

Data and Data Visualization

Machine learning, and therefore a large part of AI, is based on statistical analysis of data. In this notebook, you'll examine some fundamental concepts related to data and data visualization.

Introduction to Data

Statistics are based on data, which consist of a collection of pieces of information about things you want to study. This information can take the form of descriptions, quantities, measurements, and other observations. Typically, we work with related data items in a *dataset*, which often consists of a collection of *observations* or *cases*. Most commonly, we think about this dataset as a table that consists of a row for each observation, and a column for each individual data point related to that observation - we variously call these data points *attributes* or *features*, and they each describe a specific characteristic of the thing we're observing.

Let's take a look at a real example. In 1886, Francis Galton conducted a study into the relationship between heights of parents and their (adult) children. Run the Python code below to view the data he collected (you can safely ignore a deprecation warning if it is displayed):

In [4]: `import statsmodels.api as sm`

```
df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data
df
```

```
/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/statsmodels
/datasets/utils.py:192: FutureWarning: `item` has been deprecated and will be
removed in a future version
    return dataset_meta["Title"].item()
```

Out[4]:

	family	father	mother	midparentHeight	children	childNum	gender	childHeight
0	001	78.5	67.0	75.43	4	1	male	73.2
1	001	78.5	67.0	75.43	4	2	female	69.2
2	001	78.5	67.0	75.43	4	3	female	69.0
3	001	78.5	67.0	75.43	4	4	female	69.0
4	002	75.5	66.5	73.66	4	1	male	73.5
...
929	203	62.0	66.0	66.64	3	1	male	64.0
930	203	62.0	66.0	66.64	3	2	female	62.0

Types of Data

Now, let's take a closer look at this data (you can click the left margin next to the dataset to toggle between full height and a scrollable pane). There are 933 observations, each one recording information pertaining to an individual child. The information recorded consists of the following features:

- **family**: An identifier for the family to which the child belongs.
- **father**: The height of the father.
- **** mother****: The height of the mother.
- **midparentHeight**: The mid-point between the father and mother's heights (calculated as $(father + 1.08 \times mother) \div 2$)
- **children**: The total number of children in the family.
- **childNum**: The number of the child to whom this observation pertains (Galton numbered the children in descending order of height, with male children listed before female children)
- **gender**: The gender of the child to whom this observation pertains.
- **childHeight**: The height of the child to whom this observation pertains.

It's worth noting that there are several distinct types of data recorded here. To begin with, there are some features that represent *qualities*, or characteristics of the child - for example, gender. Other features represent a *quantity* or measurement, such as the child's height. So broadly speaking, we can divide data into *qualitative* and *quantitative* data.

Qualitative Data

Let's take a look at qualitative data first. This type of data is categorical - it is used to categorize or identify the entity being observed. Sometimes you'll see features of this type described as *factors*.

Nominal Data

In his observations of children's height, Galton assigned an identifier to each family and he recorded the gender of each child. Note that even though the **family** identifier is a number, it is not a measurement or quantity. Family 002 is not "greater" than family 001, just as a **gender** value of "male" does not indicate a larger or smaller value than "female". These are simply named values for some characteristic of the child, and as such they're known as *nominal* data.

Ordinal Data

So what about the **childNum** feature? It's not a measurement or quantity - it's just a way to identify individual children within a family. However, the number assigned to each child has some additional meaning - the numbers are ordered. You can find similar data that is text-based; for example, data about training courses might include a "level" attribute that indicates the level of the course as "basic", "intermediate", or "advanced". This type of data, where the value is not itself a quantity or measurement, but it indicates some sort of inherent order or hierarchy, is known as *ordinal* data.

Quantitative Data

Now let's turn our attention to the features that indicate some kind of quantity or measurement.

Discrete Data

Galton's observations include the number of **children** in each family. This is a *discrete* quantitative data value - it's something we *count* rather than *measure*. You can't, for example, have 2.33 children!

Continuous Data

The data set also includes height values for **father**, **mother**, **midparentHeight**, and **childHeight**. These are measurements along a scale, and as such they're described as *continuous* quantitative data values that we *measure* rather than *count*.

Visualizing Data

Data visualization is one of the key ways in which we can examine data and get insights from it. If a picture is worth a thousand words, then a good graph or chart is worth any number of tables of data.

Let's examine some common kinds of data visualization:

Bar Charts

A *bar chart* is a good way to compare numeric quantities or counts across categories. For example, in the Galton dataset, you might want to compare the number of female and male children.

Here's some Python code to create a bar chart showing the number of children of each gender.

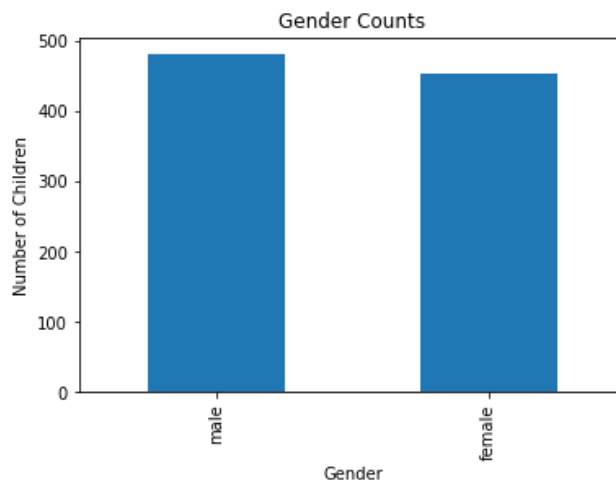
```
In [5]: import statsmodels.api as sm

df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data

# Create a data frame of gender counts
genderCounts = df['gender'].value_counts()

# Plot a bar chart
%matplotlib inline
from matplotlib import pyplot as plt

genderCounts.plot(kind='bar', title='Gender Counts')
plt.xlabel('Gender')
plt.ylabel('Number of Children')
plt.show()
```



From this chart, you can see that there are slightly more male children than female children; but the data is reasonably evenly split between the two genders.

Bar charts are typically used to compare categorical (qualitative) data values; but in some cases you might treat a discrete quantitative data value as a category. For example, in the Galton dataset the number of children in each family could be used as a way to categorize families. We might want to see how many families have one child, compared to how many have two children, etc.

Here's some Python code to create a bar chart showing family counts based on the number of children in the family.

```
In [6]: import statsmodels.api as sm

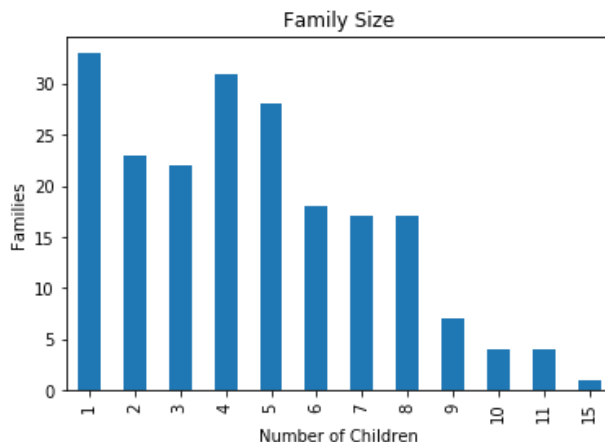
df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data

# Create a data frame of child counts
# there's a row for each child, so we need to filter to one row per family to a
families = df[['family', 'children']].drop_duplicates()
# Now count number of rows for each 'children' value, and sort by the index (ch
childCounts = families['children'].value_counts().sort_index()

# Plot a bar chart
%matplotlib inline
from matplotlib import pyplot as plt

childCounts.plot(kind='bar', title='Family Size')
plt.xlabel('Number of Children')
plt.ylabel('Families')
plt.show()
```

```
/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/statsmodels
/datasets/utils.py:192: FutureWarning: `item` has been deprecated and will be
removed in a future version
    return dataset_meta["Title"].item()
```



Note that the code sorts the data so that the categories on the x axis are in order - attention to this sort of detail can make your charts easier to read. In this case, we can see that the most common number of children per family is 1, followed by 5 and 6. Comparatively fewer families have more than 8 children.

Histograms

Bar charts work well for comparing categorical or discrete numeric values. When you need to compare continuous quantitative values, you can use a similar style of chart called a *histogram*. Histograms differ from bar charts in that they group the continuous values into ranges or *bins* - so the chart doesn't show a bar for each individual value, but rather a bar for each range of binned values. Because these bins represent continuous data rather than discrete data, the bars aren't separated by a gap. Typically, a histogram is used to show the relative frequency of values in the dataset.

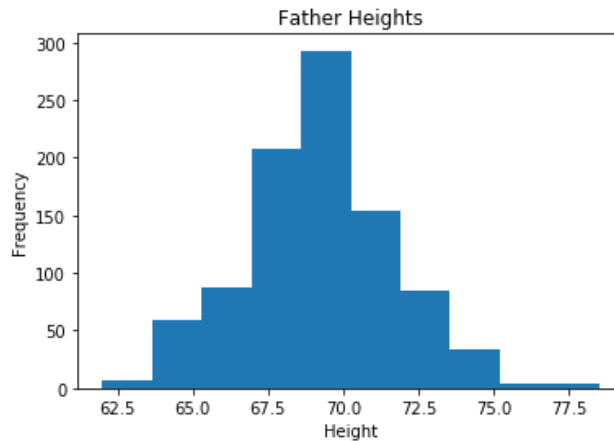
Here's some Python code to create a histogram of the **father** values in the Galton dataset, which record the father's height:

```
In [7]: import statsmodels.api as sm

df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data

# Plot a histogram of midparentHeight
%matplotlib inline
from matplotlib import pyplot as plt

df['father'].plot.hist(title='Father Heights')
plt.xlabel('Height')
plt.ylabel('Frequency')
plt.show()
```



The histogram shows that the most frequently occurring heights tend to be in the mid-range. There are fewer extremely short or extremely tall fathers.

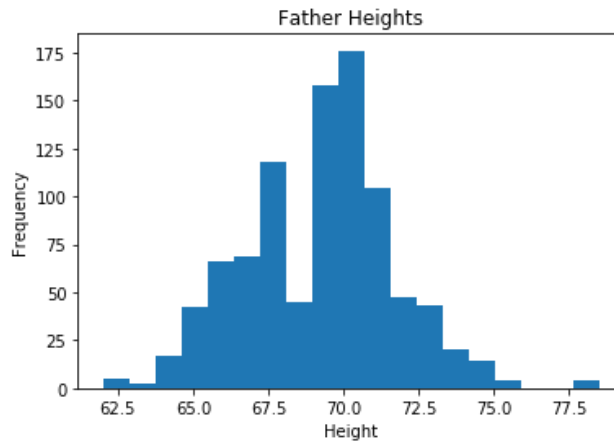
In the histogram above, the number of bins (and their corresponding ranges, or *bin widths*) was determined automatically by Python. In some cases you may want to explicitly control the number of bins, as this can help you see detail in the distribution of data values that otherwise you might miss. The following code creates a histogram for the same father's height values, but explicitly distributes them over 20 bins (19 are specified, and Python adds one):

```
In [8]: import statsmodels.api as sm

df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data

# Plot a histogram of midparentHeight
%matplotlib inline
from matplotlib import pyplot as plt

df['father'].plot.hist(title='Father Heights', bins=19)
plt.xlabel('Height')
plt.ylabel('Frequency')
plt.show()
```



We can still see that the most common heights are in the middle, but there's a notable drop in the number of fathers with a height between 67.5 and 70.

Pie Charts

Pie charts are another way to compare relative quantities of categories. They're not commonly used by data scientists, but they can be useful in many business contexts with manageable numbers of categories because they not only make it easy to compare relative quantities by categories; they also show those quantities as a proportion of the whole set of data.

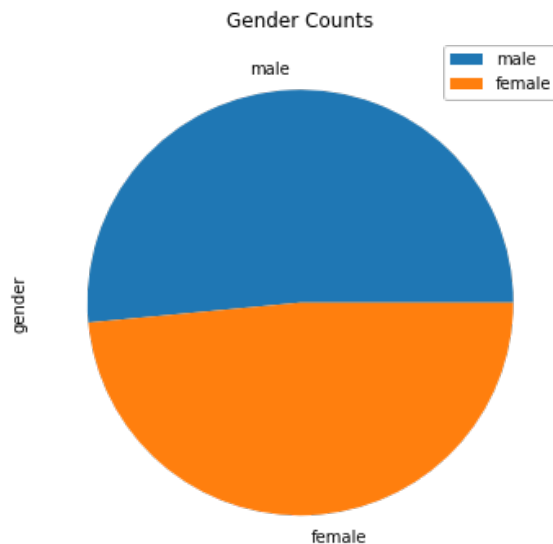
Here's some Python to show the gender counts as a pie chart:

```
In [9]: import statsmodels.api as sm

df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data

# Create a data frame of gender counts
genderCounts = df['gender'].value_counts()

# Plot a pie chart
%matplotlib inline
from matplotlib import pyplot as plt
genderCounts.plot(kind='pie', title='Gender Counts', figsize=(6,6))
plt.legend()
plt.show()
```



Note that the chart includes a *legend* to make it clear what category each colored area in the pie chart represents. From this chart, you can see that males make up slightly more than half of the overall number of children; with females accounting for the rest.

Scatter Plots

Often you'll want to compare quantitative values. This can be especially useful in data science scenarios where you are exploring data prior to building a machine learning model, as it can help identify apparent relationships between numeric features. Scatter plots can also help identify potential outliers - values that are significantly outside of the normal range of values.

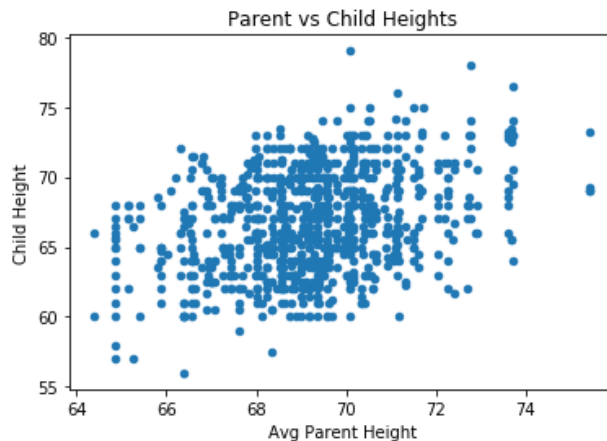
The following Python code creates a scatter plot that plots the intersection points for **midparentHeight** on the x axis, and **childHeight** on the y axis:

```
In [10]: import statsmodels.api as sm

df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data

# Create a data frame of heights (father vs child)
parentHeights = df[['midparentHeight', 'childHeight']]

# Plot a scatter plot chart
%matplotlib inline
from matplotlib import pyplot as plt
parentHeights.plot(kind='scatter', title='Parent vs Child Heights', x='midparentHeight', y='childHeight')
plt.xlabel('Avg Parent Height')
plt.ylabel('Child Height')
plt.show()
```



In a scatter plot, each dot marks the intersection point of the two values being plotted. In this chart, most of the heights are clustered around the center; which indicates that most parents and children tend to have a height that is somewhere in the middle of the range of heights observed. At the bottom left, there's a small cluster of dots that show some parents from the shorter end of the range who have children that are also shorter than their peers. At the top right, there are a few extremely tall parents who have extremely tall children. It's also interesting to note that the top left and bottom right of the chart are empty - there aren't any cases of extremely short parents with extremely tall children or vice-versa.

Line Charts

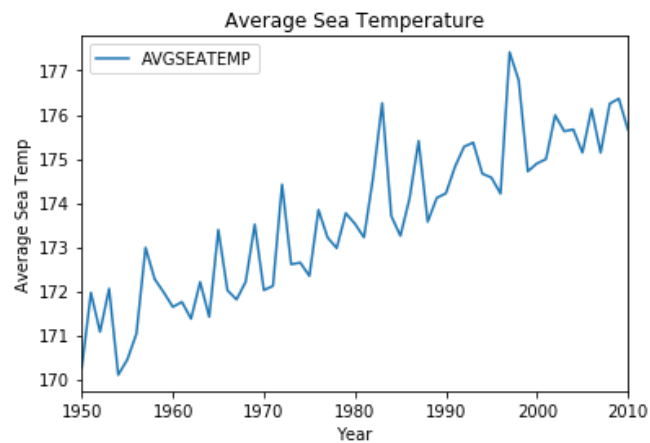
Line charts are a great way to see changes in values along a series - usually (but not always) based on a time period. The Galton dataset doesn't include any data of this type, so we'll use a different dataset that includes observations of sea surface temperature between 1950 and 2010 for this example:


```
In [11]: import statsmodels.api as sm

df = sm.datasets.elnino.load_pandas().data

df['AVGSEATEMP'] = df.mean(1)

# Plot a line chart
%matplotlib inline
from matplotlib import pyplot as plt
df.plot(title='Average Sea Temperature', x='YEAR', y='AVGSEATEMP')
plt.xlabel('Year')
plt.ylabel('Average Sea Temp')
plt.show()
```



The line chart shows the temperature trend from left to right for the period of observations. From this chart, you can see that the average temperature fluctuates from year to year, but the general trend shows an increase.

Statistics Fundamentals

Statistics is primarily about analyzing data samples, and that starts with understanding the distribution of data in a sample.

Analyzing Data Distribution

A great deal of statistical analysis is based on the way that data values are distributed within the dataset. In this section, we'll explore some statistics that you can use to tell you about the values in a dataset.

Measures of Central Tendency

The term *measures of central tendency* sounds a bit grand, but really it's just a fancy way of saying that we're interested in knowing where the middle value in our data is. For example, suppose decide to conduct a study into the comparative salaries of people who graduated from the same school. You might record the results like this:

Name	Salary
Dan	50,000
Joann	54,000
Pedro	50,000
Rosie	189,000
Ethan	55,000
Vicky	40,000
Frederic	59,000

Now, some of the former-students may earn a lot, and others may earn less; but what's the salary in the middle of the range of all salaries?

Mean

A common way to define the central value is to use the *mean*, often called the *average*. This is calculated as the sum of the values in the dataset, divided by the number of observations in the dataset. When the dataset consists of the full population, the mean is represented by the Greek symbol μ (*mu*), and the formula is written like this:

$$\mu = \frac{\sum_{i=1}^N X_i}{N}$$

More commonly, when working with a sample, the mean is represented by \bar{x} (*x-bar*), and the formula is written like this (note the lower case letters used to indicate values from a sample):

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

In the case of our list of heights, this can be calculated as:

$$\bar{x} = \frac{50000 + 54000 + 50000 + 189000 + 55000 + 40000 + 59000}{7}$$

Which is **71,000**.

In technical terminology, \bar{x} is a *statistic* (an estimate based on a sample of data) and μ is a *parameter* (a true value based on the entire population). A lot of the time, the parameters for the full population will be impossible (or at the very least, impractical) to measure; so we use statistics obtained from a representative sample to approximate them. In this case, we can use the sample mean of salary for our selection of surveyed students to try to estimate the actual average salary of all students who graduate from our school.

In [1]: `import pandas as pd`

```
df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],  
                  'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000]})  
  
print (df['Salary'].mean())
```

71000.0

So, is **71,000** really the central value? Or put another way, would it be reasonable for a graduate of this school to expect to earn \$71,000? After all, that's the average salary of a graduate from this school.

