7_Exercise_Examine_real_world_data

October 16, 2021

1 Exploring data with Python - real world data

Last time, we looked at grades for our student data, and investigated this visually with histograms and box plots. Now we will look into more complex cases, describe the data more fully, and discuss how to make basic comparisons between data.

1.0.1 Real world data distributions

Last time, we looked at grades for our student data, and estimated from this sample what the full population of grades might look like. Just to refresh, lets take a look at this data again.

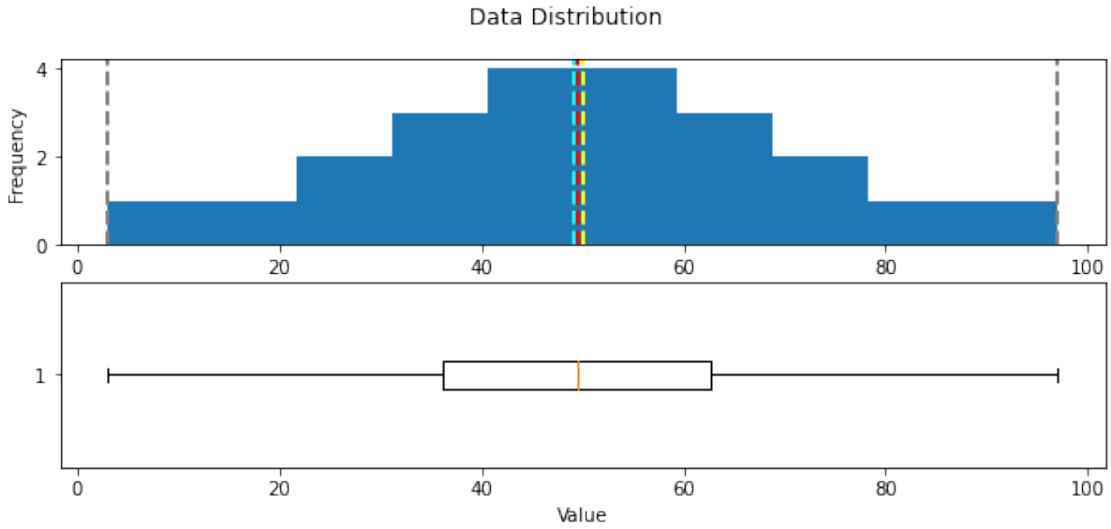
Run the code below to print out the data and make a histogram + boxplot that show the grades for our sample of students.

[ ]:
```
import pandas as pd
from matplotlib import pyplot as plt

# Load data from a text file
#!wget https://raw.githubusercontent.com/MicrosoftDocs/
 ↪mslearn-introduction-to-machine-learning/main/Data/ml-basics/grades.csv
df_students = pd.read_csv('grades.csv',delimiter=',',header='infer')

# Remove any rows with missing data
df_students = df_students.dropna(axis=0, how='any')

# Calculate who passed, assuming '60' is the grade needed to pass
passes  = pd.Series(df_students['Grade'] >= 60)

# Save who passed to the Pandas dataframe
df_students = pd.concat([df_students, passes.rename("Pass")], axis=1)


# Print the result out into this notebook
print(df_students)


# Create a function that we can re-use
def show_distribution(var_data):
    '''
```

```
    This function will make a distribution (graph) and display it
    '''

    # Get statistics
    min_val = var_data.min()
    max_val = var_data.max()
    mean_val = var_data.mean()
    med_val = var_data.median()
    mod_val = var_data.mode()[0]

    print('Minimum:{:.2f}\nMean:{:.2f}\nMedian:{:.2f}\nMode:{:.2f}\nMaximum:{:.
↪2f}\n'.format(min_val,
                                                                              ␣
↪               mean_val,
                                                                              ␣
↪               med_val,
                                                                              ␣
↪               mod_val,
                                                                              ␣
↪               max_val))

    # Create a figure for 2 subplots (2 rows, 1 column)
    fig, ax = plt.subplots(2, 1, figsize = (10,4))

    # Plot the histogram
    ax[0].hist(var_data)
    ax[0].set_ylabel('Frequency')

    # Add lines for the mean, median, and mode
    ax[0].axvline(x=min_val, color = 'gray', linestyle='dashed', linewidth = 2)
    ax[0].axvline(x=mean_val, color = 'cyan', linestyle='dashed', linewidth = 2)
    ax[0].axvline(x=med_val, color = 'red', linestyle='dashed', linewidth = 2)
    ax[0].axvline(x=mod_val, color = 'yellow', linestyle='dashed', linewidth =␣
↪2)
    ax[0].axvline(x=max_val, color = 'gray', linestyle='dashed', linewidth = 2)

    # Plot the boxplot
    ax[1].boxplot(var_data, vert=False)
    ax[1].set_xlabel('Value')

    # Add a title to the Figure
    fig.suptitle('Data Distribution')

    # Show the figure
    fig.show()
```

```
show_distribution(df_students['Grade'])
```

```
         Name  StudyHours  Grade   Pass
0         Dan       10.00   50.0  False
1       Joann       11.50   50.0  False
2       Pedro        9.00   47.0  False
3       Rosie       16.00   97.0   True
4       Ethan        9.25   49.0  False
5       Vicky        1.00    3.0  False
6    Frederic       11.50   53.0  False
7      Jimmie        9.00   42.0  False
8      Rhonda        8.50   26.0  False
9    Giovanni       14.50   74.0   True
10  Francesca       15.50   82.0   True
11      Rajab       13.75   62.0   True
12    Naiyana        9.00   37.0  False
13       Kian        8.00   15.0  False
14      Jenny       15.50   70.0   True
15     Jakeem        8.00   27.0  False
16     Helena        9.00   36.0  False
17      Ismat        6.00   35.0  False
18      Anila       10.00   48.0  False
19       Skye       12.00   52.0  False
20     Daniel       12.50   63.0   True
21      Aisha       12.00   64.0   True
Minimum:3.00
Mean:49.18
Median:49.50
Mode:50.00
Maximum:97.00


C:\Users\aduzo\Anaconda3\lib\site-packages\ipykernel_launcher.py:63:
UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
```

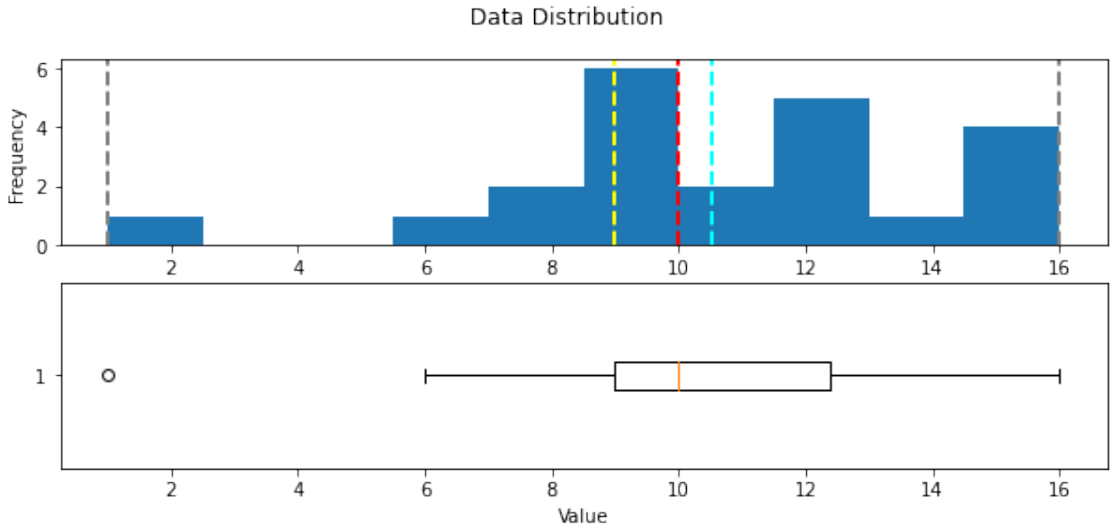


As you might recall, our data had the mean and mode at the center, with data spread symmetrically from there.

Now let's take a look at the distribution of the study hours data.

[ ]:
```
# Get the variable to examine
col = df_students['StudyHours']
# Call the function
show_distribution(col)
```

```
Minimum:1.00
Mean:10.52
Median:10.00
Mode:9.00
Maximum:16.00


C:\Users\aduzo\Anaconda3\lib\site-packages\ipykernel_launcher.py:63:
UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
```



StudyHour 1.0 at index 5 is an outlier.

The distribution of the study time data is significantly different from that of the grades.

Note that the whiskers of the box plot only begin at around 6.0, indicating that the vast majority of the first quarter of the data is above this value. The minimum is marked with an **o**, indicating that it is statistically an *outlier* - a value that lies significantly outside the range of the rest of the distribution.

Outliers can occur for many reasons. Maybe a student meant to record "10" hours of study time, but entered "1" and missed the "0". Or maybe the student was abnormally lazy when it comes to studying! Either way, it's a statistical anomaly that doesn't represent a typical student. Let's see what the distribution looks like without it.

