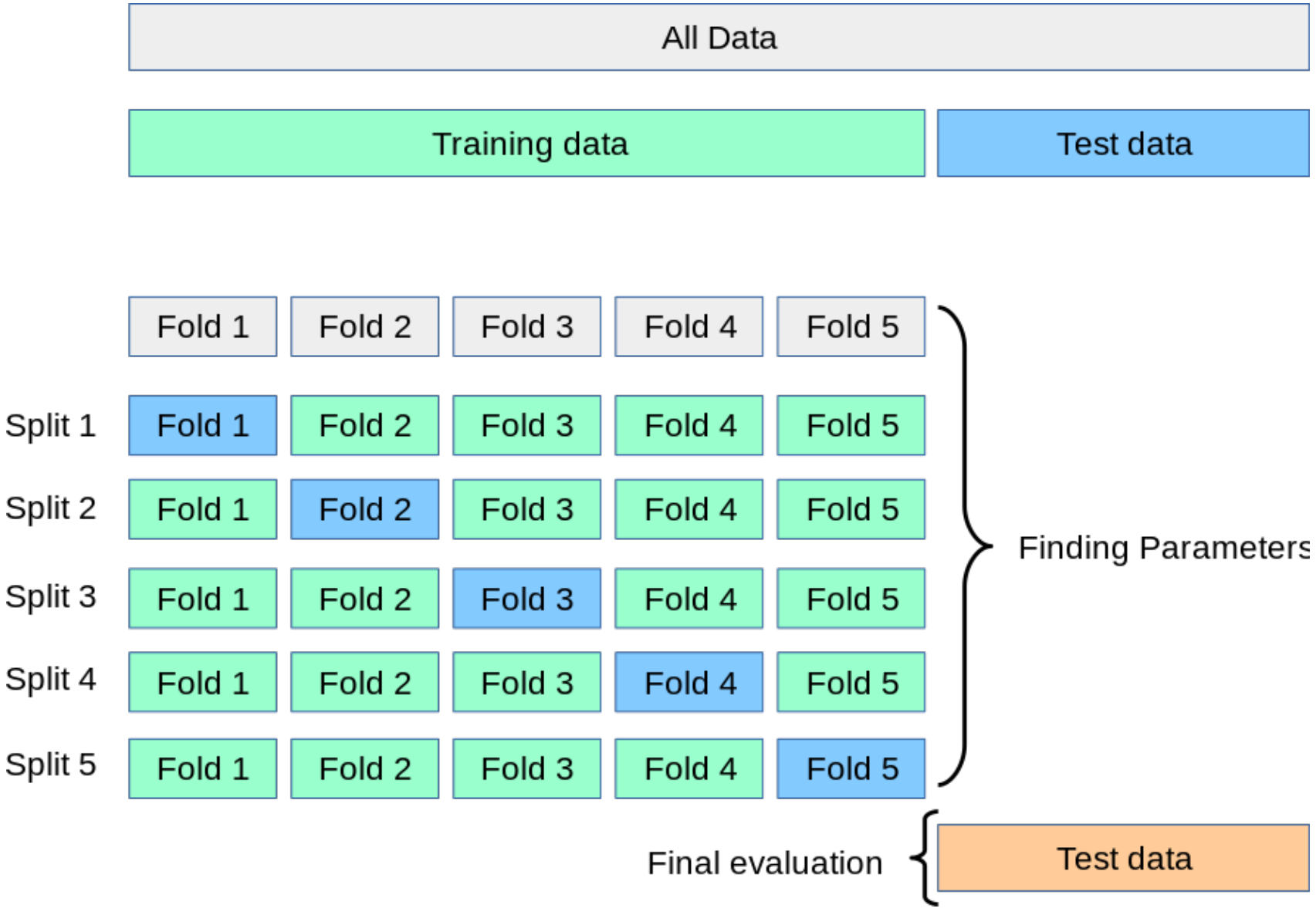


Figure 2: Courtesy of the `scikit-learn` documentation

Logic of K-Fold Cross-Validation

- In prediction problems, we care about how the model performs out-of-sample
- For a given parameter (or combination of parameters), we can run k models, where each is trained on $(k-1)/k$ of the data and makes out-of-sample predictions on $1/k$ of the data
- We then average performance across all k models for each parameter and pick the parameter that does the best
- This will help us prevent overfitting to our training data

Pseudo-Code for K-Fold Cross-Validation

1. Split dataset into training data and validation data
2. Split the training data into k folds
3. For each parameter:
    a. Train k models, where each is trained on all but one of the folds
    b. Make predictions on the unseen fold for each model
    c. Average the scoring metric (e.g. MSE) across all models
4. Choose the parameter that minimizes the scoring metric
5. Evaluate final model on validation data

Appendix: LASSO

LASSO

- "Least Absolute Squares and Shrinkage Operator"
- Two primary uses:
  - Prediction: if we have many regressors, OLS will overfit the data
  - Variable Selection: LASSO will "choose" the most useful regressors, depending on the regularization penalty
- We want to pick the regularization penalty α to make the best out-of-sample predictions

LASSO Estimator

β_{LASSO} solves the problem

$$\min_{\hat{\beta}} \underbrace{\sum_{i=1}^{n} \left(y - \mathbb{X}\hat{\beta}\right)^2}_{\text{Sum of Squared Errors}} + \underbrace{\alpha|\hat{\beta}|}_{\text{Regularization Penalty}}$$

- For smaller α, more coefficients have nonzero value, the prediction function is more complex
- For larger α, more coefficients are zero, the prediction function is less complex
- Pick optimal α via cross-validation