If you look closely at the salaries, you can see that out of the seven former students, six earn less than the mean salary. The data is *skewed* by the fact that Rosie has clearly managed to find a much higher-paid job than her classmates.

Median

OK, let's see if we can find another definition for the central value that more closely reflects the expected earning potential of students attending our school. Another measure of central tendency we can use is the *median*. To calculate the median, we need to sort the values into ascending order and then find the middle-most value. When there are an odd number of observations, you can find the position of the median value using this formula (where n is the number of observations):

$$\frac{n + 1}{2}$$

Remember that this formula returns the *position* of the median value in the sorted list; not the value itself.

If the number of observations is even, then things are a little (but not much) more complicated. In this case you calculate the median as the average of the two middle-most values, which are found like this:

$$\frac{n}{2} \quad \text{and} \quad \frac{n}{2} + 1$$

So, for our graduate salaries; first let's sort the dataset:

Salary
40,000
50,000
50,000
54,000
55,000
59,000
189,000

There's an odd number of observations (7), so the median value is at position $(7 + 1) \div 2$; in other words, position 4:

Salary
40,000
50,000
50,000
>54,000
55,000
59,000
189,000

So the median salary is **54,000**.

The `pandas.DataFrame` class in Python has a ***median*** function to find the median:

```
In [2]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000]})

print (df['Salary'].median())

54000.0
```

Mode

Another related statistic is the *mode*, which indicates the most frequently occurring value. If you think about it, this is potentially a good indicator of how much a student might expect to earn when they graduate from the school; out of all the salaries that are being earned by former students, the mode is earned by more than any other.

Looking at our list of salaries, there are two instances of former students earning **50,000**, but only one instance each for all other salaries:

Salary
40,000
>50,000
>50,000
54,000
55,000
59,000
189,000

The mode is therefore **50,000**.

As you might expect, the *pandas.dataframe* class has a ***mode*** function to return the mode:

```
In [3]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000]})

print (df['Salary'].mode())

0    50000
dtype: int64
```

Multimodal Data

It's not uncommon for a set of data to have more than one value as the mode. For example, suppose Ethan receives a raise that takes his salary to **59,000**:

Salary
40,000
>50,000
>50,000
54,000
>59,000
>59,000
189,000

Now there are two values with the highest frequency. This dataset is *bimodal*. More generally, when there is more than one mode value, the data is considered *multimodal*.

The `pandas.DataFrame.mode` function returns all of the modes:

```
In [4]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                  'Salary': [50000, 54000, 50000, 189000, 59000, 40000, 59000]})

print (df['Salary'].mode())

0    50000
1    59000
dtype: int64
```

Distribution and Density

Now we know something about finding the center, we can start to explore how the data is distributed around it. What we're interested in here is understanding the general "shape" of the data distribution so that we can begin to get a feel for what a 'typical' value might be expected to be.

We can start by finding the extremes - the minimum and maximum. In the case of our salary data, the lowest paid graduate from our school is Vicky, with a salary of **40,000**; and the highest-paid graduate is Rosie, with **189,000**.

The `pandas.DataFrame` class has *min* and *max* functions to return these values.

Run the following code to compare the minimum and maximum salaries to the central measures we calculated previously:

```
In [5]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                  'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000]})

print ('Min: ' + str(df['Salary'].min()))
print ('Mode: ' + str(df['Salary'].mode()[0]))
print ('Median: ' + str(df['Salary'].median()))
print ('Mean: ' + str(df['Salary'].mean()))
print ('Max: ' + str(df['Salary'].max()))

Min: 40000
Mode: 50000
Median: 54000.0
Mean: 71000.0
Max: 189000
```

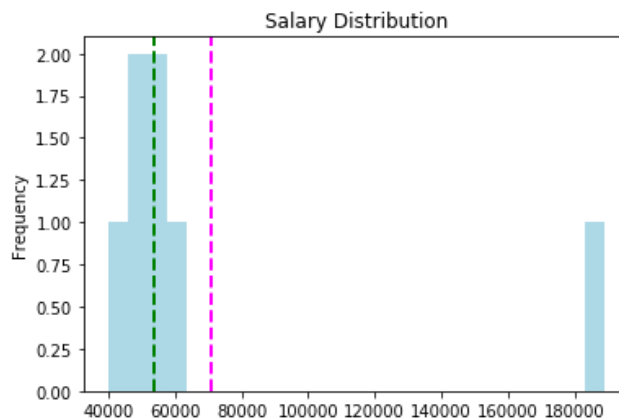
We can examine these values, and get a sense for how the data is distributed - for example, we can see that the *mean* is closer to the *max* than the *median*, and that both are closer to the *min* than to the *max*.

However, it's generally easier to get a sense of the distribution by visualizing the data. Let's start by creating a histogram of the salaries, highlighting the *mean* and *median* salaries (the *min*, *max* are fairly self-evident, and the *mode* is wherever the highest bar is):

```
In [6]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                  'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000]})

salary = df['Salary']
salary.plot.hist(title='Salary Distribution', color='lightblue', bins=25)
plt.axvline(salary.mean(), color='magenta', linestyle='dashed', linewidth=2)
plt.axvline(salary.median(), color='green', linestyle='dashed', linewidth=2)
plt.show()
```



The **mean** and **median** are shown as dashed lines. Note the following:

- *Salary* is a continuous data value - graduates could potentially earn any value along the scale, even down to a fraction of cent.
- The number of bins in the histogram determines the size of each salary band for which we're counting frequencies. Fewer bins means merging more individual salaries together to be counted as a group.
- The majority of the data is on the left side of the histogram, reflecting the fact that most graduates earn between 40,000 and 55,000
- The mean is a higher value than the median and mode.
- There are gaps in the histogram for salary bands that nobody earns.

The histogram shows the relative frequency of each salary band, based on the number of bins. It also gives us a sense of the *density* of the data for each point on the salary scale. With enough data points, and small enough bins, we could view this density as a line that shows the shape of the data distribution.

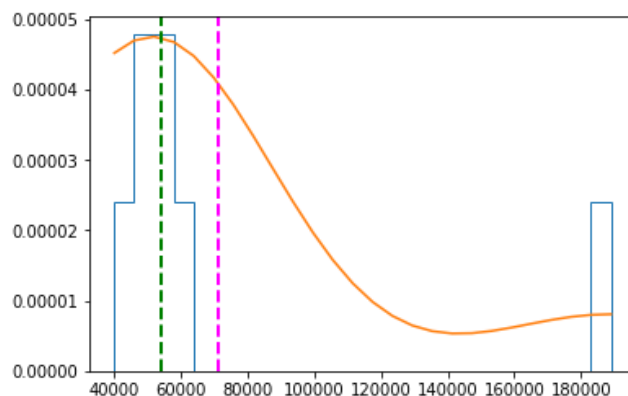
Run the following cell to show the density of the salary data as a line on top of the histogram:

```
In [7]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                  'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000]})

salary = df['Salary']
density = stats.gaussian_kde(salary)
n, x, _ = plt.hist(salary, histtype='step', normed=True, bins=25)
plt.plot(x, density(x)*5)
plt.axvline(salary.mean(), color='magenta', linestyle='dashed', linewidth=2)
plt.axvline(salary.median(), color='green', linestyle='dashed', linewidth=2)
plt.show()
```

```
/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/ipykernel_
auncher.py:12: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.
1. Use 'density' instead.
   if sys.path[0] == '':
```



Note that the density line takes the form of an asymmetric curve that has a "peak" on the left and a long tail on the right. We describe this sort of data distribution as being *skewed*; that is, the data is not distributed symmetrically but "bunched together" on one side. In this case, the data is bunched together on the left, creating a long tail on the right; and is described as being *right-skewed* because some infrequently occurring high values are pulling the *mean* to the right.

Let's take a look at another set of data. We know how much money our graduates make, but how many hours per week do they need to work to earn their salaries? Here's the data:

Name	Hours
Dan	41
Joann	40
Pedro	36
Rosie	30
Ethan	35
Vicky	39
Frederic	40

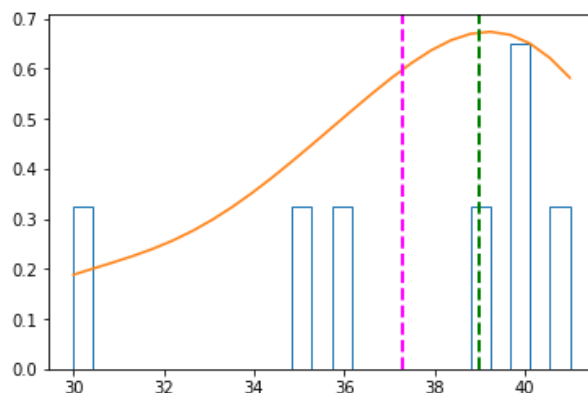
Run the following code to show the distribution of the hours worked:

```
In [8]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                           'Hours': [41, 40, 36, 30, 35, 39, 40]})

hours = df['Hours']
density = stats.gaussian_kde(hours)
n, x, _ = plt.hist(hours, histtype='step', normed=True, bins=25)
plt.plot(x, density(x)*7)
plt.axvline(hours.mean(), color='magenta', linestyle='dashed', linewidth=2)
plt.axvline(hours.median(), color='green', linestyle='dashed', linewidth=2)
plt.show()
```

```
/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/ipykernel_launcher.py:12: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.
1. Use 'density' instead.
   if sys.path[0] == '':
```



Once again, the distribution is skewed, but this time it's **left-skewed**. Note that the curve is asymmetric with the **mean** to the left of the **median** and the **mode**; and the average weekly working hours skewed to the lower end.

Once again, Rosie seems to be getting the better of the deal. She earns more than her former classmates for working fewer hours. Maybe a look at the test scores the students achieved on their final grade at school might help explain her success:

Name	Grade
Dan	50
Joann	50
Pedro	46
Rosie	95
Ethan	50
Vicky	5
Frederic	57

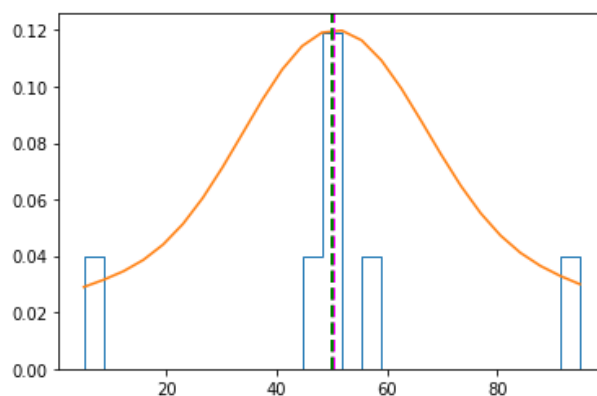
Let's take a look at the distribution of these grades:

```
In [9]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})
```

```
grade = df['Grade']
density = stats.gaussian_kde(grade)
n, x, _ = plt.hist(grade, histtype='step', normed=True, bins=25)
plt.plot(x, density(x)*7.5)
plt.axvline(grade.mean(), color='magenta', linestyle='dashed', linewidth=2)
plt.axvline(grade.median(), color='green', linestyle='dashed', linewidth=2)
plt.show()
```

```
/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/ipykernel_launcher.py:12: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.
1. Use 'density' instead.
   if sys.path[0] == '':
```



This time, the distribution is symmetric, forming a "bell-shaped" curve. The *mean*, *median*, and mode are at the same location, and the data tails off evenly on both sides from a central peak.

Statisticians call this a *normal* distribution (or sometimes a *Gaussian* distribution), and it occurs quite commonly in many scenarios due to something called the *Central Limit Theorem*, which reflects the way continuous probability works - more about that later.

Skewness and Kurtosis

You can measure *skewness* (in which direction the data is skewed and to what degree) and *kurtosis* (how "peaked" the data is) to get an idea of the shape of the data distribution. In Python, you can use the ***skew*** and ***kurt*** functions to find this:

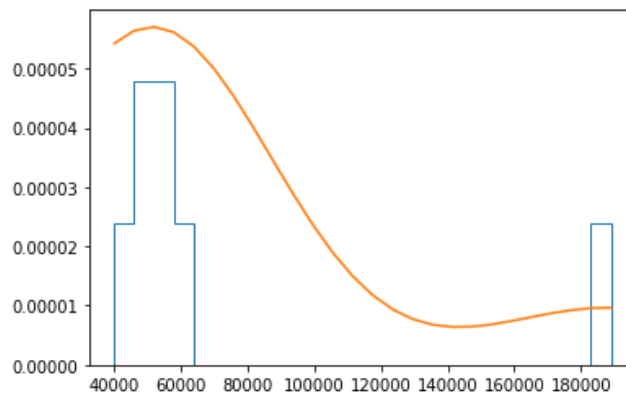
```
In [10]: %matplotlib inline
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import scipy.stats as stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 30, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

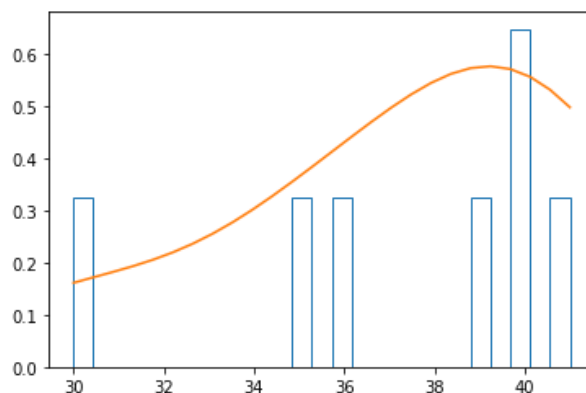
numcols = ['Salary', 'Hours', 'Grade']
for col in numcols:
    print(df[col].name + ' skewness: ' + str(df[col].skew()))
    print(df[col].name + ' kurtosis: ' + str(df[col].kurt()))
    density = stats.gaussian_kde(df[col])
    n, x, _ = plt.hist(df[col], histtype='step', normed=True, bins=25)
    plt.plot(x, density(x)*6)
    plt.show()
    print('\n')
```

Salary skewness: 2.57316410755049
Salary kurtosis: 6.719828837773431

/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/ipykernel_launcher.py:17: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.
1. Use 'density' instead.



Hours skewness: -1.194570307262883
Hours kurtosis: 0.9412265624999989



Grade skewness: -0.06512433009682762
Grade kurtosis: 2.7484764913773034

Now let's look at the distribution of a real dataset - let's see how the heights of the father's measured in Galton's study of parent and child heights are distributed:

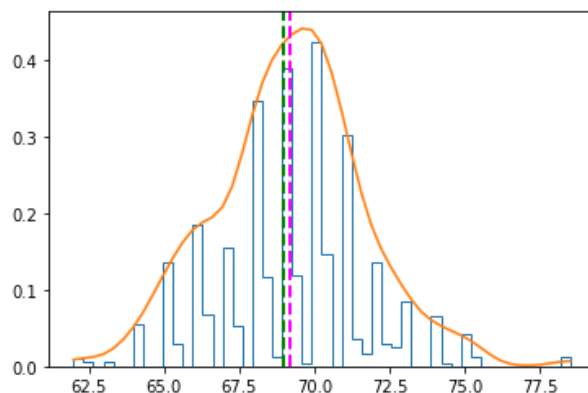
```
In [11]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats

import statsmodels.api as sm

df = sm.datasets.get_rdataset('GaltonFamilies', package='HistData').data

fathers = df['father']
density = stats.gaussian_kde(fathers)
n, x, _ = plt.hist(fathers, histtype='step', normed=True, bins=50)
plt.plot(x, density(x)*2.5)
plt.axvline(fathers.mean(), color='magenta', linestyle='dashed', linewidth=2)
plt.axvline(fathers.median(), color='green', linestyle='dashed', linewidth=2)
plt.show()
```

```
/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/statsmodels
/datasets/utils.py:192: FutureWarning: `item` has been deprecated and will be
removed in a future version
    return dataset_meta["Title"].item()
/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/ipykernel_l
auncher.py:14: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.
1. Use 'density' instead.
```



As you can see, the father's height measurements are approximately normally distributed - in other words, they form a more or less *normal* distribution that is symmetric around the mean.

Measures of Variance

We can see from the distribution plots of our data that the values in our dataset can vary quite widely. We can use various measures to quantify this variance.

Range

A simple way to quantify the variance in a dataset is to identify the difference between the lowest and highest values. This is called the *range*, and is calculated by subtracting the minimum value from the maximum value.

The following Python code creates a single Pandas dataframe for our school graduate data, and calculates the *range* for each of the numeric features:

```
In [12]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 30, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

numcols = ['Salary', 'Hours', 'Grade']
for col in numcols:
    print(df[col].name + ' range: ' + str(df[col].max() - df[col].min()))

Salary range: 149000
Hours range: 11
Grade range: 90
```

Percentiles and Quartiles

The range is easy to calculate, but it's not a particularly useful statistic. For example, a range of 149,000 between the lowest and highest salary does not tell us which value within that range a graduate is most likely to earn - it doesn't tell us anything about how the salaries are distributed around the mean within that range. The range tells us very little about the comparative position of an individual value within the distribution - for example, Frederic scored 57 in his final grade at school; which is a pretty good score (it's more than all but one of his classmates); but this isn't immediately apparent from a score of 57 and range of 90.

Percentiles

A percentile tells us where a given value is ranked in the overall distribution. For example, 25% of the data in a distribution has a value lower than the 25th percentile; 75% of the data has a value lower than the 75th percentile, and so on. Note that half of the data has a value lower than the 50th percentile - so the 50th percentile is also the median!

Let's examine Frederic's grade using this approach. We know he scored 57, but how does he rank compared to his fellow students?

Well, there are seven students in total, and five of them scored less than Frederic; so we can calculate the percentile for Frederic's grade like this:

$$\frac{5}{7} \times 100 \approx 71.4$$

So Frederic's score puts him at the 71.4th percentile in his class.

In Python, you can use the ***percentileofscore*** function in the *scipy.stats* package to calculate the percentile for a given value in a set of values:

```
In [13]: import pandas as pd
from scipy import stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                    'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                    'Hours': [41, 40, 36, 30, 35, 39, 40],
                    'Grade': [50, 50, 46, 95, 50, 5, 57]})

print(stats.percentileofscore(df['Grade'], 57, 'strict'))
```

71.42857142857143

We've used the strict definition of percentile; but sometimes it's calculated as being the percentage of values that are less than *or equal to* the value you're comparing. In this case, the calculation for Frederic's percentile would include his own score:

$$\frac{6}{7} \times 100 \approx 85.7$$

You can calculate this way in Python by using the **weak** mode of the **percentileofscore** function:

```
In [14]: import pandas as pd
from scipy import stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                    'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                    'Hours': [41, 40, 36, 30, 35, 39, 40],
                    'Grade': [50, 50, 46, 95, 50, 5, 57]})

print(stats.percentileofscore(df['Grade'], 57, 'weak'))
```

85.71428571428571

We've considered the percentile of Frederic's grade, and used it to rank him compared to his fellow students. So what about Dan, Joann, and Ethan? How do they compare to the rest of the class? They scored the same grade (50), so in a sense they share a percentile.