[ ]:
```
df_students[df_students.StudyHours>1]
```

[ ]:

	Name	StudyHours	Grade	Pass
0	Dan	10.00	50.0	False
1	Joann	11.50	50.0	False
2	Pedro	9.00	47.0	False
3	Rosie	16.00	97.0	True
4	Ethan	9.25	49.0	False
6	Frederic	11.50	53.0	False
7	Jimmie	9.00	42.0	False
8	Rhonda	8.50	26.0	False
9	Giovanni	14.50	74.0	True
10	Francesca	15.50	82.0	True
11	Rajab	13.75	62.0	True
12	Naiyana	9.00	37.0	False
13	Kian	8.00	15.0	False
14	Jenny	15.50	70.0	True

```
15     Jakeem        8.00   27.0  False
16     Helena        9.00   36.0  False
17      Ismat        6.00   35.0  False
18      Anila       10.00   48.0  False
19       Skye       12.00   52.0  False
20     Daniel       12.50   63.0   True
21      Aisha       12.00   64.0   True
```

[ ]:
```
type(df_students[df_students.StudyHours>1])
```

[ ]:
```
pandas.core.frame.DataFrame
```

notice vicky that is index number 5 is abscent

[ ]:
```
df_students[df_students.StudyHours>1]['StudyHours']
```

[ ]:
```
0     10.00
1     11.50
2      9.00
3     16.00
4      9.25
6     11.50
7      9.00
8      8.50
9     14.50
10    15.50
11    13.75
12     9.00
13     8.00
14    15.50
15     8.00
16     9.00
17     6.00
18    10.00
19    12.00
20    12.50
21    12.00
Name: StudyHours, dtype: float64
```

[ ]:
```
type(df_students[df_students.StudyHours>1]['StudyHours'])
```

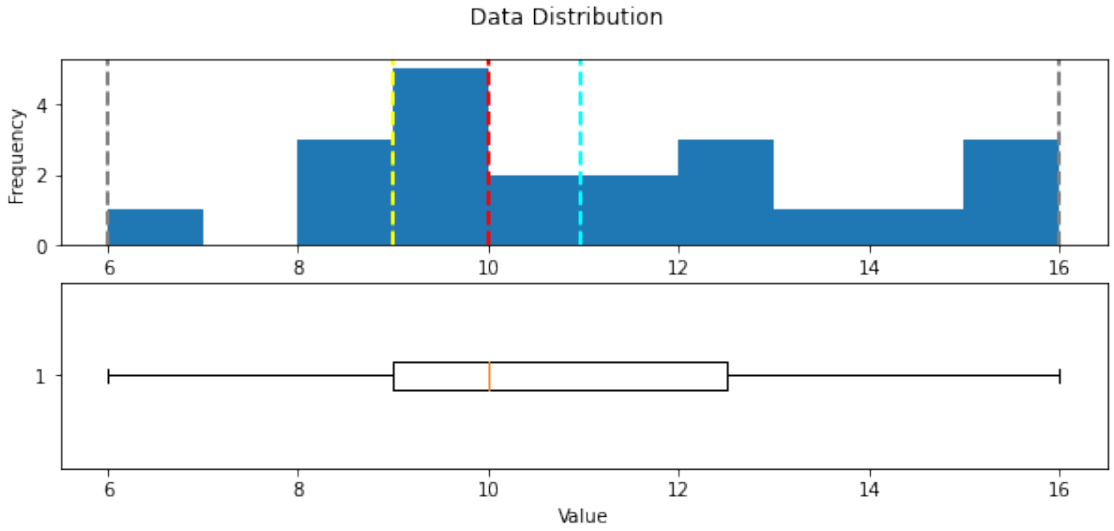
[ ]:
```
pandas.core.series.Series
```

[ ]:
```
# Get the variable to examine
# We will only get students who have studied more than one hour
col = df_students[df_students.StudyHours>1]['StudyHours']

# Call the function
```

```
show_distribution(col)
```

```
Minimum:6.00
Mean:10.98
Median:10.00
Mode:9.00
Maximum:16.00
```

```
C:\Users\aduzo\Anaconda3\lib\site-packages\ipykernel_launcher.py:63:
UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
```



Notice the x axis, which is value now start from 6.

For learning purposes we have just treated the value **1** is a true outlier here and excluded it. In the real world, though, it would be unusual to exclude data at the extremes without more justification when our sample size is so small. This is because the smaller our sample size, the more likely it is that our sampling is a bad representation of the whole population (here, the population means grades for all students, not just our 22). For example, if we sampled study time for another 1000 students, we might find that it's actually quite common to not study much!

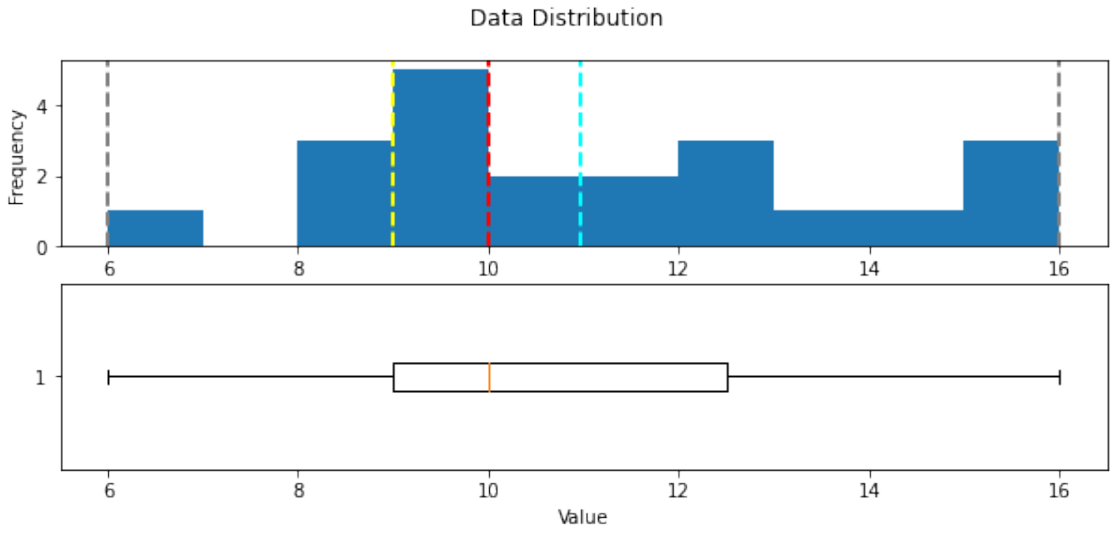
When we have more data available, our sample becomes more reliable. This makes it easier to consider outliers as being values that fall below or above percentiles within which most of the data lie. For example, the following code uses the Pandas **quantile** function to exclude observations below the 0.01th percentile (the value above which 99% of the data reside).

[ ]:
```
# calculate the 0.01th percentile
q01 = df_students.StudyHours.quantile(0.01)
q01
```

[ ]: 2.05

[ ]:
```
# Get the variable to examine
col = df_students[df_students.StudyHours >q01]['StudyHours']
# Call the function
show_distribution(col)
```

```
Minimum:6.00
Mean:10.98
Median:10.00
Mode:9.00
Maximum:16.00


C:\Users\aduzo\Anaconda3\lib\site-packages\ipykernel_launcher.py:63:
UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
```

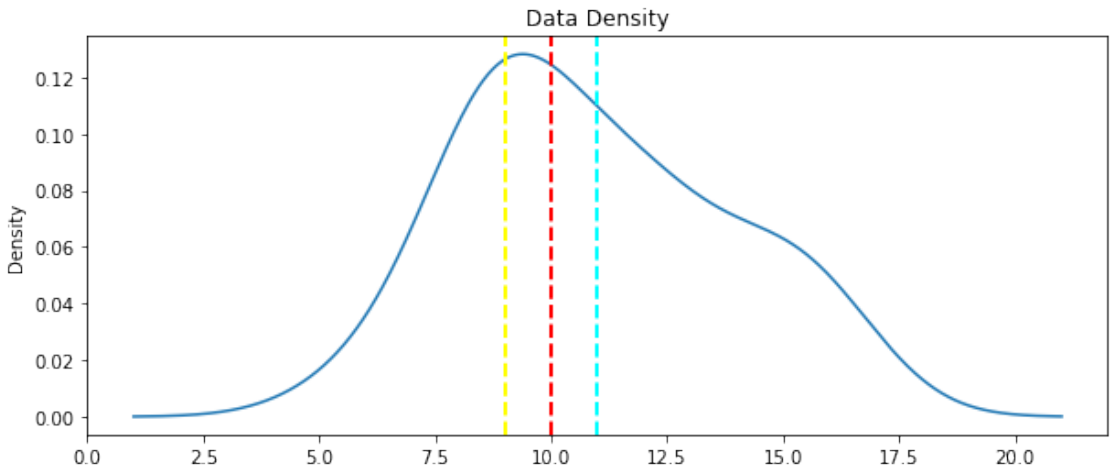


> **Tip**: You can also eliminate outliers at the upper end of the distribution by defining a threshold at a high percentile value - for example, you could use the **quantile** function to find the 0.99 percentile below which 99% of the data reside.

With the outliers removed, the box plot shows all data within the four quartiles. Note that the distribution is not symmetric like it is for the grade data though - there are some students with very high study times of around 16 hours, but the bulk of the data is between 7 and 13 hours; The few extremely high values pull the mean towards the higher end of the scale.

Let's look at the density for this distribution.

```
[ ]: def show_density(var_data):
        fig = plt.figure(figsize=(10,4))

        # Plot density
        var_data.plot.density()

        # Add titles and labels
        plt.title('Data Density')

        # Show the mean, median, and mode
        plt.axvline(x=var_data.mean(), color = 'cyan', linestyle='dashed',␣
     ↪linewidth = 2)
        plt.axvline(x=var_data.median(), color = 'red', linestyle='dashed',␣
     ↪linewidth = 2)
        plt.axvline(x=var_data.mode()[0], color = 'yellow', linestyle='dashed',␣
     ↪linewidth = 2)

        # Show the figure
        plt.show()

    # Get the density of StudyHours
    show_density(col)
```



This kind of distribution is called *right skewed*. The mass of the data is on the left side of the distribution, creating a long tail to the right because of the values at the extreme high end; which pull the mean to the right.

Measures of variance So now we have a good idea where the middle of the grade and study hours data distributions are. However, there's another aspect of the distributions we should examine: how much variability is there in the data?

Typical statistics that measure variability in the data include:

- **Range**: The difference between the maximum and minimum. There's no built-in function for this, but it's easy to calculate using the **min** and **max** functions.
- **Variance**: The average of the squared difference from the mean. You can use the built-in **var** function to find this.
- **Standard Deviation**: The square root of the variance. You can use the built-in **std** function to find this.

[ ]:
```
for col_name in ['Grade', 'StudyHours']:
    col = df_students[col_name]
    rng = col.max() - col.min()
    var = col.var()
    std = col.std()
    print('\n{}:\n - Range: {:.2f}\n - Variance: {:.2f}\n - Std.Dev: {:.2f}'.
 ↪format(col_name, rng, var, std))
```

```
Grade:
 - Range: 94.00
 - Variance: 472.54
 - Std.Dev: 21.74

StudyHours:
 - Range: 15.00
 - Variance: 12.16
 - Std.Dev: 3.49
```

Of these statistics, the standard deviation is generally the most useful. It provides a measure of variance in the data on the same scale as the data itself (so grade points for the Grade distribution and hours for the StudyHours distribution). The higher the standard deviation, the more variance there is when comparing values in the distribution to the distribution mean - in other words, the data is more spread out.

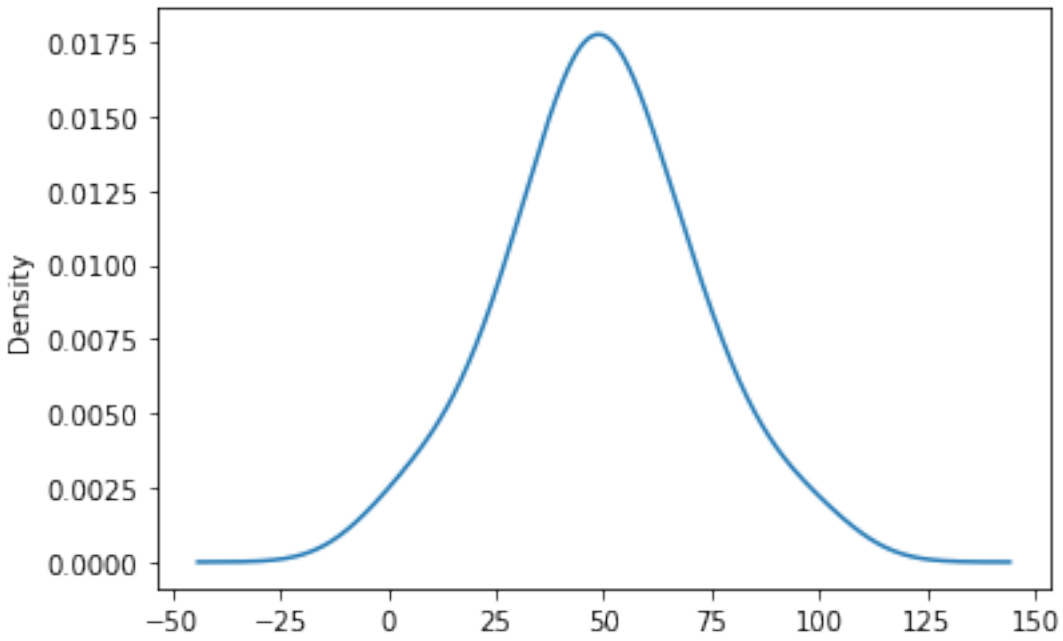
When working with a *normal* distribution, the standard deviation works with the particular characteristics of a normal distribution to provide even greater insight. Run the cell below to see the relationship between standard deviations and the data in the normal distribution.

[ ]:
```
import scipy.stats as stats

# Get the Grade column
col = df_students['Grade']

#get the density
density = stats.gaussian_kde(col)
density
```

[ ]:
```
<scipy.stats.kde.gaussian_kde at 0x1a502762188>
```

[ ]:
```
# Plot the density
col.plot.density()
```

[ ]: <AxesSubplot:ylabel='Density'>



[ ]:
```
# Get the mean and standard deviation
s= col.std()
m = col.mean()

# Annotate 1 stdev
x1 =[m-s, m+s]
print('mean: {:.3f} \nstandard deviation: {:.3f} \ninterval: [{:.3f},{:.3f}]'.
 ↪format(m,s,m-s,m+s))
```

```
mean: 49.182
standard deviation: 21.738
interval: [27.444,70.920]
```