To deal with this *grouped* scenario, we can average the percentage rankings for the matching scores. We treat half of the scores matching the one we're ranking as if they are below it, and half as if they are above it. In this case, there were three matching scores of 50, and for each of these we calculate the percentile as if 1 was below and 1 was above. So the calculation for a percentile for Joann based on scores being less than or equal to 50 is:

$$\left(\frac{4}{7}\right) \times 100 \approx 57.14$$

The value of **4** consists of the two scores that are below Joann's score of 50, Joann's own score, and half of the scores that are the same as Joann's (of which there are two, so we count one).

In Python, the **percentileofscore** function has a **rank** function that calculates grouped percentiles like this:


```
In [15]: import pandas as pd
from scipy import stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 30, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

print(stats.percentileofscore(df['Grade'], 50, 'rank'))
```

57.142857142857146

Quartiles

Rather than using individual percentiles to compare data, we can consider the overall spread of the data by dividing those percentiles into four *quartiles*. The first quartile contains the values from the minimum to the 25th percentile, the second from the 25th percentile to the 50th percentile (which is the median), the third from the 50th percentile to the 75th percentile, and the fourth from the 75th percentile to the maximum.

In Python, you can use the **quantile** function of the *pandas.dataframe* class to find the threshold values at the 25th, 50th, and 75th percentiles (*quantile* is a generic term for a ranked position, such as a percentile or quartile).

Run the following code to find the quartile thresholds for the weekly hours worked by our former students:

```
In [16]: # Quartiles
import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 17, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

print(df['Hours'].quantile([0.25, 0.5, 0.75]))
```

0.25 35.5
0.50 39.0
0.75 40.0
Name: Hours, dtype: float64

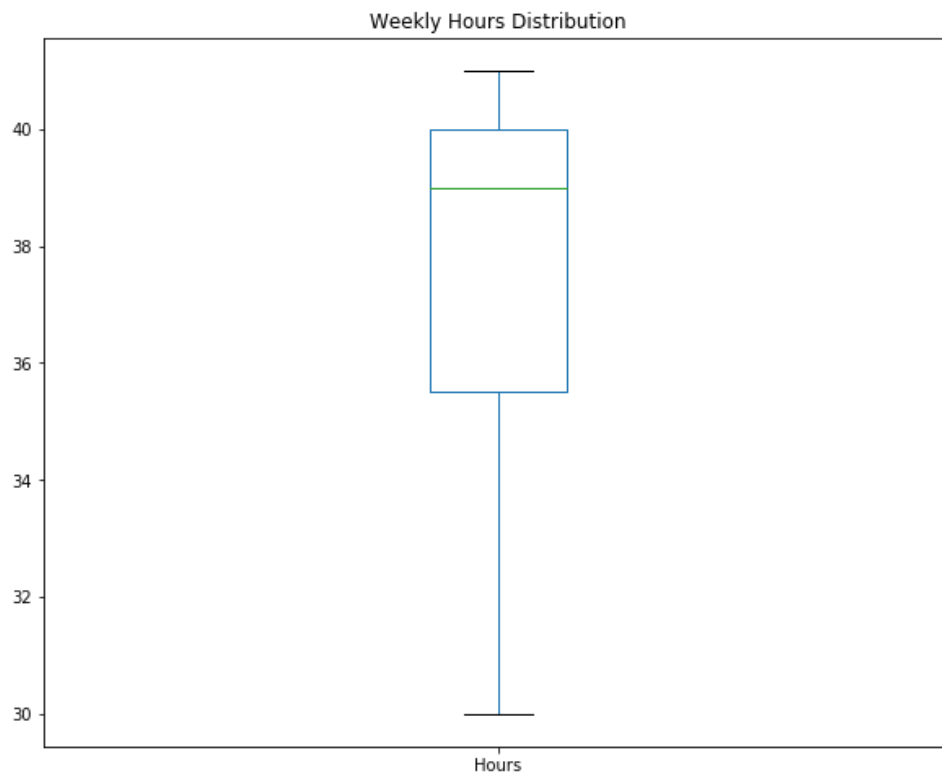
It's usually easier to understand how data is distributed across the quartiles by visualizing it. You can use a histogram, but many data scientists use a kind of visualization called a *box plot* (or a *box and whiskers* plot).

Let's create a box plot for the weekly hours:

```
In [17]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                  'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                  'Hours': [41, 40, 36, 30, 35, 39, 40],
                  'Grade': [50, 50, 46, 95, 50, 5, 57]})

# Plot a box-whisker chart
df['Hours'].plot(kind='box', title='Weekly Hours Distribution', figsize=(10,8))
plt.show()
```



The box plot consists of:

- A rectangular *box* that shows where the data between the 25th and 75th percentile (the second and third quartile) lie. This part of the distribution is often referred to as the *interquartile range* - it contains the middle 50 data values.
- *Whiskers* that extend from the box to the bottom of the first quartile and the top of the fourth quartile to show the full range of the data.
- A line in the box that shows that location of the median (the 50th percentile, which is also the threshold between the second and third quartile)

In this case, you can see that the interquartile range is between 35 and 40, with the median nearer the top of that range. The range of the first quartile is from around 30 to 35, and the fourth quartile is from 40 to 41.

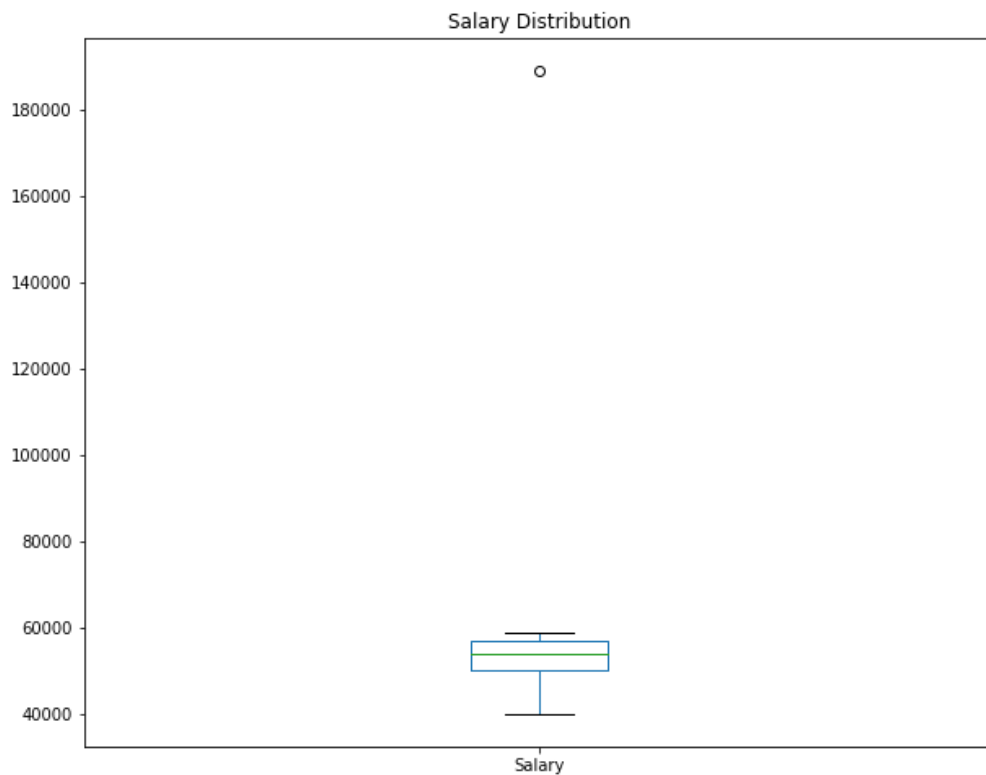
Outliers

Let's take a look at another box plot - this time showing the distribution of the salaries earned by our former classmates:

```
In [18]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 30, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

# Plot a box-whisker chart
df['Salary'].plot(kind='box', title='Salary Distribution', figsize=(10,8))
plt.show()
```



So what's going on here?

Well, as we've already noticed, Rosie earns significantly more than her former classmates. So much more in fact, that her salary has been identified as an *outlier*. An outlier is a value that is so far from the center of the distribution compared to other values that it skews the distribution by affecting the mean. There are all sorts of reasons that you might have outliers in your data, including data entry errors, failures in sensors or data-generating equipment, or genuinely anomalous values.

So what should we do about it?

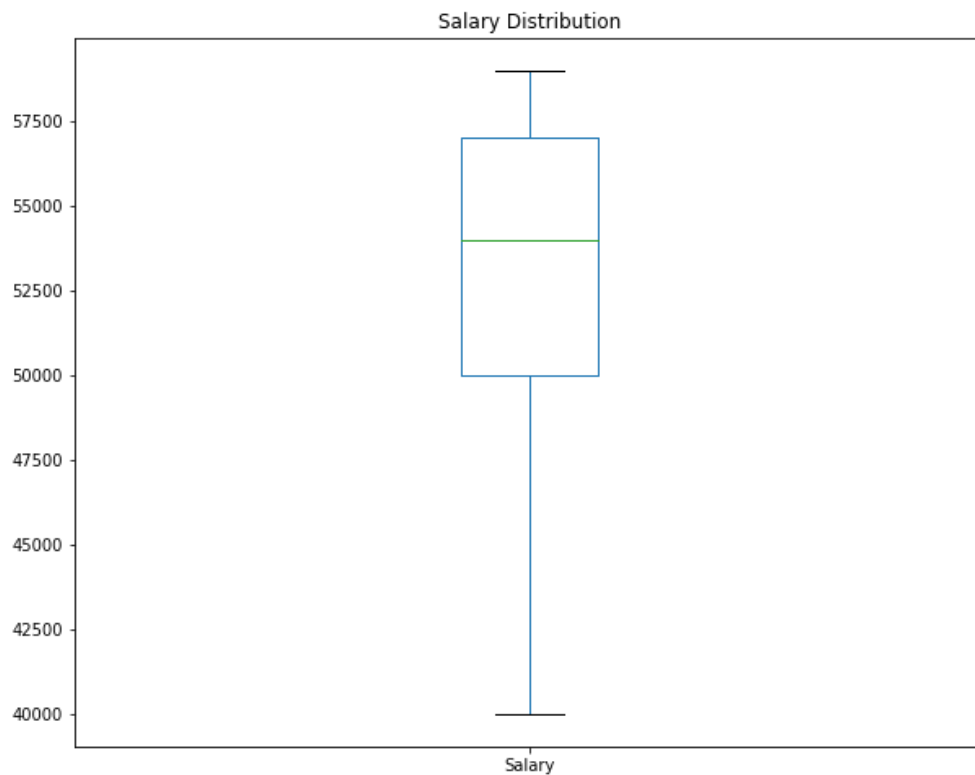
This really depends on the data, and what you're trying to use it for. In this case, let's assume we're trying to figure out what's a reasonable expectation of salary for a graduate of our school to earn. Ignoring for the moment that we have an extremely small dataset on which to base our judgement, it looks as if Rosie's salary could be either an error (maybe she mis-typed it in the form used to collect data) or a genuine anomaly (maybe she became a professional athlete or some other extremely highly paid job). Either way, it doesn't seem to represent a salary that a typical graduate might earn.

Let's see what the distribution of the data looks like without the outlier:

```
In [19]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 17, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

# Plot a box-whisker chart
df['Salary'].plot(kind='box', title='Salary Distribution', figsize=(10,8), show
plt.show())
```



Now it looks like there's a more even distribution of salaries. It's still not quite symmetrical, but there's much less overall variance. There's potentially some cause here to disregard Rosie's salary data when we compare the salaries, as it is tending to skew the analysis.

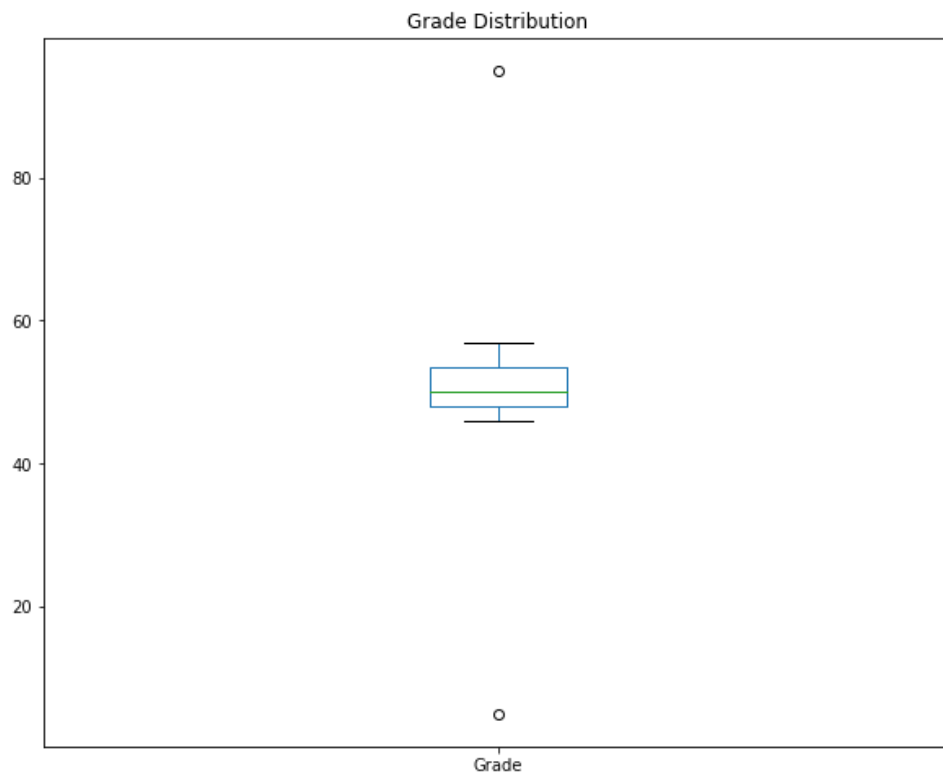
So is that OK? Can we really just ignore a data value we don't like?

Again, it depends on what you're analyzing. Let's take a look at the distribution of final grades:

```
In [20]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 17, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

# Plot a box-whisker chart
df['Grade'].plot(kind='box', title='Grade Distribution', figsize=(10,8))
plt.show()
```

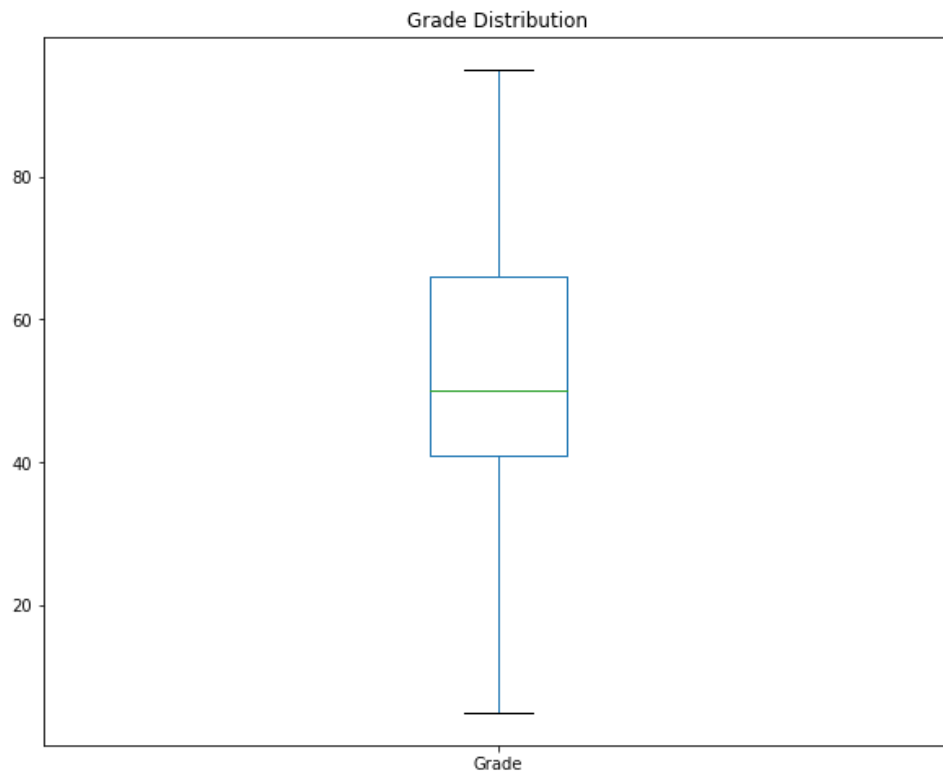


Once again there are outliers, this time at both ends of the distribution. However, think about what this data represents. If we assume that the grade for the final test is based on a score out of 100, it seems reasonable to expect that some students will score very low (maybe even 0) and some will score very well (maybe even 100); but most will get a score somewhere in the middle. The reason that the low and high scores here look like outliers might just be because we have so few data points. Let's see what happens if we include a few more students in our data:

```
In [21]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]]})

# Plot a box-whisker chart
df['Grade'].plot(kind='box', title='Grade Distribution', figsize=(10,8))
plt.show()
```



With more data, there are some more high and low scores; so we no longer consider the isolated cases to be outliers.

The key point to take away here is that you need to really understand the data and what you're trying to do with it, and you need to ensure that you have a reasonable sample size, before determining what to do with outlier values.

Variance and Standard Deviation

We've seen how to understand the *spread* of our data distribution using the range, percentiles, and quartiles; and we've seen the effect of outliers on the distribution. Now it's time to look at how to measure the amount of variance in the data.

Variance

Variance is measured as the average of the squared difference from the mean. For a full population, it's indicated by a squared Greek letter *sigma* (σ^2) and calculated like this:

$$\sigma^2 = \frac{\sum_{i=1}^N (X_i - \mu)^2}{N}$$

For a sample, it's indicated as s^2 calculated like this:

$$s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}$$

In both cases, we sum the difference between the individual data values and the mean and square the result. Then, for a full population we just divide by the number of data items to get the average. When using a sample, we divide by the total number of items **minus 1** to correct for sample bias.

Let's work this out for our student grades (assuming our data is a sample from the larger student population).

First, we need to calculate the mean grade:

$$\bar{x} = \frac{50 + 50 + 46 + 95 + 50 + 5 + 57}{7} \approx 50.43$$

Then we can plug that into our formula for the variance:

$$s^2 = \frac{(50 - 50.43)^2 + (50 - 50.43)^2 + (46 - 50.43)^2 + (95 - 50.43)^2 + (50 - 50.43)^2 + (5 - 50.43)^2 + (57 - 50.43)^2}{7 - 1}$$

So:

$$s^2 = \frac{0.185 + 0.185 + 19.625 + 1986.485 + 0.185 + 2063.885 + 43.165}{6}$$

Which simplifies to:

$$s^2 = \frac{4113.715}{6}$$

Giving the result:

$$s^2 \approx 685.619$$

The higher the variance, the more spread your data is around the mean.

In Python, you can use the **var** function of the *pandas.dataframe* class to calculate the variance of a column in a dataframe:

```
In [22]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                  'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                  'Hours': [41, 40, 36, 17, 35, 39, 40],
                  'Grade': [50, 50, 46, 95, 50, 5, 57]})

print(df['Grade'].var())

685.6190476190476
```

Standard Deviation

To calculate the variance, we squared the difference of each value from the mean. If we hadn't done this, the numerator of our fraction would always end up being zero (because the mean is at the center of our values). However, this means that the variance is not in the same unit of measurement as our data - in our case, since we're calculating the variance for grade points, it's in grade points squared; which is not very helpful.