[ ]:
```
help(density)
```

```
Help on gaussian_kde in module scipy.stats.kde object:

class gaussian_kde(builtins.object)
 |  gaussian_kde(dataset, bw_method=None, weights=None)
 |
 |  Representation of a kernel-density estimate using Gaussian kernels.
 |
```

```
|  Kernel density estimation is a way to estimate the probability density
|  function (PDF) of a random variable in a non-parametric way.
|  `gaussian_kde` works for both uni-variate and multi-variate data.   It
|  includes automatic bandwidth determination.  The estimation works best for
|  a unimodal distribution; bimodal or multi-modal distributions tend to be
|  oversmoothed.
|
|  Parameters
|  ----------
|  dataset : array_like
|      Datapoints to estimate from. In case of univariate data this is a 1-D
|      array, otherwise a 2-D array with shape (# of dims, # of data).
|  bw_method : str, scalar or callable, optional
|      The method used to calculate the estimator bandwidth.  This can be
|      'scott', 'silverman', a scalar constant or a callable.  If a scalar,
|      this will be used directly as `kde.factor`.  If a callable, it should
|      take a `gaussian_kde` instance as only parameter and return a scalar.
|      If None (default), 'scott' is used.  See Notes for more details.
|  weights : array_like, optional
|      weights of datapoints. This must be the same shape as dataset.
|      If None (default), the samples are assumed to be equally weighted
|
|  Attributes
|  ----------
|  dataset : ndarray
|      The dataset with which `gaussian_kde` was initialized.
|  d : int
|      Number of dimensions.
|  n : int
|      Number of datapoints.
|  neff : int
|      Effective number of datapoints.
|
|      .. versionadded:: 1.2.0
|  factor : float
|      The bandwidth factor, obtained from `kde.covariance_factor`, with which
|      the covariance matrix is multiplied.
|  covariance : ndarray
|      The covariance matrix of `dataset`, scaled by the calculated bandwidth
|      (`kde.factor`).
|  inv_cov : ndarray
|      The inverse of `covariance`.
|
|  Methods
|  -------
|  evaluate
|  __call__
|  integrate_gaussian
```

```
|  integrate_box_1d
|  integrate_box
|  integrate_kde
|  pdf
|  logpdf
|  resample
|  set_bandwidth
|  covariance_factor
|
|  Notes
|  -----
|  Bandwidth selection strongly influences the estimate obtained from the KDE
|  (much more so than the actual shape of the kernel).  Bandwidth selection
|  can be done by a "rule of thumb", by cross-validation, by "plug-in
|  methods" or by other means; see [3]_, [4]_ for reviews.  `gaussian_kde`
|  uses a rule of thumb, the default is Scott's Rule.
|
|  Scott's Rule [1]_, implemented as `scotts_factor`, is::
|
|      n**(-1./(d+4)),
|
|  with ``n`` the number of data points and ``d`` the number of dimensions.
|  In the case of unequally weighted points, `scotts_factor` becomes::
|
|      neff**(-1./(d+4)),
|
|  with ``neff`` the effective number of datapoints.
|  Silverman's Rule [2]_, implemented as `silverman_factor`, is::
|
|      (n * (d + 2) / 4.)**(-1. / (d + 4)).
|
|  or in the case of unequally weighted points::
|
|      (neff * (d + 2) / 4.)**(-1. / (d + 4)).
|
|  Good general descriptions of kernel density estimation can be found in [1]_
|  and [2]_, the mathematics for this multi-dimensional implementation can be
|  found in [1]_.
|
|  With a set of weighted samples, the effective number of datapoints ``neff``
|  is defined by::
|
|      neff = sum(weights)^2 / sum(weights^2)
|
|  as detailed in [5]_.
|
|  References
|  ----------
```

```
|  .. [1] D.W. Scott, "Multivariate Density Estimation: Theory, Practice, and
|         Visualization", John Wiley & Sons, New York, Chicester, 1992.
|  .. [2] B.W. Silverman, "Density Estimation for Statistics and Data
|         Analysis", Vol. 26, Monographs on Statistics and Applied Probability,
|         Chapman and Hall, London, 1986.
|  .. [3] B.A. Turlach, "Bandwidth Selection in Kernel Density Estimation: A
|         Review", CORE and Institut de Statistique, Vol. 19, pp. 1-33, 1993.
|  .. [4] D.M. Bashtannyk and R.J. Hyndman, "Bandwidth selection for kernel
|         conditional density estimation", Computational Statistics & Data
|         Analysis, Vol. 36, pp. 279-298, 2001.
|  .. [5] Gray P. G., 1969, Journal of the Royal Statistical Society.
|         Series A (General), 132, 272
|
|  Examples
|  --------
|  Generate some random two-dimensional data:
|
|  >>> from scipy import stats
|  >>> def measure(n):
|  …     "Measurement model, return two coupled measurements."
|  …     m1 = np.random.normal(size=n)
|  …     m2 = np.random.normal(scale=0.5, size=n)
|  …     return m1+m2, m1-m2
|
|  >>> m1, m2 = measure(2000)
|  >>> xmin = m1.min()
|  >>> xmax = m1.max()
|  >>> ymin = m2.min()
|  >>> ymax = m2.max()
|
|  Perform a kernel density estimate on the data:
|
|  >>> X, Y = np.mgrid[xmin:xmax:100j, ymin:ymax:100j]
|  >>> positions = np.vstack([X.ravel(), Y.ravel()])
|  >>> values = np.vstack([m1, m2])
|  >>> kernel = stats.gaussian_kde(values)
|  >>> Z = np.reshape(kernel(positions).T, X.shape)
|
|  Plot the results:
|
|  >>> import matplotlib.pyplot as plt
|  >>> fig, ax = plt.subplots()
|  >>> ax.imshow(np.rot90(Z), cmap=plt.cm.gist_earth_r,
|  …           extent=[xmin, xmax, ymin, ymax])
|  >>> ax.plot(m1, m2, 'k.', markersize=2)
|  >>> ax.set_xlim([xmin, xmax])
|  >>> ax.set_ylim([ymin, ymax])
|  >>> plt.show()
```

```
|
|  Methods defined here:
|
|  __call__ = evaluate(self, points)
|
|  __init__(self, dataset, bw_method=None, weights=None)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  covariance_factor = scotts_factor(self)
|
|  evaluate(self, points)
|      Evaluate the estimated pdf on a set of points.
|
|      Parameters
|      ----------
|      points : (# of dimensions, # of points)-array
|          Alternatively, a (# of dimensions,) vector can be passed in and
|          treated as a single point.
|
|      Returns
|      -------
|      values : (# of points,)-array
|          The values at each point.
|
|      Raises
|      ------
|      ValueError : if the dimensionality of the input points is different than
|                   the dimensionality of the KDE.
|
|  integrate_box(self, low_bounds, high_bounds, maxpts=None)
|      Computes the integral of a pdf over a rectangular interval.
|
|      Parameters
|      ----------
|      low_bounds : array_like
|          A 1-D array containing the lower bounds of integration.
|      high_bounds : array_like
|          A 1-D array containing the upper bounds of integration.
|      maxpts : int, optional
|          The maximum number of points to use for integration.
|
|      Returns
|      -------
|      value : scalar
|          The result of the integral.
|
|  integrate_box_1d(self, low, high)
|      Computes the integral of a 1D pdf between two bounds.
```

```
|
|      Parameters
|      ----------
|      low : scalar
|          Lower bound of integration.
|      high : scalar
|          Upper bound of integration.
|
|      Returns
|      -------
|      value : scalar
|          The result of the integral.
|
|      Raises
|      ------
|      ValueError
|          If the KDE is over more than one dimension.
|
|  integrate_gaussian(self, mean, cov)
|      Multiply estimated density by a multivariate Gaussian and integrate
|      over the whole space.
|
|      Parameters
|      ----------
|      mean : aray_like
|          A 1-D array, specifying the mean of the Gaussian.
|      cov : array_like
|          A 2-D array, specifying the covariance matrix of the Gaussian.
|
|      Returns
|      -------
|      result : scalar
|          The value of the integral.
|
|      Raises
|      ------
|      ValueError
|          If the mean or covariance of the input Gaussian differs from
|          the KDE's dimensionality.
|
|  integrate_kde(self, other)
|      Computes the integral of the product of this  kernel density estimate
|      with another.
|
|      Parameters
|      ----------
|      other : gaussian_kde instance
|          The other kde.
```

```
|
|      Returns
|      -------
|      value : scalar
|          The result of the integral.
|
|      Raises
|      ------
|      ValueError
|          If the KDEs have different dimensionality.
|
|  logpdf(self, x)
|      Evaluate the log of the estimated pdf on a provided set of points.
|
|  pdf(self, x)
|      Evaluate the estimated pdf on a provided set of points.
|
|      Notes
|      -----
|      This is an alias for `gaussian_kde.evaluate`.  See the ``evaluate``
|      docstring for more details.
|
|  resample(self, size=None, seed=None)
|      Randomly sample a dataset from the estimated pdf.
|
|      Parameters
|      ----------
|      size : int, optional
|          The number of samples to draw.  If not provided, then the size is
|          the same as the effective number of samples in the underlying
|          dataset.
|      seed : None or int or `np.random.RandomState`, optional
|          If `seed` is None, random variates are drawn by the RandomState
|          singleton used by np.random.
|          If `seed` is an int, a new `np.random.RandomState` instance is used,
|          seeded with seed.
|          If `seed` is already a `np.random.RandomState instance`, then that
|          `np.random.RandomState` instance is used.
|          Specify `seed` for reproducible drawing of random variates.
|
|      Returns
|      -------
|      resample : (self.d, `size`) ndarray
|          The sampled dataset.
|
|  scotts_factor(self)
|      Computes the coefficient (`kde.factor`) that
|      multiplies the data covariance matrix to obtain the kernel covariance
```

```
|      matrix. The default is `scotts_factor`.  A subclass can overwrite this
|      method to provide a different method, or set it through a call to
|      `kde.set_bandwidth`.
|
|  set_bandwidth(self, bw_method=None)
|      Compute the estimator bandwidth with given method.
|
|      The new bandwidth calculated after a call to `set_bandwidth` is used
|      for subsequent evaluations of the estimated density.
|
|      Parameters
|      ----------
|      bw_method : str, scalar or callable, optional
|          The method used to calculate the estimator bandwidth.  This can be
|          'scott', 'silverman', a scalar constant or a callable.  If a
|          scalar, this will be used directly as `kde.factor`.  If a callable,
|          it should take a `gaussian_kde` instance as only parameter and
|          return a scalar.  If None (default), nothing happens; the current
|          `kde.covariance_factor` method is kept.
|
|      Notes
|      -----
|      .. versionadded:: 0.11
|
|      Examples
|      --------
|      >>> import scipy.stats as stats
|      >>> x1 = np.array([-7, -5, 1, 4, 5.])
|      >>> kde = stats.gaussian_kde(x1)
|      >>> xs = np.linspace(-10, 10, num=50)
|      >>> y1 = kde(xs)
|      >>> kde.set_bandwidth(bw_method='silverman')
|      >>> y2 = kde(xs)
|      >>> kde.set_bandwidth(bw_method=kde.factor / 3.)
|      >>> y3 = kde(xs)
|
|      >>> import matplotlib.pyplot as plt
|      >>> fig, ax = plt.subplots()
|      >>> ax.plot(x1, np.full(x1.shape, 1 / (4. * x1.size)), 'bo',
|      …         label='Data points (rescaled)')
|      >>> ax.plot(xs, y1, label='Scott (default)')
|      >>> ax.plot(xs, y2, label='Silverman')
|      >>> ax.plot(xs, y3, label='Const (1/3 * Silverman)')
|      >>> ax.legend()
|      >>> plt.show()
|
|  silverman_factor(self)
|      Compute the Silverman factor.
```

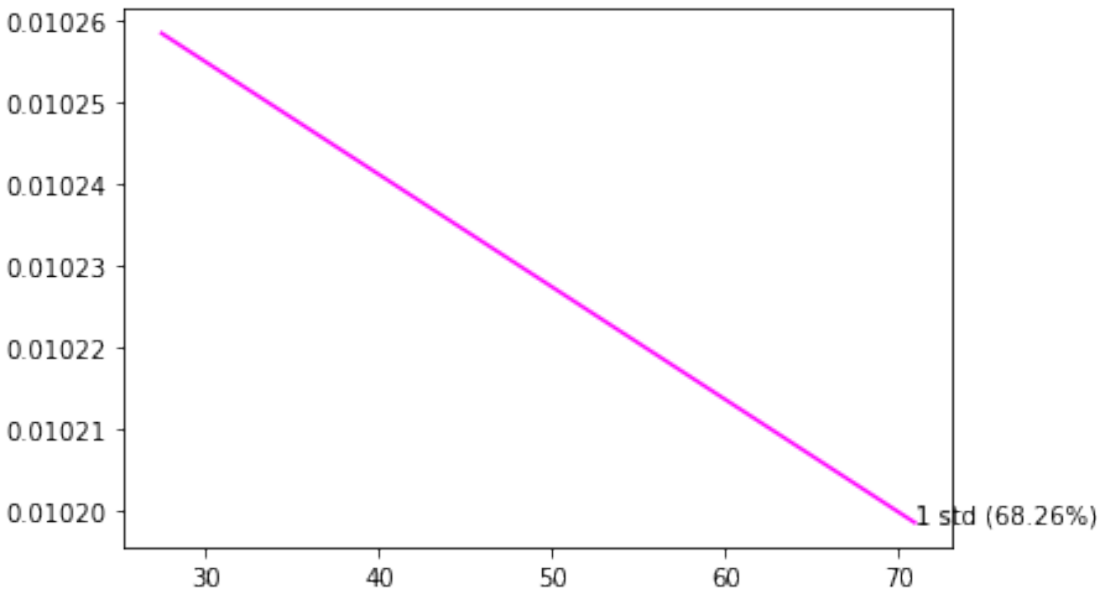
```
|
|      Returns
|      -------
|      s : float
|          The silverman factor.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  neff
|
|  weights
```

[ ]:
```
# probability density function(PDF) given mean and std
y1 = density(x1)
y1
```

[ ]:
```
array([0.01025848, 0.01019866])
```

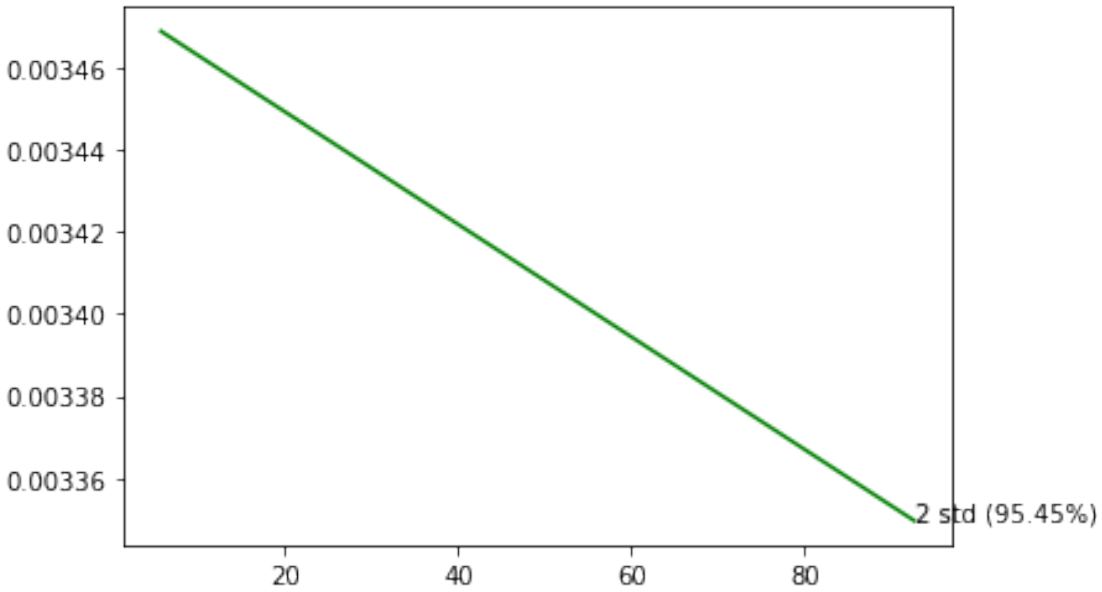
[ ]:
```
plt.plot(x1,y1, color='magenta')
plt.annotate('1 std (68.26%)', (x1[1],y1[1]))
```

[ ]:
```
Text(70.91972968538256, 0.01019866148226649, '1 std (68.26%)')
```



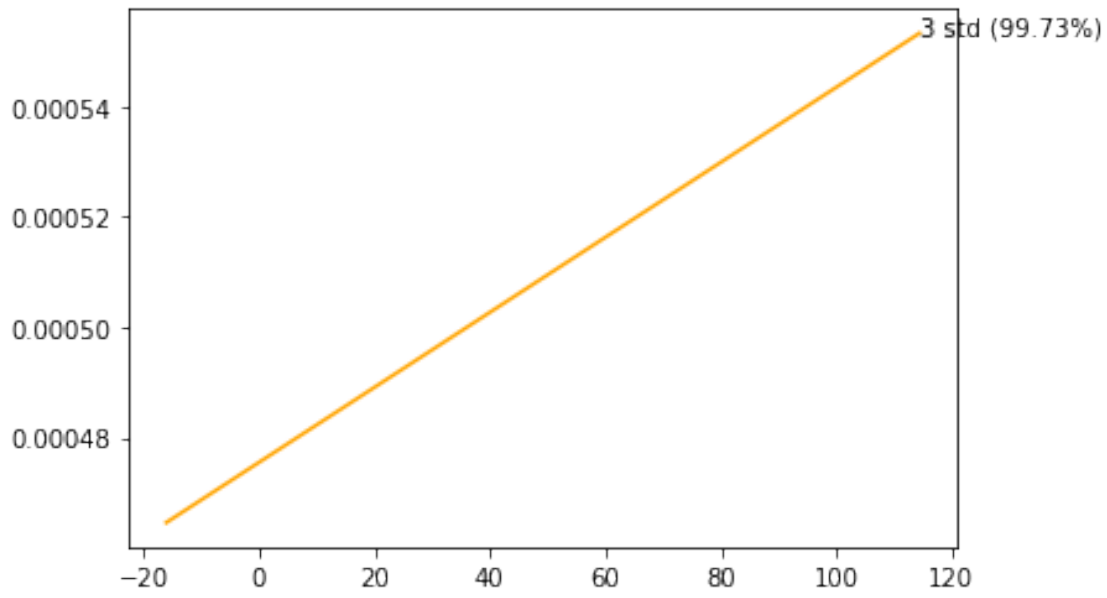
[ ]:
```
# Annotate 2 stdevs
x2 = [m-(s*2), m+(s*2)]
y2 = density(x2)
plt.plot(x2,y2, color='green')
plt.annotate('2 std (95.45%)', (x2[1],y2[1]))
```

[ ]:
```
Text(92.65764118894694, 0.003349755490270128, '2 std (95.45%)')
```



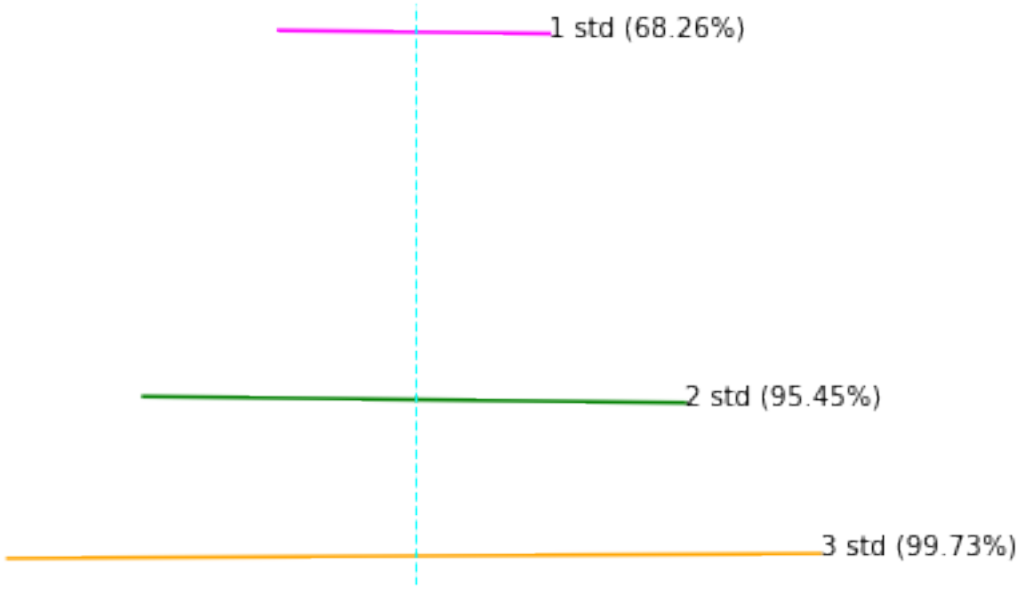
[ ]:
```
# Annotate 3 stdevs
x3 = [m-(s*3), m+(s*3)]
y3 = density(x3)
plt.plot(x3,y3, color='orange')
plt.annotate('3 std (99.73%)', (x3[1],y3[1]))
```

[ ]:
```
Text(114.39555269251133, 0.0005533522860635102, '3 std (99.73%)')
```



[ ]:
```
# Show the location of the mean
plt.axvline(col.mean(), color='cyan', linestyle='dashed', linewidth=1)

plt.axis('off')

plt.show()
```

[ ]:
```
# Annotate 1 stdev
x1 = [m-s, m+s]
y1 = density(x1)
plt.plot(x1,y1, color='magenta')
plt.annotate('1 std (68.26%)', (x1[1],y1[1]))

# Annotate 2 stdevs
x2 = [m-(s*2), m+(s*2)]
y2 = density(x2)
plt.plot(x2,y2, color='green')
plt.annotate('2 std (95.45%)', (x2[1],y2[1]))

# Annotate 3 stdevs
x3 = [m-(s*3), m+(s*3)]
y3 = density(x3)
plt.plot(x3,y3, color='orange')
plt.annotate('3 std (99.73%)', (x3[1],y3[1]))

# Show the location of the mean
plt.axvline(col.mean(), color='cyan', linestyle='dashed', linewidth=1)

plt.axis('off')

plt.show()
```



The horizontal lines show the percentage of data within 1, 2, and 3 standard deviations of the mean (plus or minus).

In any normal distribution: - Approximately 68.26% of values fall within one standard deviation from the mean. - Approximately 95.45% of values fall within two standard deviations from the mean. - Approximately 99.73% of values fall within three standard deviations from the mean.

So, since we know that the mean grade is 49.18, the standard deviation is 21.74, and distribution of grades is approximately normal; we can calculate that 68.26% of students should achieve a grade between 27.44 and 70.92.

The descriptive statistics we've used to understand the distribution of the student data variables are the basis of statistical analysis; and because they're such an important part of exploring your data, there's a built-in **Describe** method of the DataFrame object that returns the main descriptive statistics for all numeric columns.

[ ]:
```
df_students.describe()
```

[ ]:

	StudyHours	Grade
count	22.000000	22.000000
mean	10.522727	49.181818
std	3.487144	21.737912
min	1.000000	3.000000
25%	9.000000	36.250000
50%	10.000000	49.500000
75%	12.375000	62.750000
max	16.000000	97.000000

1.1 Comparing data

Now that you know something about the statistical distribution of the data in your dataset, you're ready to examine your data to identify any apparent relationships between variables.