To get the measure of variance back into the same unit of measurement, we need to find its square root:

$$s = \sqrt{685.619} \approx 26.184$$

So what does this value represent?

It's the *standard deviation* for our grades data. More formally, it's calculated like this for a full population:

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (X_i - \mu)^2}{N}}$$

Or like this for a sample:

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

Note that in both cases, it's just the square root of the corresponding variance formula!

In Python, you can calculate it using the **std** function:

```
In [23]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                  'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                  'Hours': [41, 40, 36, 17, 35, 39, 40],
                  'Grade': [50, 50, 46, 95, 50, 5, 57]})

print(df['Grade'].std())

26.184328282754315
```

Standard Deviation in a Normal Distribution

In statistics and data science, we spend a lot of time considering *normal* distributions; because they occur so frequently. The standard deviation has an important relationship to play in a normal distribution.

Run the following cell to show a histogram of a *standard normal* distribution (which is a distribution with a mean of 0 and a standard deviation of 1):


```
In [24]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats

# Create a random standard normal distribution
df = pd.DataFrame(np.random.randn(100000, 1), columns=['Grade'])

# Plot the distribution as a histogram with a density curve
grade = df['Grade']
density = stats.gaussian_kde(grade)
n, x, _ = plt.hist(grade, color='lightgrey', normed=True, bins=100)
plt.plot(x, density(x))

# Get the mean and standard deviation
s = df['Grade'].std()
m = df['Grade'].mean()

# Annotate 1 stdev
x1 = [m-s, m+s]
y1 = [0.25, 0.25]
plt.plot(x1, y1, color='magenta')
plt.annotate('1s (68.26%)', (x1[1], y1[1]))

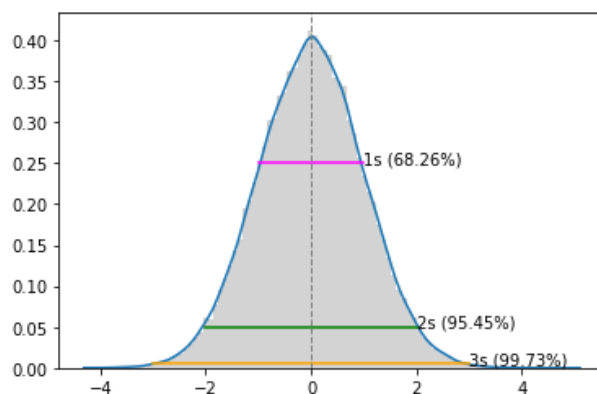
# Annotate 2 stdevs
x2 = [m-(s*2), m+(s*2)]
y2 = [0.05, 0.05]
plt.plot(x2, y2, color='green')
plt.annotate('2s (95.45%)', (x2[1], y2[1]))

# Annotate 3 stdevs
x3 = [m-(s*3), m+(s*3)]
y3 = [0.005, 0.005]
plt.plot(x3, y3, color='orange')
plt.annotate('3s (99.73%)', (x3[1], y3[1]))

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

plt.show()
```

/home/craig/workspace/ml_maths_py/venv/lib/python3.6/site-packages/ipykernel_launcher.py:13: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.
1. Use 'density' instead.
del sys.path[0]



The horizontal colored 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.

Z Score

So in a normal (or close to normal) distribution, standard deviation provides a way to evaluate how far from a mean a given range of values falls, allowing us to compare where a particular value lies within the distribution. For example, suppose Rosie tells you she was the highest scoring student among her friends - that doesn't really help us assess how well she scored. She may have scored only a fraction of a point above the second-highest scoring student. Even if we know she was in the top quartile; if we don't know how the rest of the grades are distributed it's still not clear how well she performed compared to her friends.

However, if she tells you how many standard deviations higher than the mean her score was, this will help you compare her score to that of her classmates.

So how do we know how many standard deviations above or below the mean a particular value is? We call this a *Z Score*, and it's calculated like this for a full population:

$$Z = \frac{x - \mu}{\sigma}$$

or like this for a sample:

$$Z = \frac{x - \bar{x}}{s}$$

So, let's examine Rosie's grade of 95. Now that we know the *mean* grade is 50.43 and the *standard deviation* is 26.184, we can calculate the Z Score for this grade like this:

$$Z = \frac{95 - 50.43}{26.184} = 1.702$$

So Rosie's grade is 1.702 standard deviations above the mean.

Summarizing Data Distribution in Python

We've seen how to obtain individual statistics in Python, but you can also use the ***describe*** function to retrieve summary statistics for all numeric columns in a dataframe. These summary statistics include many of the statistics we've examined so far (though it's worth noting that the *median* is not included):

```
In [25]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 17, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

print(df.describe())
```

	Salary	Hours	Grade
count	7.000000	7.000000	7.000000
mean	71000.000000	35.428571	50.428571
std	52370.475143	8.423324	26.184328
min	40000.000000	17.000000	5.000000
25%	50000.000000	35.500000	48.000000
50%	54000.000000	39.000000	50.000000
75%	57000.000000	40.000000	53.500000
max	189000.000000	41.000000	95.000000

Comparing Data

You'll often want to compare data in your dataset, to see if you can discern trends or relationships.

Univariate Data

Univariate data is data that consist of only one variable or feature. While it may initially seem as though there's not much we can do to analyze univariate data, we've already seen that we can explore its distribution in terms of measures of central tendency and measures of variance. We've also seen how we can visualize this distribution using histograms and box plots.

Here's a reminder of how you can visualize the distribution of univariate data, using our student grade data with a few additional observations in the sample:

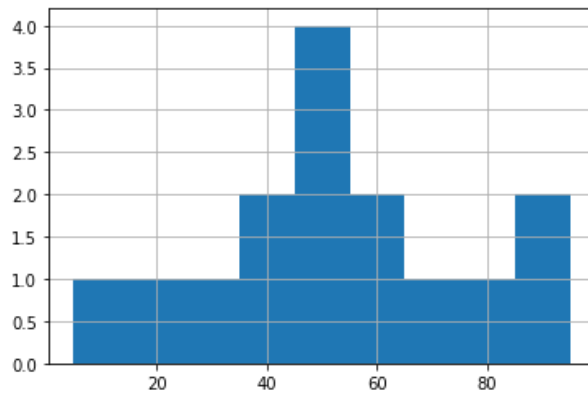
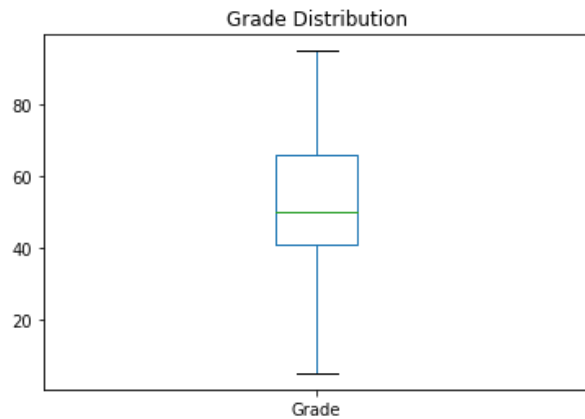
```

In [9]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Grade': [50,50,46,95,50,5,57,42,26,72,78,60,40,17,85]]})

plt.figure()
df['Grade'].plot( kind='box', title='Grade Distribution')
plt.figure()
df['Grade'].hist(bins=9)
plt.show()
print(df.describe())
print('median: ' + str(df['Grade'].median()))

```



```

      Grade
count  15.000000
mean   51.533333
std    24.642781
min     5.000000
25%    41.000000
50%    50.000000
75%    66.000000
max    95.000000
median: 50.0

```

Bivariate and Multivariate Data

It can often be useful to compare *bivariate* data; in other words, compare two variables, or even more (in which case we call it *multivariate* data).

For example, our student data includes three numeric variables for each student: their salary, the number of hours they work per week, and their final school grade. Run the following code to see an enlarged sample of this data as a table:

```
In [10]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4
                             'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                             'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

df[['Name', 'Salary', 'Hours', 'Grade']]
```

Out[10]:

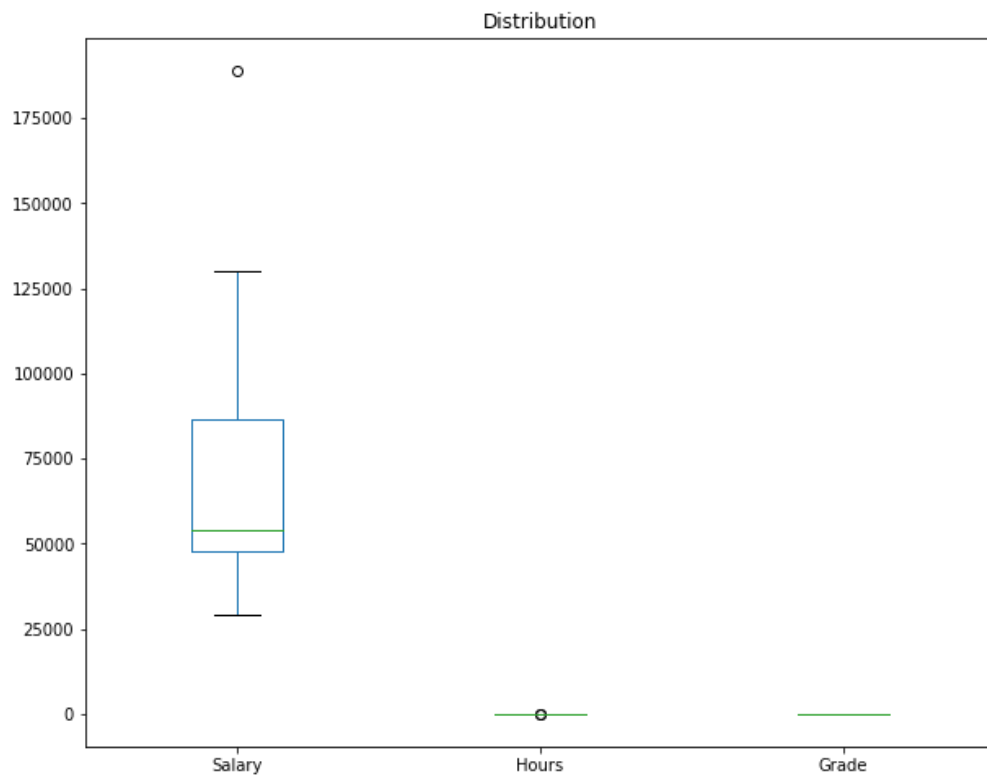
	Name	Salary	Hours	Grade
0	Dan	50000	41	50
1	Joann	54000	40	50
2	Pedro	50000	36	46
3	Rosie	189000	17	95
4	Ethan	55000	35	50
5	Vicky	40000	39	5
6	Frederic	59000	40	57
7	Jimmie	42000	45	42
8	Rhonda	47000	41	26
9	Giovanni	78000	35	72
10	Francesca	119000	30	78
11	Rajab	95000	33	60
12	Naiyana	49000	38	40
13	Kian	29000	47	17
14	Jenny	130000	24	85

Let's suppose you want to compare the distributions of these variables. You might simply create a boxplot for each variable, like this:

```
In [11]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4],
                   'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                   'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

df.plot(kind='box', title='Distribution', figsize = (10,8))
plt.show()
```



Hmm, that's not particularly useful is it?

The problem is that the data are all measured in different scales. Salaries are typically in tens of thousands, while hours and grades are in single or double digits.

Normalizing Data

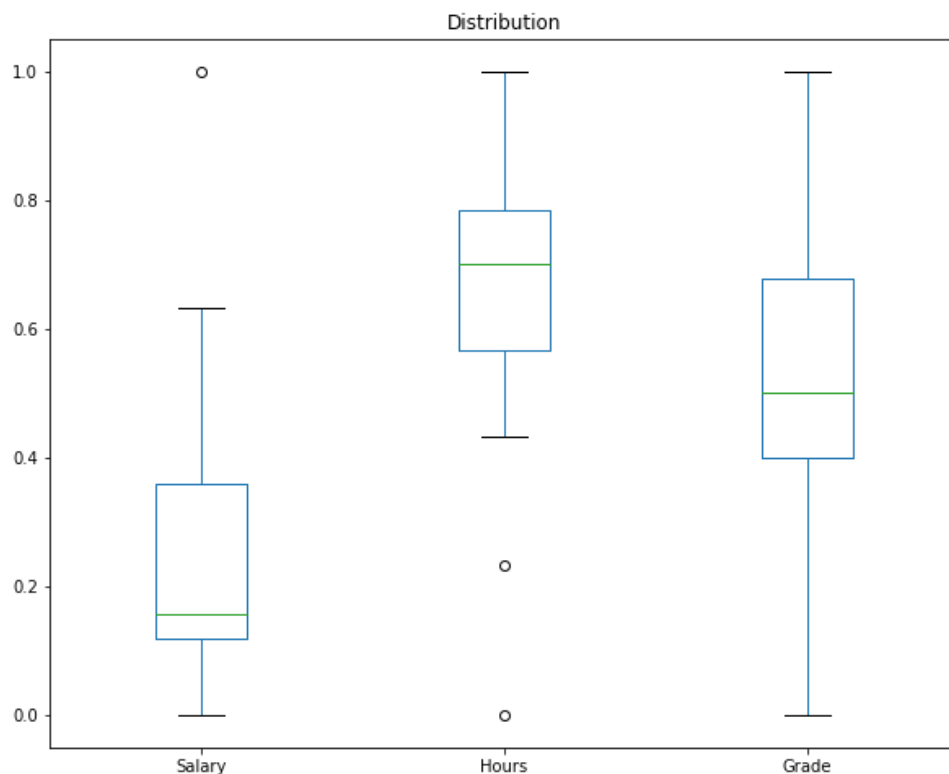
When you need to compare data in different units of measurement, you can *normalize* or *scale* the data so that the values are measured in the same proportional scale. For example, in Python you can use a MinMax scaler to normalize multiple numeric variables to a proportional value between 0 and 1 based on their minimum and maximum values. Run the following cell to do this:

```
In [12]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4
                             'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                             'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

# Normalize the data
scaler = MinMaxScaler()
df[['Salary', 'Hours', 'Grade']] = scaler.fit_transform(df[['Salary', 'Hours',

# Plot the normalized data
df.plot(kind='box', title='Distribution', figsize = (10,8))
plt.show()
```



Now the numbers on the y axis aren't particularly meaningful, but they're on a similar scale.

Comparing Bivariate Data with a Scatter Plot

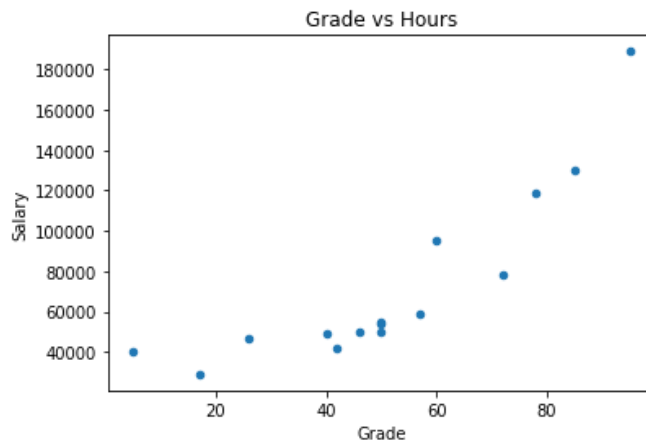
When you need to compare two numeric values, a scatter plot can be a great way to see if there is any apparent relationship between them so that changes in the value of one variable affect the value of the other.

Let's look at a scatter plot of *Salary* and *Grade*:


```
In [13]: %matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4
                             'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                             'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

# Create a scatter plot of Salary vs Grade
df.plot(kind='scatter', title='Grade vs Hours', x='Grade', y='Salary')
plt.show()
```



Look closely at the scatter plot. Can you see a diagonal trend in the plotted points, rising up to the right? It looks as though the higher the student's grade is, the higher their salary is.

You can see the trend more clearly by adding a *line of best fit* (sometimes called a *trendline*) to the plot:

```

In [14]: %matplotlib inline
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4
                             'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                             'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

# Create a scatter plot of Salary vs Grade
df.plot(kind='scatter', title='Grade vs Salary', x='Grade', y='Salary')

# Add a line of best fit
plt.plot(np.unique(df['Grade']), np.poly1d(np.polyfit(df['Grade'], df['Salary']
plt.show()

```



The line of best fit makes it clearer that there is some apparent *colinearity* between these variables (the relationship is *colinear* if one variable's value increases or decreases in line with the other).

Correlation

The apparently colinear relationship you saw in the scatter plot can be verified by calculating a statistic that quantifies the relationship between the two variables. The statistic usually used to do this is *correlation*, though there is also a statistic named *covariance* that is sometimes used. Correlation is generally preferred because the value it produces is more easily interpreted.

A correlation value is always a number between **-1** and **1**.

- A positive value indicates a positive correlation (as the value of variable *x* increases, so does the value of variable *y*).
- A negative value indicates a negative correlation (as the value of variable *x* increases, the value of variable *y* decreases).
- The closer to zero the correlation value is, the weaker the correlation between *x* and *y*.
- A correlation of exactly zero means there is no apparent relationship between the variables.

The formula to calculate correlation is:

$$r_{x,y} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 (y_i - \bar{y})^2}}$$

$r_{x,y}$ is the notation for the *correlation between x and y*.

The formula is pretty complex, but fortunately Python makes it very easy to calculate the correlation by using the **corr** function:

```
In [15]: import pandas as pd

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000],
                   'Hours': [41, 40, 36, 17, 35, 39, 40],
                   'Grade': [50, 50, 46, 95, 50, 5, 57]})

# Calculate the correlation between *Salary* and *Grade*
print(df['Grade'].corr(df['Salary']))

0.8149286388911882
```

In this case, the correlation is just over 0.8; making it a reasonably high positive correlation that indicates salary increases in line with grade.

Let's see if we can find a correlation between *Grade* and *Hours*:

```
In [16]: %matplotlib inline
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

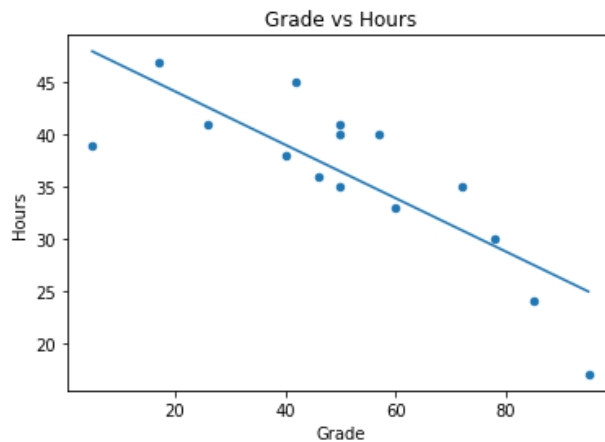
df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                           'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4
                           'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                           'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

r = df['Grade'].corr(df['Hours'])
print('Correlation: ' + str(r))

# Create a scatter plot of Salary vs Grade
df.plot(kind='scatter', title='Grade vs Hours', x='Grade', y='Hours')

# Add a line of best fit-
plt.plot(np.unique(df['Grade']), np.polyd(np.polyfit(df['Grade'], df['Hours'],
plt.show()
```

Correlation: -0.8109119058459785



In this case, the correlation value is just under -0.8; meaning a fairly strong negative correlation in which the number of hours worked decreases as the grade increases. The line of best fit on the scatter plot corroborates this statistic.

It's important to remember that *correlation is not causation*. In other words, even though there's an apparent relationship, you can't say for sure that one variable is the cause of the other. In this example, we can say that students who achieved higher grades tend to work shorter hours; but we **can't** say that those who work shorter hours do so *because* they achieved a high grade!

Least Squares Regression

In the previous examples, we drew a line on a scatter plot to show the *best fit* of the data. In many cases, your initial attempts to identify any colinearity might involve adding this kind of line by hand (or just mentally visualizing it); but as you may suspect from the use of the `numpy.polyfit` function in the code above, there are ways to calculate the coordinates for this line mathematically. One of the most commonly used techniques is *least squares regression*, and that's what we'll look at now.

Cast your mind back to when you were learning how to solve linear equations, 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 of the line.

In the case of our scatter plot for our former-student's working hours, we already have our values for x (*Grade*) and y (*Hours*), 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 the a new y value on that line for each of our x (*Grade*) values - to avoid confusion, we'll call this new y value $f(x)$ (because it's the output from a linear equation function based on x). The difference between the original y (*Hours*) value and the $f(x)$ value is the *error* between our regression line of best fit and the actual *Hours* worked by the former 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*.

So how do we accomplish this? First we need to calculate the slope (m), which we do using this formula (in which n is the number of observations in our data sample):

$$m = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

After we've calculated the slope (m), we can use it to calculate the intercept (b) like this:

$$b = \frac{\sum y - m(\sum x)}{n}$$

Let's look at a simple example that compares the number of hours of nightly study each student undertook with the final grade the student achieved:

Name	Study	Grade
Dan	1	50
Joann	0.75	50
Pedro	0.6	46
Rosie	2	95
Ethan	1	50
Vicky	0.2	5
Frederic	1.2	57

First, let's take each x (Study) and y (Grade) pair and calculate x^2 and xy , because we're going to need these to work out the slope:

Name	Study	Grade	x^2	xy
Dan	1	50	1	50
Joann	0.75	50	0.55	37.5

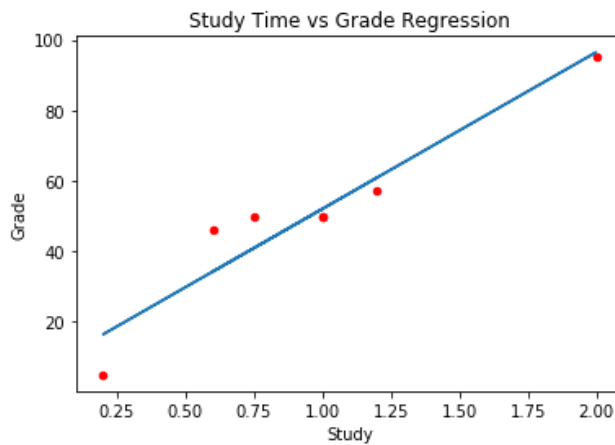
```
In [17]: %matplotlib inline
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Study': [1, 0.75, 0.6, 2, 1, 0.2, 1.2],
                   'Grade': [50, 50, 46, 95, 50, 5, 57],
                   'fx': [52.0159, 40.9106, 34.2480, 96.4321, 52.0149, 16.4811, 60.898]}

# Create a scatter plot of Study vs Grade
df.plot(kind='scatter', title='Study Time vs Grade Regression', x='Study', y='Grade')

# Plot the regression line
plt.plot(df['Study'], df['fx'])

plt.show()
```



In this case, the line fits the middle values fairly well, but is less accurate for the outlier at the low end. This is often the case, which is why statisticians and data scientists often *treat* outliers by removing them or applying a threshold value; though in this example there are too few data points to conclude that the data points are really outliers.

Let's look at a slightly larger dataset and apply the same approach to compare *Grade* and *Salary*:

```

In [18]: %matplotlib inline
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky'],
                   'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4],
                   'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                   'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

# Calculate least squares regression line
df['x2'] = df['Grade']**2
df['xy'] = df['Grade'] * df['Salary']
x = df['Grade'].sum()
y = df['Salary'].sum()
x2 = df['x2'].sum()
xy = df['xy'].sum()
n = df['Grade'].count()
m = ((n*xy) - (x*y))/((n*x2)-(x**2))
b = (y - (m*x))/n
df['fx'] = (m*df['Grade']) + b
df['error'] = df['fx'] - df['Salary']

print('slope: ' + str(m))
print('y-intercept: ' + str(b))

# Create a scatter plot of Grade vs Salary
df.plot(kind='scatter', title='Grade vs Salary Regression', x='Grade', y='Salary')

# Plot the regression line
plt.plot(df['Grade'], df['fx'])

plt.show()

# Show the original x,y values, the f(x) value, and the error
df[['Grade', 'Salary', 'fx', 'error']]

```

slope: 1516.1378856076408
 y-intercept: -5731.639038313754



Out[18]:

	Grade	Salary	fx	error
0	50	50000	70075.255242	20075.255242
1	50	54000	70075.255242	16075.255242
2	46	50000	64010.703700	14010.703700
3	95	189000	138301.460094	-50698.539906
4	50	55000	70075.255242	15075.255242
5	5	40000	1849.050390	-38150.949610

In this case, we used Python expressions to calculate the *slope* and *y-intercept* using the same approach and formula as before. In practice, Python provides great support for statistical operations like this; and you can use the ***linregress*** function in the *scipy.stats* package to retrieve the *slope* and *y-intercept* (as well as the *correlation*, *p-value*, and *standard error*) for a matched array of *x* and *y* values (we'll discuss *p-values* later!).

Here's the Python code to calculate the regression line variables using the ***linregress*** function:


```

In [19]: %matplotlib inline
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from scipy import stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4
                             'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                             'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

# Get the regression line slope and intercept
m, b, r, p, se = stats.linregress(df['Grade'], df['Salary'])

df['fx'] = (m*df['Grade']) + b
df['error'] = df['fx'] - df['Salary']

print('slope: ' + str(m))
print('y-intercept: ' + str(b))

# Create a scatter plot of Grade vs Salary
df.plot(kind='scatter', title='Grade vs Salary Regression', x='Grade', y='Salary')

# Plot the regression line
plt.plot(df['Grade'], df['fx'])

plt.show()

# Show the original x,y values, the f(x) value, and the error
df[['Grade', 'Salary', 'fx', 'error']]

```

slope: 1516.1378856076406
 y-intercept: -5731.639038313733



Out[19]:

	Grade	Salary	fx	error
0	50	50000	70075.255242	20075.255242
1	50	54000	70075.255242	16075.255242
2	46	50000	64010.703700	14010.703700
3	95	189000	138301.460094	-50698.539906
4	50	55000	70075.255242	15075.255242
5	5	40000	1849.050390	-38150.949610
6	57	59000	80688.220441	21688.220441
7	42	42000	57946.152157	15946.152157
8	26	47000	33687.945987	-13312.054013

Note that the *slope* and *y-intercept* values are the same as when we worked them out using the formula.

Similarly to the simple study hours example, the regression line doesn't fit the outliers very well. In this case, the extremes include a student who scored only 5, and a student who scored 95. Let's see what happens if we remove these students from our sample:

```
In [20]: %matplotlib inline
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from scipy import stats

df = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky',
                             'Salary': [50000, 54000, 50000, 189000, 55000, 40000, 59000, 42000, 4
                             'Hours': [41, 40, 36, 17, 35, 39, 40, 45, 41, 35, 30, 33, 38, 47, 24],
                             'Grade': [50, 50, 46, 95, 50, 5, 57, 42, 26, 72, 78, 60, 40, 17, 85]})

df = df[(df['Grade'] > 5) & (df['Grade'] < 95)]

# Get the regression line slope and intercept
m, b, r, p, se = stats.linregress(df['Grade'], df['Salary'])

df['fx'] = (m*df['Grade']) + b
df['error'] = df['fx'] - df['Salary']

print('slope: ' + str(m))
print('y-intercept: ' + str(b))

# Create a scatter plot of Grade vs Salary
df.plot(kind='scatter', title='Grade vs Salary Regression', x='Grade', y='Salary')

# Plot the regression line
plt.plot(df['Grade'], df['fx'])

plt.show()

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

slope: 1424.5008823224111
y-intercept: -7822.237984844818



Out[20]:

	Grade	Salary	fx	error
0	50	50000	63402.806131	13402.806131
1	50	54000	63402.806131	9402.806131
2	46	50000	57704.802602	7704.802602
4	50	55000	63402.806131	8402.806131
6	57	59000	73374.312308	14374.312308
7	42	42000	52006.799073	10006.799073
8	26	47000	29214.784956	-17785.215044
9	72	78000	94741.825542	16741.825542

With the outliers removed, the line is a slightly better overall fit to the data.

One of the neat things about regression is that it gives us a formula and some constant values that we can use to estimate a y value for any x value. We just need to apply the linear function using the *slope* and *y-intercept* values we've calculated from our sample data. For example, suppose a student named Fabio graduates from our school with a final grade of **62**. We can use our linear function with the *slope* and *y-intercept* values we calculated with Python to estimate what salary he can expect to earn:

$$f(x) = (1424.50 \times 62) - 7822.24 \approx 80,497$$

Probability

Many of the problems we try to solve using statistics are to do with *probability*. For example, what's the probable salary for a graduate who scored a given score in their final exam at school? Or, what's the likely height of a child given the height of his or her parents?

It therefore makes sense to learn some basic principles of probability as we study statistics.

Probability Basics

Let's start with some basic definitions and principles.

- An **experiment** or **trial** is an action with an uncertain outcome, such as tossing a coin.
- A **sample space** is the set of all possible outcomes of an experiment. In a coin toss, there's a set of two possible outcomes (*heads* and *tails*).
- A **sample point** is a single possible outcome - for example, *heads*
- An **event** is a specific outcome of single instance of an experiment - for example, tossing a coin and getting *tails*.
- **Probability** is a value between 0 and 1 that indicates the likelihood of a particular event, with 0 meaning that the event is impossible, and 1 meaning that the event is inevitable. In general terms, it's calculated like this:

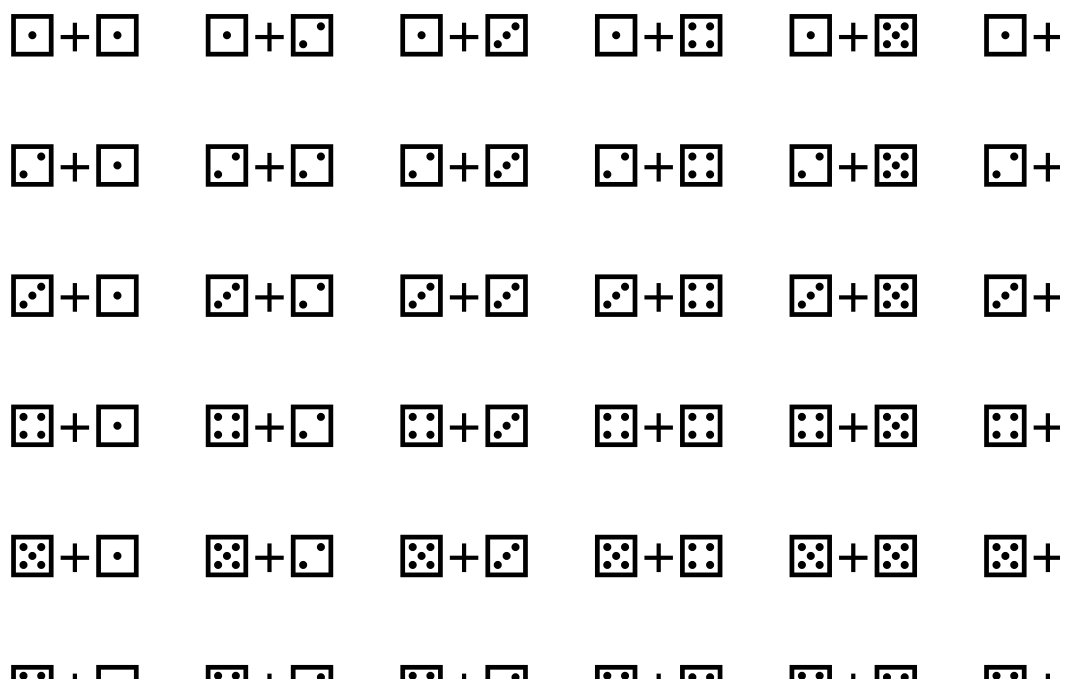
$$\text{probability of an event} = \frac{\text{Number of sample points that produce the event}}{\text{Total number of sample points in the sample space}}$$

For example, the probability of getting *heads* when tossing a coin is $\frac{1}{2}$ - there is only one side of the coin that is designated *heads*. and there are two possible outcomes in the sample space (*heads* and *tails*). So the probability of getting *heads* in a single coin toss is 0.5 (or 50% when expressed as a percentage).

Let's look at another example. Suppose you throw two dice, hoping to get 7.

The dice throw itself is an *experiment* - you don't know the outcome until the dice have landed and settled.

The *sample space* of all possible outcomes is every combination of two dice - 36 *sample points*:





Conditional Probability and Dependence

Events can be:

- *Independent* (events that are not affected by other events)
- *Dependent* (events that are conditional on other events)
- *Mutually Exclusive* (events that can't occur together)

Independent Events

Imagine you toss a coin. The sample space contains two possible outcomes: heads () or tails ()

The probability of getting *heads* is $\frac{1}{2}$, and the probability of getting *tails* is also $\frac{1}{2}$. Let's toss a coin...



OK, so we got *heads*. Now, let's toss the coin again:



It looks like we got *heads* again. If we were to toss the coin a third time, what's the probability that we'd get *heads*?

Although you might be tempted to think that a *tail* is overdue, the fact is that each coin toss is an independent event. The outcome of the first coin toss does not affect the second coin toss (or the third, or any number of other coin tosses). For each independent coin toss, the probability of getting *heads* (or *tails*) remains $\frac{1}{2}$, or 50%.

Run the following Python code to simulate 10,000 coin tosses by assigning a random value of 0 or 1 to *heads* and *tails*. Each time the coin is tossed, the probability of getting *heads* or *tails* is 50%, so you should expect approximately half of the results to be *heads* and half to be *tails* (it won't be exactly half, due to a little random variation; but it should be close):

```
In [1]: %matplotlib inline
import random

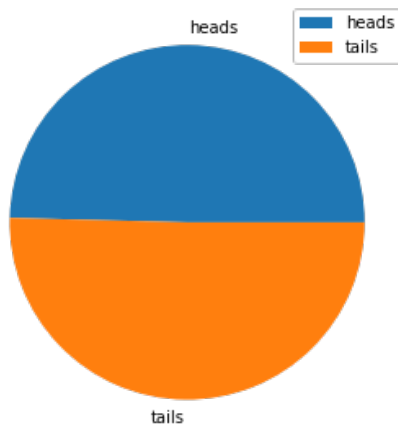
# Create a list with 2 element (for heads and tails)
heads_tails = [0,0]

# loop through 10000 trials
trials = 10000
trial = 0
while trial < trials:
    trial = trial + 1
    # Get a random 0 or 1
    toss = random.randint(0,1)
    # Increment the list element corresponding to the toss result
    heads_tails[toss] = heads_tails[toss] + 1

print (heads_tails)

# Show a pie chart of the results
from matplotlib import pyplot as plt
plt.figure(figsize=(5,5))
plt.pie(heads_tails, labels=['heads', 'tails'])
plt.legend()
plt.show()

[4960, 5040]
```



Combining Independent Events

Now, let's ask a slightly different question. What is the probability of getting three *heads* in a row? Since the probability of a heads on each independent toss is $\frac{1}{2}$, you might be tempted to think that the same probability applies to getting three *heads* in a row; but actually, we need to treat getting three *heads* as it's own event, which is the combination of three independent events. To combine independent events like this, we need to multiply the probability of each event. So:

$$\text{☀} = \frac{1}{2}$$

$$\text{☀} \text{☀} = \frac{1}{2} \times \frac{1}{2}$$

$$\text{☀} \text{☀} \text{☀} = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2}$$

So the probability of tossing three *heads* in a row is $0.5 \times 0.5 \times 0.5$, which is 0.125 (or 12.5%).

Run the code below to simulate 10,000 trials of flipping a coin three times:

```
# Count the number of 3xHeads results
h3 = 0

# Create a list of all results
results = []

# loop through 10000 trials
trials = 10000
trial = 0
while trial < trials:
    trial = trial + 1
    # Flip three coins
    result = ['H' if random.randint(0,1) == 1 else 'T',
              'H' if random.randint(0,1) == 1 else 'T',
              'H' if random.randint(0,1) == 1 else 'T']
    results.append(result)
    # If it's three heads, add it to the count
    h3 = h3 + int(result == ['H', 'H', 'H'])

# What proportion of trials produced 3x heads
print("%.2f%%" % ((h3/trials)*100))

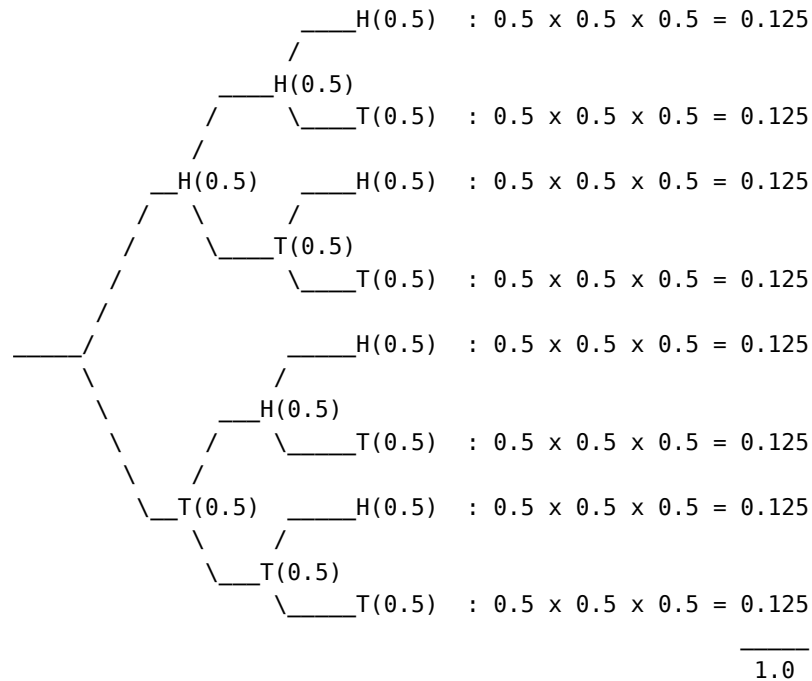
# Show all the results
print(results)
```

[illegible]

The output shows the percentage of times a trial resulted in three heads (which should be somewhere close to 12.5%). You can count the number of *['H', 'H', 'H']* entries in the full list of results to verify this if you like!

Probability Trees

You can represent a series of events and their probabilities as a probability tree:



Starting at the left, you can follow the branches in the tree that represent each event (in this case a coin toss result of *heads* or *tails* at each branch). Multiplying the probability of each branch of your path through the tree gives you the combined probability for an event composed of all of the events in the path. In this case, you can see from the tree that you are equally likely to get any sequence of three *heads* or *tails* results (so three *heads* is just as likely as three *tails*, which is just as likely as *head-tail-head*, *tail-head-tail*, or any other combination!)

Note that the total probability for all paths through the tree adds up to 1.

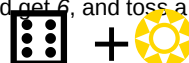
Combined Event Probability Notation

When calculating the probability of combined events, we assign a letter such as **A** or **B** to each event, and we use the *intersection* (\cap) symbol to indicate that we want the combined probability of multiple events. So we could assign the letters **A**, **B**, and **C** to each independent coin toss in our sequence of three tosses, and express the combined probability like this:

$$P(A \cap B \cap C) = P(A) \times P(B) \times P(C)$$

Combining Events with Different Probabilities

Imagine you have created a new game that mixes the excitement of coin-tossing with the thrill of die-rolling! The objective of the game is to roll a die and get 6, and toss a coin and get *heads*:



On each turn of the game, a player rolls the die and tosses the coin.

How can we calculate the probability of winning?

There are two independent events required to win: a die-roll of 6 (which we'll call event **A**), and a coin-toss of *heads* (which we'll call event **B**)

Dependent Events

Let's return to our deck of 52 cards from which we're going to draw one card. The sample space can be summarized like this:

13 x  13 x  13 x  13 x 

There are two black suits (*spades* and *clubs*) and two red suits (*hearts* and *diamonds*); with 13 cards in each suit. So the probability of drawing a black card (event **A**) and the probability of drawing a red card (event **B**) can be calculated like this:

$$P(A) = \frac{13 + 13}{52} = \frac{26}{52} = 0.5 \quad P(B) = \frac{13 + 13}{52} = \frac{26}{52} = 0.5$$

Now let's draw a card from the deck:



We drew a heart, which is red. So, assuming we don't replace the card back into the deck, this changes the sample space as follows:

12 x  13 x  13 x  13 x 

The probabilities for **A** and **B** are now:

$$P(A) = \frac{13 + 13}{51} = \frac{26}{51} = 0.51 \quad P(B) = \frac{12 + 13}{51} = \frac{25}{51} = 0.49$$

Now let's draw a second card:



We drew a diamond, so again this changes the sample space for the next draw:

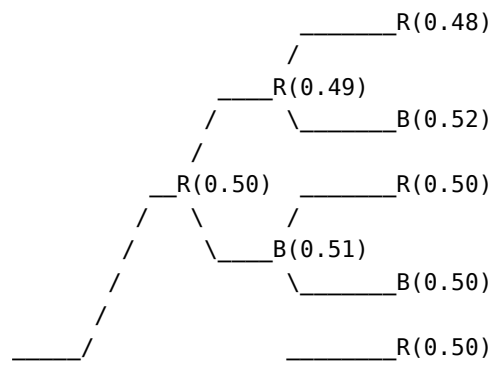
12 x  13 x  13 x  12 x 

The probabilities for **A** and **B** are now:

$$P(A) = \frac{13 + 13}{50} = \frac{26}{50} = 0.52 \quad P(B) = \frac{12 + 12}{50} = \frac{24}{50} = 0.48$$

So it's clear that one event can affect another; in this case, the probability of drawing a card of a particular color on the second draw depends on the color of card drawn on the previous draw. We call these *dependent* events.

Probability trees are particularly useful when looking at dependent events. Here's a probability tree for drawing red or black cards as the first three draws from a deck of cards:



Binomial Variables and Distributions

Now that we know something about probability, let's apply that to statistics. Statistics is about inferring measures for a full population based on samples, allowing for random variation; so we're going to have to consider the idea of a *random variable*.

A random variable is a number that can vary in value. For example, the temperature on a given day, or the number of students taking a class.

Binomial Variables

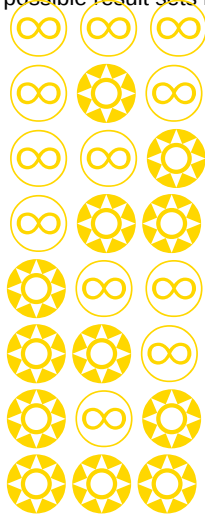
One particular type of random variable that we use in statistics is a *binomial* variable. A binomial variable is used to count how frequently an event occurs in a fixed number of repeated independent experiments. The event in question must have the same probability of occurring in each experiment, and indicates the success or failure of the experiment; with a probability p of success, which has a complement of $1 - p$ as the probability of failure (we often call this kind of experiment a *Bernoulli Trial* after Swiss mathematician Jacob Bernoulli).

For example, suppose we flip a coin three times, counting *heads* as success. We can define a binomial variable to represent the number of successful coin flips (that is, the number of times we got *heads*).

Let's examine this in more detail.

We'll call our variable X , and as stated previously it represents the number of times we flip *heads* in a series of three coin flips. Let's start by examining all the possible values for X .

We're flipping the coin three times, with a probability of $1/2$ of success on each flip. The possible results include none of the flips resulting in *heads*, all of the flips resulting in *heads*, or any combination in between. There are two possible outcomes from each flip, and there are three flips, so the total number of possible result sets is 2^3 , which is 8. Here they are:



In these results, our variable X , representing the number of successful events (getting *heads*), can vary from 0 to 3. We can write that like this:

$$X = \{0, 1, 2, 3\}$$

When we want to indicate a specific outcome for a random variable, we use write the variable in lower case, for example x . So what's the probability that $x = 0$ (meaning that out of our three flips we got no *heads*)?

We can easily see, that there is 1 row in our set of possible outcomes that contains no *heads*, so:

$$P(x = 0) = \frac{1}{8}$$

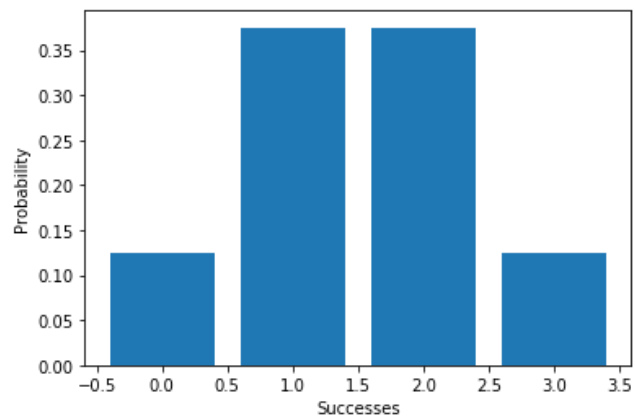
```
In [3]: %matplotlib inline
from scipy import special as sps
from matplotlib import pyplot as plt
import numpy as np

trials = 3

possibilities = 2**trials
x = np.array(range(0, trials+1))

p = np.array([sps.comb(trials, i, exact=True)/possibilities for i in x])

# Set up the graph
plt.xlabel('Successes')
plt.ylabel('Probability')
plt.bar(x, p)
plt.show()
```



Allowing for Bias

Previously, we calculated the probability for each possible value of a random variable by simply dividing the number of combinations for that value by the total number of possible outcomes. This works if the probability of the event being tested is equal for failure and success; but of course, not all experiments have an equal chance of success or failure. Some include a bias that makes success more or less likely - so we need to be a little more thorough in our calculations to allow for this.

Suppose you're flying off to some exotic destination, and you know that there's a one in four chance that the airport security scanner will trigger a random search for each passenger that goes through. If you watch five passengers go through the scanner, how many will be stopped for a random search?

It's tempting to think that there's a one in four chance, so a quarter of the passengers will be stopped; but remember that the searches are triggered randomly for thousands of passengers that pass through the airport each day. It's possible that none of the next five passengers will be searched; all five of them will be searched, or some other value in between will be searched.

Even though the probabilities of being searched or not searched are not the same, this is still a binomial variable. There are a fixed number of independent experiments (five passengers passing through the security scanner), the outcome of each experiment is either success (a search is triggered) or failure (no search is triggered), and the probability of being searched does not change for each passenger.

There are five experiments in which a passenger goes through the security scanner, let's call this **n**.

For each passenger, the probability of being searched is $\frac{1}{4}$ or 0.25. We'll call this **p**.

The complement of **p** (in other words, the probability of *not* being searched) is **1-p**, in this case $\frac{3}{4}$ or 0.75.

So, what's the probability that out of our **n** experiments, three result in a search (let's call that **k**) and the remaining ones (there will be **n-k** of them, which is two) don't?

- The probability of three passengers being searched is $0.25 \times 0.25 \times 0.25$ which is the same as 0.25^3 . Using our generic variables, this is **p^k**.
- The probability that the rest don't get searched is 0.75×0.75 , or 0.75^2 . In terms of our variables, this is **1-p^(n-k)**.
- The combined probability of three searches and two non-searches is therefore $0.25^3 \times 0.75^2$ (approximately 0.088). Using our variables, this is:

$$p^k (1 - p)^{(n-k)}$$

This formula enables us to calculate the probability for a single combination of **n** passengers in which **k** experiments had a successful outcome. In this case, it enables us to calculate that the probability of three passengers out of five being searched is approximately 0.088. However, we need to consider that there are multiple ways this can happen. The first three passengers could get searched; or the last three; or the first, third, and fifth, or any other possible combination of 3 from 5.

There are two possible outcomes for each experiment; so the total number of possible combinations of five passengers being searched or not searched is 2^5 or 32. So within those 32 sets of possible result combinations, how many have three searches? We can use the ${}_nC_k$ formula to calculate this:

$${}_5C_3 = \frac{5!}{3!(5-3)!} = \frac{120}{6 \times 4} = \frac{120}{24} = 5$$

So 5 out of our 32 combinations had 3 searches and 2 non-searches.

To find the probability of any combination of 3 searches out of 5 passengers, we need to multiply the number of possible combinations by the probability for a single combination - in this case $\frac{5}{32} \times 0.088$, which is 0.01375, or 13.75%.

So our complete formula to calculate the probability of **k** events from **n** experiments with probability **p** is:

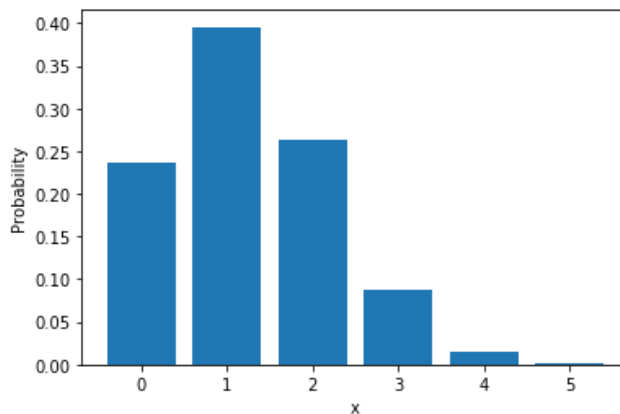
$$P(X = k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{(n-k)}$$

```
In [4]: %matplotlib inline
from scipy.stats import binom
from matplotlib import pyplot as plt
import numpy as np

n = 5
p = 0.25
x = np.array(range(0, n+1))

prob = np.array([binom.pmf(k, n, p) for k in x])

# Set up the graph
plt.xlabel('x')
plt.ylabel('Probability')
plt.bar(x, prob)
plt.show()
```



You can see from the bar chart that with this small value for n , the distribution is right-skewed.

Recall that in our coin flipping experiment, when the probability of failure vs success was equal, the resulting distribution was symmetrical. With an unequal probability of success in each experiment, the bias has the effect of skewing the overall probability mass.

However, try increasing the value of n in the code above to 10, 20, and 50; re-running the cell each time. With more observations, the *central limit theorem* starts to take effect and the distribution starts to look more symmetrical - with enough observations it starts to look like a *normal* distribution.

There is an important distinction here - the *normal* distribution applies to *continuous* variables, while the *binomial* distribution applies to *discrete* variables. However, the similarities help in a number of statistical contexts where the number of observations (experiments) is large enough for the *central limit theorem* to make the distribution of binomial variable values behave like a *normal* distribution.

Working with the Binomial Distribution

Now that you know how to work out a binomial distribution for a repeated experiment, it's time to take a look at some statistics that will help us quantify some aspects of probability.

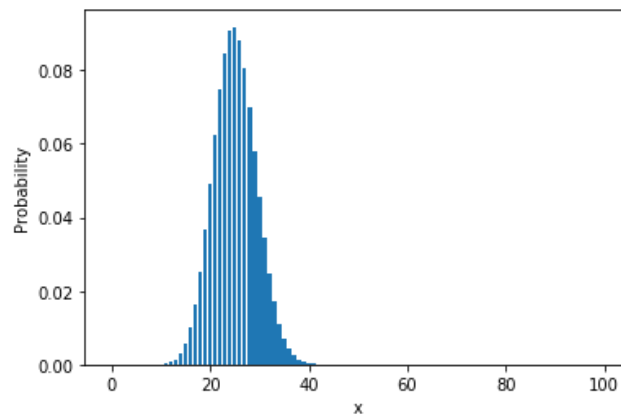
Let's increase our n value to 100 so that we're looking at the number of searches per 100 passengers. This gives us the binomial distribution graphed by the following code:

```
In [5]: %matplotlib inline
from scipy.stats import binom
from matplotlib import pyplot as plt
import numpy as np

n = 100
p = 0.25
x = np.array(range(0, n+1))

prob = np.array([binom.pmf(k, n, p) for k in x])

# Set up the graph
plt.xlabel('x')
plt.ylabel('Probability')
plt.bar(x, prob)
plt.show()
```



Mean (Expected Value)

We can calculate the mean of the distribution like this:

$$\mu = np$$

So for our airport passengers, this is:

$$\mu = 100 \times 0.25 = 25$$

When we're talking about a probability distribution, the mean is usually referred to as the *expected value*. In this case, for any 100 passengers we can reasonably expect 25 of them to be searched.

Variance and Standard Deviation

Obviously, we can't search a quarter of a passenger - the expected value reflects the fact that there is variation, and indicates an average value for our binomial random variable. To get an indication of how much variability there actually is in this scenario, we can calculate the variance and standard deviation.

For variance of a binomial probability distribution, we can use this formula:

$$\sigma^2 = np(1 - p)$$

So for our airport passengers:

$$\sigma^2 = 100 \times 0.25 \times 0.75 = 18.75$$

To convert this to standard deviation we just take the square root:

$$\sigma = \sqrt{np(1 - p)}$$

So:

$$\sigma = \sqrt{18.75} \approx 4.33$$

So for every 100 passengers, we can expect 25 searches with a standard deviation of 4.33

In Python, you can use the **mean**, **var**, and **std** functions from the `scipy.stats.binom` package to return binomial distribution statistics for given values of *n* and *p*:

In [6]: `from scipy.stats import binom`

```
n = 100
p = 0.25

print(binom.mean(n,p))
print(binom.var(n,p))
print(binom.std(n,p))

25.0
18.75
4.330127018922194
```


Working with Sampling Distributions

Most statistical analysis involves working with distributions - usually of sample data.

Sampling and Sampling Distributions

As we discussed earlier, when working with statistics, we usually base our calculations on a sample and not the full population of data. This means we need to allow for some variation between the sample statistics and the true parameters of the full population.

In the previous example, we knew the probability that a security search would be triggered was 25%, so it's pretty easy to calculate that the expected value for a random variable indicating the number of searches per 100 passengers is 25. What if we hadn't known the probability of a search? How could we estimate the expected mean number of searches for a given number of passengers based purely on sample data collected by observing passengers go through security?

Creating a Proportion Distribution from a Sample

We know that each passenger will either be searched or not searched, and we can assign the values **0** (for not searched) and **1** (for searched) to these outcomes. We can conduct a Bernoulli trial in which we sample 16 passengers and calculate the fraction (or *proportion*) of passengers that were searched (which we'll call **p**), and the remaining proportion of passengers (which are the ones who weren't searched, and can be calculated as **1-p**).

Let's say we record the following values for our 16-person sample:

0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0

In this sample, there were 3 searches out of 16 passengers; which as a proportion is $\frac{3}{16}$ or 0.1875. This is our proportion (or **p**); but because we know that this is based on a sample, we call it **p̂** (or p-hat). The remaining proportion of passengers is 1-p; in this case 1 - 0.1875, which is 0.8125.

The data itself is *qualitative* (categorical) - we're indicating "no search" or "search"; but because we're using numeric values (0 and 1), we can treat these values as numeric and create a binomial distribution from them - it's the simplest form of a binomial distribution - a Bernoulli distribution with two values.

Because we're treating the results as a numeric distribution, we can also calculate statistics like *mean* and *standard deviation*:

To calculate these, you can use the following formulae:

$$\mu_{\hat{p}} = \hat{p}$$
$$\sigma_{\hat{p}} = \sqrt{\hat{p}(1 - \hat{p})}$$

The mean is just the value of **p̂**, so in the case of the passenger search sample it is 0.1875.

The standard deviation is calculated as:

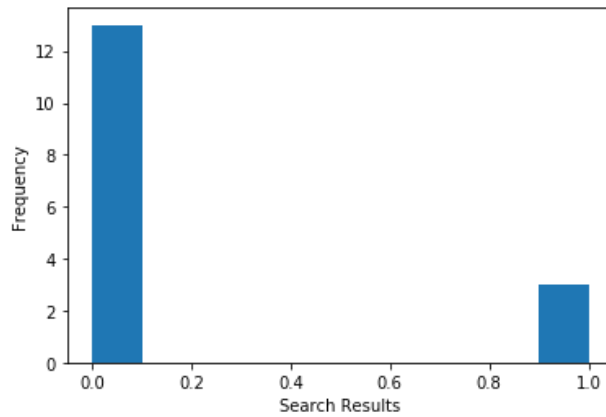
$$\sigma_{\hat{p}} = \sqrt{0.1875 \times 0.8125} \approx 0.39$$

We can use Python to plot the sample distribution and calculate the mean and standard deviation of our sample like this:

```
In [1]: %matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

searches = np.array([0,1,0,0,1,0,0,0,0,0,0,0,1,0,0,0])

# Set up the graph
plt.xlabel('Search Results')
plt.ylabel('Frequency')
plt.hist(searches)
plt.show()
print('Mean: ' + str(np.mean(searches)))
print('StDev: ' + str(np.std(searches)))
```



Mean: 0.1875
StDev: 0.3903123748998999

When talking about probability, the *mean* is also known as the *expected value*; so based on our single sample of 16 passengers, should we expect the proportion of searched passengers to be 0.1875 (18.75%)?

Well, using a single sample like this can be misleading because the number of searches can vary with each sample. Another person observing 100 passengers may get a (very) different result from you. One way to address this problem is to take multiple samples and combine the resulting means to form a *sampling* distribution. This will help us ensure that the distribution and statistics of our sample data is closer to the true values; even if we can't measure the full population.

Creating a Sampling Distribution of a Sample Proportion

So, let's collect multiple 16-passenger samples - here are the resulting sample proportions for 12 samples:

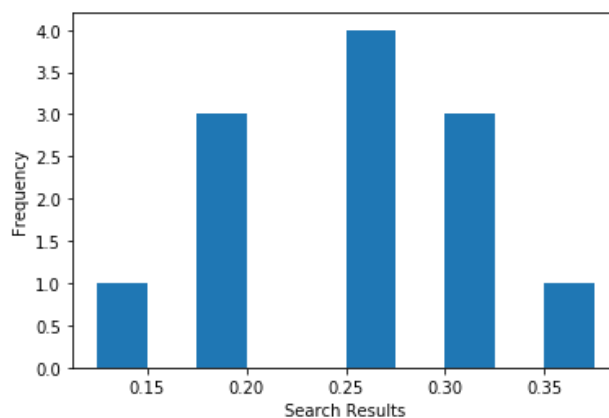
Sample	Result
\hat{p}_1	0.1875
\hat{p}_2	0.2500
\hat{p}_3	0.3125
\hat{p}_4	0.1875
\hat{p}_5	0.1250
\hat{p}_6	0.3750
\hat{p}_7	0.2500
\hat{p}_8	0.1875
\hat{p}_9	0.3125
\hat{p}_{10}	0.2500
\hat{p}_{11}	0.2500
\hat{p}_{12}	0.3125

We can plot these as a sampling distribution like this:

```
In [2]: %matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

searches = np.array([0.1875,0.25,0.3125,0.1875,0.125,0.375,0.25,0.1875,0.3125,0.25,0.25,0.3125])

# Set up the graph
plt.xlabel('Search Results')
plt.ylabel('Frequency')
plt.hist(searches)
plt.show()
```



The Central Limit Theorem

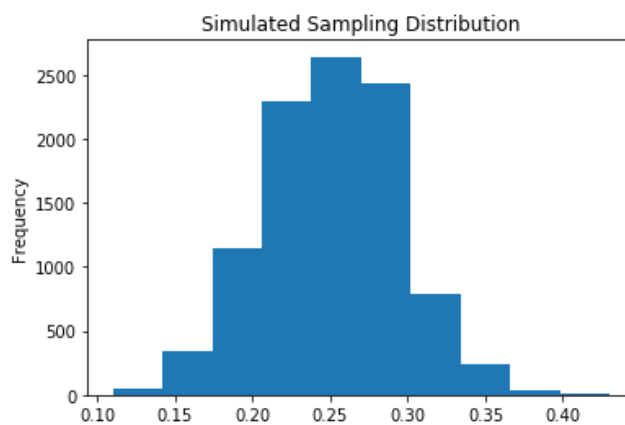
You saw previously with the binomial probability distribution, with a large enough sample size (the n value indicating the number of binomial experiments), the distribution of values for a random variable started to form an approximately *normal* curve. This is the effect of the *central limit theorem*, and it applies to any distribution of sample data if the size of the sample is large enough. For our airport passenger data, if we collect a large enough number of samples, each based on a large enough number of passenger observations, the sampling distribution will be approximately normal. The larger the sample size, the closer to a perfect *normal* distribution the data will be, and the less variance around the mean there will be.

Run the cell below to see a simulated distribution created by 10,000 random 100-passenger samples:

```
In [3]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

n, p, s = 100, 0.25, 10000
df = pd.DataFrame(np.random.binomial(n,p,s)/n, columns=['p-hat'])

# Plot the distribution as a histogram
means = df['p-hat']
means.plot.hist(title='Simulated Sampling Distribution')
plt.show()
print ('Mean: ' + str(means.mean()))
print ('Std: ' + str(means.std()))
```



Mean: 0.24923
Std: 0.04373006210644096

Mean and Standard Error of a Sampling Distribution of Proportion

The sampling distribution is created from the means of multiple samples, and its mean is therefore the mean of all the sample means. For a distribution of proportion means, this is considered to be the same as p (the population mean). In the case of our passenger search samples, this is 0.25.

Because the sampling distribution is based on means, and not totals, its standard deviation is referred to as its *standard error*, and its formula is:

$$\sigma_{\hat{p}} = \sqrt{\frac{p(1-p)}{n}}$$

In this formula, n is the size of each sample; and we divide by this to correct for the error introduced by the average values used in the sampling distribution. In this case, our samples were based on observing 16-passengers, so:

$$\sigma_{\hat{p}} = \sqrt{\frac{0.25 \times 0.75}{16}} \approx 0.11$$

In our simulation of 100-passenger samples, the mean remains 0.25. The standard error is:

$$\sigma_{\hat{p}} = \sqrt{\frac{0.25 \times 0.75}{100}} \approx 0.043$$

Note that the effect of the central limit theorem is that as you increase the number and/or size of samples, the mean remains constant but the amount of variance around it is reduced.

Being able to calculate the mean (or *expected value*) and standard error is useful, because we can apply these to what we know about an approximately normal distribution to estimate probabilities for particular values. For example, we know that in a normal distribution, around 95.4% of the values are within two standard deviations of the mean. If we apply that to our sampling distribution of ten thousand 100-passenger samples, we can determine that the proportion of searched passengers in 95.4% of the samples was between 0.164 (16.4%) and 0.336 (36.6%).

How do we know this?

We know that the mean is **0.25** and the standard error (which is the same thing as the standard deviation for our sampling distribution) is **0.043**. We also know that because this is a *normal* distribution, **95.4%** of the data lies within two standard deviations (so 2×0.043) of the mean, so the value for 95.4% of our samples is $0.25 \pm$ (*plus or minus*) 0.086.

The *plus or minus* value is known as the *margin of error*, and the range of values within it is known as a *confidence interval* - we'll look at these in more detail later. For now, run the following cell to see a visualization of this interval:

```

In [4]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

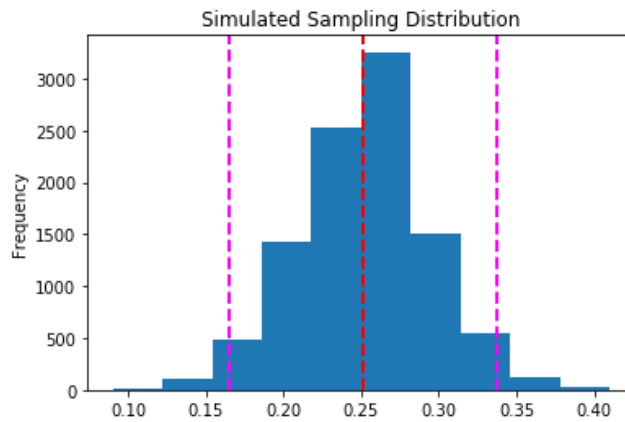
n, p, s = 100, 0.25, 10000
df = pd.DataFrame(np.random.binomial(n,p,s)/n, columns=['p-hat'])

# Plot the distribution as a histogram
means = df['p-hat']
m = means.mean()
sd = means.std()
moel = m - (sd * 2)
moe2 = m + (sd * 2)

means.plot.hist(title='Simulated Sampling Distribution')

plt.axvline(m, color='red', linestyle='dashed', linewidth=2)
plt.axvline(moel, color='magenta', linestyle='dashed', linewidth=2)
plt.axvline(moe2, color='magenta', linestyle='dashed', linewidth=2)
plt.show()

```



Creating a Sampling Distribution of Sample Means

In the previous example, we created a sampling distribution of proportions; which is a suitable way to handle discrete values, like the number of passengers searched or not searched. When you need to work with continuous data, you use slightly different formulae to work with the sampling distribution.

For example, suppose we want to examine the weight of the hand luggage carried by each passenger. It's impractical to weigh every bag that is carried through security, but we could weigh one or more samples, for say, 5 passengers at a time, on twelve occasions. We might end up with some data like this:

Sample	Weights
1	[4.020992,2.143457,2.260409,2.339641,4.699211]
2	[3.38532,4.438345,3.170228,3.499913,4.489557]
3	[3.338228,1.825221,3.53633,3.507952,2.698669]
4	[2.992756,3.292431,3.38148,3.479455,3.051273]
5	[2.969977,3.869029,4.149342,2.785682,3.03557]
6	[3.138055,2.535442,3.530052,3.029846,2.881217]
7	[1.596558,1.486385,3.122378,3.684084,3.501813]
8	[2.997384,3.818661,3.118434,3.455269,3.026508]
9	[4.078268,2.283018,3.606384,4.555053,3.344701]
10	[2.532509,3.064274,3.32908,2.981303,3.915995]
11	[4.078268,2.283018,3.606384,4.555053,3.344701]
12	[2.532509,3.064274,3.32908,2.981303,3.915995]

Just as we did before, we could take the mean of each of these samples and combine them to form a sampling distribution of the sample means (which we'll call \bar{X} , and which will contain a mean for each sample, which we'll label \bar{x}_n):

Sample	Mean Weight
\bar{x}_1	3.092742
\bar{x}_2	3.7966726
\bar{x}_3	2.98128
\bar{x}_4	3.239479
\bar{x}_5	3.36192
\bar{x}_6	3.0229224
\bar{x}_7	2.6782436
\bar{x}_8	3.2832512
\bar{x}_9	3.5734848
\bar{x}_{10}	3.1646322
\bar{x}_{11}	3.5734848
\bar{x}_{12}	3.1646322

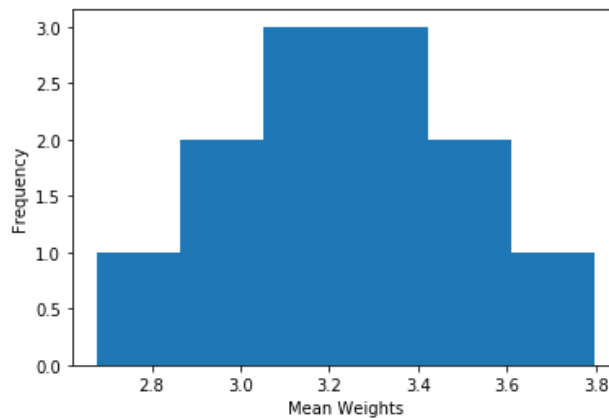
We can plot the distribution for the sampling distribution like this:

```
In [5]: %matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

meanweights = np.array([3.092742,
                        3.7966726,
                        2.98128,
                        3.239479,
                        3.36192,
                        3.0229224,
                        2.6782436,
                        3.2832512,
                        3.5734848,
                        3.1646322,
                        3.5734848,
                        3.1646322])

# Set up the graph
plt.xlabel('Mean Weights')
plt.ylabel('Frequency')
plt.hist(meanweights, bins=6)
plt.show()

print('Mean: ' + str(meanweights.mean()))
print('Std: ' + str(meanweights.std()))
```



Mean: 3.2443954
Std: 0.2903283632058937

Just as before, as we increase the sample size, the central limit theorem ensures that our sampling distribution starts to approximate a normal distribution. Our current distribution is based on the means generated from twelve samples, each containing 5 weight observations. Run the following code to see a distribution created from a simulation of 10,000 samples each containing weights for 500 passengers:

This may take a few minutes to run. The code is not the most efficient way to generate a sample distribution, but it reflects the principle that our sampling distribution is made up of the means from multiple samples. In reality, you could simulate the sampling by just creating a single sample from the **random.normal** function with a larger **n** value.


```

In [*]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

mu, sigma, n = 3.2, 1.2, 500
samples = list(range(0, 10000))

# data will hold all of the sample data
data = np.array([])

# sampling will hold the means of the samples
sampling = np.array([])

# Perform 10,000 samples
for s in samples:
    # In each sample, get 500 data points from a normal distribution
    sample = np.random.normal(mu, sigma, n)
    data = np.append(data, sample)
    sampling = np.append(sampling, sample.mean())

# Create a dataframe with the sampling of means
df = pd.DataFrame(sampling, columns=['mean'])

# Plot the distribution as a histogram
means = df['mean']
means.plot.hist(title='Simulated Sampling Distribution', bins=100)
plt.show()

# Print the Mean and StdDev for the full sample and for the sampling distribution
print('Sample Mean: ' + str(data.mean()))
print('Sample StdDev: ' + str(data.std()))
print('Sampling Mean: ' + str(means.mean()))
print('Sampling StdErr: ' + str(means.std()))

```

Mean and Variance of the Sampling Distribution

The following variables are printed beneath the histogram:

- **Sample Mean:** This is the mean for the complete set of sample data - all 10,000 x 500 bag weights.
- **Sample StdDev:** This is the standard deviation for the complete set of sample data - all 10,000 x 500 bag weights.
- **Sampling Mean:** This is the mean for the sampling distribution - the means of the means!
- **Sampling StdErr:** This is the standard deviation (or *standard error*) for the sampling distribution

If we assume that \mathbf{X} is a random variable representing every possible bag weight, then its mean (indicated as $\mu_{\mathbf{x}}$) is the population mean (μ). The mean of the $\bar{\mathbf{X}}$ sampling distribution (which is indicated as $\mu_{\bar{\mathbf{x}}}$) is considered to have the same value. Or, as an equation:

$$\mu_{\mathbf{x}} = \mu_{\bar{\mathbf{x}}}$$

In this case, the full population mean is unknown (unless we weigh every bag in the world!), but we do have the mean of the full set of sample observations we collected ($\bar{\mathbf{x}}$), and if we check the values generated by Python for the sample mean and the sampling mean, they're more or less the same: around 3.2.

To find the standard deviation of the sample mean, which is technically the *standard error*, we can use this formula:

$$\sigma_{\bar{\mathbf{x}}} = \frac{\sigma}{\sqrt{n}}$$

In this formula, σ is the population standard deviation and n is the size of each sample.

Since our the population standard deviation is unknown, we can use the full sample standard deviation instead:

$$SE_{\bar{\mathbf{x}}} \approx \frac{s}{\sqrt{n}}$$

In this case, the standard deviation of our set of sample data is around 1.2, and we have used 500 variables in each sample to calculate our sample means, so:

$$SE_{\bar{\mathbf{x}}} \approx \frac{1.2}{\sqrt{500}} = \frac{1.2}{22.36} \approx 0.053$$

Confidence Intervals

A confidence interval is a range of values around a sample statistic within which we are confident that the true parameter lies. For example, our bag weight sampling distribution is based on samples of the weights of bags carried by passengers through our airport security line. We know that the mean weight (the *expected value* for the weight of a bag) in our sampling distribution is 3.2, and we assume this is also the population mean for all bags; but how confident can we be that the true mean weight of all carry-on bags is close to the value?

Let's start to put some precision onto these terms. We could state the question another way. What's the range of weights within which are confident that the mean weight of a carry-on bag will be 95% of the time? To calculate this, we need to determine the range of values within which the population mean weight is likely to be in 95% of samples. This is known as a *confidence interval*; and it's based on the Z-scores inherent in a normal distribution.

Confidence intervals are expressed as a sample statistic \pm (*plus or minus*) a margin of error. To calculate the margin of error, you need to determine the confidence level you want to find (for example, 95%), and determine the Z score that marks the threshold above or below which the values that are *not* within the chosen interval reside. For example, to calculate a 95% confidence interval, you need the critical Z scores that exclude 5% of the values under the curve; with 2.5% of them being lower than the values in the confidence interval range, and 2.5% being higher. In a normal distribution, 95% of the area under the curve is between a Z score of ± 1.96 . The following table shows the critical Z values for some other popular confidence interval ranges:

Confidence	Z Score
90%	1.645
95%	1.96
99%	2.576

To calculate a confidence interval around a sample statistic, we simply calculate the *standard error* for that statistic as described previously, and multiply this by the appropriate Z score for the confidence interval we want.

To calculate the 95% confidence interval margin of error for our bag weights, we multiply our standard error of 0.053 by the Z score for a 95% confidence level, which is 1.96:

$$MoE = 0.053 \times 1.96 = 0.10388$$

So we can say that we're confident that the population mean weight is in the range of the sample mean ± 0.10388 with 95% confidence. Thanks to the central limit theorem, if we used an even bigger sample size, the confidence interval would become smaller as the amount of variance in the distribution is reduced. If the number of samples were infinite, the standard error would be 0 and the confidence interval would become a certain value that reflects the true mean weight for all carry-on bags:

$$\lim_{n \rightarrow \infty} \frac{\sigma}{\sqrt{n}} = 0$$

In Python, you can use the `scipy.stats.norm.interval***` function to calculate a confidence interval for a normal distribution. Run the following code to recreate the sampling distribution for bag searches with the same parameters, and display the 95% confidence interval for the mean (again, this may take some time to run):

```

In [*]: %matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats

mu, sigma, n = 3.2, 1.2, 500
samples = list(range(0, 10000))

# data will hold all of the sample data
data = np.array([])

# sampling will hold the means of the samples
sampling = np.array([])

# Perform 10,000 samples
for s in samples:
    # In each sample, get 500 data points from a normal distribution
    sample = np.random.normal(mu, sigma, n)
    data = np.append(data, sample)
    sampling = np.append(sampling, sample.mean())

# Create a dataframe with the sampling of means
df = pd.DataFrame(sampling, columns=['mean'])

# Get the Mean, StdDev, and 95% CI of the means
means = df['mean']
m = means.mean()
sd = means.std()
ci = stats.norm.interval(0.95, m, sd)

# Plot the distribution, mean, and CI
means.plot.hist(title='Simulated Sampling Distribution', bins=100)
plt.axvline(m, color='red', linestyle='dashed', linewidth=2)
plt.axvline(ci[0], color='magenta', linestyle='dashed', linewidth=2)
plt.axvline(ci[1], color='magenta', linestyle='dashed', linewidth=2)
plt.show()

# Print the Mean, StdDev and 95% CI
print ('Sampling Mean: ' + str(m))
print ('Sampling StdErr: ' + str(sd))
print ('95% Confidence Interval: ' + str(ci))

```

Hypothesis Testing

Single-Sample, One-Sided Tests

Our students have completed their school year, and been asked to rate their statistics class on a scale between -5 (terrible) and 5 (fantastic). The statistics class is taught online to tens of thousands of students, so to assess its success, we'll take a random sample of 50 ratings.

Run the following code to draw 50 samples.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

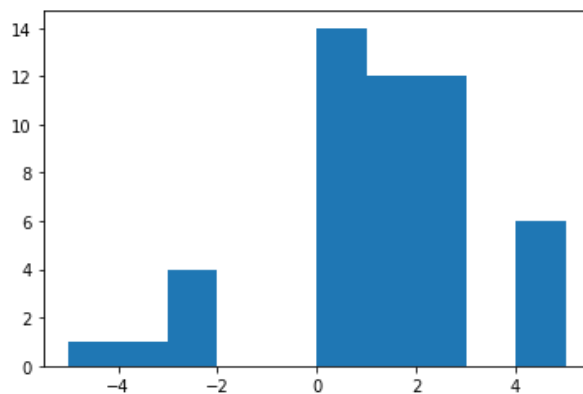
np.random.seed(123)
lo = np.random.randint(-5, -1, 6)
mid = np.random.randint(0, 3, 38)
hi = np.random.randint(4, 6, 6)
sample = np.append(lo, np.append(mid, hi))
print("Min:" + str(sample.min()))
print("Max:" + str(sample.max()))
print("Mean:" + str(sample.mean()))

plt.hist(sample)
plt.show()
```

Min:-5

Max:5

Mean:0.84



A question we might immediately ask is: "how do students tend to like the class"? In this case, possible ratings were between -5 and 5, with a "neutral" score of 0. In other words, if our average score is above zero, then students tend to enjoy the course.

In the sample above, the mean score is above 0 (in other words, people liked the class in this data). If you had actually run this course and saw this data, it might lead you to believe that the overall mean rating for this class (i.e., not just the sample) is likely to be positive.

There is an important point to be made, though: this is just a sample, and you want to make a statement not just about your sample but the whole population from which it came. In other words, you want to know how the class was received overall, but you only have access to a limited set of data. This is often the case when analyzing data.

So, how can you test your belief that your positive looking *sample* reflects the fact that the course does tend to get good evaluations, that your *population* mean (not just your sample mean) is positive?

We start by defining two hypotheses:

- The *null* hypothesis (H_0) is that the population mean for all of the ratings is *not* higher than 0, and the fact that our sample mean is higher than this is due to random chance in our sample selection.
- The *alternative* hypothesis (H_1) is that the population mean is actually higher than 0, and the fact that our sample mean is higher than this means that our sample correctly detected this trend.

You can write these as mutually exclusive expressions like this:

$$H_0 : \mu \leq 0$$

$$H_1 : \mu > 0$$

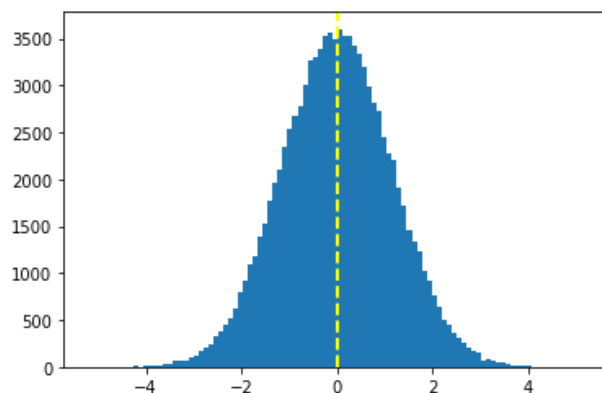
So how do we test these hypotheses? Because they are mutually exclusive, if we can show the null is probably not true, then we are safe to reject it and conclude that people really do like our online course. But how do we do that?

Well, if the *null* hypothesis is true, the sampling distribution for ratings with a sample size of 50 will be a normal distribution with a mean of 0. Run the following code to visualize this, with the mean of 0 shown as a yellow dashed line.

(The code just generates a normal distribution with a mean of 0 and a standard deviation that makes it approximate a sampling distribution of 50 random means between -5 and 5 - don't worry too much about the actual values, it's just to illustrate the key points!)

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

pop = np.random.normal(0, 1.15, 100000)
plt.hist(pop, bins=100)
plt.axvline(pop.mean(), color='yellow', linestyle='dashed', linewidth=2)
plt.show()
```



This illustrates all the *sample* results you could get if the null hypothesis was true (that is, the rating population mean is actually 0). Note that if the null hypothesis is true, it's still *possible* to get a sample with a mean ranging from just over -5 to just under 5. The question is how *probable* is it to get a sample with a mean as high we did for our 50-rating sample under the null hypothesis? And how improbable would it *need* to be for us to conclude that the null is, in fact, a poor explanation for our data?

Well, we measure distance from the mean in standard deviations, so we need to find out how many standard deviations above the null-hypothesized population mean of 0 our sample mean is, and measure the area under the distribution curve from this point on - that will give us the probability of observing a mean that is *at least* as high as our sample mean. We call the number of standard deviations above the mean where our sample mean is found the *test statistic* (or sometimes just *t-statistic*), and we call the area under the curve from this point (representing the probability of observing a sample mean this high or greater) the *p-value*.

So the p-value tells us how probable our sample mean is when the null is true, but we need to set a threshold under which we consider this to be too improbable to be explained by random chance alone. We call this threshold our *critical value*, and we usually indicate it using the Greek letter alpha (α). You can use any value you think is appropriate for α - commonly a value of 0.05 (5%) is used, but there's nothing special about this value.

We calculate the t-statistic by performing a statistical test. Technically, when the standard deviation of the population is known, we call it a z-test (because a *normal* distribution is often called a *z-distribution* and we measure variance from the mean in multiples of standard deviation known as *z-scores*). When the standard deviation of the population is not known, the test is referred to as a *t-test* and based on an adjusted version of a normal distribution called a *student's t distribution*, in which the distribution is "flattened" to allow for more sample variation depending on the sample size. Generally, with a sample size of 30 or more, a t-test is approximately equivalent to a z-test.

Specifically, in this case we're performing a *single sample* test (we're comparing the mean from a single sample of ratings against the hypothesized population mean), and it's a *one-tailed* test (we're checking to see if the sample mean is *greater than* the null-hypothesized population mean - in other words, in the *right* tail of the distribution).

The general formula for one-tailed, single-sample t-test is:

$$t = \frac{\bar{x} - \mu}{s \div \sqrt{n}}$$

In this formula, \bar{x} is the sample mean, μ is the population mean, s is the standard deviation, and n is the sample size. You can think of the numerator of this equation (the expression at the top of the fraction) as a *signal*, and the denominator (the expression at the bottom of the fraction) as being *noise*. The signal measures the difference between the statistic and the null-hypothesized value, and the noise represents the random variance in the data in the form of standard deviation (or standard error). The t-statistic is the ratio of signal to noise, and measures the number of standard errors between the null-hypothesized value and the observed sample mean. A large value tells you that your "result" or "signal" was much larger than you would typically expect by chance.

Fortunately, most programming languages used for statistical analysis include functions to perform a t-test, so you rarely need to manually calculate the results using the formula.

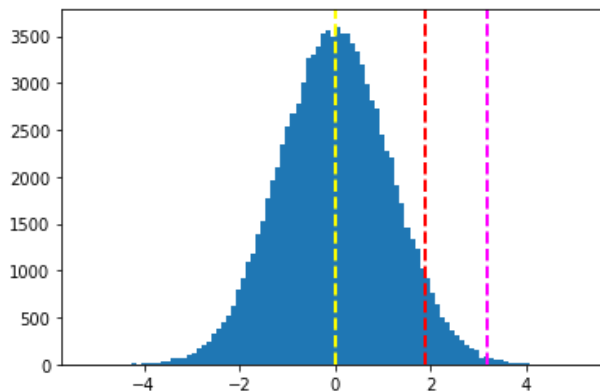
Run the code below to run a single-sample t-test comparing our sample mean for ratings to a hypothesized population mean of 0, and visualize the resulting t-statistic on the normal distribution for the null hypothesis.

```
In [3]: from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# T-Test
t,p = stats.ttest_1samp(sample, 0)
# ttest_1samp is 2-tailed, so half the resulting p-value to get a 1-tailed p-value
p1 = '%f' % (p/2)
print ("t-statistic:" + str(t))
print("p-value:" + str(p1))

# calculate a 90% confidence interval. 10% of the probability is outside this,
ci = stats.norm.interval(0.90, 0, 1.15)
plt.hist(pop, bins=100)
# show the hypothesized population mean
plt.axvline(pop.mean(), color='yellow', linestyle='dashed', linewidth=2)
# show the right-tail confidence interval threshold - 5% of probability is under
plt.axvline(ci[1], color='red', linestyle='dashed', linewidth=2)
# show the t-statistic - the p-value is the area under the curve to the right of this
plt.axvline(pop.mean() + t*pop.std(), color='magenta', linestyle='dashed', linewidth=2)
plt.show()
```

```
t-statistic:2.773584905660377
p-value:0.003911
```



In the plot produced by the code above, the yellow line shows the population mean for the null hypothesis. The area under the curve to the right of the red line represents the critical value of 0.05 (or 5%). The magenta line indicates how much higher the sample mean is compared to the hypothesized population mean. This is calculated as the t-statistic (which is printed above the plot) multiplied by the standard deviation. The area under the curve to the right of this encapsulates the p-value calculated by the test (which is also printed above the plot).

So what should we conclude from these results?

Well, if the p-value is smaller than our critical value of 0.05, that means that under the null hypothesis, the probability of observing a sample mean as high as we did by random chance is low. That's a good sign for us, because it means that our sample is unlikely under the null, and therefore the null is a poor explanation for the data. We can safely *reject* the null hypothesis in favor of the alternative hypothesis - there's enough evidence to suggest that the population mean for our class ratings is greater than 0.

Conversely, if the p-value is greater than the critical value, we *fail to reject the null hypothesis* and conclude that the mean rating is not greater than 0. Note that we never actually *accept* the null hypothesis, we just conclude that there isn't enough evidence to reject it!

Two-Tailed Tests

The previous test was an example of a one-tailed test in which the p-value represents the area under one tail of the distribution curve. In this case, the area in question is under the right tail because the alternative hypothesis we were trying to show was that the true population mean is *greater than* the mean of the null hypothesis scenario.

Suppose we restated our hypotheses like this:

- The *null* hypothesis (**H₀**) is that the population mean for all of the ratings is 0, and the fact that our sample mean is higher or lower than this can be explained by random chance in our sample selection.
- The *alternative* hypothesis (**H₁**) is that the population mean is not equal to 0.

We can write these as mutually exclusive expressions like this:

$$H_0 : \mu = 0$$

$$H_1 : \mu \neq 0$$

Why would we do this? Well, in the test we performed earlier, we could only reject the null hypothesis if we had really *positive* ratings, but what if our sample data looked really *negative*? It would be a mistake to turn around and run a one-tailed test the other way, for negative ratings. Instead, we conduct a test designed for such a question: a two-tailed test.

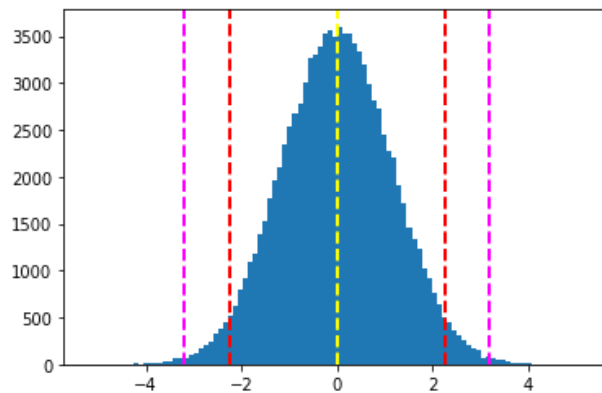
In a two-tailed test, we are willing to reject the null hypothesis if the result is significantly *greater or lower* than the null hypothesis. Our critical value (5%) is therefore split in two: the top 2.5% of the curve and the bottom 2.5% of the curve. As long as our test statistic is in that region, we are in the extreme 5% of values ($p < .05$) and we reject the null hypothesis. In other words, our p-value now needs to be below .025, but it can be in either tail of the distribution. For convenience, we usually "double" the p-value in a two-tailed test so that we don't have to remember this rule and still compare against .05 (this is known as a "two-tailed p-value"). In fact, it is assumed this has been done in all statistical analyses unless stated otherwise.

The following code shows the results of a two-tailed, single sample test of our class ratings. Note that the ***ttest_1samp*** function in the ***stats*** library returns a 2-tailed p-value by default (which is why we halved it in the previous example).

```
In [4]: from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# T-Test
t,p = stats.ttest_1samp(sample, 0)
print ("t-statistic:" + str(t))
# ttest_1samp is 2-tailed
print("p-value:" + '%f' % p)
# calculate a 95% confidence interval. 50% of the probability is outside this,
ci = stats.norm.interval(0.95, 0, 1.15)
plt.hist(pop, bins=100)
# show the hypothesized population mean
plt.axvline(pop.mean(), color='yellow', linestyle='dashed', linewidth=2)
# show the confidence interval thresholds - 5% of propability is under the cur
plt.axvline(ci[0], color='red', linestyle='dashed', linewidth=2)
plt.axvline(ci[1], color='red', linestyle='dashed', linewidth=2)
# show the t-statistic thresholds - the p-value is the area under the curve out
plt.axvline(pop.mean() - t*pop.std(), color='magenta', linestyle='dashed', line
plt.axvline(pop.mean() + t*pop.std(), color='magenta', linestyle='dashed', line
plt.show()
```

t-statistic:2.773584905660377
p-value:0.007822



Here we see that our 2-tailed p-value was clearly less than 0.05; so We reject the null hypothesis.

You may note that doubling the p-value in a two-tailed test makes it harder to reject the null. This is true; we require more evidence because we are asking a more complicated question.

Two-Sample Tests

In both of the previous examples, we compared a statistic from a single data sample to a null-hypothesized population parameter. Sometimes you might want to compare two samples against one another.

For example, let's suppose that some of the students who took the statistics course had previously studied mathematics, while other students had no previous math experience. You might hypothesize that the grades of students who had previously studied math are significantly higher than the grades of students who had not.

- The *null* hypothesis (H_0) is that the population mean grade for students with previous math studies is not greater than the population mean grade for students without any math experience, and the fact that our sample mean for math students is higher than our sample mean for non-math students can be explained by random chance in our sample selection.
- The *alternative* hypothesis (H_1) is that the population mean grade for students with previous math studies is greater than the population mean grade for students without any math experience.

We can write these as mutually exclusive expressions like this:

$$H_0 : \mu_1 \leq \mu_2$$

$$H_1 : \mu_1 > \mu_2$$

This is a one-sided test that compares two samples. To perform this test, we'll take two samples. One sample contains 100 grades for students who have previously studied math, and the other sample contains 100 grades for students with no math experience.

We won't go into the test-statistic formula here, but it's essentially the same as the one above, adapted to include information from both samples. We can easily test this in most software packages using the command for an "independent samples" t-test:

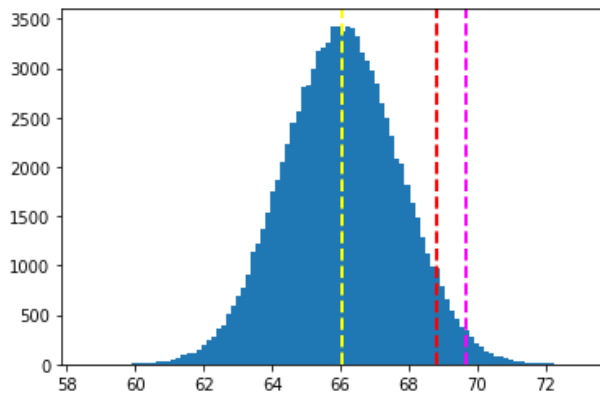
```
In [5]: import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
%matplotlib inline

np.random.seed(123)
nonMath = np.random.normal(66.0, 1.5, 100)
math = np.random.normal(66.55, 1.5, 100)
print("non-math sample mean:" + str(nonMath.mean()))
print("math sample mean:" + str(math.mean()))

# Independent T-Test
t,p = stats.ttest_ind(math, nonMath)
# ttest_ind is 2-tailed, so half the resulting p-value to get a 1-tailed p-value
p1 = '%f' % (p/2)
print("t-statistic:" + str(t))
print("p-value:" + str(p1))

pop = np.random.normal(nonMath.mean(), nonMath.std(), 100000)
# calculate a 90% confidence interval. 10% of the probability is outside this,
ci = stats.norm.interval(0.90, nonMath.mean(), nonMath.std())
plt.hist(pop, bins=100)
# show the hypothesized population mean
plt.axvline(pop.mean(), color='yellow', linestyle='dashed', linewidth=2)
# show the right-tail confidence interval threshold - 5% of probability is under
plt.axvline(ci[1], color='red', linestyle='dashed', linewidth=2)
# show the t-statistic - the p-value is the area under the curve to the right of
plt.axvline(pop.mean() + t*pop.std(), color='magenta', linestyle='dashed', linewidth=2)
plt.show()

non-math sample mean:66.04066361023553
math sample mean:66.52069665713476
t-statistic:2.140008413392296
p-value:0.016789
```



You can interpret the results of this test the same way as for the previous single-sample, one-tailed test. If the p-value (the area under the curve to the right of the magenta line) is smaller than our critical value (α) of 0.05 (the area under the curve to the right of the red line), then the difference can't be explained by chance alone; so we can reject the null hypothesis and conclude that students with previous math experience perform better on average than students without.

Alternatively, you could always compare two groups and *not* specify a direction (i.e., two-tailed). If you did this, as above, you could simply double the p-value (now .001), and you would see you could still reject the null hypothesis.

Paired Tests

In the two-sample test we conducted previously, the samples were independent; in other words there was no relationship between the observations in the first sample and the observations in the second sample. Sometimes you might want to compare statistical differences between related observations before and after some change that you believe might influence the data.

For example, suppose our students took a mid-term exam, and later took an end-of-term exam. You might hypothesise that the students will improve their grades in the end-of-term exam, after they've undertaken additional study. We could test for a general improvement on average across all students with a two-sample independent test, but a more appropriate test would be to compare the two test scores for each individual student.

To accomplish this, we need to create two samples; one for scores in the mid-term exam, the other for scores in the end-of-term exam. Then we need to compare the samples in such a way that each pair of observations for the same student are compared to one another.

This is known as a paired-samples t-test or a dependent-samples t-test. Technically, it tests whether the *changes* tend to be in the positive or negative direction.

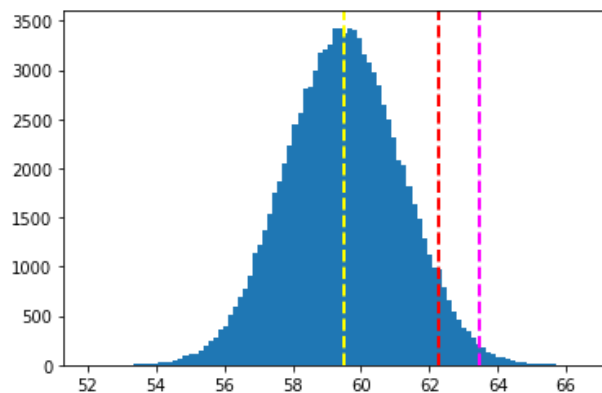
```
In [6]: import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
%matplotlib inline

np.random.seed(123)
midTerm = np.random.normal(59.45, 1.5, 100)
endTerm = np.random.normal(60.05, 1.5, 100)

# Paired (related) test
t,p = stats.ttest_rel(endTerm, midTerm)
# ttest_rel is 2-tailed, so half the resulting p-value to get a 1-tailed p-value
p1 = '%f' % (p/2)
print("t-statistic:" + str(t))
print("p-value:" + str(p1))

pop = np.random.normal(midTerm.mean(), midTerm.std(), 100000)
# calculate a 90% confidence interval. 10% of the probability is outside this,
ci = stats.norm.interval(0.90, midTerm.mean(), midTerm.std())
plt.hist(pop, bins=100)
# show the hypothesized population mean
plt.axvline(pop.mean(), color='yellow', linestyle='dashed', linewidth=2)
# show the right-tail confidence interval threshold - 5% of probability is under
plt.axvline(ci[1], color='red', linestyle='dashed', linewidth=2)
# show the t-statistic - the p-value is the area under the curve to the right of this
plt.axvline(pop.mean() + t*pop.std(), color='magenta', linestyle='dashed', linewidth=2)
plt.show()

t-statistic:2.3406857739212583
p-value:0.010627
```



In our sample, we see that scores did in fact improve, so we can reject the null hypothesis.