First of all, let's get rid of any rows that contain outliers so that we have a sample that is representative of a typical class of students. We identified that the StudyHours column contains some outliers with extremely low values, so we'll remove those rows.

[ ]:
```
df_sample = df_students[df_students['StudyHours']>1]
df_sample
```

[ ]:

	Name	StudyHours	Grade	Pass
0	Dan	10.00	50.0	False
1	Joann	11.50	50.0	False
2	Pedro	9.00	47.0	False
3	Rosie	16.00	97.0	True
4	Ethan	9.25	49.0	False
6	Frederic	11.50	53.0	False
7	Jimmie	9.00	42.0	False
8	Rhonda	8.50	26.0	False
9	Giovanni	14.50	74.0	True
10	Francesca	15.50	82.0	True
11	Rajab	13.75	62.0	True
12	Naiyana	9.00	37.0	False

13	Kian	8.00	15.0	False
14	Jenny	15.50	70.0	True
15	Jakeem	8.00	27.0	False
16	Helena	9.00	36.0	False
17	Ismat	6.00	35.0	False
18	Anila	10.00	48.0	False
19	Skye	12.00	52.0	False
20	Daniel	12.50	63.0	True
21	Aisha	12.00	64.0	True

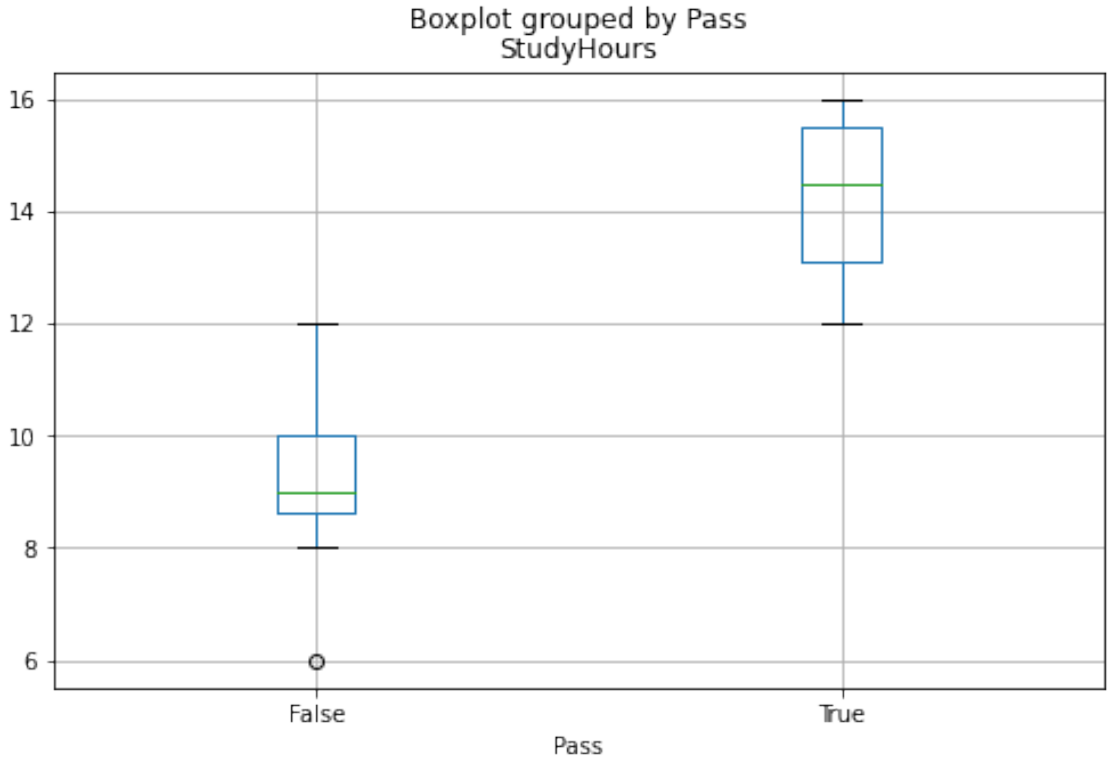
1.1.1 Comparing numeric and categorical variables

The data includes two *numeric* variables (**StudyHours** and **Grade**) and two *categorical* variables (**Name** and **Pass**). Let's start by comparing the numeric **StudyHours** column to the categorical **Pass** column to see if there's an apparent relationship between the number of hours studied and a passing grade.

To make this comparison, let's create box plots showing the distribution of StudyHours for each possible Pass value (true and false).

[ ]:
```
df_sample.boxplot(column='StudyHours', by='Pass', figsize=(8,5))
```

[ ]:
```
<AxesSubplot:title={'center':'StudyHours'}, xlabel='Pass'>
```



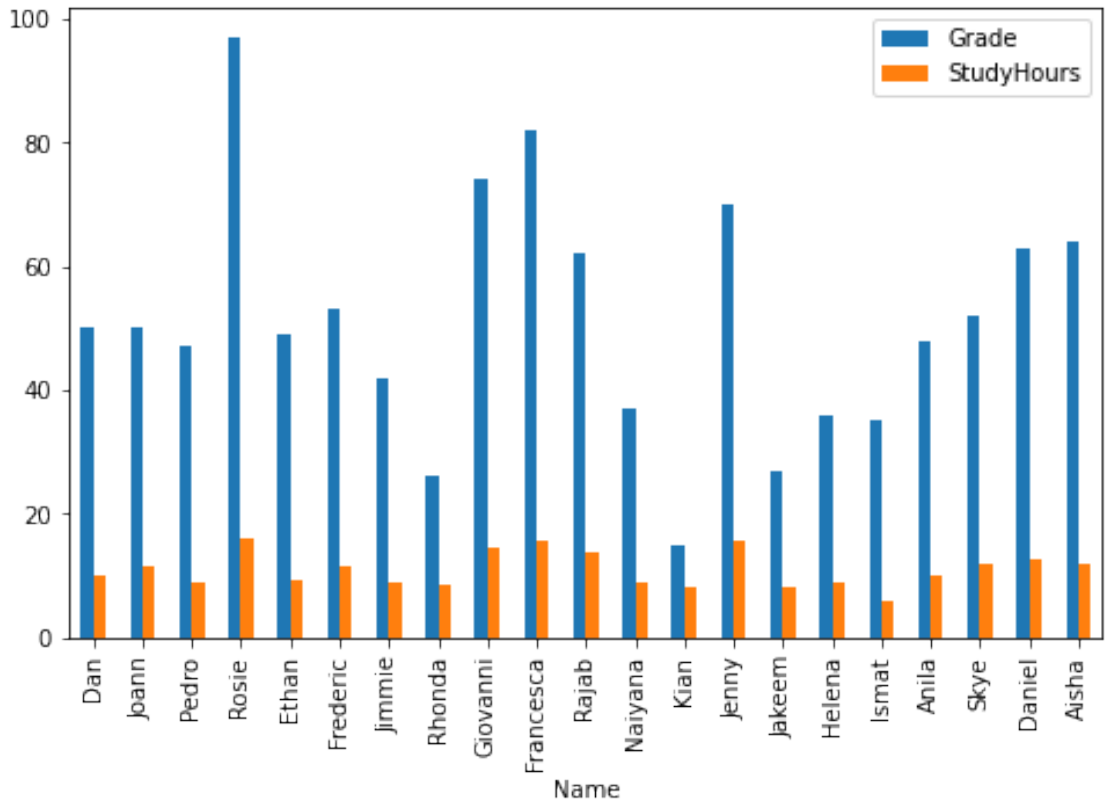
Comparing the StudyHours distributions, it's immediately apparent (if not particularly surprising) that students who passed the course tended to study for more hours than students who didn't. So if you wanted to predict whether or not a student is likely to pass the course, the amount of time they spend studying may be a good predictive feature.

1.1.2 Comparing numeric variables

Now let's compare two numeric variables. We'll start by creating a bar chart that shows both grade and study hours.

[ ]:
```
# Create a bar plot of name vs grade and study hours
df_sample.plot(x='Name', y=['Grade','StudyHours'], kind='bar', figsize=(8,5))
```

[ ]:
```
<AxesSubplot:xlabel='Name'>
```



The chart shows bars for both grade and study hours for each student; but it's not easy to compare because the values are on different scales. Grades are measured in grade points, and range from 3 to 97; while study time is measured in hours and ranges from 1 to 16.

A common technique when dealing with numeric data in different scales is to *normalize* the data so that the values retain their proportional distribution, but are measured on the same scale. To accomplish this, we'll use a technique called *MinMax* scaling that distributes the values proportionally

on a scale of 0 to 1. You could write the code to apply this transformation; but the **Scikit-Learn** library provides a scaler to do it for you.

[ ]:
```
df_sample[['Name','Grade','StudyHours']].head()
```

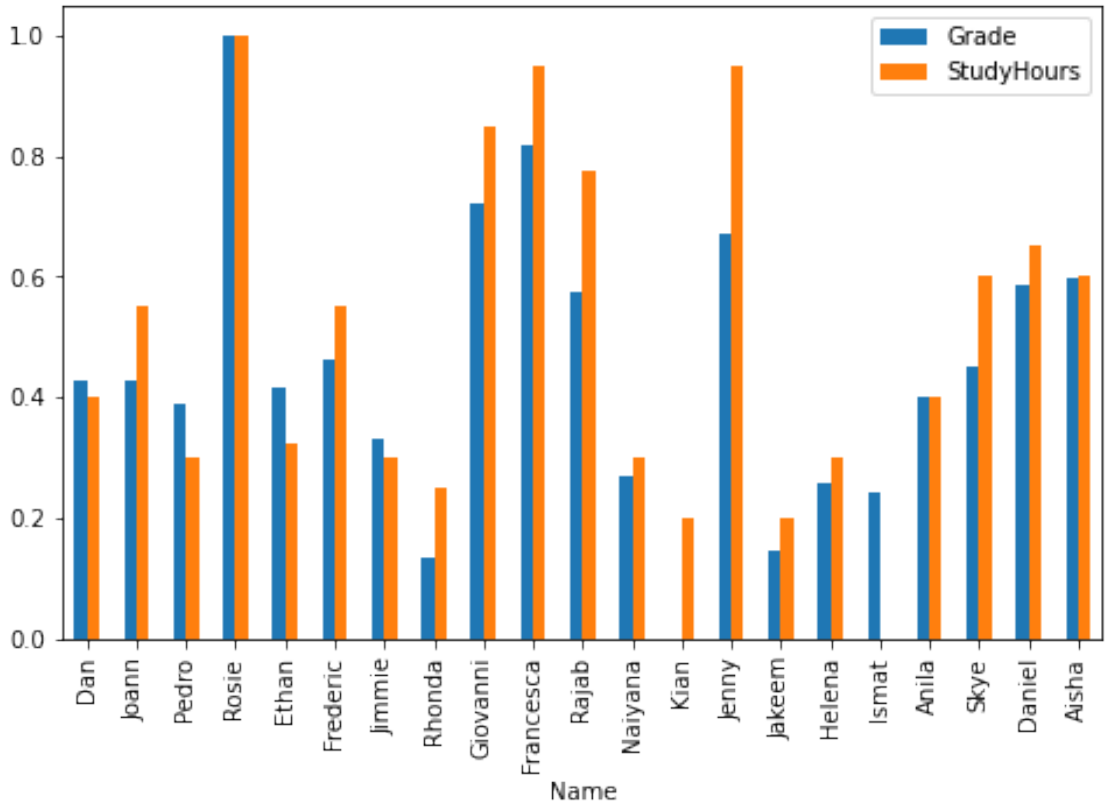
[ ]:
```
    Name  Grade  StudyHours
0    Dan   50.0       10.00
1  Joann   50.0       11.50
2  Pedro   47.0        9.00
3  Rosie   97.0       16.00
4  Ethan   49.0        9.25
```

[ ]:
```
from sklearn.preprocessing import MinMaxScaler

# Get a scaler object
scaler = MinMaxScaler()

# Create a new dataframe from the scaled values
df_normalized = df_sample[['Name','Grade','StudyHours']].copy()
df_normalized.head()
```

[ ]:
```
    Name  Grade  StudyHours
0    Dan   50.0       10.00
1  Joann   50.0       11.50
2  Pedro   47.0        9.00
3  Rosie   97.0       16.00
4  Ethan   49.0        9.25
```

[ ]:
```
# Normalize the numeric columns
df_normalized[['Grade','StudyHours']] = scaler.
 ↪fit_transform(df_normalized[['Grade','StudyHours']])

df_normalized.plot(x='Name', y=['Grade','StudyHours'],kind='bar', figsize=(8,5))
```

[ ]:
```
<AxesSubplot:xlabel='Name'>
```



With the data normalized, it's easier to see an apparent relationship between grade and study time. It's not an exact match, but it definitely seems like students with higher grades tend to have studied more.

So there seems to be a correlation between study time and grade; and in fact, there's a statistical *correlation* measurement we can use to quantify the relationship between these columns.

[ ]:
```
df_normalized.Grade.corr(df_normalized.StudyHours)
```

[ ]:
```
0.9117666413789677
```

The correlation statistic is a value between -1 and 1 that indicates the strength of a relationship. Values above 0 indicate a *positive* correlation (high values of one variable tend to coincide with high values of the other), while values below 0 indicate a *negative* correlation (high values of one variable tend to coincide with low values of the other). In this case, the correlation value is close to 1; showing a strongly positive correlation between study time and grade.

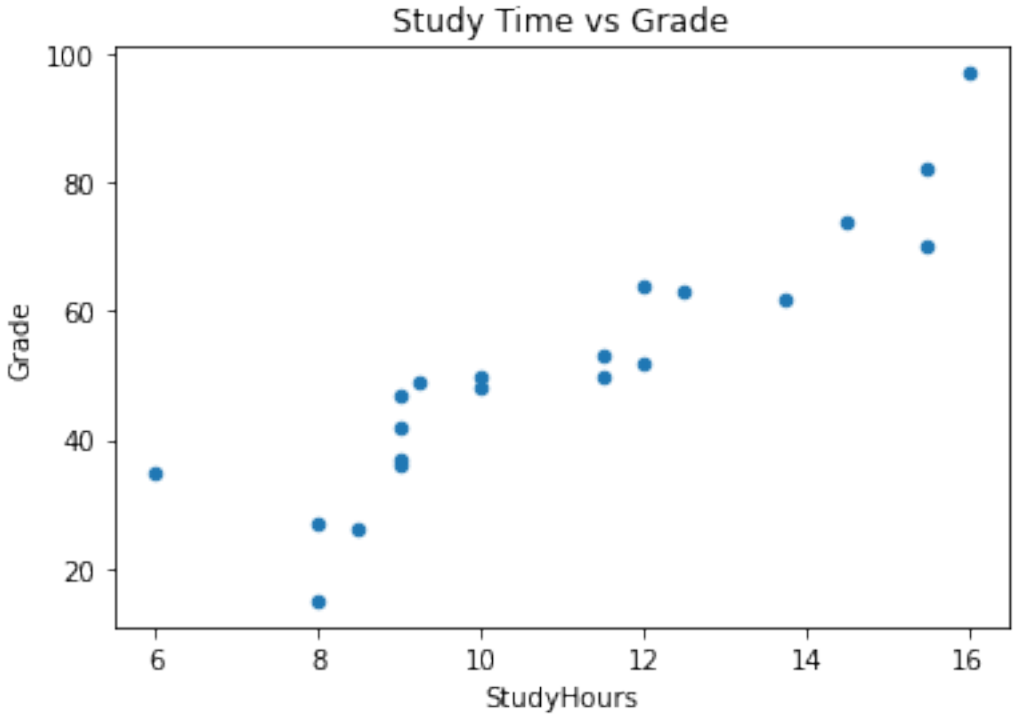
> **Note**: Data scientists often quote the maxim "*correlation* is not *causation*". In other words, as tempting as it might be, you shouldn't interpret the statistical correlation as explaining *why* one of the values is high. In the case of the student data, the statistics demonstrates that students with high grades tend to also have high amounts of study time; but this is not the same as proving that they achieved high grades *because* they studied a lot. The statistic could equally be used as evidence to support the nonsensical

conclusion that the students studied a lot *because* their grades were going to be high.

Another way to visualise the apparent correlation between two numeric columns is to use a *scatter* plot.

```
[ ]: # Create a scatter plot
df_sample.plot.scatter(title='Study Time vs Grade', x='StudyHours',y='Grade')
```

```
[ ]: <AxesSubplot:title={'center':'Study Time vs Grade'}, xlabel='StudyHours',
ylabel='Grade'>
```



Again, it looks like there's a discernible pattern in which the students who studied the most hours are also the students who got the highest grades.

We can see this more clearly by adding a *regression* line (or a *line of best fit*) to the plot that shows the general trend in the data. To do this, we'll use a statistical technique called *least squares regression*.

Warning - Math Ahead!

Cast your mind back to when you were learning how to solve linear equations in school, and recall that the *slope-intercept* form of a linear equation looks like this:

$y = mx + b$

In this equation, y and x are the coordinate variables, m is the slope of the line, and b is the y-intercept (where the line goes through the Y-axis).

In the case of our scatter plot for our student data, we already have our values for x (*StudyHours*) and y (*Grade*), so we just need to calculate the intercept and slope of the straight line that lies closest to those points. Then we can form a linear equation that calculates a new y value on that line for each of our x (*StudyHours*) values - to avoid confusion, we'll call this new y value *f(x)* (because it's the output from a linear equation ***f***unction based on x). The difference between the original y (*Grade*) value and the *f(x)* value is the *error* between our regression line and the actual *Grade* achieved by the student. Our goal is to calculate the slope and intercept for a line with the lowest overall error.

Specifically, we define the overall error by taking the error for each point, squaring it, and adding all the squared errors together. The line of best fit is the line that gives us the lowest value for the sum of the squared errors - hence the name *least squares regression*.

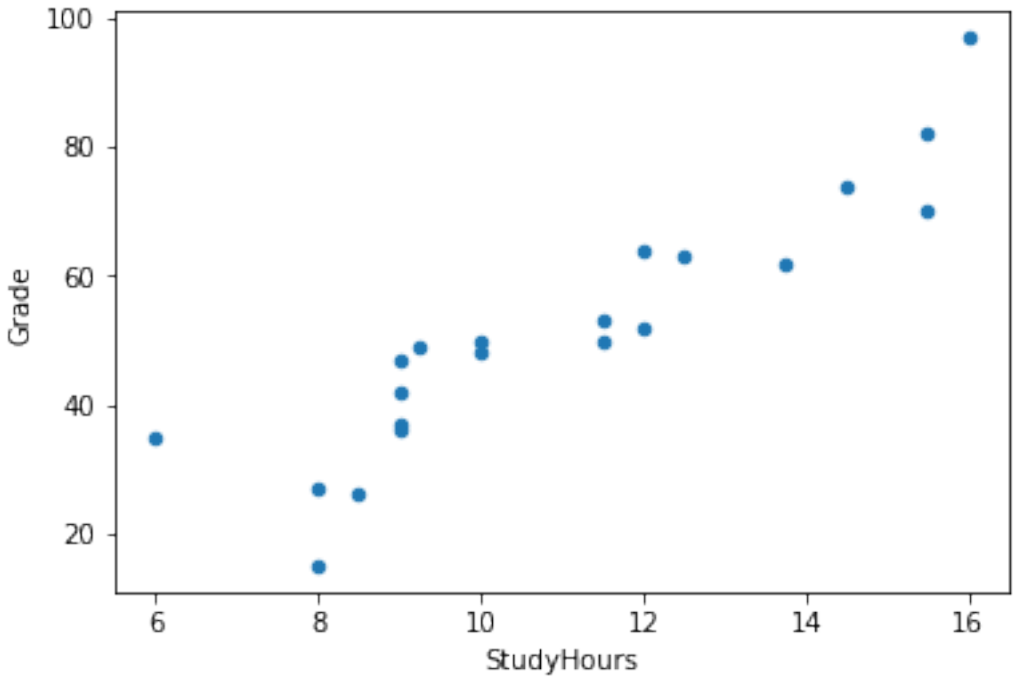
Fortunately, you don't need to code the regression calculation yourself - the **SciPy** package includes a **stats** class that provides a **linregress** method to do the hard work for you. This returns (among other things) the coefficients you need for the slope equation - slope (m) and intercept (b) based on a given pair of variable samples you want to compare.

[ ]:
```
from scipy import stats

df_regression=df_sample[['Grade','StudyHours']].copy()

# Get the regression slope and intercept
m, b, r, p, se = stats.linregress(df_regression['StudyHours'],␣
 ↪df_regression['Grade'])
print('slope: {:.4f}\ny-intercept: {:.4f}'.format(m,b))
print('so...\n f(x) = {:.4f}x + {:.4f}'.format(m,b))
```

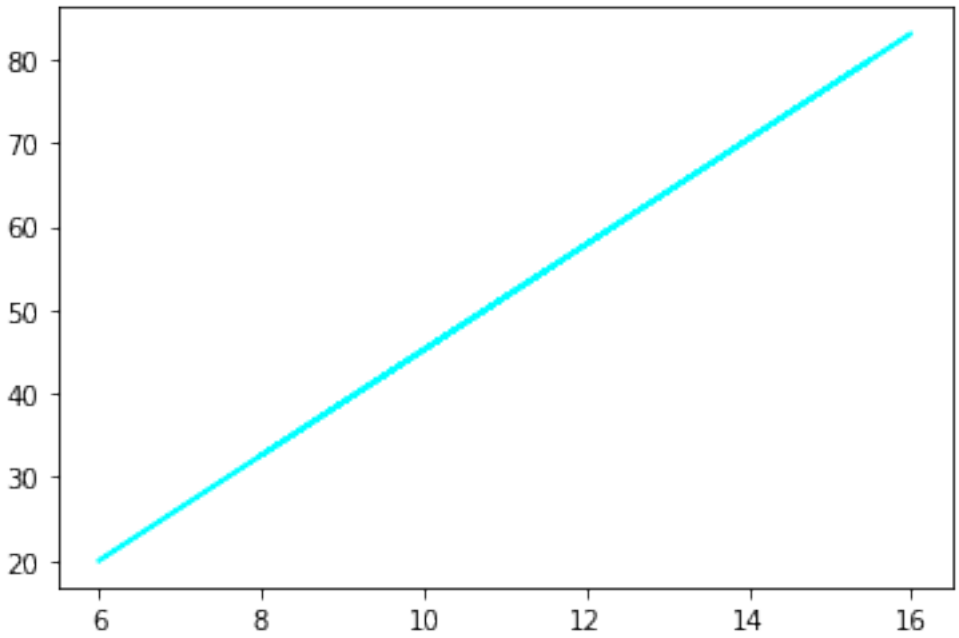
```
slope: 6.3134
y-intercept: -17.9164
so…
 f(x) = 6.3134x + -17.9164
```

[ ]:
```
# Use the function (mx + b) to calculate f(x) for each x (StudyHours) value
df_regression['fx'] =(m * df_regression['StudyHours']) + b

# Calculate the error between f(x) and the actual y (Grade) value
df_regression['error'] = df_regression['fx'] - df_regression['Grade']

# Create a scatter plot of Grade vs Salary
df_regression.plot.scatter(x='StudyHours', y='Grade')
```

[ ]:
```
<AxesSubplot:xlabel='StudyHours', ylabel='Grade'>
```



[ ]:
```
# Plot the regression line
plt.plot(df_regression['StudyHours'], df_regression['fx'], color='cyan')

# Display
plt.show()
```



Note that this time, the code plotted two distinct things - the scatter plot of the sample study hours and grades is plotted as before, and then a line of best fit based on the least squares regression coefficients is plotted.

The slope and intercept coefficients calculated for the regression line are shown above the plot.

The line is based on the $f(x)$ values calculated for each **StudyHours** value. Run the following cell to see a table that includes the following values:

- The **StudyHours** for each student.
- The **Grade** achieved by each student.
- The ***f(x)*** value calculated using the regression line coefficients.
- The *error* between the calculated ***f(x)*** value and the actual **Grade** value.

Some of the errors, particularly at the extreme ends, and quite large (up to over 17.5 grade points); but in general, the line is pretty close to the actual grades.

[ ]:
```
df_regression
```

[ ]:

	Grade	StudyHours	fx	error
0	50.0	10.00	45.217846	-4.782154
1	50.0	11.50	54.687985	4.687985
2	47.0	9.00	38.904421	-8.095579
3	97.0	16.00	83.098400	-13.901600
4	49.0	9.25	40.482777	-8.517223
6	53.0	11.50	54.687985	1.687985
7	42.0	9.00	38.904421	-3.095579
8	26.0	8.50	35.747708	9.747708
9	74.0	14.50	73.628262	-0.371738
10	82.0	15.50	79.941687	-2.058313
11	62.0	13.75	68.893193	6.893193
12	37.0	9.00	38.904421	1.904421
13	15.0	8.00	32.590995	17.590995
14	70.0	15.50	79.941687	9.941687
15	27.0	8.00	32.590995	5.590995
16	36.0	9.00	38.904421	2.904421
17	35.0	6.00	19.964144	-15.035856
18	48.0	10.00	45.217846	-2.782154
19	52.0	12.00	57.844698	5.844698
20	63.0	12.50	61.001410	-1.998590
21	64.0	12.00	57.844698	-6.155302

[ ]:
```
# Show the original x,y values, the f(x) value, and the error
df_regression[['StudyHours', 'Grade', 'fx', 'error']]
```

[ ]:

	StudyHours	Grade	fx	error
0	10.00	50.0	45.217846	-4.782154

1	11.50	50.0	54.687985	4.687985
2	9.00	47.0	38.904421	-8.095579
3	16.00	97.0	83.098400	-13.901600
4	9.25	49.0	40.482777	-8.517223
6	11.50	53.0	54.687985	1.687985
7	9.00	42.0	38.904421	-3.095579
8	8.50	26.0	35.747708	9.747708
9	14.50	74.0	73.628262	-0.371738
10	15.50	82.0	79.941687	-2.058313
11	13.75	62.0	68.893193	6.893193
12	9.00	37.0	38.904421	1.904421
13	8.00	15.0	32.590995	17.590995
14	15.50	70.0	79.941687	9.941687
15	8.00	27.0	32.590995	5.590995
16	9.00	36.0	38.904421	2.904421
17	6.00	35.0	19.964144	-15.035856
18	10.00	48.0	45.217846	-2.782154
19	12.00	52.0	57.844698	5.844698
20	12.50	63.0	61.001410	-1.998590
21	12.00	64.0	57.844698	-6.155302

1.1.3 Using the regression coefficients for prediction

Now that you have the regression coefficients for the study time and grade relationship, you can use them in a function to estimate the expected grade for a given amount of study.

[ ]:
```
# Define a function based on our regression coefficients
def f(x):
    m = 6.3134
    b = -17.9164
    return m*x + b

study_time = 14

# Get f(x) for study time
prediction =f(study_time)

# Grade can't be less than 0 or more than 100
expected_grade = max(0, min(100,prediction))

# Print the esimated grade
print ('Studying for {} hours per week may result in a grade of {:.0f}'.
 ↪format(study_time,expected_grade))
```

```
Studying for 14 hours per week may result in a grade of 70
```

So by applying statistics to sample data, you've determined a relationship between study time and grade; and encapsulated that relationship in a general function that can be used to predict a grade for a given amount of study time.

This technique is in fact the basic premise of machine learning. You can take a set of sample data that includes one or more *features* (in this case, the number of hours studied) and a known *label* value (in this case, the grade achieved) and use the sample data to derive a function that calculates predicted label values for any given set of features.

1.2 Summary

Here we've looked at:

1. What an outlier is and how to remove them
2. How data can be skewed
3. How to look at the spread of data
4. Basic ways to compare variables, such as grades and study time

1.3 Further Reading

To learn more about the Python packages you explored in this notebook, see the following documentation:

- NumPy
- Pandas
- Matplotlib

[ ]:
```
import numpy as np
```

[ ]:
```
df_array=np.array(df_regression)
df_array.shape
```

[ ]:
```
(21, 4)
```

21 rows and 4 columns

[ ]:
```
df_regression.shape
```

[ ]:
```
(21, 4)
```

[ ]:
```
df_regression['Grade'].mean()
```

[ ]:
```
51.38095238095238
```

[ ]: