Getting Started with Equations

Equations are calculations in which one or more variables represent unknown values. In this notebook, you'll learn some fundamental techniques for solving simple equations.

One Step Equations

Consider the following equation:

$$x + 16 = -25$$

The challenge here is to find the value for **x**, and to do this we need to *isolate the variable*. In this case, we need to get **x** onto one side of the "=" sign, and all of the other values onto the other side. To accomplish this we'll follow these rules:

1. Use opposite operations to cancel out the values we don't want on one side. In this case, the left side of the equation includes an addition of 16, so we'll cancel that out by subtracting 16 and the left side of the equation becomes **x + 16 - 16**.
2. Whatever you do to one side, you must also do to the other side. In this case, we subtracted 16 from the left side, so we must also subtract 16 from the right side of the equation, which becomes **-25 - 16**. Our equation now looks like this:

$$x + 16 - 16 = -25 - 16$$

Now we can calculate the values on both side. On the left side, 16 - 16 is 0, so we're left with:

$$x = -25 - 16$$

Which yields the result **-41**. Our equation is now solved, as you can see here:

$$x = -41$$

It's always good practice to verify your result by plugging the variable value you've calculated into the original equation and ensuring that it holds true. We can easily do that by using some simple Python code.

To verify the equation using Python code, place the cursor in the following cell and then click the ►| button in the toolbar.

In [1]:
```
x = -41
x + 16 == -25
```

Out[1]:
```
True
```

Two-Step Equations

The previous example was fairly simple - you could probably work it out in your head. So what about something a little more complex?

$$3x - 2 = 10$$

As before, we need to isolate the variable **x**, but this time we'll do it in two steps. The first thing we'll do is to cancel out the *constants*. A constant is any number that stands on its own, so in this case the 2 that we're subtracting on the left side is a constant. We'll use an opposite operation to cancel it out on the left side, so since the current operation is to subtract 2, we'll add 2; and of course whatever we do on the left side we also need to do on the right side, so after the first step, our equation looks like this:

$$3x - 2 + 2 = 10 + 2$$

Now the -2 and +2 on the left cancel one another out, and on the right side, 10 + 2 is 12; so the equation is now:

$$3x = 12$$

OK, time for step two - we need to deal with the *coefficients* - a coefficient is a number that is applied to a variable. In this case, our expression on the left is 3x, which means x multiplied by 3; so we can apply the opposite operation to cancel it out as long as we do the same to the other side, like this:

$$\frac{3x}{3} = \frac{12}{3}$$

3x ÷ 3 is x, so we've now isolated the variable

$$x = \frac{12}{3}$$

And we can calculate the result as $^{12}/_{3}$ which is **4**:

$$x = 4$$

Let's verify that result using Python:

In [2]:
```
x = 4
3*x - 2 == 10
```

Out[2]:
```
True
```

Combining Like Terms

Like terms are elements of an expression that relate to the same variable or constant (with the same *order* or *exponential*, which we'll discuss later). For example, consider the following equation:

$$\mathbf{5x} + 1\mathbf{- 2x} = 22$$

In this equation, the left side includes the terms **5x** and **- 2x**, both of which represent the variable **x** multiplied by a coefficent. Note that we include the sign (+ or -) in front of the value.

We can rewrite the equation to combine these like terms:

$$\mathbf{5x - 2x} + 1 = 22$$

We can then simply perform the necessary operations on the like terms to consolidate them into a single term:

$$\mathbf{3x} + 1 = 22$$

Now, we can solve this like any other two-step equation. First we'll remove the constants from the left side - in this case, there's a constant expression that adds 1, so we'll use the opposite operation to remove it and do the same on the other side:

$$3x + 1 - 1 = 22 - 1$$

That gives us:

$$3x = 21$$

Then we'll deal with the coefficients - in this case x is multiplied by 3, so we'll divide by 3 on boths sides to remove that:

$$\frac{3x}{3} = \frac{21}{3}$$

This give us our answer:

$$x = 7$$

In [3]:
```
x = 7
5*x + 1 - 2*x == 22
```

Out[3]:
```
True
```

Working with Fractions

Some of the steps in solving the equations above have involved working wth fractions - which in themselves are actually just division operations. Let's take a look at an example of an equation in which our variable is defined as a fraction:

$$\frac{x}{3} + 1 = 16$$

We follow the same approach as before, first removing the constants from the left side - so we'll subtract 1 from both sides.

$$\frac{x}{3} = 15$$

Now we need to deal with the fraction on the left so that we're left with just **x**. The fraction is $^{x}/_{3}$ which is another way of saying *x divided by 3*, so we can apply the opposite operation to both sides. In this case, we need to multiply both sides by the denominator under our variable, which is 3. To make it easier to work with a term that contains fractions, we can express whole numbers as fractions with a denominator of 1; so on the left, we can express 3 as $^{3}/_{1}$ and multiply it with $^{x}/_{3}$. Note that the notation for mutiplication is a • symbol rather than the standard *x* multiplication operator (which would cause confusion with the variable **x**) or the asterisk symbol used by most programming languages.

$$\frac{3}{1} \cdot \frac{x}{3} = 15 \cdot 3$$

This gives us the following result:

$$x = 45$$

Let's verify that with some Python code:

```
In [4]: x = 45
        x/3 + 1 == 16
Out[4]: True
```

Let's look at another example, in which the variable is a whole number, but its coefficient is a fraction:

$$\frac{2}{5}x + 1 = 11$$

As usual, we'll start by removing the constants from the variable expression; so in this case we need to subtract 1 from both sides:

$$\frac{2}{5}x = 10$$

Now we need to cancel out the fraction. The expression equates to two-fifths times x, so the opposite operation is to divide by $^{2}/_{5}$; but a simpler way to do this with a fraction is to multiply it by its *reciprocal*, which is just the inverse of the fraction, in this case $^{5}/_{2}$. Of course, we need to do this to both sides:

$$\frac{5}{2} \cdot \frac{2}{5}x = \frac{10}{1} \cdot \frac{5}{2}$$

That gives us the following result:

$$x = \frac{50}{2}$$

Which we can simplify to:

$$x = 25$$

We can confirm that with Python:

In [5]:
```
x = 25
2/5 * x + 1 ==11
```

Out[5]:
```
True
```

Equations with Variables on Both Sides

So far, all of our equations have had a variable term on only one side. However, variable terms can exist on both sides.

Consider this equation:

$$3x + 2 = 5x - 1$$

This time, we have terms that include **x** on both sides. Let's take exactly the same approach to solving this kind of equation as we did for the previous examples. First, let's deal with the constants by adding 1 to both sides. That gets rid of the -1 on the right:

$$3x + 3 = 5x$$

Now we can eliminate the variable expression from one side by subtracting 3x from both sides. That gets rid of the 3x on the left:

$$3 = 2x$$

Next, we can deal with the coefficient by dividing both sides by 2:

$$\frac{3}{2} = x$$

Now we've isolated x. It looks a little strange because we usually have the variable on the left side, so if it makes you more comfortable you can simply reverse the equation:

$$x = \frac{3}{2}$$

Finally, this answer is correct as it is; but $^{3}/_{2}$ is an improper fraction. We can simplify it to:

$$x = 1\frac{1}{2}$$

So x is $1^{1}/_{2}$ (which is of course 1.5 in decimal notation). Let's check it in Python:

In [6]:
```
x = 1.5
3*x + 2 == 5*x -1
```

Out[6]:
```
True
```

Using the Distributive Property

The distributive property is a mathematical law that enables us to distribute the same operation to terms within parenthesis. For example, consider the following equation:

$$\mathbf{4(x + 2) + 3(x - 2)} = 16$$

The equation includes two operations in parenthesis: **4(*x* + 2)** and **3(*x* - 2)**. Each of these operations consists of a constant by which the contents of the parenthesis must be multipled: for example, 4 times (*x* + 2). The distributive property means that we can achieve the same result by multiplying each term in the parenthesis and adding the results, so for the first parenthetical operation, we can multiply 4 by *x* and add it to 4 times +2; and for the second parenthetical operation, we can calculate 3 times *x* + 3 times -2). Note that the constants in the parenthesis include the sign (+ or -) that preceed them:

$$4x + 8 + 3x - 6 = 16$$

Now we can group our like terms:

$$7x + 2 = 16$$

Then we move the constant to the other side:

$$7x = 14$$

And now we can deal with the coefficient:

$$\frac{7x}{7} = \frac{14}{7}$$

Which gives us our anwer:

$$x = 2$$

Here's the original equation with the calculated value for *x* in Python:

In [7]:
```
x = 2
4*(x + 2) + 3*(x - 2) == 16
```

Out[7]:
```
True
```

Linear Equations

The equations in the previous lab included one variable, for which you solved the equation to find its value. Now let's look at equations with multiple variables. For reasons that will become apparent, equations with two variables are known as linear equations.

Solving a Linear Equation

Consider the following equation:

$$2y + 3 = 3x - 1$$

This equation includes two different variables, **x** and **y**. These variables depend on one another; the value of x is determined in part by the value of y and vice-versa; so we can't solve the equation and find absolute values for both x and y. However, we *can* solve the equation for one of the variables and obtain a result that describes a relative relationship between the variables.

For example, let's solve this equation for y. First, we'll get rid of the constant on the right by adding 1 to both sides:

$$2y + 4 = 3x$$

Then we'll use the same technique to move the constant on the left to the right to isolate the y term by subtracting 4 from both sides:

$$2y = 3x - 4$$

Now we can deal with the coefficient for y by dividing both sides by 2:

$$y = \frac{3x - 4}{2}$$

Our equation is now solved. We've isolated **y** and defined it as $^{3x-4}/_{2}$

While we can't express **y** as a particular value, we can calculate it for any value of **x**. For example, if **x** has a value of 6, then **y** can be calculated as:

$$y = \frac{3 \cdot 6 - 4}{2}$$

This gives the result $^{14}/_{2}$ which can be simplified to 7.

You can view the values of **y** for a range of **x** values by applying the equation to them using the following Python code:

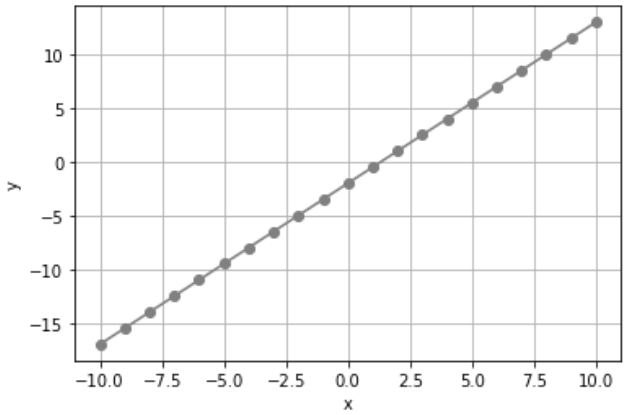
In [1]:
```
import pandas as pd

# Create a dataframe with an x column containing values from -10 to 10
df = pd.DataFrame ({'x': range(-10, 11)})

# Add a y column by applying the solved equation to x
df['y'] = (3*df['x'] - 4) / 2

#Display the dataframe
df
```

Out[1]:

	x	y
0	-10	-17.0
1	-9	-15.5
2	-8	-14.0
3	-7	-12.5
4	-6	-11.0
5	-5	-9.5
6	-4	-8.0
7	-3	-6.5
8	-2	-5.0
9	-1	-3.5
10	0	-2.0
11	1	-0.5
12	2	1.0
13	3	2.5
14	4	4.0
15	5	5.5
16	6	7.0
17	7	8.5
18	8	10.0
19	9	11.5
20	10	13.0

We can also plot these values to visualize the relationship between x and y as a line. For this reason, equations that describe a relative relationship between two variables are known as *linear equations*:

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

plt.plot(df.x, df.y, color="grey", marker = "o")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.show()
```



In a linear equation, a valid solution is described by an ordered pair of x and y values. For example, valid solutions to the linear equation above include:

- (-10, -17)
- (0, -2)
- (9, 11.5)

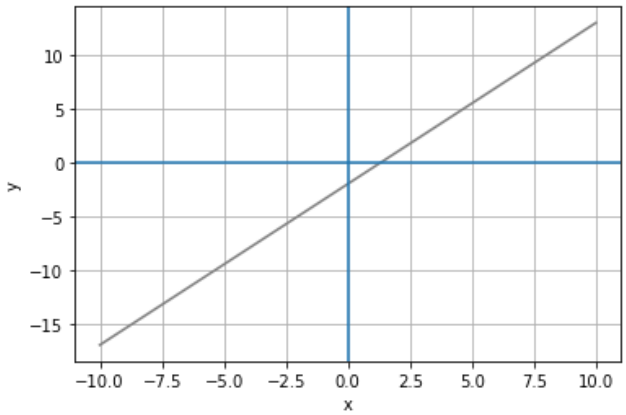
The cool thing about linear equations is that we can plot the points for some specific ordered pair solutions to create the line, and then interpolate the x value for any y value (or vice-versa) along the line.

Intercepts

When we use a linear equation to plot a line, we can easily see where the line intersects the X and Y axes of the plot. These points are known as *intercepts*. The *x-intercept* is where the line intersects the X (horizontal) axis, and the *y-intercept* is where the line intersects the Y (horizontal) axis.

Let's take a look at the line from our linear equation with the X and Y axis shown through the origin (0,0).

In [3]:
```
plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()

## add axis lines for 0,0
plt.axhline()
plt.axvline()
plt.show()
```



The x-intercept is the point where the line crosses the X axis, and at this point, the **y** value is always 0. Similarly, the y-intercept is where the line crosses the Y axis, at which point the **x** value is 0. So to find the intercepts, we need to solve the equation for **x** when **y** is 0.

For the x-intercept, our equation looks like this:

$$0 = \frac{3x - 4}{2}$$

Which can be reversed to make it look more familar with the x expression on the left:

$$\frac{3x - 4}{2} = 0$$

We can multiply both sides by 2 to get rid of the fraction:

$$3x - 4 = 0$$

Then we can add 4 to both sides to get rid of the constant on the left:

$$3x = 4$$

And finally we can divide both sides by 3 to get the value for x:

$$x = \frac{4}{3}$$

Which simplifies to:

$$x = 1\frac{1}{3}$$

So the x-intercept is $1^1/_3$ (approximately 1.333).

To get the y-intercept, we solve the equation for y when x is 0:

$$y = \frac{3 \cdot 0 - 4}{2}$$

Since 3 x 0 is 0, this can be simplified to:

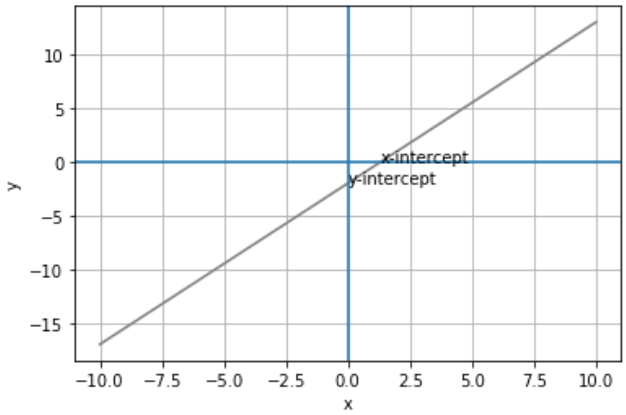
$$y = \frac{-4}{2}$$

-4 divided by 2 is -2, so:

$$y = -2$$

This gives us our y-intercept, so we can plot both intercepts on the graph:

In [4]:
```
plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()

## add axis lines for 0,0
plt.axhline()
plt.axvline()
plt.annotate('x-intercept',(1.333, 0))
plt.annotate('y-intercept',(0,-2))
plt.show()
```



The ability to calculate the intercepts for a linear equation is useful, because you can calculate only these two points and then draw a straight line through them to create the entire line for the equation.

Slope

It's clear from the graph that the line from our linear equation describes a slope in which values increase as we travel up and to the right along the line. It can be useful to quantify the slope in terms of how much **x** increases (or decreases) for a given change in **y**. In the notation for this, we use the greek letter Δ (*delta*) to represent change:

$$slope = \frac{\Delta y}{\Delta x}$$

Sometimes slope is represented by the variable ***m***, and the equation is written as:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Although this form of the equation is a little more verbose, it gives us a clue as to how we calculate slope. What we need is any two ordered pairs of x,y values for the line - for example, we know that our line passes through the following two points:

- (0,-2)
- (6,7)

We can take the x and y values from the first pair, and label them x_1 and y_1; and then take the x and y values from the second point and label them x_2 and y_2. Then we can plug those into our slope equation:

$$m = \frac{7 - -2}{6 - 0}$$

This is the same as:

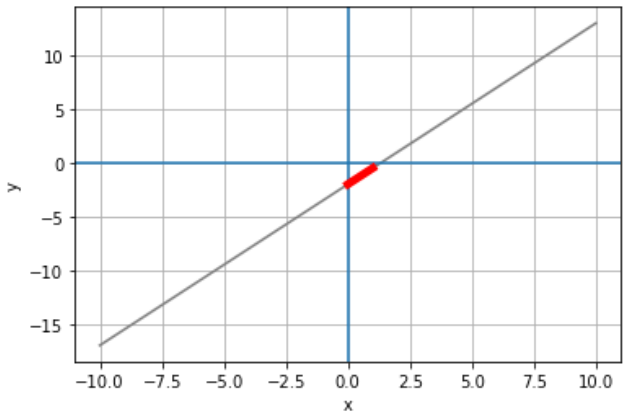
$$m = \frac{7 + 2}{6 - 0}$$

That gives us the result $^{9}/_{6}$ which is $1^{1}/_{2}$ or 1.5 .

So what does that actually mean? Well, it tells us that for every change of **1** in x, **y** changes by $1^{1}/_{2}$ or 1.5. So if we start from any point on the line and move one unit to the right (along the X axis), we'll need to move 1.5 units up (along the Y axis) to get back to the line.

You can plot the slope onto the original line with the following Python code to verify it fits:

In [5]:
```
plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()

# set the slope
m = 1.5

# get the y-intercept
yInt = -2

# plot the slope from the y-intercept for 1x
mx = [0, 1]
my = [yInt, yInt + m]
plt.plot(mx,my, color='red', lw=5)

plt.show()
```



Slope-Intercept Form

One of the great things about algebraic expressions is that you can write the same equation in multiple ways, or *forms*. The *slope-intercept form* is a specific way of writing a 2-variable linear equation so that the equation definition includes the slope and y-intercept. The generalised slope-intercept form looks like this:

$$y = mx + b$$

In this notation, ***m*** is the slope and ***b*** is the y-intercept.

For example, let's look at the solved linear equation we've been working with so far in this section:

$$y = \frac{3x - 4}{2}$$

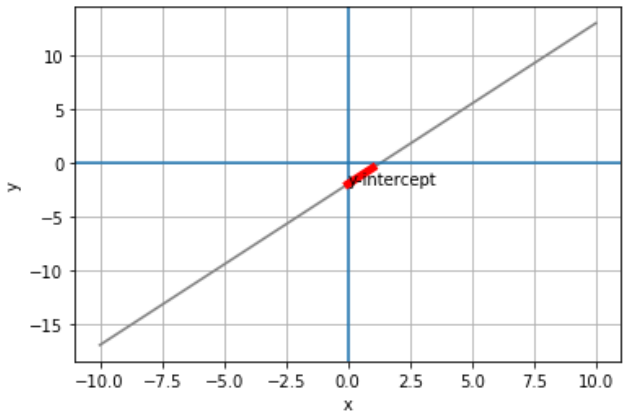
Now that we know the slope and y-intercept for the line that this equation defines, we can rewrite the equation as:

$$y = 1\frac{1}{2}x + -2$$

You can see intuitively that this is true. In our original form of the equation, to find y we multiply x by three, subtract 4, and divide by two - in other words, x is half of 3x - 4; which is 1.5x - 2. So these equations are equivalent, but the slope-intercept form has the advantages of being simpler, and including two key pieces of information we need to plot the line represented by the equation. We know the y-intecept that the line passes through (0, -2), and we know the slope of the line (for every x, we add 1.5 to y.

Let's recreate our set of test x and y values using the slope-intercept form of the equation, and plot them to prove that this describes the same line:

In [6]:
```
%matplotlib inline

import pandas as pd
from matplotlib import pyplot as plt

# Create a dataframe with an x column containing values from -10 to 10
df = pd.DataFrame ({'x': range(-10, 11)})

# Define slope and y-intercept
m = 1.5
yInt = -2

# Add a y column by applying the slope-intercept equation to x
df['y'] = m*df['x'] + yInt

# Plot the line
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()

# label the y-intercept
plt.annotate('y-intercept',(0,yInt))

# plot the slope from the y-intercept for 1x
mx = [0, 1]
my = [yInt, yInt + m]
plt.plot(mx,my, color='red', lw=5)

plt.show()
```



Systems of Equations

Imagine you are at a casino, and you have a mixture of £10 and £25 chips. You know that you have a total of 16 chips, and you also know that the total value of chips you have is £250. Is this enough information to determine how many of each denomination of chip you have?

Well, we can express each of the facts that we have as an equation. The first equation deals with the total number of chips - we know that this is 16, and that it is the number of £10 chips (which we'll call ***x***) added to the number of £25 chips (***y***).

The second equation deals with the total value of the chips (£250), and we know that this is made up of ***x*** chips worth £10 and ***y*** chips worth £25.

Here are the equations

$$x + y = 16$$
$$10x + 25y = 250$$

Taken together, these equations form a *system of equations* that will enable us to determine how many of each chip denomination we have.

Graphing Lines to Find the Intersection Point

One approach is to determine all possible values for x and y in each equation and plot them.

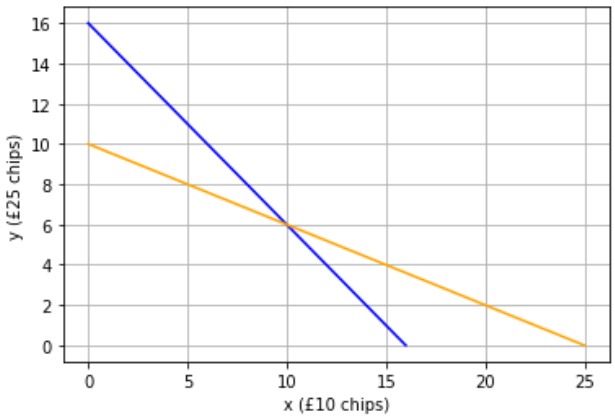
A collection of 16 chips could be made up of 16 £10 chips and no £25 chips, no £10 chips and 16 £25 chips, or any combination between these.

Similarly, a total of £250 could be made up of 25 £10 chips and no £25 chips, no £10 chips and 10 £25 chips, or a combination in between.

Let's plot each of these ranges of values as lines on a graph:

In [1]:

```
%matplotlib inline
from matplotlib import pyplot as plt

# Get the extremes for number of chips
chipsAll10s = [16, 0]
chipsAll25s = [0, 16]

# Get the extremes for values
valueAll10s = [25,0]
valueAll25s = [0,10]

# Plot the lines
plt.plot(chipsAll10s,chipsAll25s, color='blue')
plt.plot(valueAll10s, valueAll25s, color="orange")
plt.xlabel('x (£10 chips)')
plt.ylabel('y (£25 chips)')
plt.grid()

plt.show()
```



Looking at the graph, you can see that there is only a single combination of £10 and £25 chips that is on both the line for all possible combinations of 16 chips and the line for all possible combinations of £250. The point where the line intersects is (10, 6); or put another way, there are ten £10 chips and six £25 chips.

Solving a System of Equations with Elimination

You can also solve a system of equations mathematically. Let's take a look at our two equations:

$$x + y = 16$$
$$10x + 25y = 250$$

We can combine these equations to eliminate one of the variable terms and solve the resulting equation to find the value of one of the variables. Let's start by combining the equations and eliminating the x term.

We can combine the equations by adding them together, but first, we need to manipulate one of the equations so that adding them will eliminate the x term. The first equation includes the term ***x***, and the second includes the term ***10x***, so if we multiply the first equation by -10, the two x terms will cancel each other out. So here are the equations with the first one multiplied by -10:

$$-10(x + y) = -10(16)$$
$$10x + 25y = 250$$

After we apply the multiplication to all of the terms in the first equation, the system of equations look like this:

$$-10x + -10y = -160$$
$$10x + 25y = 250$$

Now we can combine the equations by adding them. The ***-10x*** and ***10x*** cancel one another, leaving us with a single equation like this:

$$15y = 90$$

We can isolate ***y*** by dividing both sides by 15:

$$y = \frac{90}{15}$$

So now we have a value for ***y***:

$$y = 6$$

So how does that help us? Well, now we have a value for ***y*** that satisfies both equations. We can simply use it in either of the equations to determine the value of ***x***. Let's use the first one:

$$x + 6 = 16$$

When we work through this equation, we get a value for ***x***:

$$x = 10$$

So now we've calculated values for ***x*** and ***y***, and we find, just as we did with the graphical intersection method, that there are ten £10 chips and six £25 chips.

You can run the following Python code to verify that the equations are both true with an ***x*** value of 10 and a ***y*** value of 6.

In [2]:
```
x = 10
y = 6
print ((x + y == 16) & ((10*x) + (25*y) == 250))
```

```
True
```

Exponentials, Radicals, and Logs

Up to this point, all of our equations have included standard arithmetic operations, such as division, multiplication, addition, and subtraction. Many real-world calculations involve exponential values in which numbers are raised by a specific power.

Exponentials

A simple case of of using an exponential is squaring a number; in other words, multipying a number by itself. For example, 2 squared is 2 times 2, which is 4. This is written like this:

$$2^2 = 2 \cdot 2 = 4$$

Similarly, 2 cubed is 2 times 2 times 2 (which is of course 8):

$$2^3 = 2 \cdot 2 \cdot 2 = 8$$

In Python, you use the ****** operator, like this example in which **x** is assigned the value of 5 raised to the power of 3 (in other words, 5 x 5 x 5, or 5-cubed):

In [1]:
```
x = 5**3
print(x)
```

```
125
```

Multiplying a number by itself twice or three times to calculate the square or cube of a number is a common operation, but you can raise a number by any exponential power. For example, the following notation shows 4 to the power of 7 (or 4 x 4 x 4 x 4 x 4 x 4 x 4), which has the value:

$$4^7 = 16384$$

In mathematical terminology, **4** is the *base*, and **7** is the *power* or *exponent* in this expression.

Radicals (Roots)

While it's common to need to calculate the solution for a given base and exponential, sometimes you'll need to calculate one or other of the elements themselves. For example, consider the following expression:

$$?^2 = 9$$

This expression is asking, given a number (9) and an exponent (2), what's the base? In other words, which number multipled by itself results in 9? This type of operation is referred to as calculating the *root*, and in this particular case it's the *square root* (the base for a specified number given the exponential **2**). In this case, the answer is 3, because 3 x 3 = 9. We show this with a √ symbol, like this:

$$\sqrt{9} = 3$$

Other common roots include the *cube root* (the base for a specified number given the exponential **3**). For example, the cube root of 64 is 4 (because 4 x 4 x 4 = 64). To show that this is the cube root, we include the exponent **3** in the √ symbol, like this:

$$\sqrt[3]{64} = 4$$

We can calculate any root of any non-negative number, indicating the exponent in the √ symbol.

The **math** package in Python includes a **sqrt** function that calculates the square root of a number. To calculate other roots, you need to reverse the exponential calculation by raising the given number to the power of 1 divided by the given exponent:

In [2]:
```
import math

# Calculate square root of 25
x = math.sqrt(25)
print (x)

# Calculate cube root of 64
cr = round(64 ** (1. / 3))
print(cr)
```

```
5.0
4
```

The code used in Python to calculate roots other than the square root reveals something about the relationship between roots and exponentials. The exponential root of a number is the same as that number raised to the power of 1 divided by the exponential. For example, consider the following statement:

$$8^{\frac{1}{3}} = \sqrt[3]{8} = 2$$

Note that a number to the power of 1/3 is the same as the cube root of that number.

Based on the same arithmetic, a number to the power of 1/2 is the same as the square root of the number:

$$9^{\frac{1}{2}} = \sqrt{9} = 3$$

You can see this for yourself with the following Python code:

In [3]:
```
import math

print (9**0.5)
print (math.sqrt(9))
```

```
3.0
3.0
```

Logarithms

Another consideration for exponential values is the requirement occassionally to determine the exponent for a given number and base. In other words, how many times do I need to multiply a base number by itself to get the given result. This kind of calculation is known as the *logarithm*.

For example, consider the following expression:

$$4^{?} = 16$$

In other words, to what power must you raise 4 to produce the result 16?

The answer to this is 2, because 4 x 4 (or 4 to the power of 2) = 16. The notation looks like this:

$$log_4(16) = 2$$

In Python, you can calculate the logarithm of a number using the **log** function in the **math** package, indicating the number and the base:

In [4]:
```
import math

x = math.log(16, 4)
print(x)
```

```
2.0
```

The final thing you need to know about exponentials and logarithms is that there are some special logarithms:

The *common* logarithm of a number is its exponential for the base **10**. You'll occassionally see this written using the usual *log* notation with the base omitted:

$$log(1000) = 3$$

Another special logarithm is something called the *natural log*, which is a exponential of a number for base ***e***, where ***e*** is a constant with the approximate value 2.718. This number occurs naturally in a lot of scenarios, and you'll see it often as you work with data in many analytical contexts. For the time being, just be aware that the natural log is sometimes written as ***ln***:

$$log_e(64) = ln(64) = 4.1589$$

The **math.log** function in Python returns the natural log (base ***e***) when no base is specified. Note that this can be confusing, as the mathematical notation *log* with no base usually refers to the common log (base **10**). To return the common log in Python, use the **math.log10** function:

In [5]:
```
import math

# Natural log of 29
print (math.log(29))

# Common log of 100
print(math.log10(100))
```

```
3.367295829986474
2.0
```

Solving Equations with Exponentials

OK, so now that you have a basic understanding of exponentials, roots, and logarithms; let's take a look at some equations that involve exponential calculations.

Let's start with what might at first glance look like a complicated example, but don't worry - we'll solve it step-by-step and learn a few tricks along the way:

$$2y = 2x^4(\frac{x^2 + 2x^2}{x^3})$$

First, let's deal with the fraction on the right side. The numerator of this fraction is $x^2 + 2x^2$ - so we're adding two exponential terms. When the terms you're adding (or subtracting) have the same exponential, you can simply add (or subtract) the coefficients. In this case, x^2 is the same as $1x^2$, which when added to $2x^2$ gives us the result $3x^2$, so our equation now looks like this:

$$2y = 2x^4(\frac{3x^2}{x^3})$$

Now that we've condolidated the numerator, let's simplify the entire fraction by dividing the numerator by the denominator. When you divide exponential terms with the same variable, you simply divide the coefficients as you usually would and subtract the exponential of the denominator from the exponential of the numerator. In this case, we're dividing $3x^2$ by $1x^3$: The coefficient 3 divided by 1 is 3, and the exponential 2 minus 3 is -1, so the result is $3x^{-1}$, making our equation:

$$2y = 2x^4(3x^{-1})$$

So now we've got rid of the fraction on the right side, let's deal with the remaining multiplication. We need to multiply $3x^{-1}$ by $2x^4$. Multiplication, is the opposite of division, so this time we'll multipy the coefficients and add the exponentials: 3 multiplied by 2 is 6, and -1 + 4 is 3, so the result is $6x^3$:

$$2y = 6x^3$$

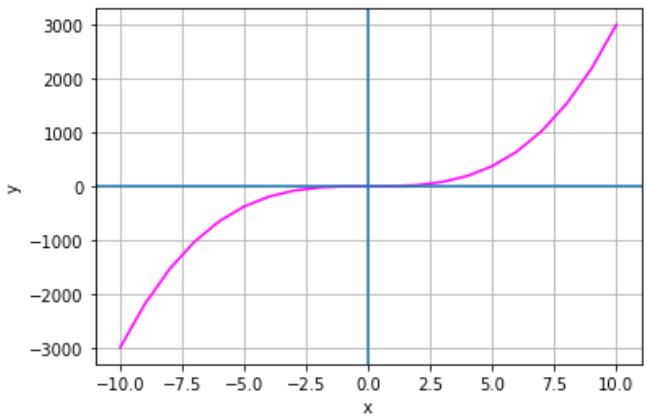
We're in the home stretch now, we just need to isolate y on the left side, and we can do that by dividing both sides by 2. Note that we're not dividing by an exponential, we simply need to divide the whole $6x^3$ term by two; and half of 6 times x^3 is just 3 times x^3:

$$y = 3x^3$$

Now we have a solution that defines y in terms of x. We can use Python to plot the line created by this equation for a set of arbitrary *x* and *y* values:

In [6]:
```
import pandas as pd

# Create a dataframe with an x column containing values from -10 to 10
df = pd.DataFrame ({'x': range(-10, 11)})

# Add a y column by applying the slope-intercept equation to x
df['y'] = 3*df['x']**3

#Display the dataframe
print(df)

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="magenta")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()
plt.show()
```

```
     x     y
0  -10 -3000
1   -9 -2187
2   -8 -1536
3   -7 -1029
4   -6  -648
5   -5  -375
6   -4  -192
7   -3   -81
8   -2   -24
9   -1    -3
10   0     0
11   1     3
12   2    24
13   3    81
14   4   192
15   5   375
16   6   648
17   7  1029
18   8  1536
19   9  2187
20  10  3000
```



Note that the line is curved. This is symptomatic of an exponential equation: as values on one axis increase or decrease, the values on the other axis scale *exponentially* rather than *linearly*.

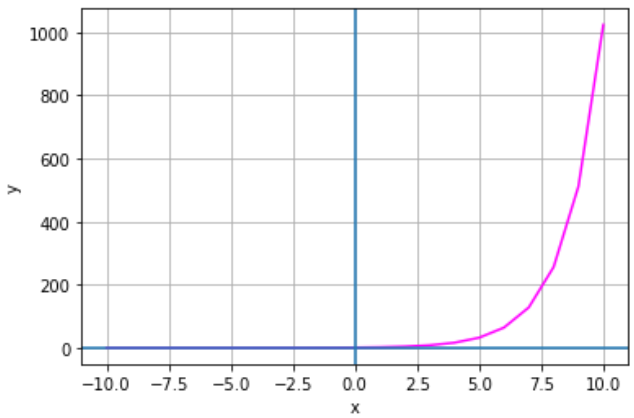
Let's look at an example in which x is the exponential, not the base:

$$y = 2^x$$

We can still plot this as a line:

In [7]:
```
import pandas as pd

# Create a dataframe with an x column containing values from -10 to 10
df = pd.DataFrame ({'x': range(-10, 11)})

# Add a y column by applying the slope-intercept equation to x
df['y'] = 2.0**df['x']

#Display the dataframe
print(df)

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="magenta")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()
plt.show()
```

```
     x            y
0  -10     0.000977
1   -9     0.001953
2   -8     0.003906
3   -7     0.007812
4   -6     0.015625
5   -5     0.031250
6   -4     0.062500
7   -3     0.125000
8   -2     0.250000
9   -1     0.500000
10   0     1.000000
11   1     2.000000
12   2     4.000000
13   3     8.000000
14   4    16.000000
15   5    32.000000
16   6    64.000000
17   7   128.000000
18   8   256.000000
19   9   512.000000
20  10  1024.000000
```



Note that when the exponential is a negative number, Python reports the result as 0. Actually, it's a very small fractional number, but because the base is positive the exponential number will always positive. Also, note the rate at which y increases as x increases - exponential growth can be be pretty dramatic.

So what's the practical application of this?

Well, let's suppose you deposit $100 in a bank account that earns 5% interest per year. What would the balance of the account be in twenty years, assuming you don't deposit or withdraw any additional funds?

To work this out, you could calculate the balance for each year:

After the first year, the balance will be the initial deposit ($100) plus 5% of that amount:

$$y1 = 100 + (100 \cdot 0.05)$$

Another way of saying this is:

$$y1 = 100 \cdot 1.05$$

At the end of year two, the balance will be the year one balance plus 5%:

$$y2 = 100 \cdot 1.05 \cdot 1.05$$

Note that the interest for year two, is the interest for year one multiplied by itself - in other words, squared. So another way of saying this is:

$$y2 = 100 \cdot 1.05^2$$

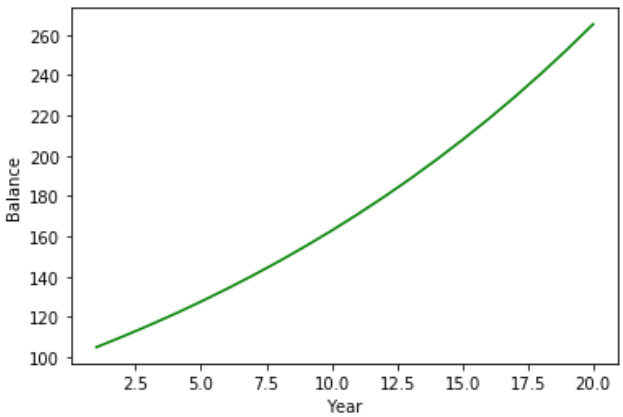
It turns out, if we just use the year as the exponent, we can easily calculate the growth after twenty years like this:

$$y20 = 100 \cdot 1.05^{20}$$

Let's apply this logic in Python to see how the account balance would grow over twenty years:

In [8]:
```
import pandas as pd

# Create a dataframe with 20 years
df = pd.DataFrame ({'Year': range(1, 21)})

# Calculate the balance for each year based on the exponential growth from inte
df['Balance'] = 100 * (1.05**df['Year'])

#Display the dataframe
print(df)

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.Year, df.Balance, color="green")
plt.xlabel('Year')
plt.ylabel('Balance')
plt.show()
```

```
    Year     Balance
0      1  105.000000
1      2  110.250000
2      3  115.762500
3      4  121.550625
4      5  127.628156
5      6  134.009564
6      7  140.710042
7      8  147.745544
8      9  155.132822
9     10  162.889463
10    11  171.033936
11    12  179.585633
12    13  188.564914
13    14  197.993160
14    15  207.892818
15    16  218.287459
16    17  229.201832
17    18  240.661923
18    19  252.695020
19    20  265.329771
```



Polynomials

Some of the equations we've looked at so far include expressions that are actually *polynomials*; but what *is* a polynomial, and why should you care?

A polynomial is an algebraic expression containing one or more *terms* that each meet some specific criteria. Specifically:

- Each term can contain:
  - Numeric values that are coefficients or constants (for example 2, -5, $^1/_7$)
  - Variables (for example, x, y)
  - Non-negative integer exponents (for example ^{2}, ^{64})
- The terms can be combined using arithmetic operations - but **not** division by a variable.

For example, the following expression is a polynomial:

$$12x^3 + 2x - 16$$

When identifying the terms in a polynomial, it's important to correctly interpret the arithmetic addition and subtraction operators as the sign for the term that follows. For example, the polynomial above contains the following three terms:

- $12x^3$
- $2x$
- -16

The terms themselves include:

- Two coefficients(12 and 2) and a constant (-16)
- A variable (x)
- An exponent (^{3})

A polynomial that contains three terms is also known as a *trinomial*. Similarly, a polynomial with two terms is known as a *binomial* and a polynomial with only one term is known as a *monomial*.

So why do we care? Well, polynomials have some useful properties that make them easy to work with. for example, if you multiply, add, or subtract a polynomial, the result is always another polynomial.

Standard Form for Polynomials

Techbnically, you can write the terms of a polynomial in any order; but the *standard form* for a polynomial is to start with the highest *degree* first and constants last. The degree of a term is the highest order (exponent) in the term, and the highest order in a polynomial determines the degree of the polynomial itself.

For example, consider the following expression:

$$3x + 4xy^2 - 3 + x^3$$

To express this as a polynomial in the standard form, we need to re-order the terms like this:

$$x^3 + 4xy^2 + 3x - 3$$

Simplifying Polynomials

We saw previously how you can simplify an equation by combining *like terms*. You can simplify polynomials in the same way.

For example, look at the following polynomial:

$$x^3 + 2x^3 - 3x - x + 8 - 3$$

In [1]:
```
from random import randint
x = randint(1,100)

(x**3 + 2*x**3 - 3*x - x + 8 - 3) == (3*x**3 - 4*x + 5)
```

Out[1]: True

Adding Polynomials

When you add two polynomials, the result is a polynomial. Here's an example:

$$(3x^3 - 4x + 5) + (2x^3 + 3x^2 - 2x + 2)$$

because this is an addition operation, you can simply add all of the like terms from both polynomials. To make this clear, let's first put the like terms together:

$$3x^3 + 2x^3 + 3x^2 - 4x - 2x + 5 + 2$$

This simplifies to:

$$5x^3 + 3x^2 - 6x + 7$$

We can verify this with Python:

In [2]:
```
from random import randint
x = randint(1,100)


(3*x**3 - 4*x + 5) + (2*x**3 + 3*x**2 - 2*x + 2) == 5*x**3 + 3*x**2 - 6*x + 7
```

Out[2]: True

Subtracting Polynomials

Subtracting polynomials is similar to adding them but you need to take into account that one of the polynomials is a negative.

Consider this expression:

$$(2x^2 - 4x + 5) - (x^2 - 2x + 2)$$

The key to performing this calculation is to realize that the subtraction of the second polynomial is really an expression that adds -1(x^2 - 2x + 2); so you can use the distributive property to multiply each of the terms in the polynomial by -1 (which in effect simply reverses the sign for each term). So our expression becomes:

$$(2x^2 - 4x + 5) + (-x^2 + 2x - 2)$$

Which we can solve as an addition problem. First place the like terms together:

$$2x^2 + -x^2 + -4x + 2x + 5 + -2$$

Which simplifies to:

$$x^2 - 2x + 3$$

Let's check that with Python:

In [3]:
```
from random import randint
x = randint(1,100)

(2*x**2 - 4*x + 5) - (x**2 - 2*x + 2) == x**2 - 2*x + 3
```

Out[3]: True

Multiplying Polynomials

To multiply two polynomials, you need to perform the following two steps:

1. Multiply each term in the first polynomial by each term in the second polynomial.
2. Add the results of the multiplication operations, combining like terms where possible.

For example, consider this expression:

$$(x^4 + 2)(2x^2 + 3x - 3)$$

Let's do the first step and multiply each term in the first polynomial by each term in the second polynomial. The first term in the first polynomial is x^4, and the first term in the second polynomial is $2x^2$, so multiplying these gives us $2x^6$. Then we can multiply the first term in the first polynomial (x^4) by the second term in the second polynomial (3x), which gives us $3x^5$, and so on until we've multipled all of the terms in the first polynomial by all of the terms in the second polynomial, which results in this:

$$2x^6 + 3x^5 - 3x^4 + 4x^2 + 6x - 6$$

We can verify a match between this result and the original expression this with the following Python code:

In [4]:
```
from random import randint
x = randint(1,100)

(x**4 + 2)*(2*x**2 + 3*x - 3) == 2*x**6 + 3*x**5 - 3*x**4 + 4*x**2 + 6*x - 6
```

Out[4]: True

Dividing Polynomials

When you need to divide one polynomial by another, there are two approaches you can take depending on the number of terms in the divisor (the expression you're dividing by).

Dividing Polynomials Using Simplification

In the simplest case, division of a polynomial by a monomial, the operation is really just simplification of a fraction.

For example, consider the following expression:

$$(4x + 6x^2) \div 2x$$

This can also be written as:

$$\frac{4x + 6x^2}{2x}$$

One approach to simplifying this fraction is to split it it into a separate fraction for each term in the dividend (the expression we're dividing), like this:

$$\frac{4x}{2x} + \frac{6x^2}{2x}$$

Then we can simplify each fraction and add the results. For the first fraction, 2x goes into 4x twice, so the fraction simplifies to 2; and for the second, $6x^2$ is 2x mutliplied by 3x. So our answer is 2 + 3x:

$$2 + 3x$$

Let's use Python to compare the original fraction with the simplified result for an arbitrary value of ***x***:

In [5]:
```
from random import randint
x = randint(1,100)

(4*x + 6*x**2) / (2*x) == 2 + 3*x
```

Out[5]:
```
True
```

Dividing Polynomials Using Long Division

Things get a little more complicated for divisors with more than one term.

Suppose we have the following expression:

$$(x^2 + 2x - 3) \div (x - 2)$$

Another way of writing this is to use the long-division format, like this:

$$x - 2 \overline{|x^2 + 2x - 3}$$

We begin long-division by dividing the highest order divisor into the highest order dividend - so in this case we divide x into x^2. X goes into x^2 x times, so we put an x on top and then multiply it through the divisor:

$$\begin{array}{r} x \\ \hline x - 2 \overline{|x^2 + 2x - 3} \\ x^2 - 2x \end{array}$$

Now we'll subtract the remaining dividend, and then carry down the -3 that we haven't used to see what's left:

$$\begin{array}{r} x \\ \hline x - 2 \overline{|x^2 + 2x - 3} \\ \underline{-(x^2 - 2x)\quad} \\ 4x - 3 \end{array}$$

OK, now we'll divide our highest order divisor into the highest order of the remaining dividend. In this case, x goes into 4x four times, so we'll add a 4 to the top line, multiply it through the divisor, and subtract the remaining dividend:

$$\begin{array}{r} x + 4 \\ \hline x - 2 \overline{|x^2 + 2x - 3} \\ \underline{-(x^2 - 2x)\quad} \\ 4x - 3 \\ -(\quad \underline{4x - 8)} \\ 5 \end{array}$$

We're now left with just 5, which we can't divide further by x - 2; so that's our remainder, which we'll add as a fraction.

The solution to our division problem is:

$$x + 4 + \frac{5}{x - 2}$$

Once again, we can use Python to check our answer:

In [6]:
```
from random import randint
x = randint(3,100)

(x**2 + 2*x -3)/(x-2) == x + 4 + (5/(x-2))
```

Out[6]:
```
True
```

Factorization

Factorization is the process of restating an expression as the *product* of two expressions (in other words, expressions multiplied together).

For example, you can make the value **16** by performing the following multiplications of integer numbers:

- 1 x 16
- 2 x 8
- 4 x 4

Another way of saying this is that 1, 2, 4, 8, and 16 are all factors of 16.

Factors of Polynomial Expressions

We can apply the same logic to polynomial expressions. For example, consider the following monomial expression:

$$-6x^2y^3$$

You can get this value by performing the following multiplication:

$$(2xy^2)(-3xy)$$

Run the following Python code to test this with arbitrary ***x*** and ***y*** values:

In [1]:
```
from random import randint
x = randint(1,100)
y = randint(1,100)

(2*x*y**2)*(-3*x*y) == -6*x**2*y**3
```

Out[1]: True

So, we can say that **$2xy^2$** and **$-3xy$** are both factors of **$-6x^2y^3$**.

This also applies to polynomials with more than one term. For example, consider the following expression:

$$(x + 2)(2x^2 - 3y + 2) = 2x^3 + 4x^2 - 3xy + 2x - 6y + 4$$

Based on this, **$x+2$** and **$2x^2 - 3y + 2$** are both factors of **$2x^3 + 4x^2 - 3xy + 2x - 6y + 4$**.

(and if you don't believe me, you can try this with random values for x and y with the following Python code):

In [2]:
```
from random import randint
x = randint(1,100)
y = randint(1,100)

(x + 2)*(2*x**2 - 3*y + 2) == 2*x**3 + 4*x**2 - 3*x*y + 2*x - 6*y + 4
```

Out[2]: True

Greatest Common Factor

Of course, these may not be the only factors of **$-6x^2y^3$**, just as 8 and 2 are not the only factors of 16.

Additionally, 2 and 8 aren't just factors of 16; they're factors of other numbers too - for example, they're both factors of 24 (because 2 x 12 = 24 and 8 x 3 = 24). Which leads us to the question, what is the highest number that is a factor of both 16 and 24? Well, let's look at all the numbers that multiply evenly into 12 and all the numbers that multiply evenly into 24:

16	**24**
1 x 16	1 x 24
2 x **8**	2 x 12
	3 x **8**
4 x 4	4 x 6

The highest value that is a multiple of both 16 and 24 is **8**, so 8 is the *Greatest Common Factor* (or GCF) of 16 and 24.

OK, let's apply that logic to the following expressions:

$$15x^2y \qquad 9xy^3$$

So what's the greatest common factor of these two expressions?

It helps to break the expressions into their consitituent components. Let's deal with the coefficients first; we have 15 and 9. The highest value that divides evenly into both of these is **3** (3 x 5 = 15 and 3 x 3 = 9).

Now let's look at the ***x*** terms; we have x^2 and x. The highest value that divides evenly into both is these is **x** (*x* goes into *x* once and into x^2 *x* times).

Finally, for our ***y*** terms, we have y and y^3. The highest value that divides evenly into both is these is **y** (*y* goes into *y* once and into y^3 *y*•*y* times).

Putting all of that together, the GCF of both of our expression is:

$$3xy$$

An easy shortcut to identifying the GCF of an expression that includes variables with exponentials is that it will always consist of:

- The *largest* numeric factor of the numeric coefficients in the polynomial expressions (in this case 3)
- The *smallest* exponential of each variable (in this case, x and y, which technically are x^1 and y^1.

You can check your answer by dividing the original expressions by the GCF to find the coefficent expressions for the GCF (in other words, how many times the GCF divides into the original expression). The result, when multiplied by the GCF will always produce the original expression. So in this case, we need to perform the following divisions:

$$\frac{15x^2y}{3xy} \qquad \frac{9xy^3}{3xy}$$

These fractions simplify to **5x** and **$3y^2$**, giving us the following calculations to prove our factorization:

$$3xy(5x) = 15x^2y$$
$$3xy(3y^2) = 9xy^3$$

Let's try both of those in Python:

In [3]:
```
from random import randint
x = randint(1,100)
y = randint(1,100)

print((3*x*y)*(5*x) == 15*x**2*y)
print((3*x*y)*(3*y**2) == 9*x*y**3)
```

```
True
True
```

Distributing Factors

Let's look at another example. Here is a binomial expression:

$$6x + 15y$$

To factor this, we need to find an expression that divides equally into both of these expressions. In this case, we can use **3** to factor the coefficents, because 3 • 2x = 6x and 3• 5y = 15y, so we can write our original expression as:

$$6x + 15y = 3(2x) + 3(5y)$$

Now, remember the distributive property? It enables us to multiply each term of an expression by the same factor to calculate the product of the expression multiplied by the factor. We can *factor-out* the common factor in this expression to distribute it like this:

$$6x + 15y = 3(2x) + 3(5y) = \mathbf{3(2x + 5y)}$$

Let's prove to ourselves that these all evaluate to the same thing:

In [4]:
```
from random import randint
x = randint(1,100)
y = randint(1,100)

(6*x + 15*y) == (3*(2*x) + 3*(5*y)) == (3*(2*x + 5*y))
```

Out[4]:
```
True
```

For something a little more complex, let's return to our previous example. Suppose we want to add our original $15x^2y$ and $9xy^3$ expressions:

$$15x^2y + 9xy^3$$

We've already calculated the common factor, so we know that:

$$3xy(5x) = 15x^2y$$
$$3xy(3y^2) = 9xy^3$$

Now we can factor-out the common factor to produce a single expression:

$$15x^2y + 9xy^3 = \mathbf{3xy(5x + 3y^2)}$$

And here's the Python test code:

In [5]:
```
from random import randint
x = randint(1,100)
y = randint(1,100)

(15*x**2*y + 9*x*y**3) == (3*x*y*(5*x + 3*y**2))
```

Out[5]:
```
True
```

So you might be wondering what's so great about being able to distribute the common factor like this. The answer is that it can often be useful to apply a common factor to multiple terms in order to solve seemingly complex problems.

For example, consider this:

$$x^2 + y^2 + z^2 = 127$$

Now solve this equation:

$$a = 5x^2 + 5y^2 + 5z^2$$

At first glance, this seems tricky because there are three unknown variables, and even though we know that their squares add up to 127, we don't know their individual values. However, we can distribute the common factor and apply what we *do* know. Let's restate the problem like this:

$$a = 5(x^2 + y^2 + z^2)$$

Now it becomes easier to solve, because we know that the expression in parenthesis is equal to 127, so actually our equation is:

$$a = 5(127)$$

So ***a*** is 5 times 127, which is 635

Formulae for Factoring Squares

There are some useful ways that you can employ factoring to deal with expressions that contain squared values (that is, values with an exponential of 2).

Differences of Squares

Consider the following expression:

$$x^2 - 9$$

The constant *9* is 3^2, so we could rewrite this as:

$$x^2 - 3^2$$

Whenever you need to subtract one squared term from another, you can use an approach called the *difference of squares*, whereby we can factor $a^2 - b^2$ as $(a - b)(a + b)$; so we can rewrite the expression as:

$$(x - 3)(x + 3)$$

Run the code below to check this:

In [6]:
```
from random import randint
x = randint(1,100)

(x**2 - 9) == (x - 3)*(x + 3)
```

Out[6]:
```
True
```

Perfect Squares

A *perfect square* is a number multiplied by itself, for example 3 multipled by 3 is 9, so 9 is a perfect square.

When working with equations, the ability to factor between polynomial expressions and binomial perfect square expressions can be a useful tool. For example, consider this expression:

$$x^2 + 10x + 25$$

We can use 5 as a common factor to rewrite this as:

$$(x + 5)(x + 5)$$

So what happened here?

Well, first we found a common factor for our coefficients: 5 goes into 10 twice and into 25 five times (in other words, squared). Then we just expressed this factoring as a multiplication of two identical binomials *(x + 5)(x + 5)*.

Remember the rule for multiplication of polynomials is to multiple each term in the first polynomial by each term in the second polynomial and then add the results; so you can do this to verify the factorization:

- x • x = x^2
- x • 5 = 5x
- 5 • x = 5x
- 5 • 5 = 25

When you combine the two 5x terms we get back to our original expression of x^2 + 10x + 25.

Now we have an expression multipled by itself; in other words, a perfect square. We can therefore rewrite this as:

$$(x + 5)^2$$

Factorization of perfect squares is a useful technique, as you'll see when we start to tackle quadratic equations in the next section. In fact, it's so useful that it's worth memorizing its formula:

$$(a + b)^2 = a^2 + b^2 + 2ab$$

In our example, the *a* terms is ***x*** and the *b* terms is ***5***, and in standard form, our equation x^2 + *10x* + *25* is actually a^2 + *2ab* + b^2. The operations are all additions, so the order isn't actually important!

Run the following code with random values for *a* and *b* to verify that the formula works:

In [7]:
```
from random import randint
a = randint(1,100)
b = randint(1,100)

a**2 + b**2 + (2*a*b) == (a + b)**2
```

Out[7]:
```
True
```

Quadratic Equations

Consider the following equation:

$$y = 2(x - 1)(x + 2)$$

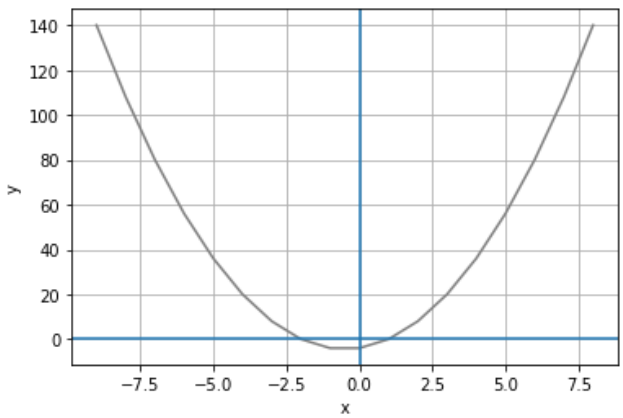
If you multiply out the factored ***x*** expressions, this equates to:

$$y = 2x^2 + 2x - 4$$

Note that the highest ordered term includes a squared variable (x^2).

Let's graph this equation for a range of ***x*** values:

In [1]:
```
import pandas as pd

# Create a dataframe with an x column containing values to plot
df = pd.DataFrame ({'x': range(-9, 9)})

# Add a y column by applying the quadratic equation to x
df['y'] = 2*df['x']**2 + 2 *df['x'] - 4

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()
plt.show()
```



Note that the graph shows a *parabola*, which is an arc-shaped line that reflects the x and y values calculated for the equation.

Now let's look at another equation that includes an x^2 term:

$$y = -2x^2 + 6x + 7$$

What does that look like as a graph?:

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

# Create a dataframe with an x column containing values to plot
df = pd.DataFrame ({'x': range(-8, 12)})

# Add a y column by applying the quadratic equation to x
df['y'] = -2*df['x']**2 + 6*df['x'] + 7

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()
plt.show()
```

Again, the graph shows a parabola, but this time instead of being open at the top, the parabola is open at the bottom.

Equations that assign a value to ***y*** based on an expression that includes a squared value for ***x*** create parabolas. If the relationship between ***y*** and ***x*** is such that ***y*** is a *positive* multiple of the x^2 term, the parabola will be open at the top; when ***y*** is a *negative* multiple of the x^2 term, then the parabola will be open at the bottom.

These kinds of equations are known as *quadratic* equations, and they have some interesting characteristics. There are several ways quadratic equations can be written, but the *standard form* for quadratic equation is:

$$y = ax^2 + bx + c$$

Where ***a***, ***b***, and ***c*** are numeric coefficients or constants.

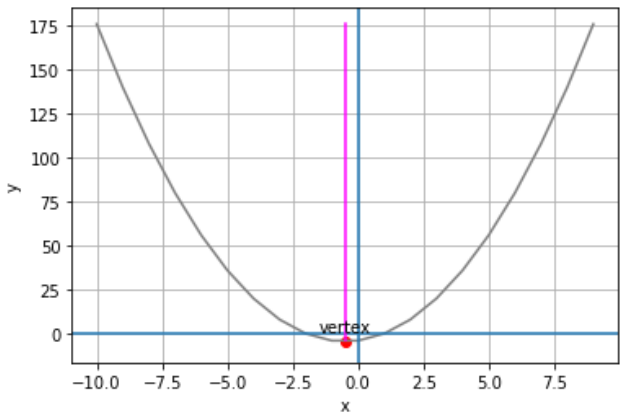
Let's start by examining the parabolas generated by quadratic equations in more detail.

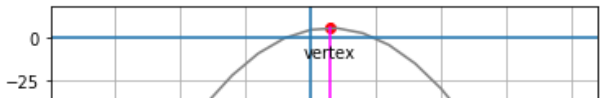
Parabola Vertex and Line of Symmetry

Parabolas are symmetrical, with x and y values converging exponentially towards the highest point (in the case of a downward opening parabola) or lowest point (in the case of an upward opening parabola). The point where the parabola meets the line of symmetry is known as the *vertex*.

Run the following cell to see the line of symmetry and vertex for the two parabolas described previously (don't worry about the calculations used to find the line of symmetry and vertex - we'll explore that later):

In [3]:
```
%matplotlib inline

def plot_parabola(a, b, c):
    import pandas as pd
    import numpy as np
    from matplotlib import pyplot as plt

    # get the x value for the line of symmetry
    vx = (-1*b)/(2*a)

    # get the y value when x is at the line of symmetry
    vy = a*vx**2 + b*vx + c

    # Create a dataframe with an x column containing values from x-10 to x+10
    minx = int(vx - 10)
    maxx = int(vx + 11)
    df = pd.DataFrame ({'x': range(minx, maxx)})

    # Add a y column by applying the quadratic equation to x
    df['y'] = a*df['x']**2 + b *df['x'] + c

    # get min and max y values
    miny = df.y.min()
    maxy = df.y.max()

    # Plot the line
    plt.plot(df.x, df.y, color="grey")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    plt.axhline()
    plt.axvline()

    # plot the line of symmetry
    sx = [vx, vx]
    sy = [miny, maxy]
    plt.plot(sx,sy, color='magenta')

    # Annotate the vertex
    plt.scatter(vx,vy, color="red")
    plt.annotate('vertex',(vx, vy), xytext=(vx - 1, (vy + 5)* np.sign(a)))

    plt.show()


plot_parabola(2, 2, -4)

plot_parabola(-2, 3, 5)
```





Parabola Intercepts

Recall that linear equations create lines that intersect the **x** and **y** axis of a graph, and we call the points where these intersections occur *intercepts*. Now look at the graphs of the parabolas we've worked with so far. Note that these parabolas both have a y-intercept; a point where the line intersects the y axis of the graph (in other words, when x is 0). However, note that the parabolas have *two* x-intercepts; in other words there are two points at which the line crosses the x axis (and y is 0). Additionally, imagine a downward opening parabola with its vertex at -1, -1. This is perfectly possible, and the line would never have an x value greater than -1, so it would have *no* x-intercepts.

Regardless of whether the parabola crosses the x axis or not, other than the vertex, for every ***y*** point in the parabola, there are *two* ***x*** points; one on the right (or positive) side of the axis of symmetry, and one of the left (or negative) side. The implications of this are what make quadratic equations so interesting. When we solve the equation for ***x***, there are *two* correct answers.

Let's take a look at an example to demonstrate this. Let's return to the first of our quadratic equations, and we'll look at it in its *factored* form:

$$y = 2(x - 1)(x + 2)$$

Now, let's solve this equation for a ***y*** value of 0. We can restate the equation like this:

$$2(x - 1)(x + 2) = 0$$

The equation is the product of two expressions **2(x - 1)** and **(x + 2)**. In this case, we know that the product of these expressions is 0, so logically *one or both of the expressions must return 0*.

Let's try the first one:

$$2(x - 1) = 0$$

If we distrbute this, we get:

$$2x - 2 = 0$$

This simplifies to:

$$2x = 2$$

Which gives us a value for *x* of **1**.

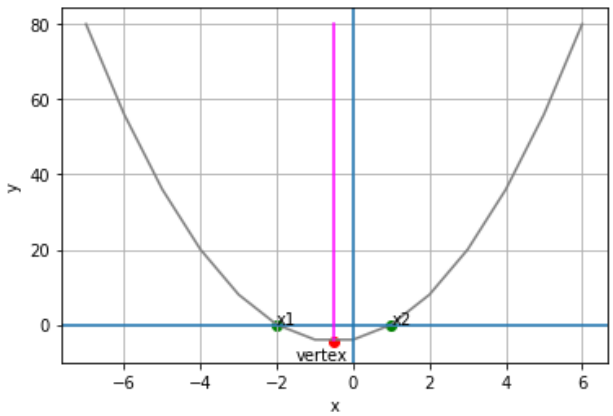
Now let's try the other expression:

$$x + 2 = 0$$

This gives us a value for *x* of **-2**.

So, when *y* is **0**, *x* is **-2** or **1**. Let's plot these points on our parabola:

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

# Assign the calculated x values
x1 = -2
x2 = 1

# Create a dataframe with an x column containing some values to plot
df = pd.DataFrame ({'x': range(x1-5, x2+6)})

# Add a y column by applying the quadratic equation to x
df['y'] = 2*(df['x'] - 1) * (df['x'] + 2)

# Get x at the line of symmetry (halfway between x1 and x2)
vx = (x1 + x2) / 2

# Get y when x is at the line of symmetry
vy = 2*(vx -1)*(vx + 2)

# get min and max y values
miny = df.y.min()
maxy = df.y.max()

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()

# Plot calculated x values for y = 0
plt.scatter([x1,x2],[0,0], color="green")
plt.annotate('x1',(x1, 0))
plt.annotate('x2',(x2, 0))

# plot the line of symmetry
sx = [vx, vx]
sy = [miny, maxy]
plt.plot(sx,sy, color='magenta')

# Annotate the vertex
plt.scatter(vx,vy, color="red")
plt.annotate('vertex',(vx, vy), xytext=(vx - 1, (vy - 5)))

plt.show()
```



So from the plot, we can see that both of the values we calculated for ***x*** align with the parabola when ***y*** is 0. Additionally, because the parabola is symmetrical, we know that every pair of ***x*** values for each ***y*** value will be equidistant from the line of symmetry, so we can calculate the ***x*** value for the line of symmetry as the average of the ***x*** values for any value of ***y***. This in turn means that we know the ***x*** coordinate for the vertex (it's on the line of symmetry), and we can use the quadratic equation to calculate ***y*** for this point.

Solving Quadratics Using the Square Root Method

The technique we just looked at makes it easy to calculate the two possible values for ***x*** when ***y*** is 0 if the equation is presented as the product two expressions. If the equation is in standard form, and it can be factored, you could do the necessary manipulation to restate it as the product of two expressions. Otherwise, you can calculate the possible values for x by applying a different method that takes advantage of the relationship between squared values and the square root.

Let's consider this equation:

$$y = 3x^2 - 12$$

Note that this is in the standard quadratic form, but there is no *b* term; in other words, there's no term that contains a coeffecient for ***x*** to the first power. This type of equation can be easily solved using the square root method. Let's restate it so we're solving for ***x*** when ***y*** is 0:

$$3x^2 - 12 = 0$$

The first thing we need to do is to isolate the x^2 term, so we'll remove the constant on the left by adding 12 to both sides:

$$3x^2 = 12$$

Then we'll divide both sides by 3 to isolate x^2:

$$x^2 = 4$$

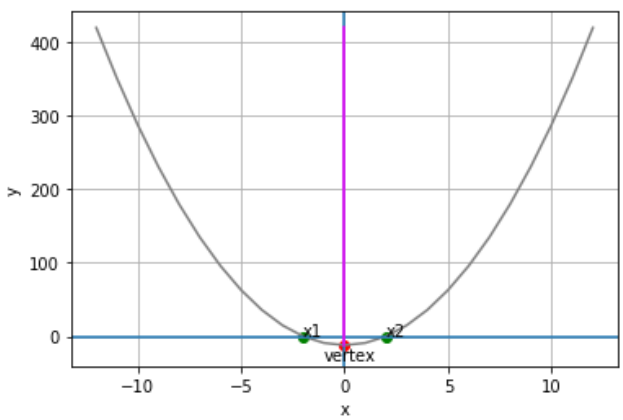
No we can isolate ***x*** by taking the square root of both sides. However, there's an additional consideration because this is a quadratic equation. The ***x*** variable can have two possibe values, so we must calculate the *principle* and *negative* square roots of the expression on the right:

$$x = \pm\sqrt{4}$$

The principle square root of 4 is 2 (because 2^2 is 4), and the corresponding negative root is -2 (because -2^2 is also 4); so *x* is **2** or **-2**.

Let's see this in Python, and use the results to calculate and plot the parabola with its line of symmetry and vertex:

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

y = 0
x1 = int(- math.sqrt(y + 12 / 3))
x2 = int(math.sqrt(y + 12 / 3))

# Create a dataframe with an x column containing some values to plot
df = pd.DataFrame ({'x': range(x1-10, x2+11)})

# Add a y column by applying the quadratic equation to x
df['y'] = 3*df['x']**2 - 12

# Get x at the line of symmetry (halfway between x1 and x2)
vx = (x1 + x2) / 2

# Get y when x is at the line of symmetry
vy = 3*vx**2 - 12

# get min and max y values
miny = df.y.min()
maxy = df.y.max()

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()

# Plot calculated x values for y = 0
plt.scatter([x1,x2],[0,0], color="green")
plt.annotate('x1',(x1, 0))
plt.annotate('x2',(x2, 0))

# plot the line of symmetry
sx = [vx, vx]
sy = [miny, maxy]
plt.plot(sx,sy, color='magenta')

# Annotate the vertex
plt.scatter(vx,vy, color="red")
plt.annotate('vertex',(vx, vy), xytext=(vx - 1, (vy - 20)))

plt.show()
```



Solving Quadratics Using the Completing the Square Method

In quadratic equations where there is a *b* term; that is, a term containing **x** to the first power, it is impossible to directly calculate the square root. However, with some algebraic manipulation, you can take advantage of the ability to factor a polynomial expression in the form $a^2 + 2ab + b^2$ as a binomial *perfect square* expression in the form $(a + b)^2$.

At first this might seem like some sort of mathematical sleight of hand, but follow through the steps carefull and you'll see that there's nothing up my sleeve!

The underlying basis of this approach is that a trinomial expression like this:

$$x^2 + 24x + 12^2$$

Can be factored to this:

$$(x + 12)^2$$

OK, so how does this help us solve a quadratic equation? Well, let's look at an example:

$$y = x^2 + 6x - 7$$

Let's start as we've always done so far by restating the equation to solve ***x*** for a ***y*** value of 0:

$$x^2 + 6x - 7 = 0$$

Now we can move the constant term to the right by adding 7 to both sides:

$$x^2 + 6x = 7$$

OK, now let's look at the expression on the left: $x^2 + 6x$. We can't take the square root of this, but we can turn it into a trinomial that will factor into a perfect square by adding a squared constant. The question is, what should that constant be? Well, we know that we're looking for an expression like $x^2 + 2\mathbf{c}x + \mathbf{c}^2$, so our constant **c** is half of the coefficient we currently have for ***x***. This is **6**, making our constant **3**, which when squared is **9** So we can create a trinomial expression that will easily factor to a perfect square by adding 9; giving us the expression $x^2 + 6x + 9$.

However, we can't just add something to one side without also adding it to the other, so our equation becomes:

$$x^2 + 6x + 9 = 16$$

So, how does that help? Well, we can now factor the trinomial expression as a perfect square binomial expression:

$$(x + 3)^2 = 16$$

And now, we can use the square root method to find x + 3:

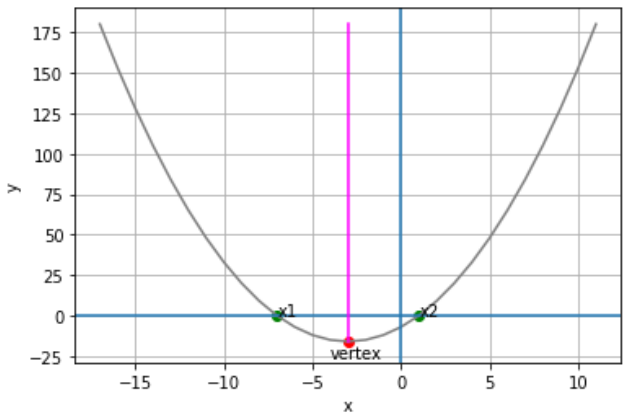
$$x + 3 = \pm\sqrt{16}$$

So, x + 3 is **-4** or **4**. We isolate ***x*** by subtracting 3 from both sides, so ***x*** is **-7** or **1**:

$$x = -7, 1$$

Let's see what the parabola for this equation looks like in Python:

In [6]:
```
import pandas as pd
import math

x1 = int(- math.sqrt(16) - 3)
x2 = int(math.sqrt(16) - 3)

# Create a dataframe with an x column containing some values to plot
df = pd.DataFrame ({'x': range(x1-10, x2+11)})

# Add a y column by applying the quadratic equation to x
df['y'] = ((df['x'] + 3)**2) - 16

# Get x at the line of symmetry (halfway between x1 and x2)
vx = (x1 + x2) / 2

# Get y when x is at the line of symmetry
vy = ((vx + 3)**2) - 16

# get min and max y values
miny = df.y.min()
maxy = df.y.max()

# Plot the line
%matplotlib inline
from matplotlib import pyplot as plt

plt.plot(df.x, df.y, color="grey")
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.axhline()
plt.axvline()

# Plot calculated x values for y = 0
plt.scatter([x1,x2],[0,0], color="green")
plt.annotate('x1',(x1, 0))
plt.annotate('x2',(x2, 0))

# plot the line of symmetry
sx = [vx, vx]
sy = [miny, maxy]
plt.plot(sx,sy, color='magenta')

# Annotate the vertex
plt.scatter(vx,vy, color="red")
plt.annotate('vertex',(vx, vy), xytext=(vx - 1, (vy - 10)))

plt.show()
```



Vertex Form

Let's look at another example of a quadratic equation in standard form:

$$y = 2x^2 - 16x + 2$$

We can start to solve this by subtracting 2 from both sides to move the constant term from the right to the left:

$$y - 2 = 2x^2 - 16x$$

Now we can factor out the coefficient for x^2, which is **2**. $2x^2$ is 2 • x^2, and -16x is 2 • 8x:

$$y - 2 = 2(x^2 - 8x)$$

Now we're ready to complete the square, so we add the square of half of the -8x coefficient on the right side to the parenthesis. Half of -8 is -4, and -4^2 is 16, so the right side of the equation becomes *$2(x^2 - 8x + 16)$*. Of course, we can't add something to one side of the equation without also adding it to the other side, and we've just added 2 • 16 (which is 32) to the right, so we must also add that to the left.

$$y - 2 + 32 = 2(x^2 - 8x + 16)$$

Now we can simplify the left and factor out a perfect square binomial expression on the right:

$$y + 30 = 2(x - 4)^2$$

We now have a squared term for ***x***, so we could use the square root method to solve the equation. However, we can also isolate ***y*** by subtracting 30 from both sides. So we end up restating the original equation as:

$$y = 2(x - 4)^2 - 30$$

Let's just quickly check our math with Python:

In [7]:
```
from random import randint
x = randint(1,100)

2*x**2 - 16*x + 2 == 2*(x - 4)**2 - 30
```

Out[7]:
```
True
```

So we've managed to take the expression ***2x^2 - 16x + 2*** and change it to ***2(x - 4)^2 - 30***. How does that help?

Well, when a quadratic equation is stated this way, it's in *vertex form*, which is generically described as:

$$y = a(x - h)^2 + k$$

The neat thing about this form of the equation is that it tells us the coordinates of the vertex - it's at ***h,k***.

So in this case, we know that the vertex of our equation is 4, -30. Moreover, we know that the line of symmetry is at ***x = 4***.

We can then just use the equation to calculate two more points, and the three points will be enough for us to determine the shape of the parabola. We can simply choose any ***x*** value we like and substitute it into the equation to calculate the corresponding ***y*** value. For example, let's calculate ***y*** when x is **0**:

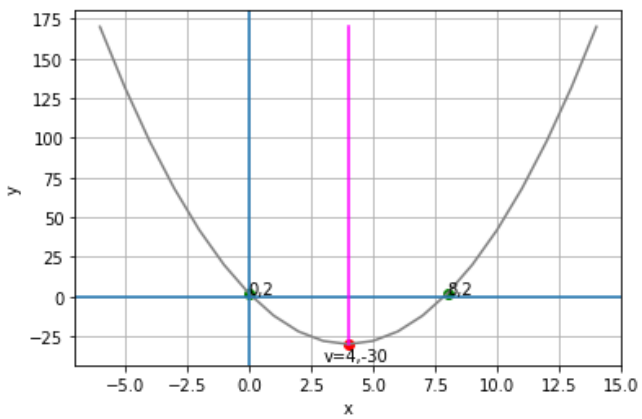
$$y = 2(0 - 4)^2 - 30$$

When we work through the equation, it gives us the answer **2**, so we know that the point 0, 2 is in our parabola.

So, we know that the line of symmetry is at ***x = h*** (which is 4), and we now know that the ***y*** value when ***x*** is 0 (***h*** - ***h***) is 2. The ***y*** value at the same distance from the line of symmetry in the negative direction will be the same as the value in the positive direction, so when ***x*** is ***h*** + ***h***, the ***y*** value will also be 2.

The following Python code encapulates all of this in a function that draws and annotates a parabola using only the ***a***, ***h***, and ***k*** values from a quadratic equation in vertex form:

In [8]:
```
def plot_parabola_from_vertex_form(a, h, k):
    import pandas as pd
    import math

    # Create a dataframe with an x column a range of x values to plot
    df = pd.DataFrame ({'x': range(h-10, h+11)})

    # Add a y column by applying the quadratic equation to x
    df['y'] = (a*(df['x'] - h)**2) + k

    # get min and max y values
    miny = df.y.min()
    maxy = df.y.max()

    # calculate y when x is 0 (h+-h)
    y = a*(0 - h)**2 + k

    # Plot the line
    %matplotlib inline
    from matplotlib import pyplot as plt

    plt.plot(df.x, df.y, color="grey")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    plt.axhline()
    plt.axvline()

    # Plot calculated y values for x = 0 (h-h and h+h)
    plt.scatter([h-h, h+h],[y,y], color="green")
    plt.annotate(str(h-h) + ',' + str(y),(h-h, y))
    plt.annotate(str(h+h) + ',' + str(y),(h+h, y))

    # plot the line of symmetry (x = h)
    sx = [h, h]
    sy = [miny, maxy]
    plt.plot(sx,sy, color='magenta')

    # Annotate the vertex (h,k)
    plt.scatter(h,k, color="red")
    plt.annotate('v=' + str(h) + ',' + str(k),(h, k), xytext=(h - 1, (k - 10)))

    plt.show()


# Call the function for the example discussed above
plot_parabola_from_vertex_form(2, 4, -30)
```



It's important to note that the vertex form specifically requires a *subtraction* operation in the factored perfect square term. For example, consider the following equation in the standard form:

$$y = 3x^2 + 6x + 2$$

The steps to solve this are:

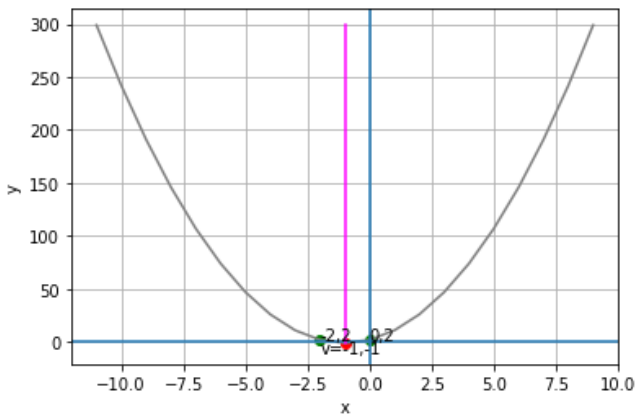
1. Move the constant to the left side:
$$y - 2 = 3x^2 + 6x$$
2. Factor the ***x*** expressions on the right:
$$y - 2 = 3(x^2 + 2x)$$
3. Add the square of half the x coefficient to the right, and the corresponding multiple on the left:
$$y - 2 + 3 = 3(x^2 + 2x + 1)$$
4. Factor out a perfect square binomial:
$$y + 1 = 3(x + 1)^2$$
5. Move the constant back to the right side:
$$y = 3(x + 1)^2 - 1$$

To express this in vertex form, we need to convert the addition in the parenthesis to a subtraction:

$$y = 3(x - -1)^2 - 1$$

Now, we can use the a, h, and k values to define a parabola:

In [9]:
```
plot_parabola_from_vertex_form(3, -1, -1)
```



Shortcuts for Solving Quadratic Equations

We've spent some time in this notebook discussing how to solve quadratic equations to determine the vertex of a parabola and the ***x*** values in relation to ***y***. It's important to understand the techniques we've used, which incude:

- Factoring
- Calculating the Square Root
- Completing the Square
- Using the vertex form of the equation

The underlying algebra for all of these techniques is the same, and this consistent algebra results in some shortcuts that you can memorize to make it easier to solve quadratic equations without going through all of the steps:

Calculating the Vertex from Standard Form

You've already seen that converting a quadratic equation to the vertex form makes it easy to identify the vertex coordinates, as they're encoded as ***h*** and ***k*** in the equation itself - like this:

$$y = a(x - \mathbf{h})^2 + \mathbf{k}$$

However, what if you have an equation in standard form?:

$$y = ax^2 + bx + c$$

There's a quick and easy technique you can apply to get the vertex coordinates.

1. To find ***h*** (which is the x-coordinate of the vertex), apply the following formula:
$$h = \frac{-b}{2a}$$
2. After you've found ***h***, use it in the quadratic equation to solve for ***k***:
$$\mathbf{k} = a\mathbf{h}^2 + b\mathbf{h} + c$$

For example, here's the quadratic equation in standard form that we previously converted to the vertex form:

$$y = 2x^2 - 16x + 2$$

To find ***h***, we perform the following calculation:

$$h = \frac{-b}{2a} \quad = \quad \frac{-1 \cdot 16}{2 \cdot 2} \quad = \quad \frac{16}{4} \quad = \quad 4$$

Then we simply plug the value we've obtained for ***h*** into the quadratic equation in order to find ***k***:

$$k = 2 \cdot (4^2) - 16 \cdot 4 + 2 \quad = \quad 32 - 64 + 2 \quad = \quad -30$$

Note that a vertex at 4,-30 is also what we previously calculated for the vertex form of the same equation:

$$y = 2(x - 4)^2 - 30$$

The Quadratic Formula

Another useful formula to remember is the *quadratic formula*, which makes it easy to calculate values for ***x*** when ***y*** is **0**; or in other words:

$$ax^2 + bx + c = 0$$

Here's the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In [10]:

```
def plot_parabola_from_formula (a, b, c):
    import math

    # Get vertex
    print('CALCULATING THE VERTEX')
    print('vx = -b / 2a')

    nb = -b
    a2 = 2*a
    print('vx = ' + str(nb) + ' / ' + str(a2))

    vx = -b/(2*a)
    print('vx = ' + str(vx))

    print('\nvy = ax^2 + bx + c')
    print('vy =' + str(a) + '(' + str(vx) + '^2) + ' + str(b) + '(' + str(vx) +

    avx2 = a*vx**2
    bvx = b*vx
    print('vy =' + str(avx2) + ' + ' + str(bvx) + ' + ' + str(c))

    vy = avx2 + bvx + c
    print('vy = ' + str(vy))

    print ('\nv = ' + str(vx) + ',' + str(vy))

    # Get +x and -x (showing intermediate calculations)
    print('\nCALCULATING -x AND +x FOR y=0')
    print('x = -b +- sqrt(b^2 - 4ac) / 2a')


    b2 = b**2
    ac4 = 4*a*c
    print('x = ' + str(nb) + '+-sqrt(' + str(b2) + ' - ' + str(ac4) + ')/' + st

    sr = math.sqrt(b2 - ac4)
    print('x = ' + str(nb) + ' +- ' + str(sr) + ' / ' + str(a2))
    print('-x = ' + str(nb) + ' - ' + str(sr) + ' / ' + str(a2))
    print('+x = ' + str(nb) + ' + ' + str(sr) + ' / ' + str(a2))

    posx = (nb + sr) / a2
    negx = (nb - sr) / a2
    print('-x = ' + str(negx))
    print('+x = ' + str(posx))


    print('\nPLOTTING THE PARABOLA')
    import pandas as pd

    # Create a dataframe with an x column a range of x values to plot
    df = pd.DataFrame ({'x': range(round(vx)-10, round(vx)+11)})

    # Add a y column by applying the quadratic equation to x
    df['y'] = a*df['x']**2 + b*df['x'] + c

    # get min and max y values
    miny = df.y.min()
    maxy = df.y.max()

    # Plot the line
    %matplotlib inline
    from matplotlib import pyplot as plt

    plt.plot(df.x, df.y, color="grey")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    plt.axhline()
```

Functions

So far in this course we've explored equations that perform algebraic operations to produce one or more results. A *function* is a way of encapsulating an operation that takes an input and produces exactly one ouput.

For example, consider the following function definition:

$$f(x) = x^2 + 2$$

This defines a function named ***f*** that accepts one input (***x***) and returns a single value that is the result calculated by the expression $x^2 + 2$.

Having defined the function, we can use it for any input value. For example:

$$f(3) = 11$$

You've already seen a few examples of Python functions, which are defined using the **def** keyword. However, the strict definition of an algebraic function is that it must return a single value. Here's an example of defining and using a Python function that meets this criteria:

In [1]:
```
# define a function to return x^2 + 2
def f(x):
    return x**2 + 2

# call the function
f(3)
```

Out[1]:
```
11
```

You can use functions in equations, just like any other term. For example, consider the following equation:

$$y = f(x) - 1$$

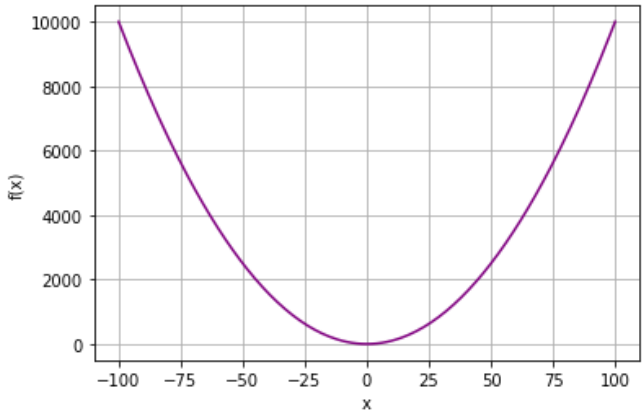
To calculate a value for ***y***, we take the ***f*** of ***x*** and subtract 1. So assuming that ***f*** is defined as previously, given an ***x*** value of 4, this equation returns a ***y*** value of **17** ($f(4)$ returns $4^2 + 2$, so 16 + 2 = 18; and then the equation subtracts 1 to give us 17). Here it is in Python:

In [2]:
```
x = 4
y = f(x) - 1
print(y)
```

```
17
```

Of course, the value returned by a function depends on the input; and you can graph this with the iput (let's call it ***x***) on one axis and the output (***f(x)***) on the other.

In [3]:
```
%matplotlib inline

import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from -100 to 100
x = np.array(range(-100, 101))

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot x against f(x)
plt.plot(x,f(x), color='purple')

plt.show()
```



As you can see (if you hadn't already figured it out), our function is a *quadratic function* - it returns a squared value that results in a parabolic graph when the output for multiple input values are plotted.

Bounds of a Function

Some functions will work for any input and may return any output. For example, consider the function ***u*** defined here:

$$u(x) = x + 1$$

This function simply adds 1 to whatever input is passed to it, so it will produce a defined output for any value of ***x*** that is a *real* number; in other words, any "regular" number - but not an *imaginary* number like √-1, or ∞ (infinity). You can specify the set of real numbers using the symbol $\mathbb{R}$ (note the double stroke). The values that can be used for ***x*** can be expressed as a *set*, which we indicate by enclosing all of the members of the set in "{...}" braces; so to indicate the set of all possible values for x such that x is a member of the set of all real numbers, we can use the following expression:

$$\{x \in \mathbb{R}\}$$

Domain of a Function

We call the set of numbers for which a function can return value it's *domain*, and in this case, the domain of ***u*** is the set of all real numbers; which is actually the default assumption for most functions.

Now consider the following function ***g***:

$$g(x) = (\frac{12}{2x})^2$$

If we use this function with an ***x*** value of **2**, we would get the output **9**; because $(12 \div (2 \bullet 2))^2$ is 9. Similarly, if we use the value **-3** for ***x***, the output will be **4**. However, what happens when we apply this function to an ***x*** value of **0**? Anything divided by 0 is undefined, so the function ***g*** doesn't work for an ***x*** value of 0.

So we need a way to denote the domain of the function ***g*** by indicating the input values for which a defined output can be returned. Specifically, we need to restrict ***x*** to a specific list of values - specifically any real number that is not 0. To indicate this, we can use the following notation:

$$\{x \in \mathbb{R} \mid x \neq 0\}$$

This is interpreted as *Any value for x where x is in the set of real numbers such that x is not equal to 0*, and we can incorporate this into the function's definition like this:

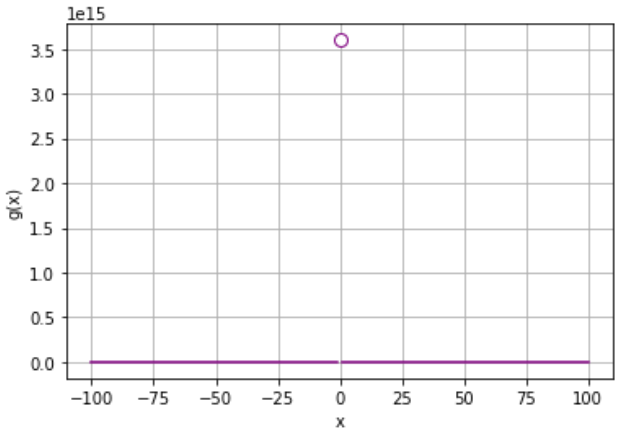
$$g(x) = (\frac{12}{2x})^2, \{x \in \mathbb{R} \mid x \neq 0\}$$

Or more simply:

$$g(x) = (\frac{12}{2x})^2, \quad x \neq 0$$

When you plot the output of a function, you can indicate the gaps caused by input values that are not in the function's domain by plotting an empty circle to show that the function is not defined at this point:

In [4]:
```
%matplotlib inline

# Define function g
def g(x):
    if x != 0:
        return (12/(2*x))**2

# Plot output from function g
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from -100 to 100
x = range(-100, 101)

# Get the corresponding y values from the function
y = [g(a) for a in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('g(x)')
plt.grid()

# Plot x against g(x)
plt.plot(x,y, color='purple')

# plot an empty circle to show the undefined point
plt.plot(0,g(0.0000001), color='purple', marker='o', markerfacecolor='w', marke

plt.show()
```



Note that the function works for every value other than 0; so the function is defined for x = 0.000000001, and for x = -0.000000001; it only fails to return a defined value for exactly 0.

OK, let's take another example. Consider this function:

$$h(x) = 2\sqrt{x}$$

Applying this function to a non-negative ***x*** value returns a meaningful output; but for any value where ***x*** is negative, the output is undefined.

We can indicate the domain of this function in its definition like this:

$$h(x) = 2\sqrt{x}, \{x \in \mathbb{R} \mid x \geq 0\}$$

This is interpreted as *Any value for x where x is in the set of real numbers such that x is greater than or equal to 0*.

Or, you might see this in a simpler format:

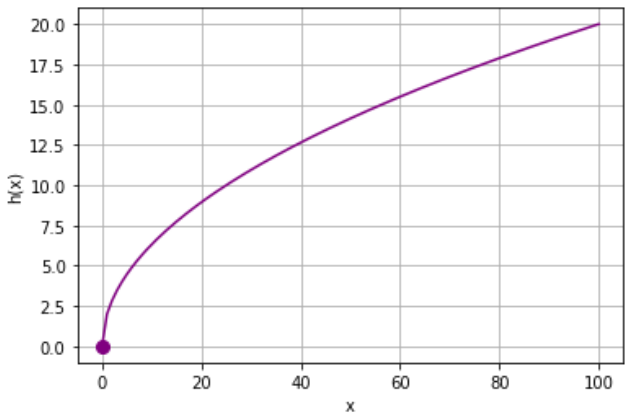
$$h(x) = 2\sqrt{x}, \quad x \geq 0$$

Note that the symbol ≥ is used to indicate that the value must be *greater than* ***or equal to**** *0; and this means that* ***0*** *is included in the set of valid values. To indicate that the value must be greater than 0, *not including 0***, use the > symbol. You can also use the equivalent symbols for *less than or equal to* (≤) and *less than* (<).

When plotting a function line that marks the end of a continuous range, the end of the line is shown as a circle, which is filled if the function includes the value at that point, and unfilled if it does not.

Here's the Python to plot function ***h***:

In [5]:
```
%matplotlib inline

def h(x):
    if x >= 0:
        import numpy as np
        return 2 * np.sqrt(x)

# Plot output from function h
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from -100 to 100
x = range(-100, 101)

# Get the corresponding y values from the function
y = [h(a) for a in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('h(x)')
plt.grid()

# Plot x against h(x)
plt.plot(x,y, color='purple')

# plot a filled circle at the end to indicate a closed interval
plt.plot(0, h(0), color='purple', marker='o', markerfacecolor='purple', markers

plt.show()
```



Sometimes, a function may be defined for a specific *interval*; for example, for all values between 0 and 5:

$$j(x) = x + 2, \quad x \geq 0 \text{ and } x \leq 5$$

In this case, the function is defined for ***x*** values between 0 and 5 *inclusive*; in other words, **0** and **5** are included in the set of defined values. This is known as a *closed* interval and can be indicated like this:

$$\{x \in \mathbb{R} \ | \ 0 \leq x \leq 5\}$$

It could also be indicated like this:

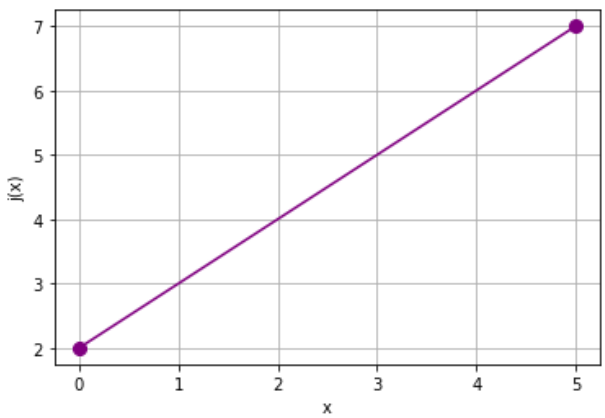
$$\{x \in \mathbb{R} \ | \ [0, 5]\}$$

If the condition in the function was **x > 0 and x < 5**, then the interval would be described as *open* and 0 and 5 would *not* be included in the set of defined values. This would be indicated using one of the following expressions:

$$\{x \in \mathbb{R} \ | \ 0 < x < 5\}$$
$$\{x \in \mathbb{R} \ | \ (0, 5)\}$$

Here's function ***j*** in Python:

In [6]:
```
%matplotlib inline

def j(x):
    if x >= 0 and x <= 5:
        return x + 2


# Plot output from function j
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from -100 to 100
x = range(-100, 101)
y = [j(a) for a in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('j(x)')
plt.grid()

# Plot x against k(x)
plt.plot(x, y, color='purple')

# plot a filled circle at the ends to indicate an open interval
plt.plot(0, j(0), color='purple', marker='o', markerfacecolor='purple', markers
plt.plot(5, j(5), color='purple', marker='o', markerfacecolor='purple', markers

plt.show()
```



Now, suppose we have a function like this:

$$k(x) = \begin{cases} 0, & \text{if } x = 0, \\ 1, & \text{if } x = 100 \end{cases}$$

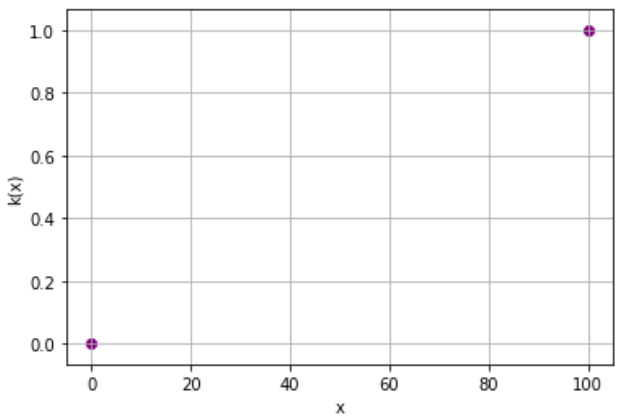
In this case, the function has highly restricted domain; it only returns a defined output for 0 and 100. No output for any other ***x*** value is defined. In this case, the set of the domain is:

$$\{0, 100\}$$

Note that this does not include all real numbers, it only includes 0 and 100.

When we use Python to plot this function, note that it only makes sense to plot a scatter plot showing the individual values returned, there is no line in between because the function is not continuous between the values within the domain.

In [7]:
```
%matplotlib inline

def k(x):
    if x == 0:
        return 0
    elif x == 100:
        return 1


# Plot output from function k
from matplotlib import pyplot as plt

# Create an array of x values from -100 to 100
x = range(-100, 101)
# Get the k(x) values for every value in x
y = [k(a) for a in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('k(x)')
plt.grid()

# Plot x against k(x)
plt.scatter(x, y, color='purple')

plt.show()
```



Range of a Function

Just as the domain of a function defines the set of values for which the function is defined, the *range* of a function defines the set of possible outputs from the function.

For example, consider the following function:

$$p(x) = x^2 + 1$$

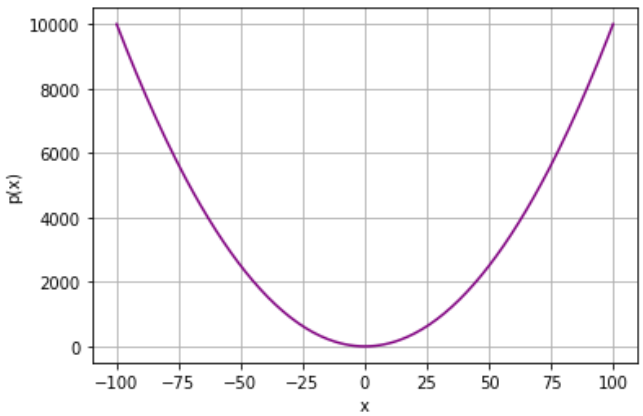
The domain of this function is all real numbers. However, this is a quadratic function, so the output values will form a parabola; and since the function has no negative coefficient or constant, it will be an upward opening parabola with a vertex that has a y value of 1.

So what does that tell us? Well, the minimum value that will be returned by this function is 1, so it's range is:

$$\{p(x) \in \mathbb{R} \mid p(x) \geq 1\}$$

Let's create and plot the function for a range of ***x*** values in Python:

In [8]:
```
%matplotlib inline

# define a function to return x^2 + 1
def p(x):
    return x**2 + 1


# Plot the function
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from -100 to 100
x = np.array(range(-100, 101))

# Set up the graph
plt.xlabel('x')
plt.ylabel('p(x)')
plt.grid()

# Plot x against f(x)
plt.plot(x,p(x), color='purple')

plt.show()
```



Note that the ***p(x)*** values in the plot drop exponentially for ***x*** values that are negative, and then rise exponentially for positive ***x*** values; but the minimum value returned by the function (for an *x* value of 0) is **1**.

Rate of Change

Functions are often visualized as a line on a graph, and this line shows how the value returned by the function changes based on changes in the input value.

Linear Rate of Change

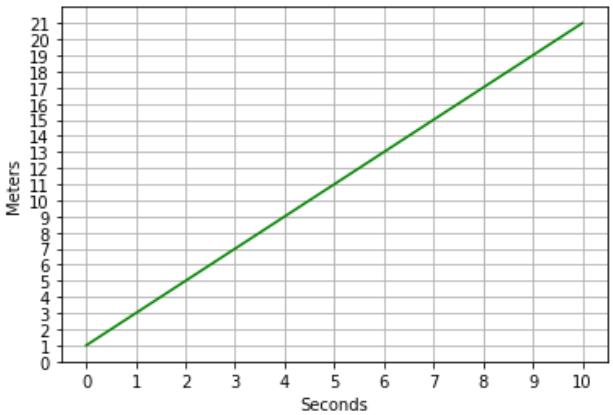
For example, imagine a function that returns the number of meters travelled by a cyclist based on the number of seconds that the cyclist has been cycling.

Here is such a function:

$$q(x) = 2x + 1$$

We can plot the output for this function for a period of 10 seconds like this:

In [1]:
```
%matplotlib inline

def q(x):
    return 2*x + 1

# Plot the function
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10
x = np.array(range(0, 11))

# Set up the graph
plt.xlabel('Seconds')
plt.ylabel('Meters')
plt.xticks(range(0,11, 1))
plt.yticks(range(0, 22, 1))
plt.grid()

# Plot x against q(x)
plt.plot(x,q(x), color='green')

plt.show()
```



It's clear from the graph that ***q*** is a *linear* function that describes a slope in which distance increases at a constant rate over time. In other words, the cyclist is travelling at a constant speed.

But what speed?

Speed, or more technically, velocity is a measure of change - it measures how the distance travelled changes over time (which is why we typically express it as a unit of distance per a unit of time, like *miles-per-hour* or *meters-per-second*). So we're looking for a way to measure the change in the line created by the function.

The change in values along the line define its *slope*, which we know from a previous lesson is represented like this:

$$m = \frac{\Delta y}{\Delta x}$$

We can calculate the slope of our function like this:

$$m = \frac{q(x)_2 - q(x)_1}{x_2 - x_1}$$

So we just need two ordered pairs of ***x*** and ***q(x)*** values from our line to apply this equation.

- After 1 second, ***x*** is 1 and ***q***(1) = **3**.
- After 10 seconds, ***x*** is 10 and ***q***(10) = 21.

So we can meassure the rate of change like this:

$$m = \frac{21 - 3}{10 - 1}$$

This is the same as:

$$m = \frac{18}{9}$$

Which simplifies to:

$$m = \frac{2}{1}$$

So our rate of change is $^2/_1$ or put another way, the cyclist is travelling at 2 meters-per-second.

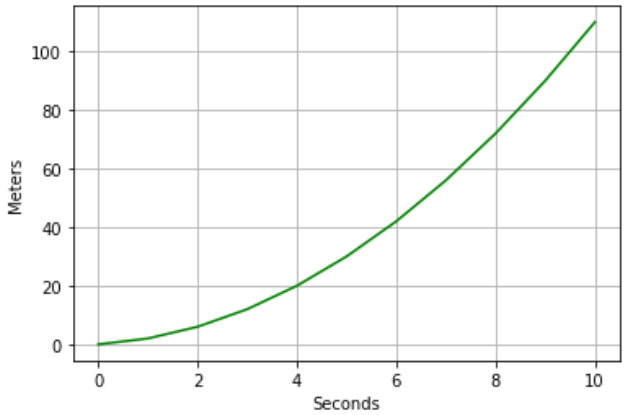
Average Rate of Change

OK, let's look at another function that calculates distance travelled for a given number of seconds:

$$r(x) = x^2 + x$$

Let's take a look at that using Python:

In [2]:
```
%matplotlib inline

def r(x):
    return x**2 + x

# Plot the function
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10
x = np.array(range(0, 11))

# Set up the graph
plt.xlabel('Seconds')
plt.ylabel('Meters')
plt.grid()

# Plot x against r(x)
plt.plot(x,r(x), color='green')

plt.show()
```



This time, the function is not linear. It's actually a quadratic function, and the line from 0 seconds to 10 seconds shows an exponential increase; in other words, the cyclist is *accelerating*.

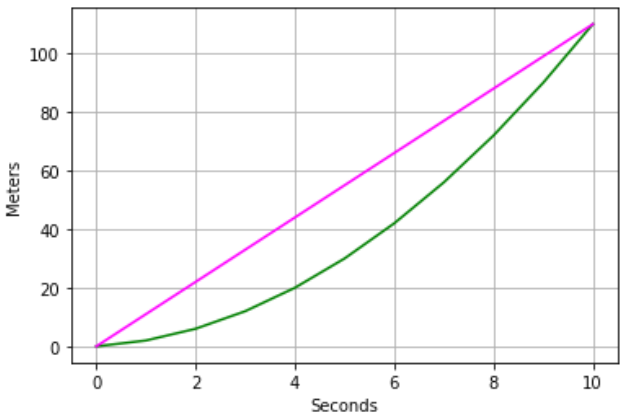
Technically, acceleration itself is a measure of change in velocity over time; and velocity, as we've already discussed, is a measure of change in distance over time. So measuring accelleration is pretty complex, and requires *differential calculus*, which we're going to cover shortly. In fact, even just measuring the velocity at a single point in time requires differential calculus; but we can use algebraic methods to calculate an *average* rate of velocity for a given period shown in the graph.

First, we need to define a *secant* line that joins two points in our exponential arc to create a straight slope. For example, a secant line for the entire 10 second time span would join the following two points:

- 0, ***r***(0)
- 10, ***r***(10)

Run the following Python code to visualize this line:

In [3]:
```
%matplotlib inline

def r(x):
    return (x)**2 + x

# Plot the function
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10
x = np.array(range(0, 11))

# Create an array for the secant line
s = np.array([0,10])

# Set up the graph
plt.xlabel('Seconds')
plt.ylabel('Meters')
plt.grid()

# Plot x against r(x)
plt.plot(x,r(x), color='green')

# Plot the secant line
plt.plot(s,r(s), color='magenta')

plt.show()
```



Now, because the secant line is straight, we can apply the slope formula we used for a linear function to calculate the average velocity for the 10 second period:

- At 0 seconds, ***x*** is 0 and ***r***(0) = **0**.
- At 10 seconds, ***x*** is 10 and ***r***(10) = 110.

So we can meassure the rate of change like this:

$$m = \frac{110 - 0}{10 - 0}$$

This is the same as:

$$m = \frac{110}{10}$$

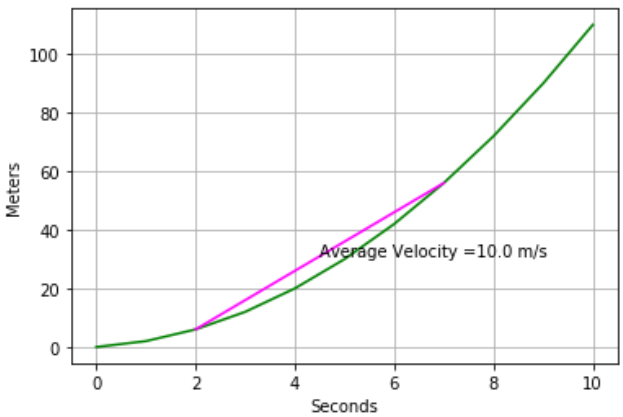
Which simplifies to:

$$m = \frac{11}{1}$$

So our rate of change is $^{11}/_{1}$ or put another way, the cyclist is travelling at an average velocity of 11 meters-per-second over the 10-second period.

Of course, we can measure the average velocity between any two points on the exponential line. Use the following Python code to show the secant line for the period between 2 and 7 seconds, and calculate the average velocity for that period

In [4]:
```
%matplotlib inline

def r(x):
    return x**2 + x

# Plot the function
import numpy as np
from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10
x = np.array(range(0, 11))

# Create an array for the secant line
s = np.array([2,7])

# Calculate rate of change
x1 = s[0]
x2 = s[-1]
y1 = r(x1)
y2 = r(x2)
a = (y2 - y1)/(x2 - x1)


# Set up the graph
plt.xlabel('Seconds')
plt.ylabel('Meters')
plt.grid()

# Plot x against r(x)
plt.plot(x,r(x), color='green')

# Plot the secant line
plt.plot(s,r(s), color='magenta')

plt.annotate('Average Velocity =' + str(a) + ' m/s',((x2+x1)/2, (y2+y1)/2))

plt.show()
```



Limits

You can use algebraeic methods to calculate the rate of change over a function interval by joining two points on the function with a secant line and measuring its slope. For example, a function might return the distance travelled by a cyclist in a period of time, and you can use a secant line to measure the average velocity between two points in time. However, this doesn't tell you the cyclist's vecolcity at any single point in time - just the average speed over an interval.

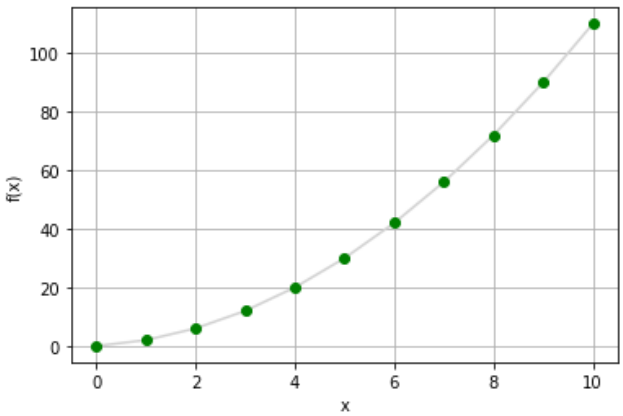
To find the cyclist's velocity at a specific point in time, you need the ability to find the slope of a curve at a given point. *Differential Calculus* enables us to do through the use of *derivatives*. We can use derivatives to find the slope at a specific x value by calculating a delta for x_1 and x_2 values that are infinitesimally close together - so you can think of it as measuring the slope of a tiny straight line that comprises part of the curve.

Introduction to Limits

However, before we can jump straight into derivatives, we need to examine another aspect of differential calculus - the *limit* of a function; which helps us measure how a function's value changes as the x_2 value approaches x_1

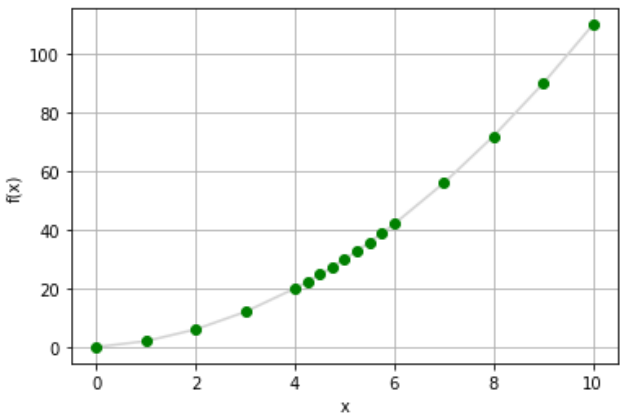
To better understand limits, let's take a closer look at our function, and note that although we graph the function as a line, it is in fact made up of individual points. Run the following cell to show the points that we've plotted for integer values of $\boldsymbol{x}$ - the line is created by interpolating the points in between:

In [1]:
```
%matplotlib inline

# Here's the function
def f(x):
    return x**2 + x

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Get the corresponding y values from the function
y = [f(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='lightgrey', marker='o', markeredgecolor='green', markerfac

plt.show()
```



We know from the function that the ***f(x)*** values are calculated by squaring the ***x*** value and adding ***x***, so we can easily calculate points in between and show them - run the following code to see this:

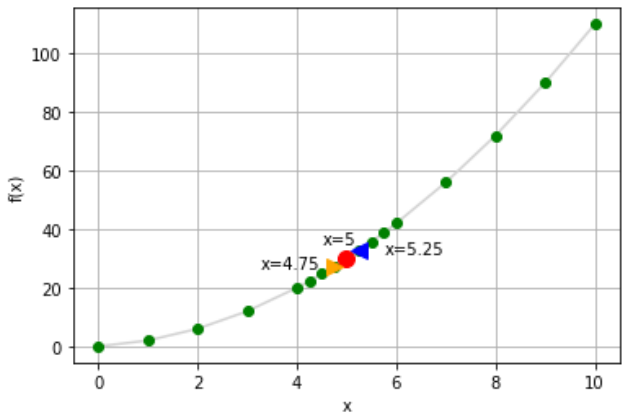
In [2]:
```
%matplotlib inline

# Here's the function
def f(x):
    return x**2 + x

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0,5))
x.append(4.25)
x.append(4.5)
x.append(4.75)
x.append(5)
x.append(5.25)
x.append(5.5)
x.append(5.75)
x = x + list(range(6,11))

# Get the corresponding y values from the function
y = [f(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='lightgrey', marker='o', markeredgecolor='green', markerfac

plt.show()
```



Now we can see more clearly that this function line is formed of a continuous series of points, so theoretically for any given value of ***x*** there is a point on the line, and there is an adjacent point on either side with a value that is as close to ***x*** as possible, but not actually ***x***.

Run the following code to visualize a specific point for *x* = *5*, and try to identify the closest point either side of it:

In [3]:
```
%matplotlib inline

# Here's the function
def f(x):
    return x**2 + x

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0,5))
x.append(4.25)
x.append(4.5)
x.append(4.75)
x.append(5)
x.append(5.25)
x.append(5.5)
x.append(5.75)
x = x + list(range(6,11))

# Get the corresponding y values from the function
y = [f(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='lightgrey', marker='o', markeredgecolor='green', markerfac

zx = 5
zy = f(zx)
plt.plot(zx, zy, color='red', marker='o', markersize=10)
plt.annotate('x=' + str(zx),(zx, zy), xytext=(zx - 0.5, zy + 5))

# Plot f(x) when x = 5.1
posx = 5.25
posy = f(posx)
plt.plot(posx, posy, color='blue', marker='<', markersize=10)
plt.annotate('x=' + str(posx),(posx, posy), xytext=(posx + 0.5, posy - 1))

# Plot f(x) when x = 4.9
negx = 4.75
negy = f(negx)
plt.plot(negx, negy, color='orange', marker='>', markersize=10)
plt.annotate('x=' + str(negx),(negx, negy), xytext=(negx - 1.5, negy - 1))

plt.show()
```



You can see the point where ***x*** is 5, and you can see that there are points shown on the graph that appear to be right next to this point (at *x=4.75* and *x=5.25*). However, if we zoomed in we'd see that there are still gaps that could be filled by other values of ***x*** that are even closer to 5; for example, 4.9 and 5.1, or 4.999 and 5.001. If we could zoom infinitely close to the line we'd see that no matter how close a value you use (for example, 4.999999999999), there is always a value that's fractionally closer (for example, 4.9999999999999).

So what we can say is that there is a hypothetical number that's as close as possible to our desired value of *x* without actually being *x*, but we can't express it as a real number. Instead, we express its symbolically as a *limit*, like this:

$$\lim_{x \to 5} f(x)$$

This is interpreted as *the limit of function f(x) as *x* approaches 5*.

Limits and Continuity

The function ***f(x)*** is *continuous* for all real numbered values of ***x***. Put simply, this means that you can draw the line created by the function without lifting your pen (we'll look at a more formal definition later in this course).

However, this isn't necessarily true of all functions. Consider function ***g(x)*** below:

$$g(x) = -(\frac{12}{2x})^2$$

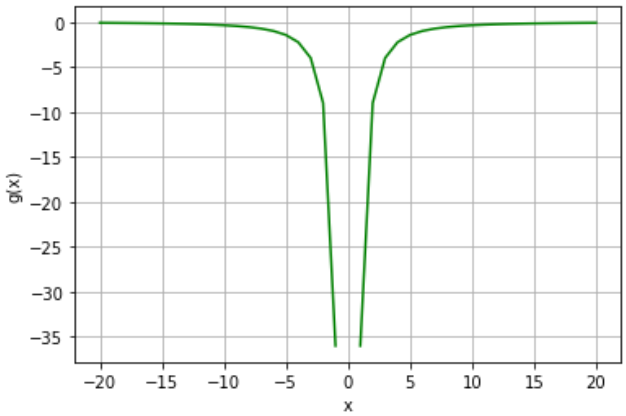
This function is a little more complex than the previous one, but the key thing to note is that it requires a division by *2x*. Now, ask yourself; what would happen if you applied this function to an *x* value of **0**?

Well, 2 • 2 is 0, and anything divided by 0 is *undefined*. So the *domain* of this function does not include 0; in other words, the function is defined when *x* is any real number such that *x is not equal to 0*. The function should therefore be written like this:

$$g(x) = -(\frac{12}{2x})^2, \quad x \neq 0$$

So why is this important? Let's investigate by running the following Python code to define the function and plot it for a set of arbitrary of values:

In [4]:
```
%matplotlib inline

# Define function g
def g(x):
    if x != 0:
        return -(12/(2*x))**2

# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-20, 21)

# Get the corresponding y values from the function
y = [g(a) for a in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('g(x)')
plt.grid()

# Plot x against g(x)
plt.plot(x,y, color='green')

plt.show()
```



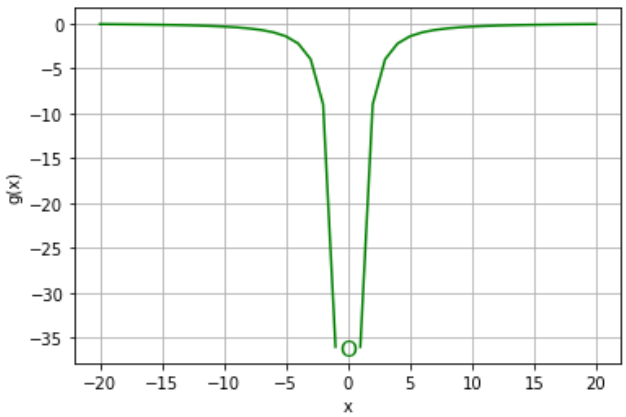
Look closely at the plot, and note the gap the line where *x* = 0. This indicates that the function is not defined here.The *domain* of the function (it's set of possible input values) not include 0, and it's *range* (the set of possible output values) does not include a value for x=0.

This is a *non-continuous* function - in other words, it includes at least one gap when plotted (so you couldn't plot it by hand without lifting your pen). Specifically, the function is non-continuous at x=0.

By convention, when a non-continuous function is plotted, the points that form a continuous line (or *interval*) are shown as a line, and the end of each line where there is a discontinuity is shown as a circle, which is filled if the value at that point is included in the line and empty if the value is not included in the line.

In this case, the function produces two intervals with a gap between them where the function is not defined, so we can show the discontinuous point as an unfilled circle - run the following code to visualize this with Python:

In [5]:
```
%matplotlib inline

# Define function g
def g(x):
    if x != 0:
        return -(12/(2*x))**2

# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-20, 21)


# Get the corresponding y values from the function
y = [g(a) for a in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('g(x)')
plt.grid()

# Plot x against g(x)
plt.plot(x,y, color='green')

# plot a circle at the gap (or close enough anyway!)
xy = (0,g(1))
plt.annotate('O',xy, xytext=(-0.7, -37),fontsize=14,color='green')

plt.show()
```



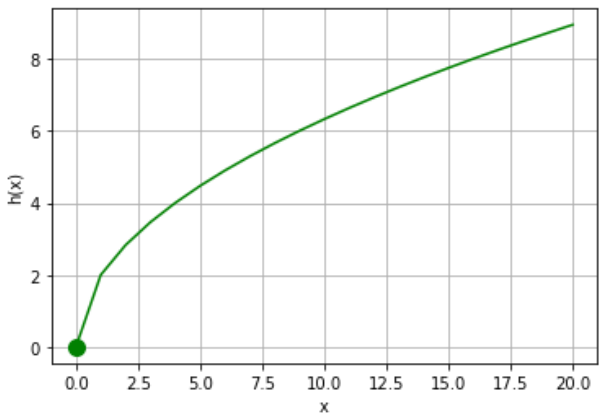
There are a number of reasons a function might be non-continuous. For example, consider the following function:

$$h(x) = 2\sqrt{x}, \quad x \geq 0$$

Applying this function to a non-negative ***x*** value returns a valid output; but for any value where ***x*** is negative, the output is undefined, because the square root of a negative value is not a real number.

Here's the Python to plot function ***h***:

In [6]:
```
%matplotlib inline

def h(x):
    if x >= 0:
        import numpy as np
        return 2 * np.sqrt(x)

# Plot output from function h
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-20, 21)

# Get the corresponding y values from the function
y = [h(a) for a in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('h(x)')
plt.grid()

# Plot x against h(x)
plt.plot(x,y, color='green')

# plot a circle close enough to the h(-x) limit for our purposes!
plt.plot(0, h(0), color='green', marker='o', markerfacecolor='green', markersiz

plt.show()
```



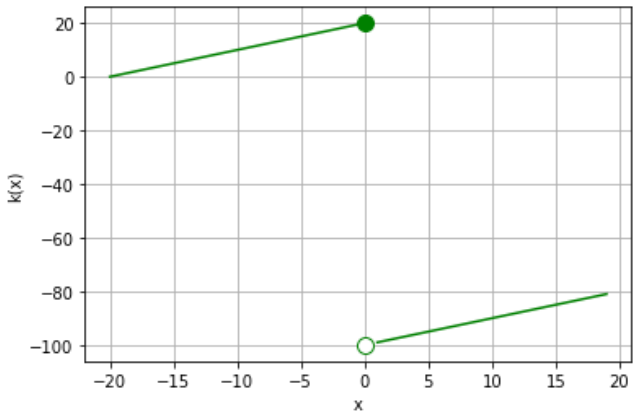
Now, suppose we have a function like this:

$$k(x) = \begin{cases} x + 20, & \text{if } x \leq 0, \\ x - 100, & \text{otherwise} \end{cases}$$

In this case, the function's domain includes all real numbers, but its output is still non-continuous because of the way different values are returned depending on the value of *x*. The *range* of possible outputs for *k(x ≤ 0)* is ≤ 20, and the range of output values for *k(x > 0)* is x > -100.

Let's use Python to plot function ***k***:

In [7]:
```
%matplotlib inline

def k(x):
    import numpy as np
    if x <= 0:
        return x + 20
    else:
        return x - 100

# Plot output from function h
from matplotlib import pyplot as plt

# Create an array of x values for each non-contonuous interval
x1 = range(-20, 1)
x2 = range(1, 20)

# Get the corresponding y values from the function
y1 = [k(i) for i in x1]
y2 = [k(i) for i in x2]

# Set up the graph
plt.xlabel('x')
plt.ylabel('k(x)')
plt.grid()

# Plot x against k(x)
plt.plot(x1,y1, color='green')
plt.plot(x2,y2, color='green')

# plot a circle at the interval ends
plt.plot(0, k(0), color='green', marker='o', markerfacecolor='green', markersiz
plt.plot(0, k(0.0001), color='green', marker='o', markerfacecolor='w', markersi

plt.show()
```



Finding Limits of Functions Graphically

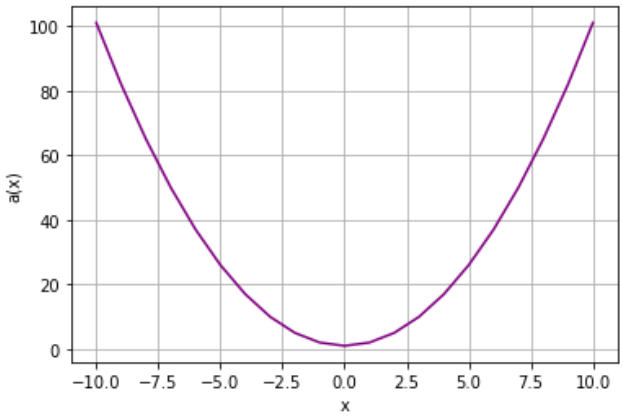
So the question arises, how do we find a value for the limit of a function at a specific point?

Let's explore this function, ***a***:

$$a(x) = x^2 + 1$$

We can start by plotting it:

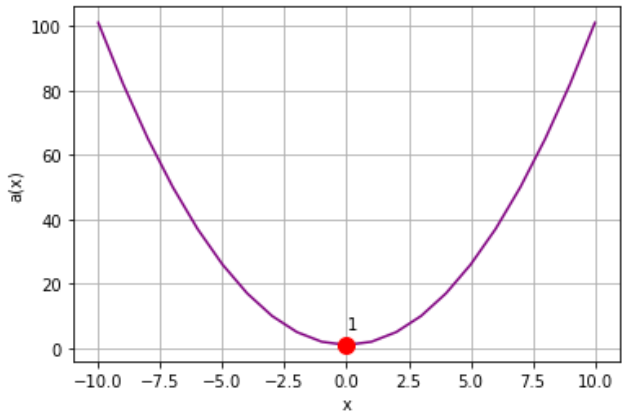
In [8]:
```
%matplotlib inline

# Define function a
def a(x):
    return x**2 + 1


# Plot output from function a
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [a(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('a(x)')
plt.grid()

# Plot x against a(x)
plt.plot(x,y, color='purple')

plt.show()
```



Note that this function is continuous at all points, there are no gaps in its range. However, the range of the function is *{a(x) ≥ 1}* (in other words, all real numbers that are greater than or equal to 1). For negative values of ***x***, the function appears to return ever-decreasing values as ***x*** gets closer to 0, and for positive values of ***x***, the function appears to return ever-increasing values as ***x*** gets further from 0; but it never returns 0.

Let's plot the function for an ***x*** value of 0 and find out what the ***a(0)*** value is returned:

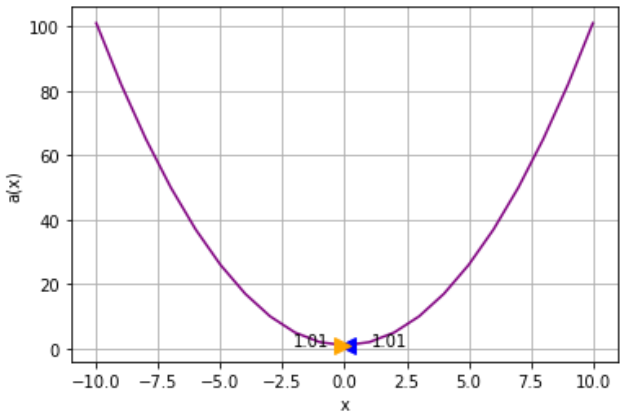
In [9]:
```
%matplotlib inline

# Define function a
def a(x):
    return x**2 + 1


# Plot output from function a
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [a(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('a(x)')
plt.grid()

# Plot x against a(x)
plt.plot(x,y, color='purple')

# Plot a(x) when x = 0
zx = 0
zy = a(zx)
plt.plot(zx, zy, color='red', marker='o', markersize=10)
plt.annotate(str(zy),(zx, zy), xytext=(zx, zy + 5))

plt.show()
```



OK, so ***a(0)*** returns **1**.

What happens if we use ***x*** values that are very slightly higher or lower than 0?

In [10]:
```
%matplotlib inline

# Define function a
def a(x):
    return x**2 + 1


# Plot output from function a
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [a(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('a(x)')
plt.grid()

# Plot x against a(x)
plt.plot(x,y, color='purple')

# Plot a(x) when x = 0.1
posx = 0.1
posy = a(posx)
plt.plot(posx, posy, color='blue', marker='<', markersize=10)
plt.annotate(str(posy),(posx, posy), xytext=(posx + 1, posy))

# Plot a(x) when x = -0.1
negx = -0.1
negy = a(negx)
plt.plot(negx, negy, color='orange', marker='>', markersize=10)
plt.annotate(str(negy),(negx, negy), xytext=(negx - 2, negy))

plt.show()
```



These ***x*** values return ***a(x)*** values that are just slightly above 1, and if we were to keep plotting numbers that are increasingly close to 0, for example 0.0000000001 or -0.0000000001, the function would still return a value that is just slightly greater than 1. The limit of function *a(x)* as *x* approaches 0, is 1; and the notation to indicate this is:

$$\lim_{x \to 0} a(x) = 1$$

This reflects a more formal definition of function continuity. Previously, we stated that a function is continuous at a point if you can draw it at that point without lifting your pen. The more mathematical definition is that a function is continuous at a point if the limit of the function as it approaches that point from both directions is equal to the function's value at that point. In this case, as we approach x = 0 from both sides, the limit is 1; and the value of *a(0)* is also 1; so the function is continuous at x = 0.

Limits at Non-Continuous Points

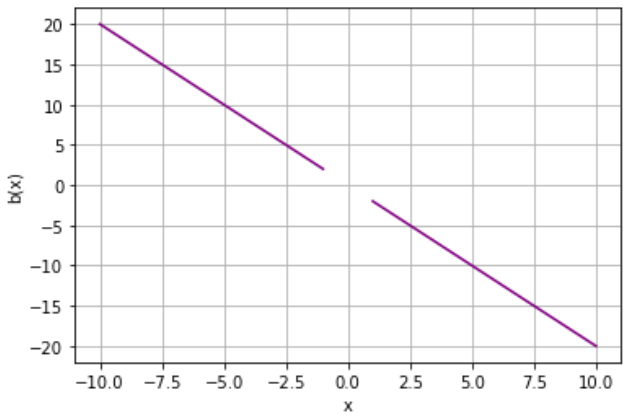
Let's try another function, which we'll call ***b***:

$$b(x) = -2x^2 \cdot \frac{1}{x}, \quad x \neq 0$$

Note that this function has a domain that includes all real number values of *x* such that *x* does not equal 0. In other words, the function will return a valid output for any number other than 0.

Let's create it and plot it with Python:

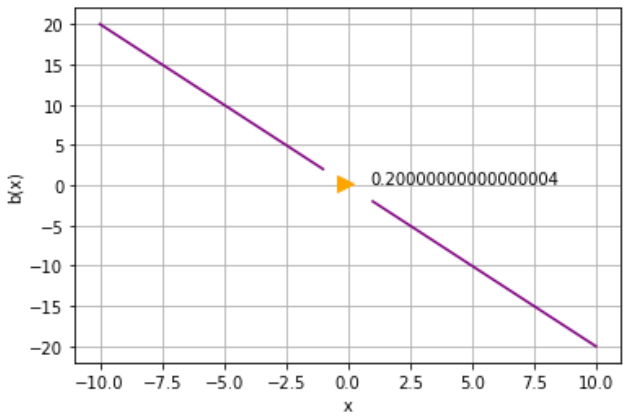
In [11]:
```
%matplotlib inline

# Define function b
def b(x):
    if x != 0:
        return (-2*x**2) * 1/x


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [b(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('b(x)')
plt.grid()

# Plot x against b(x)
plt.plot(x,y, color='purple')

plt.show()
```



The output from this function contains a gap in the line where x = 0. It seems that not only does the *domain* of the function (the values that can be passed in as *x*) exclude 0; but the *range* of the function (the set of values that can be returned from it) also excludes 0.

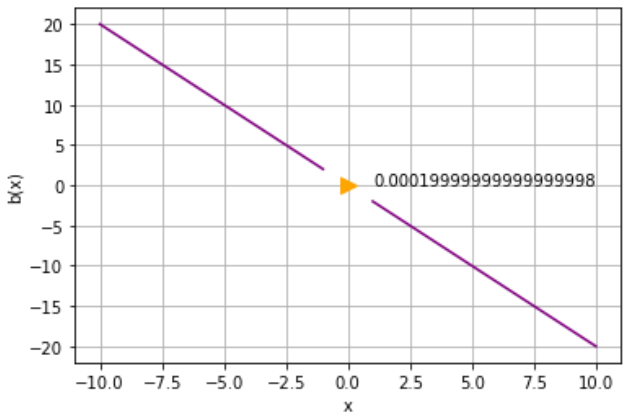
We can't evaluate the function for an *x* value of 0, but we can see what it returns for a value that is just very slightly less than 0:

In [12]:
```
%matplotlib inline

# Define function b
def b(x):
    if x != 0:
        return (-2*x**2) * 1/x


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [b(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('b(x)')
plt.grid()

# Plot x against b(x)
plt.plot(x,y, color='purple')

# Plot b(x) for x = -0.1
negx = -0.1
negy = b(negx)
plt.plot(negx, negy, color='orange', marker='>', markersize=10)
plt.annotate(str(negy),(negx, negy), xytext=(negx + 1, negy))

plt.show()
```



We can even try a negative *x* value that's a little closer to 0.

In [13]:
```
%matplotlib inline

# Define function b
def b(x):
    if x != 0:
        return (-2*x**2) * 1/x


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [b(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('b(x)')
plt.grid()

# Plot x against b(x)
plt.plot(x,y, color='purple')

# Plot b(x) for x = -0.0001
negx = -0.0001
negy = b(negx)
plt.plot(negx, negy, color='orange', marker='>', markersize=10)
plt.annotate(str(negy),(negx, negy), xytext=(negx + 1, negy))

plt.show()
```



So as the value of *x* gets closer to 0 from the left (negative), the value of *b(x)* is decreasing towards 0. We can show this with the following notation:

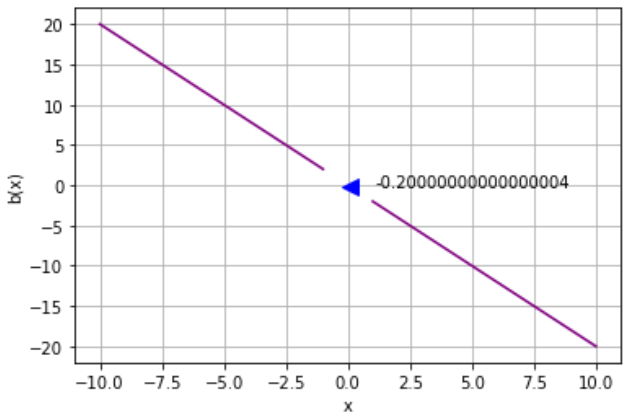
$$\lim_{x \to 0^-} b(x) = 0$$

Note that the arrow points to 0^- (with a minus sign) to indicate that we're describing the limit as we approach 0 from the negative side.

So what about the positive side?

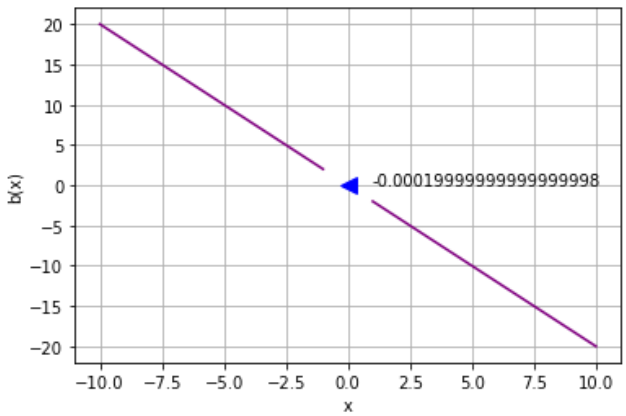
Let's see what the function value is when *x* is 0.1:

In [14]:
```
%matplotlib inline

# Define function b
def b(x):
    if x != 0:
        return (-2*x**2) * 1/x


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [b(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('b(x)')
plt.grid()

# Plot x against b(x)
plt.plot(x,y, color='purple')

# Plot b(x) for x = 0.1
posx = 0.1
posy = b(posx)
plt.plot(posx, posy, color='blue', marker='<', markersize=10)
plt.annotate(str(posy),(posx, posy), xytext=(posx + 1, posy))

plt.show()
```



What happens if we decrease the value of *x* so that it's even closer to 0?

In [15]:
```
%matplotlib inline

# Define function b
def b(x):
    if x != 0:
        return (-2*x**2) * 1/x


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the function
y = [b(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('b(x)')
plt.grid()

# Plot x against b(x)
plt.plot(x,y, color='purple')

# Plot b(x) for x = 0.0001
posx = 0.0001
posy = b(posx)
plt.plot(posx, posy, color='blue', marker='<', markersize=10)
plt.annotate(str(posy),(posx, posy), xytext=(posx + 1, posy))

plt.show()
```



As with the negative side, as *x* approaches 0 from the positive side, the value of *b(x)* gets closer to 0; and we can show that like this:

$$\lim_{x \to 0^+} b(x) = 0$$

Now, even although the function is not defined at x = 0; since the limit as we approach x = 0 from the negative side is 0, and the limit when we approach x = 0 from the positive side is also 0; we can say that the overall, or *two-sided* limit for the function at x = 0 is 0:

$$\lim_{x \to 0} b(x) = 0$$

So can we therefore just ignore the gap and say that the function is *continuous* at x = 0? Well, recall that the formal definition for continuity is that to be continuous at a point, the function's limit as we approach the point in both directions must be equal to the function's value at that point. In this case, the two-sided limit as we approach x = 0 is 0, but *b(0)* is not defined; so the function is ***non-continuous*** at x = 0.

One-Sided Limits

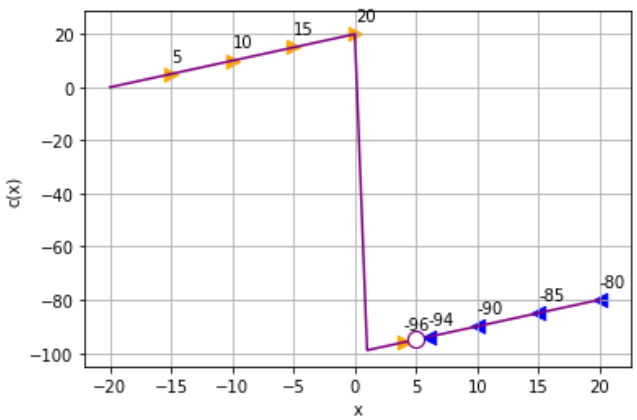
Let's take a look at a different function. We'll call this one ***c***:

$$c(x) = \begin{cases} x + 20, & \text{if } x \leq 0, \\ x - 100, & \text{otherwise} \end{cases}$$

In this case, the function's domain includes all real numbers, but its range is still non-continuous because of the way different values are returned depending on the value of *x*. The range of possible outputs for *c(x ≤ 0)* is ≤ 20, and the range of output values for *c(x > 0)* is x ≥ -100.

Let's use Python to plot function ***c*** with some values for *c(x)* marked on the line

In [16]:
```
%matplotlib inline

def c(x):
    import numpy as np
    if x <= 0:
        return x + 20
    else:
        return x - 100

# Plot output from function h
from matplotlib import pyplot as plt

# Create arrays of x values
x1 = range(-20, 6)
x2 = range(6, 21)

# Get the corresponding y values from the function
y1 = [c(i) for i in x1]
y2 = [c(i) for i in x2]

# Set up the graph
plt.xlabel('x')
plt.ylabel('c(x)')
plt.grid()

# Plot x against c(x)
plt.plot(x1,y1, color='purple')
plt.plot(x2,y2, color='purple')

# plot a circle close enough to the c limits for our purposes!
plt.plot(5, c(5), color='purple', marker='o', markerfacecolor='purple', markers
plt.plot(5, c(5.001), color='purple', marker='o', markerfacecolor='w', markersi

# plot some points from the +ve direction
posx = [20, 15, 10, 6]
posy = [c(i) for i in posx]
plt.scatter(posx, posy, color='blue', marker='<', s=70)
for p in posx:
    plt.annotate(str(c(p)),(p, c(p)),xytext=(p, c(p) + 5))

# plot some points from the -ve direction
negx = [-15, -10, -5, 0, 4]
negy = [c(i) for i in negx]
plt.scatter(negx, negy, color='orange', marker='>', s=70)
for n in negx:
    plt.annotate(str(c(n)),(n, c(n)),xytext=(n, c(n) + 5))

plt.show()
```



The plot of the function shows a line in which the *c(x)* value increases towards 25 as *x* approaches 5 from the negative side:

$$\lim_{x \to 5^-} c(x) = 25$$

However, the *c(x)* value decreases towards -95 as *x* approaches 5 from the positive side:

$$\lim_{x \to 5^+} c(x) = -95$$

So what can we say about the two-sided limit of this function at x = 5?

The limit as we approach x = 5 from the negative side is *not* equal to the limit as we approach x = 5 from the positive side, so no two-sided limit exists for this function at that point:

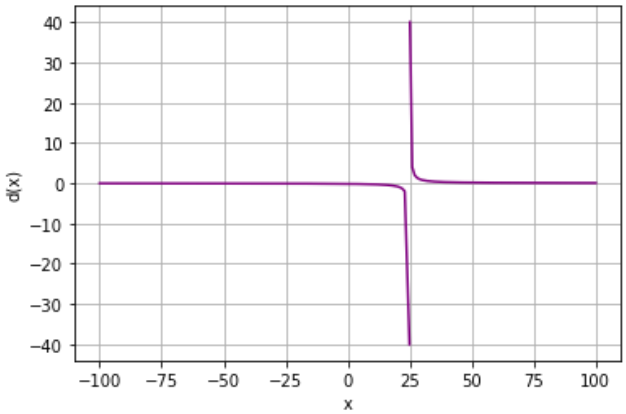
$$\lim_{x \to 5} \text{does not exist}$$

Asymptotes and Infinity

OK, time to look at another function:

$$d(x) = \frac{4}{x - 25}, \quad x \neq 25$$

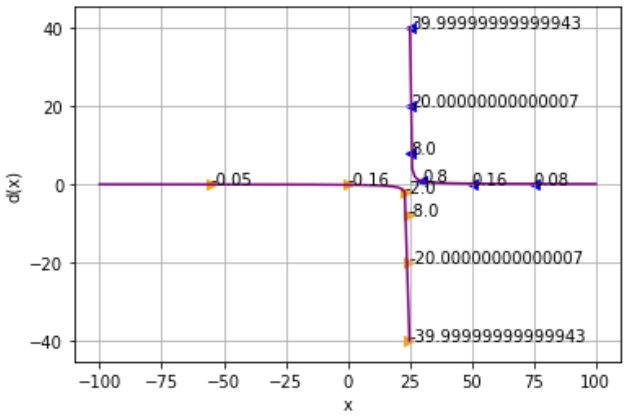
In [17]:
```
%matplotlib inline

# Define function d
def d(x):
    if x != 25:
        return 4 / (x - 25)


# Plot output from function d
from matplotlib import pyplot as plt

# Create an array of x values
x = list(range(-100, 24))
x.append(24.9) # Add some fractional x
x.append(25)   # values around
x.append(25.1) # 25 for finer-grain results
x = x + list(range(26, 101))
# Get the corresponding y values from the function
y = [d(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('d(x)')
plt.grid()

# Plot x against d(x)
plt.plot(x,y, color='purple')

plt.show()
```



What's the limit of *d* as *x* approaches 25?

We can plot a few points to help us:

In [18]:
```
%matplotlib inline

# Define function d
def d(x):
    if x != 25:
        return 4 / (x - 25)


# Plot output from function d
from matplotlib import pyplot as plt

# Create an array of x values
x = list(range(-100, 24))
x.append(24.9) # Add some fractional x
x.append(25)   # values around
x.append(25.1) # 25 for finer-grain results
x = x + list(range(26, 101))
# Get the corresponding y values from the function
y = [d(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('d(x)')
plt.grid()

# Plot x against d(x)
plt.plot(x,y, color='purple')

# plot some points from the +ve direction
posx = [75, 50, 30, 25.5, 25.2, 25.1]
posy = [d(i) for i in posx]
plt.scatter(posx, posy, color='blue', marker='<')
for p in posx:
    plt.annotate(str(d(p)),(p, d(p)))

# plot some points from the -ve direction
negx = [-55, 0, 23, 24.5, 24.8, 24.9]
negy = [d(i) for i in negx]
plt.scatter(negx, negy, color='orange', marker='>')
for n in negx:
    plt.annotate(str(d(n)),(n, d(n)))

plt.show()
```



From these plotted values, we can see that as *x* approaches 25 from the negative side, *d(x)* is decreasing, and as *x* approaches 25 from the positive side, *d(x)* is increasing. As *x* gets closer to 25, *d(x)* increases or decreases more significantly.

If we were to plot every fractional value of *d(x)* for *x* values between 24.9 and 25, we'd see a line that decreases indefintely, getting closer and closer to the x = 25 vertical line, but never actually reaching it. Similarly, plotting every *x* value between 25 and 25.1 would result in a line going up indefinitely, but always staying to the right of the vertical x = 25 line.

The x = 25 line in this case is an *asymptote* - a line to which a curve moves ever closer but never actually reaches. The positive limit for x = 25 in this case in not a real numbered value, but *infinity*:

$$\lim_{x \to 25^+} d(x) = \infty$$

Conversely, the negative limit for x = 25 is negative infinity:

$$\lim_{x \to 25^-} d(x) = -\infty$$

Finding Limits Numerically Using a Table

Up to now, we've estimated limits for a point graphically by examining a graph of a function. You can also approximate limits by creating a table of x values and the corresponding function values either side of the point for which you want to find the limits.

For example, let's return to our ***a*** function:

$$a(x) = x^2 + 1$$

If we want to find the limits as x is approaching 0, we can apply the function to some values either side of 0 and view them as a table. Here's some Python code to do that:

In [19]:
```
# Define function a
def a(x):
    return x**2 + 1


import pandas as pd

# Create a dataframe with an x column containing values either side of 0
df = pd.DataFrame ({'x': [-1, -0.5, -0.2, -0.1, -0.01, 0, 0.01, 0.1, 0.2, 0.5,

# Add an a(x) column by applying the function to x
df['a(x)'] = a(df['x'])

#Display the dataframe
df
```

Out[19]:

	x	a(x)
0	-1.00	2.0000
1	-0.50	1.2500
2	-0.20	1.0400
3	-0.10	1.0100
4	-0.01	1.0001
5	0.00	1.0000
6	0.01	1.0001
7	0.10	1.0100
8	0.20	1.0400
9	0.50	1.2500
10	1.00	2.0000

Looking at the output, you can see that the function values are getting closer to 1 as x approaches 0 from both sides, so:

$$\lim_{x \to 0} a(x) = 1$$

Additionally, you can see that the actual value of the function when x = 0 is also 1, so:

$$\lim_{x \to 0} a(x) = a(0)$$

Which according to our earlier definition, means that the function is continuous at 0.

However, you should be careful not to assume that the limit when x is approaching 0 will always be the same as the value when x = 0; even when the function is defined for x = 0.

For example, consider the following function:

$$e(x) = \begin{cases} 5, & \text{if } x = 0, \\ 1 + x^2, & \text{otherwise} \end{cases}$$

Let's see what the function returns for *x* values either side of 0 in a table:

In [20]:
```
# Define function e
def e(x):
    if x == 0:
        return 5
    else:
        return 1 + x**2

import pandas as pd
# Create a dataframe with an x column containing values either side of 0
x= [-1, -0.5, -0.2, -0.1, -0.01, 0, 0.01, 0.1, 0.2, 0.5, 1]
y =[e(i) for i in x]
df = pd.DataFrame ({' x':x, 'e(x)': y })
df
```

Out[20]:

	x	e(x)
0	-1.00	2.0000
1	-0.50	1.2500
2	-0.20	1.0400
3	-0.10	1.0100
4	-0.01	1.0001
5	0.00	5.0000
6	0.01	1.0001
7	0.10	1.0100
8	0.20	1.0400
9	0.50	1.2500
10	1.00	2.0000

As before, you can see that as the *x* values approach 0 from both sides, the value of the function gets closer to 1, so:

$$\lim_{x \to 0} e(x) = 1$$

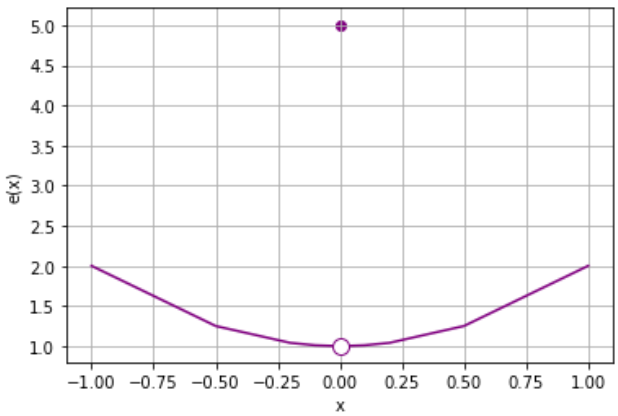
However the actual value of the function when x = 0 is 5, not 1; so:

$$\lim_{x \to 0} e(x) \neq e(0)$$

Which according to our earlier definition, means that the function is non-continuous at 0.

Run the following cell to see what this looks like as a graph:

In [21]:
```
%matplotlib inline

# Define function e
def e(x):
    if x == 0:
        return 5
    else:
        return 1 + x**2

from matplotlib import pyplot as plt

x= [-1, -0.5, -0.2, -0.1, -0.01, 0.01, 0.1, 0.2, 0.5, 1]
y =[e(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('e(x)')
plt.grid()

# Plot x against e(x)
plt.plot(x, y, color='purple')
# (we're cheating slightly - we'll manually plot the discontinous point...)
plt.scatter(0, e(0), color='purple')
# (... and overplot the gap)
plt.plot(0, 1, color='purple', marker='o', markerfacecolor='w', markersize=10)
plt.show()
```



Determining Limits Analytically

We've seen how to estimate limits visually on a graph, and by creating a table of *x* and *f(x)* values either side of a point. There are also some mathematical techniques we can use to calculate limits.

Direct Substitution

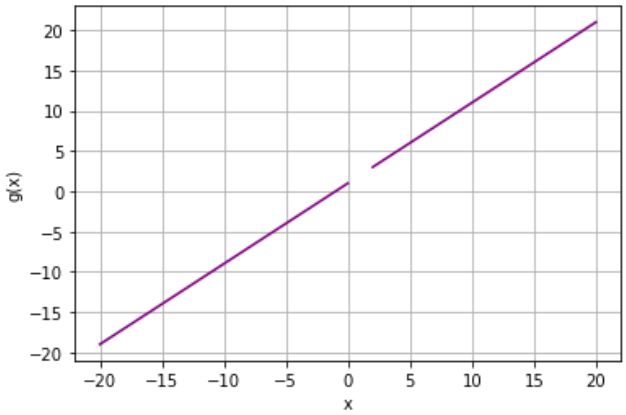
Recall that our definition for a function to be continuous at a point is that the two-directional limit must exist and that it must be equal to the function value at that point. It therefore follows, that if we know that a function is continuous at a given point, we can determine the limit simply by evaluating the function for that point.

For example, let's consider the following function ***g***:

$$g(x) = \frac{x^2 - 1}{x - 1}, x \neq 1$$

Run the following code to see this function as a graph:

In [22]:
```
%matplotlib inline

# Define function f
def g(x):
    if x != 1:
        return (x**2 - 1) / (x - 1)


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x= range(-20, 21)
y =[g(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('g(x)')
plt.grid()

# Plot x against g(x)
plt.plot(x,y, color='purple')

plt.show()
```



Now, suppose we need to find the limit of ***g(x)*** as ***x*** approaches **4**. We can try to find this by simply substituting 4 for the *x* values in the function:

$$g(4) = \frac{4^2 - 1}{4 - 1}$$

This simplifies to:

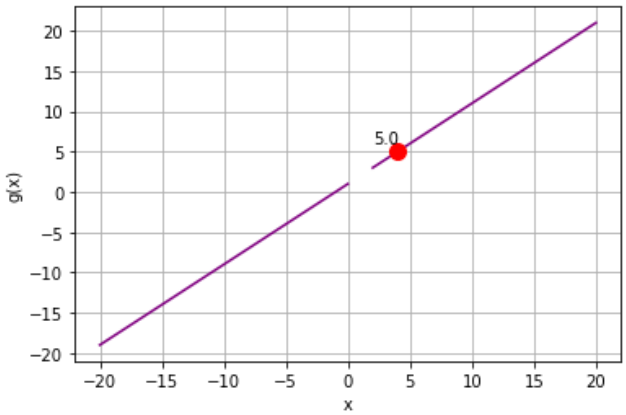
$$g(4) = \frac{15}{3}$$

So:

$$\lim_{x \to 4} g(x) = 5$$

Let's take a look:

In [23]:
```
%matplotlib inline

# Define function g
def g(x):
    if x != 1:
        return (x**2 - 1) / (x - 1)


# Plot output from function f
from matplotlib import pyplot as plt

# Create an array of x values
x= range(-20, 21)
y =[g(i) for i in x]

# Set the x point we're interested in
zx = 4

plt.xlabel('x')
plt.ylabel('g(x)')
plt.grid()

# Plot x against g(x)
plt.plot(x,y, color='purple')

# Plot g(x) when x = 0
zy = g(zx)
plt.plot(zx, zy, color='red', marker='o', markersize=10)
plt.annotate(str(zy),(zx, zy), xytext=(zx - 2, zy + 1))

plt.show()

print ('Limit as x -> ' + str(zx) + ' = ' + str(zy))
```




```
Limit as x -> 4 = 5.0
```

Factorization

OK, now let's try to find the limit of ***g(x)*** as ***x*** approaches **1**.

We know from the function definition that the function is not defined at x = 1, but we're not trying to find the *value* of ***g(x)*** when x *equals* 1; we're trying to find the *limit* of ***g(x)*** as x *approaches* 1.

The direct substitution approach won't work in this case:

$$g(1) = \frac{1^2 - 1}{1 - 1}$$

Simplifies to:

$$g(1) = \frac{0}{0}$$

Anything divided by 0 is undefined; so all we've done is to confirm that the function is not defined at this point. You might be tempted to assume that this means the limit does not exist, but $^0/_0$ is a special case; it's what's known as the *indeterminate form*; and there may be a way to solve this problem another way.

We can factor the $x^2 - 1$ numerator in the definition of ***g*** as as $(x - 1)(x + 1)$, so the limit equation can we rewritten like this:

$$\lim_{x \to a} g(x) = \frac{(x - 1)(x + 1)}{x - 1}$$

The ***x - 1*** in the numerator and the ***x - 1*** in the denominator cancel each other out:

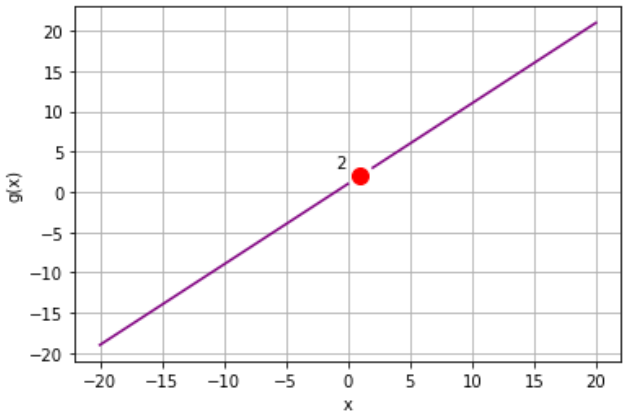
$$\lim_{x \to a} g(x) = x + 1$$

So we can now use substitution for $x = 1$ to calculate the limit as $1 + 1$:

$$\lim_{x \to 1} g(x) = 2$$

Let's see what that looks like:

In [24]:
```
%matplotlib inline

# Define function g
def f(x):
    if x != 1:
        return (x**2 - 1) / (x - 1)


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x= range(-20, 21)
y =[g(i) for i in x]

# Set the x point we're interested in
zx = 1

# Calculate the limit of g(x) when x->zx using the factored equation
zy = zx + 1

plt.xlabel('x')
plt.ylabel('g(x)')
plt.grid()

# Plot x against g(x)
plt.plot(x,y, color='purple')

# Plot the limit of g(x)
zy = zx + 1
plt.plot(zx, zy, color='red', marker='o', markersize=10)
plt.annotate(str(zy),(zx, zy), xytext=(zx - 2, zy + 1))

plt.show()

print ('Limit as x -> ' + str(zx) + ' = ' + str(zy))
```




```
Limit as x -> 1 = 2
```

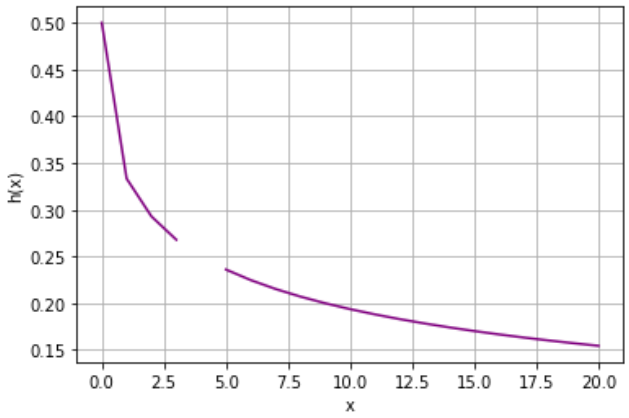
Rationalization

Let's look at another function:

$$h(x) = \frac{\sqrt{x} - 2}{x - 4}, x \neq 4 \text{ and } x \geq 0$$

Run the following cell to plot this function as a graph:

In [25]:
```
%matplotlib inline

# Define function h
def h(x):
    import math
    if x >= 0 and x != 4:
        return (math.sqrt(x) - 2) / (x - 4)


# Plot output from function h
from matplotlib import pyplot as plt

# Create an array of x values
x= range(-20, 21)
y =[h(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('h(x)')
plt.grid()

# Plot x against h(x)
plt.plot(x,y, color='purple')

plt.show()
```



To find the limit of ***h(x)*** as ***x*** approaches **4**, we can't use the direct substitution method because the function is not defined at that point. However, we can take an alternative approach by multiplying both the numerator and denominator in the function by the *conjugate* of the numerator to *rationalize* the square root term (a conjugate is a binomial formed by reversing the sign of the second term of a binomial):

$$\lim_{x \to a} h(x) = \frac{\sqrt{x} - 2}{x - 4} \cdot \frac{\sqrt{x} + 2}{\sqrt{x} + 2}$$

This simplifies to:

$$\lim_{x \to a} h(x) = \frac{(\sqrt{x})^2 - 2^2}{(x - 4)(\sqrt{x} + 2)}$$

The $\sqrt{x^2}$ is x, and 2^2 is 4, so we can simplify the numerator as follows:

$$\lim_{x \to a} h(x) = \frac{x - 4}{(x - 4)(\sqrt{x} + 2)}$$

Now we can cancel out the *x - 4* in both the numerator and denominator:

$$\lim_{x \to a} h(x) = \frac{1}{\sqrt{x} + 2}$$

So for x approaching 4, this is:

$$\lim_{x \to 4} h(x) = \frac{1}{\sqrt{4} + 2}$$

This simplifies to:

$$\lim_{x \to 4} h(x) = \frac{1}{2 + 2}$$

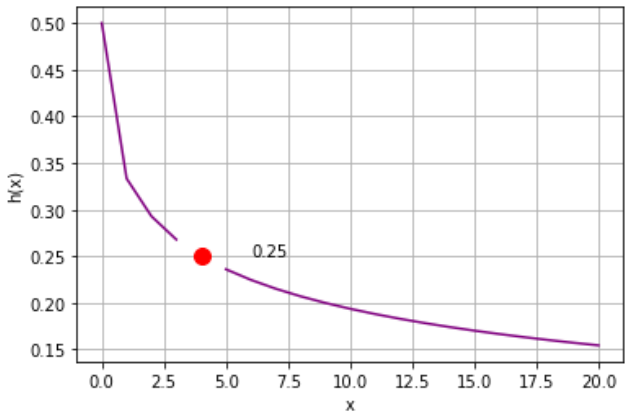
Which is of course:

$$\lim_{x \to 4} h(x) = \frac{1}{4}$$

So the limit of ***h(x)*** as ***x*** approaches **4** is $^1/_4$ or 0.25.

Let's calculate and plot this with Python:

In [26]:
```
%matplotlib inline

# Define function h
def h(x):
    import math
    if x >= 0 and x != 4:
        return (math.sqrt(x) - 2) / (x - 4)


# Plot output from function h
from matplotlib import pyplot as plt

# Create an array of x values
x= range(-20, 21)
y =[h(i) for i in x]

# Specify the point we're interested in
zx = 4

# Calculate the limit of f(x) when x->zx using factored equation
import math
zy = 1 / ((math.sqrt(zx)) + 2)

plt.xlabel('x')
plt.ylabel('h(x)')
plt.grid()

# Plot x against h(x)
plt.plot(x,y, color='purple')

# Plot the limit of h(x) when x->zx
plt.plot(zx, zy, color='red', marker='o', markersize=10)
plt.annotate(str(zy),(zx, zy), xytext=(zx + 2, zy))

plt.show()

print ('Limit as x -> ' + str(zx) + ' = ' + str(zy))
```




```
Limit as x -> 4 = 0.25
```

Rules for Limit Operations

When you are working with functions and limits, you may want to combine limits using arithmetic operations. There are some intuitive rules for doing this.

Let's define two simple functions, ***j***:

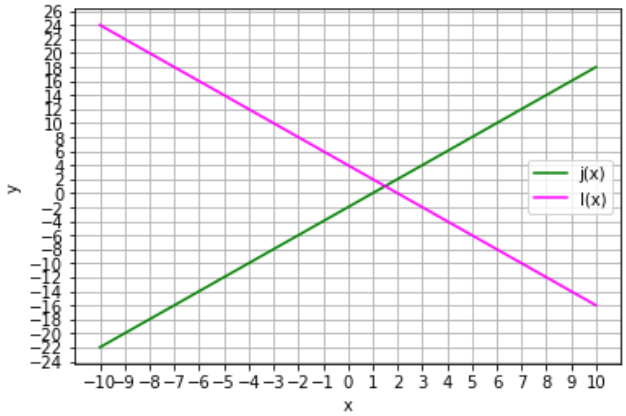
$$j(x) = 2x - 2$$

and ***l***:

$$l(x) = -2x + 4$$

Run the cell below to plot these functions:

In [27]:
```
%matplotlib inline

# Define function j
def j(x):
    return x * 2 - 2

# Define function l
def l(x):
    return -x * 2 + 4


# Plot output from functions j and l
from matplotlib import pyplot as plt

# Create an array of x values
x = range(-10, 11)

# Get the corresponding y values from the functions
jy = [j(i) for i in x]
ly = [l(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.xticks(range(-10,11, 1))
plt.ylabel('y')
plt.yticks(range(-30,30, 2))
plt.grid()

# Plot x against j(x)
plt.plot(x,jy, color='green', label='j(x)')

# Plot x against l(x)
plt.plot(x,ly, color='magenta', label='l(x)')

plt.legend()

plt.show()
```



Addition of Limits

First, let's look at the rule for addition:

$$\lim_{x \to a}(j(x) + l(x)) = \lim_{x \to a} j(x) + \lim_{x \to a} l(x)$$

What we're saying here, is that the limit of *j(x) + l(x)* as *x* approaches *a*, is the same as the limit of *j(x)* as *x* approaches *a* added to the limit of *l(x)* as *x* approaches *a*.

Looking at the graph for our functions ***j*** and ***l***, let's apply this rule to an *a* value of **8**.

By visually inspecting the graph, you can see that as *x* approaches 8 from either direction, *j(x)* gets closer to 14, so:

$$\lim_{x \to 8} j(x) = 14$$

Similarly, as *x* approaches 8 from either direction, *l(x)* gets closer to -12, so:

$$\lim_{x \to 8} l(x) = -12$$

So based on the addition rule:

$$\lim_{x \to 8}(j(x) + l(x)) = 14 + -12 = 2$$

Subtraction of Limits

Here's the rule for subtraction:

$$\lim_{x \to a}(j(x) - l(x)) = \lim_{x \to a} j(x) - \lim_{x \to a} l(x)$$

As you've probably noticed, this is consistent with the rule of addition. Based on an *a* value of 8 (and the limits we identified for this *a* value above), we can apply this rule like this:

$$\lim_{x \to 8}(j(x) - l(x)) = 14 - -12 = 26$$

Multiplication of Limits

Here's the rule for multiplication:

$$\lim_{x \to a}(j(x) \cdot l(x)) = \lim_{x \to a} j(x) \cdot \lim_{x \to a} l(x)$$

Again, you can apply this to the limits as x approached an *a* value of 8 we identified previously:

$$\lim_{x \to 8}(j(x) \cdot l(x)) = 14 \cdot -12 = -168$$

This rule also applies to multipying a limit by a constant:

$$\lim_{x \to a} c \cdot l(x) = c \cdot \lim_{x \to a} l(x)$$

So for an *a* value of 8 and a constant *c* value of 3, this equates to:

$$\lim_{x \to 8} 3 \cdot l(x) = 3 \cdot -12 = -36$$

Division of Limits

For division, assuming the limit of *l(x)* when x is approaching *a* is not 0:

$$\lim \frac{j(x)}{} = \frac{\lim_{x \to a} j(x)}{}$$

Differentiation and Derivatives

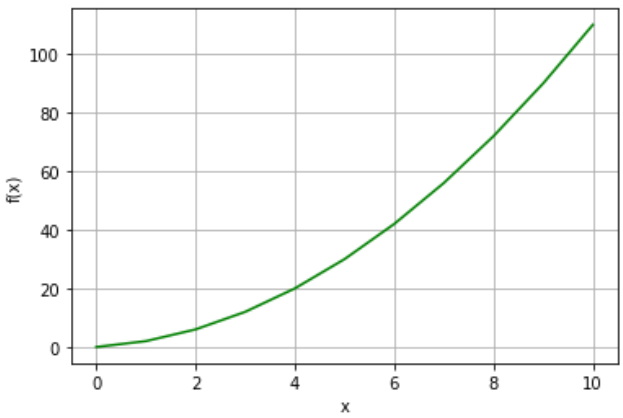
So far in this course, you've learned how to evaluate limits for points on a line. Now you're going to build on that knowledge and look at a calculus technique called *differentiation*. In differentiation, we use our knowledge of limits to calculate the *derivative* of a function in order to determine the rate of change at an individual point on a line.

Let's remind ourselves of the problem we're trying to solve, here's a function:

$$f(x) = x^2 + x$$

We can visualize part of the line that this function defines using the folllowing Python code:

In [1]:
```
%matplotlib inline

# Here's the function
def f(x):
    return x**2 + x

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [f(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

plt.show()
```



Now, we know that we can calculate the average rate of change for a given interval on the line by calculating the slope for a secant line that connects two points on the line. For example, we can calculate the average change for the interval between x=4 and x=6 by dividing the change (or *delta*, indicated as Δ) in the value of *f(x)* by the change in the value of *x*:

$$m = \frac{\Delta f(x)}{\Delta x}$$

The delta for *f(x)* is calculated by subtracting the *f(x)* values of our points, and the delta for *x* is calculated by subtracting the *x* values of our points; like this:

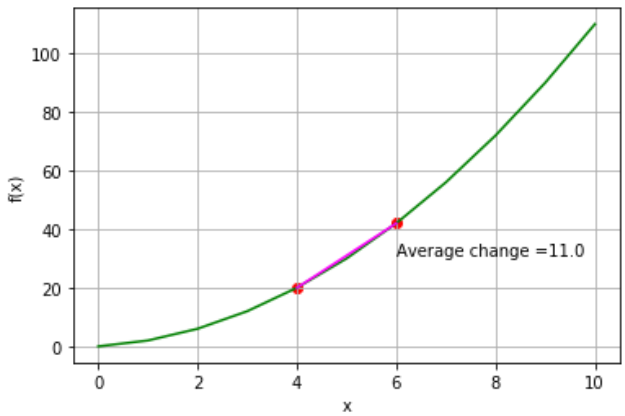
$$m = \frac{f(x)_2 - f(x)_1}{x_2 - x_1}$$

So for the interval between x=4 and x=6, that's:

$$m = \frac{f(6) - f(4)}{6 - 4}$$

We can calculate and plot this using the following Python:

In [2]:
```
%matplotlib inline

def f(x):
    return x**2 + x

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [f(i) for i in x]

# Set the a values
x1 = 4
x2 = 6

# Get the corresponding f(x) values
y1 = f(x1)
y2 = f(x2)

# Calculate the slope by dividing the deltas
a = (y2 - y1)/(x2 - x1)

# Create an array of x values for the secant line
sx = [x1,x2]

# Use the function to get the y values
sy = [f(i) for i in sx]

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the interval points
plt.scatter([x1,x2],[y1,y2], c='red')

# Plot the secant line
plt.plot(sx,sy, color='magenta')

# Display the calculated average rate of change
plt.annotate('Average change =' + str(a),(x2, (y2+y1)/2))

plt.show()
```

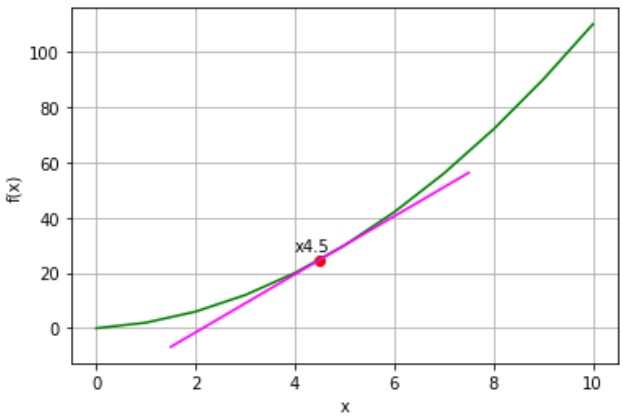


The average rate of change for the interval between x=4 and x=6 is $^{11}/_{1}$ (or simply 11), meaning that for every **1** added to *x*, *f(x)* increases by **11**. Put another way, if x represents time in seconds and f(x) represents distance in meters, the average rate of change for distance over time (in other words, *velocity*) for the 4 to 6 second interval is 11 meters-per-second.

So far, this is just basic algebra; but what if instead of the average rate of change over an interval, we want to calculate the rate of change at a single point, say, where x = 4.5?

One approach we could take is to create a secant line between the point at which we want the slope and another point on the function line that is infintesimally close to it. So close in fact that the secant line is actually a tangent that goes through both points. We can then calculate the slope for the secant line as before. This would look something like the graph produced by the following code:

In [3]:
```
%matplotlib inline

def f(x):
    return x**2 + x

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [f(i) for i in x]

# Set the x1 point, arbitrarily 5
x1 = 4.5
y1 = f(x1)

# Set the x2 point, very close to x1
x2 = 5.000000001
y2 = f(x2)

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the point
plt.scatter(x1,y1, c='red')
plt.annotate('x' + str(x1),(x1,y1), xytext=(x1-0.5, y1+3))

# Approximate the tangent slope and plot it
m = (y2-y1)/(x2-x1)
xMin = x1 - 3
yMin = y1 - (3*m)
xMax = x1 + 3
yMax = y1 + (3*m)
plt.plot([xMin,xMax],[yMin,yMax], color='magenta')

plt.show()
```



Calculating a Derivative

In the Python code above, we created the (almost) tangential secant line by specifying a point that is very close to the point at which we want to calculate the rate of change. This is adequate to show the line conceptually in the graph, but it's not a particularly generalizable (or accurate) way to actually calculate the line so that we can get the rate of change at any given point.

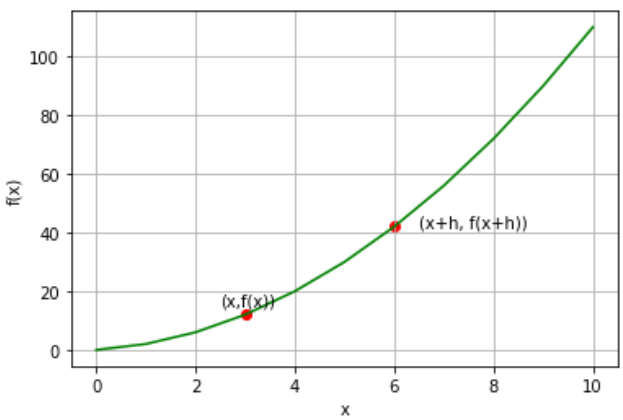
If only we knew of a way to calculate a point on the line that is as close as possible to point with a given *x* value.

Oh wait, we do! It's a *limit*.

So how do we apply a limit in this scenario? Well, let's start by examining our general approach to calculating slope in a little more detail.Our tried and tested approach is to plot a secant line between two points at different values of x, so let's plot an arbitrary (*x,y*) point, and then add an arbitrary amount to *x*, which we'll call *h*. Then we know that we can plot a secant line between (*x,f(x)*) and (*x+h,f(x+h)*) and find its slope.

Run the cell below to see these points:

In [4]:
```
%matplotlib inline

def f(x):
    return x**2 + x


from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [f(i) for i in x]

# Set the x point
x1 = 3
y1 = f(x1)

# set the increment
h = 3

# set the x+h point
x2 = x1+h
y2 = f(x2)

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the x point
plt.scatter(x1,y1, c='red')
plt.annotate('(x,f(x))',(x1,y1), xytext=(x1-0.5, y1+3))

# Plot the x+h point
plt.scatter(x2,y2, c='red')
plt.annotate('(x+h, f(x+h))',(x2,y2), xytext=(x2+0.5, y2))

plt.show()
```



As we saw previously, our formula to calculate slope is:

$$m = \frac{\Delta f(x)}{\Delta x}$$

The delta for *f(x)* is calculated by subtracting the *f(x + h)* and *f(x)* values of our points, and the delta for *x* is just the difference between *x* and *x + h*; in other words, *h*:

$$m = \frac{f(x+h) - f(x)}{h}$$

What we actually need is the slope at the shortest possible distance between x and x+h, so we're looking for the smallest possible value of *h*. In other words, we need the limit as *h* approaches 0.

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

This equation is generalizable, and we can use it as the definition of a function to help us find the slope at any given value of *x* on the line, and it's what we call the *derivative* of our original function (which in this case is called *f*). This is generally indicated in *Lagrange* notation like this:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

You'll also sometimes see derivatives written in *Leibniz's* notation like this:

$$\frac{d}{dx} f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Note: *Some textbooks use* ***h*** *to symbolize the difference between* $\boldsymbol{x_0}$ *and* $\boldsymbol{x_1}$*, while others use* $\boldsymbol{\Delta x}$*. It makes no diffrerence which symbolic value you use.*

Alternate Form for a Derivative

The formula above shows the generalized form for a derivative. You can use the derivative function to get the slope at any given point, for example to get the slope at point *a* you could just plug the value for *a* into the generalized derivative function:

$$f'(\mathbf{a}) = \lim_{h \to 0} \frac{f(\mathbf{a}+h) - f(\mathbf{a})}{h}$$

Or you could use the alternate form, which is specific to point *a*:

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}$$

These are mathematically equivalent.

Finding the Derivative for a Specific Point

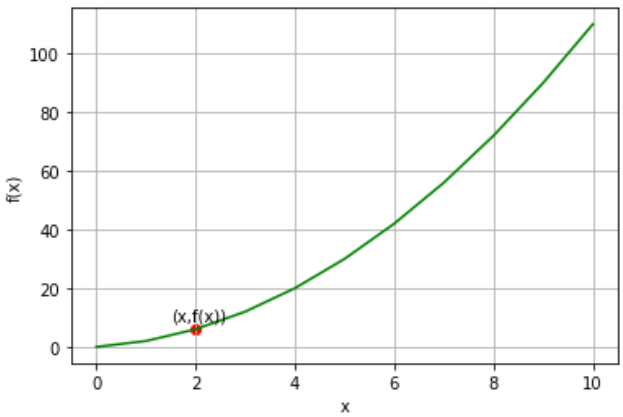
It's easier to understand differentiation by seeing it in action, so let's use it to find the derivitive for a specific point in the function ***f***.

Here's the definition of function ***f***:

$$f(x) = x^2 + x$$

Let's say we want to find ***f'(2)*** (the derivative for ***f*** when ***x*** is 2); so we're trying to find the slope at the point shown by the following code:

In [5]:
```
%matplotlib inline

def f(x):
    return x**2 + x

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [f(i) for i in x]

# Set the point
x1 = 2
y1 = f(x1)

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the point
plt.scatter(x1,y1, c='red')
plt.annotate('(x,f(x))',(x1,y1), xytext=(x1-0.5, y1+3))

plt.show()
```



Here's our generalized formula for finding a derivative at a specific point (*a*):

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

So let's just start by plugging our *a* value in:

$$f'(\mathbf{2}) = \lim_{h \to 0} \frac{f(\mathbf{2}+h) - f(\mathbf{2})}{h}$$

We know that ***f(x)*** encapsulates the equation ***x² + x***, so we can rewrite our derivative equation as:

$$f'(2) = \lim_{h \to 0} \frac{((2+h)^2 + 2 + h) - (2^2 + 2)}{h}$$

We can apply the distribution property to $(2+h)^2$ using the rule that $(a+b)^2 = a^2 + b^2 + 2ab$:

$$f'(2) = \lim_{h \to 0} \frac{(4 + h^2 + 4h + 2 + h) - (2^2 + 2)}{h}$$

Then we can simplify $2^2 + 2$ (2^2 is 4, plus 2 gives is 6):

$$f'(2) = \lim_{h \to 0} \frac{(4 + h^2 + 4h + 2 + h) - 6}{h}$$

We can combine like terms on the left side of the numerator to make things a little clearer:

$$f'(2) = \lim_{h \to 0} \frac{(h^2 + 5h + 6) - 6}{h}$$

Which combines even further to get rid of the *6*:

$$f'(2) = \lim_{h \to 0} \frac{h^2 + 5h}{h}$$

And finally, we can simplify the fraction:

$$f'(2) = \lim_{h \to 0} h + 5$$

To get the limit when *h* is approaching 0, we can use direct substitution for h:

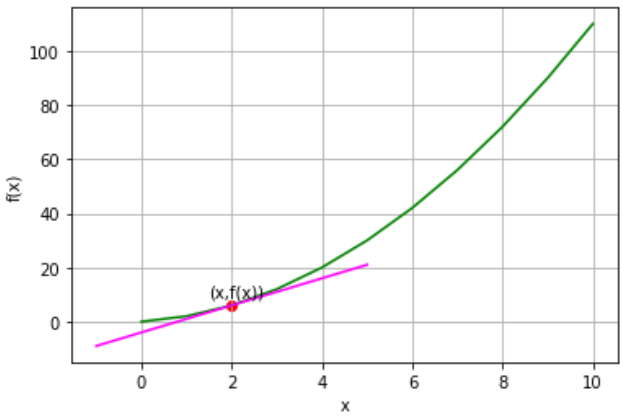
$$f'(2) = 0 + 5$$

so:

$$f'(2) = 5$$

Let's draw a tangent line with that slope on our graph to see if it looks right:

In [6]:
```
%matplotlib inline

def f(x):
    return x**2 + x


from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [f(i) for i in x]

# Set the point
x1 = 2
y1 = f(x1)

# Specify the derivative we calculated above
m = 5

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the point
plt.scatter(x1,y1, c='red')
plt.annotate('(x,f(x))',(x1,y1), xytext=(x1-0.5, y1+3))

# Plot the tangent line using the derivative we calculated
xMin = x1 - 3
yMin = y1 - (3*m)
xMax = x1 + 3
yMax = y1 + (3*m)
plt.plot([xMin,xMax],[yMin,yMax], color='magenta')

plt.show()
```



Finding a Derivative for Any Point

Now let's put it all together and define a function that we can use to find the derivative for any point in the ***f*** function:

Here's our general derivative function again:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

We know that ***f(x)*** encapsulates the equation ***$x^2 + x$***, so we can rewrite our derivative equation as:

$$f'(x) = \lim_{h \to 0} \frac{((x+h)^2 + x + h) - (x^2 + x)}{h}$$

We can apply the distribution property to ***$(x + h)^2$*** using the rule that $(a + b)^2 = a^2 + b^2 + 2ab$:

$$f'(x) = \lim_{h \to 0} \frac{(x^2 + h^2 + 2xh + x + h) - (x^2 + x)}{h}$$

Then we can use the distributive property to expand ***$-(x^2 + x)$***, which is the same thing as $-1(x^2 + x)$, to ***$-x^2 - x$***:

$$f'(x) = \lim_{h \to 0} \frac{x^2 + h^2 + 2xh + x + h - x^2 - x}{h}$$

We can combine like terms on the numerator to make things a little clearer:

$$f'(x) = \lim_{h \to 0} \frac{h^2 + 2xh + h}{h}$$

And finally, we can simplify the fraction:

$$f'(x) = \lim_{h \to 0} 2x + h + 1$$

To get the limit when *h* is approaching 0, we can use direct substitution for h:

$$f'(x) = 2x + 0 + 1$$

so:

$$f'(x) = 2x + 1$$

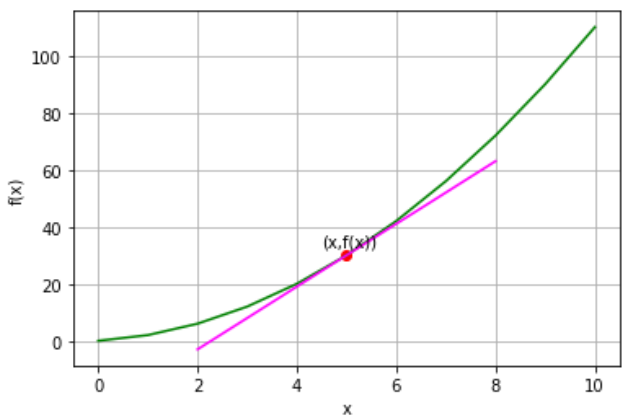
Now we have a function for the derivative of ***f***, which we can apply to any *x* value to find the slope of the function at ***f(x***).

For example, let's find the derivative of ***f*** with an *x* value of 5:

$$f'(5) = 2 \cdot 5 + 1 = 10 + 1 = 11$$

Let's use Python to define the ***f(x)*** and ***f'(x)*** functions, plot ***f(5)*** and show the tangent line for ***f'(5)***:

In [7]:
```
%matplotlib inline

# Create function f
def f(x):
    return x**2 + x

# Create derivative function for f
def fd(x):
    return (2 * x) + 1

from matplotlib import pyplot as plt

# Create an array of x values from 0 to 10 to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [f(i) for i in x]

# Set the point
x1 = 5
y1 = f(x1)

# Calculate the derivative using the derivative function
m = fd(x1)

# Set up the graph
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the point
plt.scatter(x1,y1, c='red')
plt.annotate('(x,f(x))',(x1,y1), xytext=(x1-0.5, y1+3))

# Plot the tangent line using the derivative we calculated
xMin = x1 - 3
yMin = y1 - (3*m)
xMax = x1 + 3
yMax = y1 + (3*m)
plt.plot([xMin,xMax],[yMin,yMax], color='magenta')

plt.show()
```



Differentiability

It's important to realize that a function may not be *differentiable* at every point; that is, you might not be able to calculate the derivative for every point on the function line.

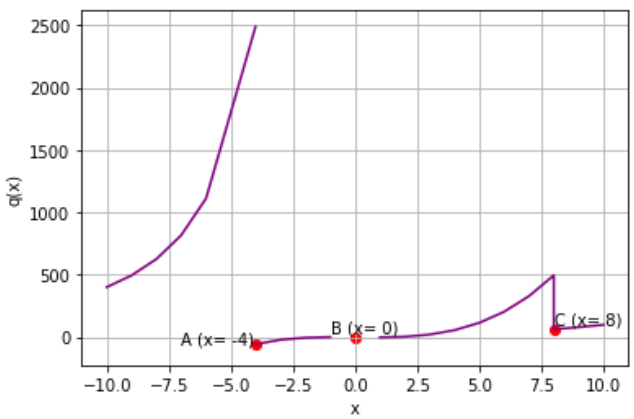
To be differentiable at a given point:

- The function must be *continuous* at that point.
- The tangent line at that point cannot be vertical
- The line must be *smooth* at that point (that is, it cannot take on a sudden change of direction at the point)

For example, consider the following (somewhat bizarre) function:

$$q(x) = \begin{cases} \frac{40{,}000}{x^2}, & \text{if } x < -4, \\ (x^2 - 2) \cdot (x - 1), & \text{if } x \neq 0 \text{ and } x \geq -4 \text{ and } x < 8, \\ (x^2 - 2), & \text{if } x \neq 0 \text{ and } x \geq 8 \end{cases}$$

In [8]:
```
%matplotlib inline

# Define function q
def q(x):
    if x != 0:
        if x < -4:
            return 40000 / (x**2)
        elif x < 8:
            return (x**2 - 2) * x - 1
        else:
            return (x**2 - 2)


# Plot output from function g
from matplotlib import pyplot as plt

# Create an array of x values
x = list(range(-10, -5))
x.append(-4.01)
x2 = list(range(-4,8))
x2.append(7.9999)
x2 = x2 + list(range(8,11))

# Get the corresponding y values from the function
y = [q(i) for i in x]
y2 = [q(i) for i in x2]

# Set up the graph
plt.xlabel('x')
plt.ylabel('q(x)')
plt.grid()

# Plot x against q(x)
plt.plot(x,y, color='purple')
plt.plot(x2,y2, color='purple')


plt.scatter(-4,q(-4), c='red')
plt.annotate('A (x= -4)',(-5,q(-3.9)), xytext=(-7, q(-3.9)))

plt.scatter(0,0, c='red')
plt.annotate('B (x= 0)',(0,0), xytext=(-1, 40))

plt.scatter(8,q(8), c='red')
plt.annotate('C (x= 8)',(8,q(8)), xytext=(8, 100))

plt.show()
```



The points marked on this graph are non-differentiable:

- Point **A** is non-continuous - the limit from the negative side is infinity, but the limit from the positive side ≈ -57
- Point **B** is also non-continuous - the function is not defined at x = 0.
- Point **C** is defined and continuous, but the sharp change in direction makes it non-differentiable.

Derivatives of Equations

We've been talking about derivatves of *functions*, but it's important to remember that functions are just named operations that return a value. We can apply what we know about calculating derivatives to any equation, for example:

$$\frac{d}{dx}(2x + 6)$$

Note that we generally switch to *Leibniz's* notation when finding derivatives of equations that are not encapsulated as functions; but the approach for solving this example is exactly the same as if we had a hypothetical function with the definition *2x + 6*:

$$\frac{d}{dx}(2x + 6) = \lim_{h \to 0} \frac{(2(x + h) + 6) - (2x + 6)}{h}$$

After factoring out the* 2(x+h)* on the left and the *-(2x - 6)* on the right, this is:

$$\frac{d}{dx}(2x + 6) = \lim_{h \to 0} \frac{2x + 2h + 6 - 2x - 6}{h}$$

We can simplify this to:

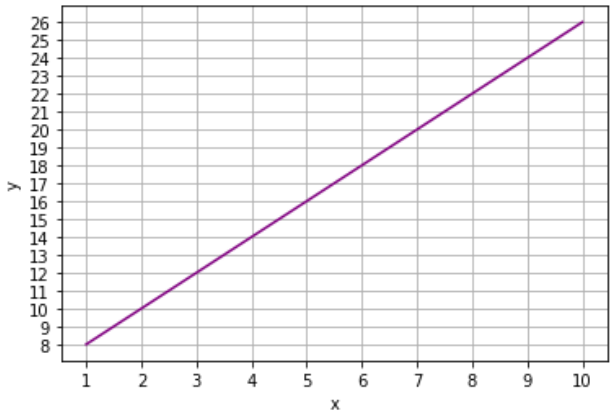
$$\frac{d}{dx}(2x + 6) = \lim_{h \to 0} \frac{2h}{h}$$

Now we can factor *h* out entirely, so at any point:

$$\frac{d}{dx}(2x + 6) = 2$$

If you run the Python code below to plot the line created by the equation, you'll see that it does indeed have a constant slope of 2:

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

# Create an array of x values from 0 to 10 to plot
x = list(range(1, 11))

# Use the function to get the y values
y = [(2*i) + 6 for i in x]

# Set up the graph
plt.xlabel('x')
plt.xticks(range(1,11, 1))
plt.ylabel('y')
plt.yticks(range(8,27, 1))
plt.grid()

# Plot the function
plt.plot(x,y, color='purple')


plt.show()
```



Derivative Rules and Operations

When working with derivatives, there are some rules, or shortcuts, that you can apply to make your life easier.

Basic Derivative Rules

Let's start with some basic rules that it's useful to know.

- If *f(x)* = *C* (where *C* is a constant), then *f'(x)* = 0

  This makes sense if you think about it for a second. No matter what value you use for *x*, the function returns the same constant value; so the graph of the function will be a horizontal line. There's no rate of change in a horiziontal line, so its slope is 0 at all points. This is true of any constant, including symbolic constants like π (pi).

  So, for example:

  $$f(x) = 6 \quad \therefore \quad f'(x) = 0$$

  `Or:`

  $$f(x) = \pi \quad \therefore \quad f'(x) = 0$$

- If *f(x)* = *Cg(x)*, then *f'(x)* = *Cg'(x)*

  This rule tells us that if a function is equal to a second function multiplied by a constant, then the derivative of the first function will be equal to the derivative of the second function multiplied by the same constant. For example:

  $$f(x) = 2g(x) \quad \therefore \quad f'(x) = 2g'(x)$$

- If *f(x)* = *g(x)* + *h(x)*, then *f'(x)* = *g'(x)* + *h'(x)*

  In other words, if a function is the sum of two other functions, then the derivative of the first function is the sum of the derivatives of the other two functions. For example:

  $$f(x) = g(x) + h(x) \quad \therefore \quad f'(x) = g'(x) + h'(x)$$

  `Of course, this also applies to subtraction:`

  $$f(x) = k(x) - l(x) \quad \therefore \quad f'(x) = k'(x) - l'(x)$$

As discussed previously, functions are just equations encapsulated as a named entity that return a value; and the rules can be applied to any equation. For example:

$$\frac{d}{dx}(2x + 6) = \frac{d}{dx}2x + \frac{d}{dx}6$$

So we can take advantage of the rules to make the calculation a little easier:

$$\frac{d}{dx}(2x) = \lim_{h \to 0} \frac{2(x + h) - 2x}{h}$$

After factoring out the* 2(x+h)* on the left, this is:

$$\frac{d}{dx}(2x) = \lim_{h \to 0} \frac{2x + 2h - 2x}{h}$$

We can simplify this to:

$$\frac{d}{dx}(2x) = \lim_{h \to 0} \frac{2h}{h}$$

The Quotient Rule

The *quotient rule* applies to functions that are defined as a quotient of one expression divided by another; for example:

$$r(x) = \frac{s(x)}{t(x)}$$

In this situation, you can apply the following quotient rule to find the derivative of *r(x)*:

$$r'(x) = \frac{s'(x)t(x) - s(x)t'(x)}{[t(x)]^2}$$

Here are our definitions for *s(x)* and *t(x)*:

$$s(x) = 3x^2$$

$$t(x) = 2x$$

Let's start with *s'(x)*:

$$s'(x) = \lim_{h \to 0} \frac{3(x+h)^2 - 3x^2}{h}$$

This factors out to:

$$s'(x) = \lim_{h \to 0} \frac{3x^2 + 3h^2 + 6xh - 3x^2}{h}$$

Which when we cancel out the $3x^2$ and $-3x^2$ is:

$$s'(x) = \lim_{h \to 0} \frac{3h^2 + 6xh}{h}$$

Which simplifies to:

$$s'(x) = \lim_{h \to 0} 3h + 6x$$

With *h* approaching 0, this is:

$$s'(x) = 6x$$

Now let's look at *t'(x)*:

$$t'(x) = \lim_{h \to 0} \frac{2(x+h) - 2x}{h}$$

We can just factor out the *2(x + h)* on the left:

$$t'(x) = \lim_{h \to 0} \frac{2x + 2h - 2x}{h}$$

Which can be cleaned up to:

$$t'(x) = \lim_{h \to 0} \frac{2h}{h}$$

Enabling us to factor *h* out completely to give a constant derivative of *2*:

$$t'(x) = 2$$

So now we can calculate the derivative for the quotient of these functions by plugging the function definitions and the derivatives we've calculated for them into the quotient rule equation:

The Chain Rule

The *chain rule* takes advantage of the fact that equations can be encapsulated as functions, and since functions contain equations, it's possible to nest one function within another.

For example, consider the following function:

$$u(x) = 2x^2$$

We could view the definition of *u(x)* as a composite of two functions,; an *inner* function that calculates x^2, and an *outer* function that multiplies the result of the inner function by 2.

$$u(x) = \overbrace{2(\underbrace{x^2}_{\text{inner}})}^{\text{outer}}$$

To make things simpler, we can name these inner and outer functions:

$$i(x) = x^2$$

$$o(x) = 2x$$

Note that *x* indicates the input for each function. Function *i* takes its input and squares it, and function *o* takes its input and multiplies it by 2. When we use these as a composite function, the *x* value input into the outer function will be the output from the inner function.

Let's take a look at how we can apply these functions to get back to our original *u* function:

$$u(x) = o(i(x))$$

So first we need to find the output of the inner *i* function so we can use at as the input value for the outer *o* function. Well, that's easy, we know the definition of *i* (square the input), so we can just plug it in:

$$u(x) = o(x^2)$$

We also know the definition for the outer *o* function (multiply the input by 2), so we can just apply that to the input:

$$u(x) = 2x^2$$

OK, so now we know how to form a composite function. The *chain rule* can be stated like this:

$$\frac{d}{dx}[o(i(x))] = o'(i(x)) \cdot i'(x)$$

Alright, let's start by plugging the output of the inner *i(x)* function in:

$$\frac{d}{dx}[o(i(x))] = o'(x^2) \cdot i'(x)$$

Now let's use that to calculate the derivative of *o*, replacing each *x* in the equation with the output from the *i* function (x^2):

$$o'(x) = \lim_{h \to 0} \frac{2(x^2 + h) - 2x^2}{h}$$

This factors out to:

$$o'(x) = \lim_{h \to 0} \frac{2x^2 + 2h - 2x^2}{h}$$

Which when we cancel out the $2x^2$ and $-2x^2$ is:

$$o'(x) = \lim_{h \to 0} \frac{2h}{h}$$

Critical Points and Optimization

We've explored various techniques that we can use to calculate the derivative of a function at a specific *x* value; in other words, we can determine the *slope* of the line created by the function at any point on the line.

This ability to calculate the slope means that we can use derivatives to determine some interesting properties of the function.

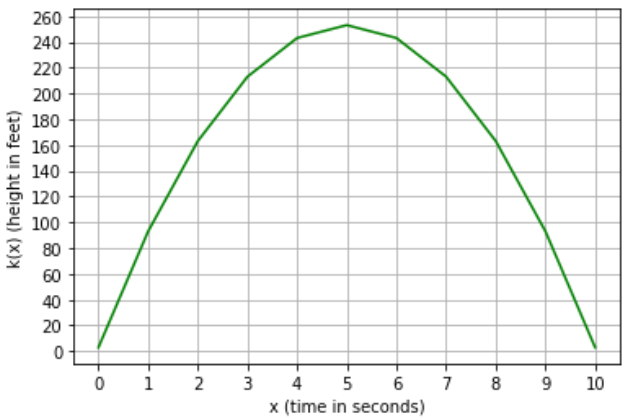
Function Direction at a Point

Consider the following function, which represents the trajectory of a ball that has been kicked on a football field:

$$k(x) = -10x^2 + 100x + 3$$

Run the Python code below to graph this function and see the trajectory of the ball over a period of 10 seconds.

In [1]:
```
%matplotlib inline

# Create function k
def k(x):
    return -10*(x**2) + (100*x)  + 3

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [k(i) for i in x]

# Set up the graph
plt.xlabel('x (time in seconds)')
plt.ylabel('k(x) (height in feet)')
plt.xticks(range(0,15, 1))
plt.yticks(range(-200, 500, 20))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

plt.show()
```



By looking at the graph of this function, you can see that it describes a parabola in which the ball rose in height before falling back to the ground. On the graph, it's fairly easy to see when the ball was rising and when it was falling.

Of course, we can also use derivative to determine the slope of the function at any point. We can apply some of the rules we've discussed previously to determine the derivative function:

- We can add together the derivatives of the individual terms (***-10x^2***, ***100x***, and ***3***) to find the derivative of the entire function.
- The *power* rule tells us that the derivative of ***-10x^2*** is ***-10 • 2x***, which is ***-20x***.
- The *power* rule also tells us that the derivative of ***100x*** is ***100***.
- The derivative of any constant, such as ***3*** is ***0***.

So:

$$k'(x) = -20x + 100 + 0$$

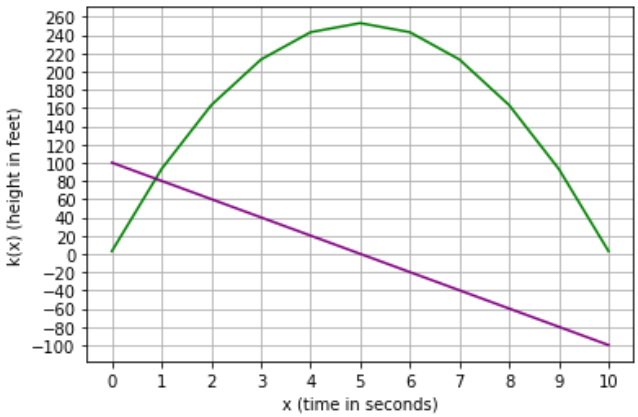
Which of course simplifies to:

$$k'(x) = -20x + 100$$

Now we can use this derivative function to find the slope for any value of ***x***.

Run the cell below to see a graph of the function and its derivative function:

In [2]:
```
%matplotlib inline

# Create function k
def k(x):
    return -10*(x**2) + (100*x)  + 3

def kd(x):
    return -20*x + 100

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [k(i) for i in x]

# Use the derivative function to get the derivative values
yd = [kd(i) for i in x]

# Set up the graph
plt.xlabel('x (time in seconds)')
plt.ylabel('k(x) (height in feet)')
plt.xticks(range(0,15, 1))
plt.yticks(range(-200, 500, 20))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the derivative
plt.plot(x,yd, color='purple')

plt.show()
```



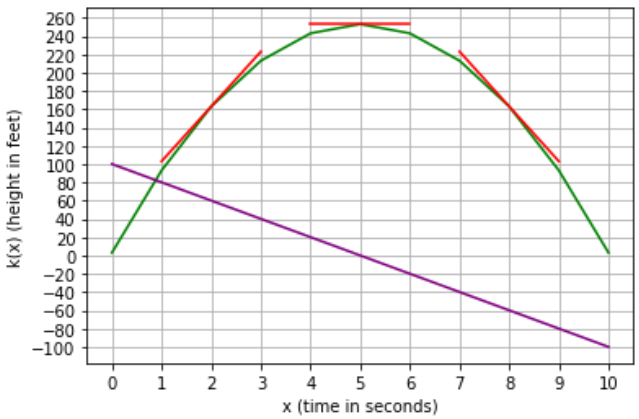
Look closely at the purple line representing the derivative function, and note that it is a constant decreasing value - in other words, the slope of the function is reducing linearly as x increases. Even though the function value itself is increasing for the first half of the parabola (while the ball is rising), the slope is becoming less steep (the ball is not rising at such a high rate), until finally the ball reaches its apogee and the slope becomes negative (the ball begins falling).

Note also that the point where the derivative line crosses 0 on the y-axis is also the point where the function value stops increasing and starts decreasing. When the slope has a positive value, the function is increasing; and when the slope has a negative value, the function is decreasing.

The fact that the derivative line crosses 0 at the highest point of the function makes sense if you think about it logically. If you were to draw the tangent line representing the slope at each point, it would be rotating clockwise throughout the graph, initially pointing up and to the right as the ball rises, and turning until it is pointing down and right as the ball falls. At the highest point, the tangent line would be perfectly horizontal, representing a slope of 0.

Run the following code to visualize this:

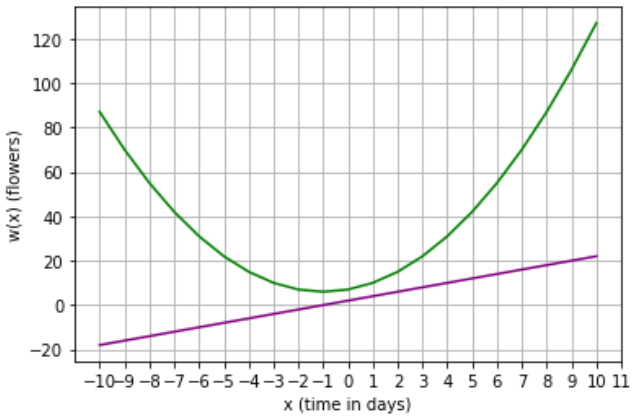
In [3]:
```
%matplotlib inline

# Create function k
def k(x):
    return -10*(x**2) + (100*x)  + 3

def kd(x):
    return -20*x + 100

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [k(i) for i in x]

# Use the derivative function to get the derivative values
yd = [kd(i) for i in x]

# Set up the graph
plt.xlabel('x (time in seconds)')
plt.ylabel('k(x) (height in feet)')
plt.xticks(range(0,15, 1))
plt.yticks(range(-200, 500, 20))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the derivative
plt.plot(x,yd, color='purple')

# Plot tangent slopes for x = 2, 5, and 8
x1 = 2
x2 = 5
x3 = 8
plt.plot([x1-1,x1+1],[k(x1)-(kd(x1)),k(x1)+(kd(x1))], color='red')
plt.plot([x2-1,x2+1],[k(x2)-(kd(x2)),k(x2)+(kd(x2))], color='red')
plt.plot([x3-1,x3+1],[k(x3)-(kd(x3)),k(x3)+(kd(x3))], color='red')

plt.show()
```



Now consider the following function, which represents the number of flowers growing in a flower bed before and after the spraying of a fertilizer:

$$w(x) = x^2 + 2x + 7$$

In [4]:
```
%matplotlib inline

# Create function w
def w(x):
    return (x**2) + (2*x) + 7

def wd(x):
    return 2*x + 2

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(-10, 11))

# Use the function to get the y values
y = [w(i) for i in x]

# Use the derivative function to get the derivative values
yd = [wd(i) for i in x]

# Set up the graph
plt.xlabel('x (time in days)')
plt.ylabel('w(x) (flowers)')
plt.xticks(range(-10,15, 1))
plt.yticks(range(-200, 500, 20))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the derivative
plt.plot(x,yd, color='purple')

plt.show()
```



Note that the green line represents the function, showing the number of flowers for 10 days before and after the fertilizer treatment. Before treatment, the number of flowers was in decline, and after treatment the flower bed started to recover.

The derivative function is shown in purple, and once again shows a linear change in slope. This time, the slope is increasing at a constant rate; and once again, the derivative function line crosses 0 at the lowest point in the function line (in other words, the slope changed from negative to positive when the flowers started to recover).

Critical Points

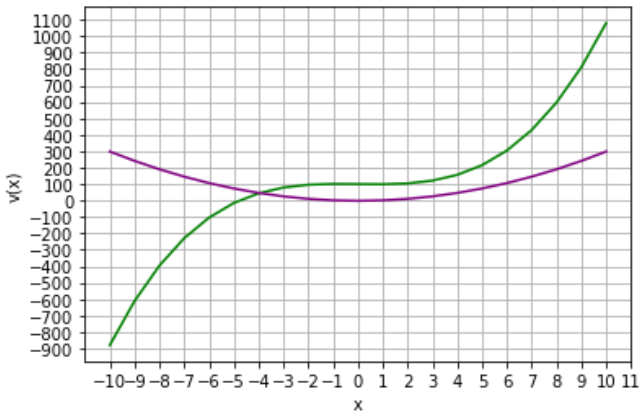
From what we've seen so far, it seems that there is a relationship between a function reaching an extreme value (a maximum or a minimum), and a derivative value of 0. This makes intuitive sense; the derivative represents the slope of the line, so when a function changes from a negative slope to a positive slope, or vice-versa, the derivative must pass through 0.

However, you need to be careful not to assume that just because the derivative is 0 at a given point, that this point represents the minimum or maximum of the function. For example, consider the following function:

$$v(x) = x^3 - 2x + 100$$

Run the following Python code to visualize this function and its corresponding derivative function:

In [5]:
```
%matplotlib inline

# Create function v
def v(x):
    return (x**3) - (2*x) + 100

def vd(x):
    return 3*(x**2) - 2

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(-10, 11))

# Use the function to get the y values
y = [v(i) for i in x]

# Use the derivative function to get the derivative values
yd = [vd(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('v(x)')
plt.xticks(range(-10,15, 1))
plt.yticks(range(-1000, 2000, 100))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the derivative
plt.plot(x,yd, color='purple')

plt.show()
```



Note that in this case, the purple derivative function line passes through 0 as the green function line transitions from a *concave downwards* slope (a slope that is decreasing) to a *concave upwards* slope (a slope that is increasing). The slope flattens out to 0, forming a "saddle" before the it starts increasing.

What we can learn from this is that interesting things seem to happen to the function when the derivative is 0. We call points where the derivative crosses 0 *critical points*, because they indicate that the function is changing direction. When a function changes direction from positive to negative, it forms a peak (or a *local maximum*), when the function changes direction from negative to positive it forms a trough (or *local minimum*), and when it maintains the same overall direction but changes the concavity of the slope it creates an *inflexion point*.

Finding Minima and Maxima

A common use of calculus is to find minimum and maximum points in a function. For example, we might want to find out how many seconds it took for the kicked football to reach its maximum height, or how long it took for our fertilizer to be effective in reversing the decline of flower growth.

We've seen that when a function changes direction to create a maximum peak or a minimum trough, the derivative of the function is 0, so a step towards finding these extreme points might be to simply find all of the points in the function where the derivative is 0. For example, here's our function for the kicked football:

$$k(x) = -10x^2 + 100x + 3$$

From this, we've calculated the function for the derivative as:

$$k'(x) = -20x + 100$$

We can then solve the derivative equation for an f'(x) value of 0:

$$-20x + 100 = 0$$

We can remove the constant by subtracting 100 to both sides:

$$-20x = -100$$

Multiplying both sides by -1 gets rid of the negative values (this isn't strictly necessary, but makes the equation a little less confusing)

$$20x = 100$$

So:

$$x = 5$$

So we know that the derivative will be 0 when *x* is 5, but is this a minimum, a maximum, or neither? It could just be an inflexion point, or the entire function could be a constant value with a slope of 0) Without looking at the graph, it's difficult to tell.

Second Order Derivatives

The solution to our problem is to find the derivative of the derivative! Until now, we've found the derivative of a function, and indicated it as ***f'(x)***. Technically, this is known as the *prime* derivative; and it describes the slope of the function. Since the derivative function is itself a function, we can find its derivative, which we call the *second order* (or sometimes just *second*) derivative. This is indicated like this: ***f''(x)***.

So, here's our function for the kicked football:

$$k(x) = -10x^2 + 100x + 3$$

Here's the function for the prime derivative:

$$k'(x) = -20x + 100$$

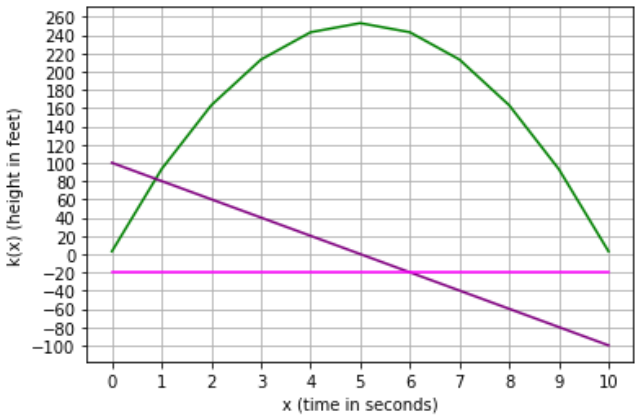
And using a combination of the power rule and the constant rule, here's the function for the second derivative:

$$k''(x) = -20$$

Now, without even drawing the graph, we can see that the second derivative has a constant value; so we know that the slope of the prime derivative is linear; and because it's a negative value, we know that it is decreasing. So when the prime derivative crosses 0, it we know that the slope of the function is decreasing linearly; so the point at *x=0* must be a maximum point.

Run the following code to plot the function, the prime derivative, and the second derivative for the kicked

In [6]:
```
%matplotlib inline

# Create function k
def k(x):
    return -10*(x**2) + (100*x)  + 3

def kd(x):
    return -20*x + 100

def k2d(x):
    return -20

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(0, 11))

# Use the function to get the y values
y = [k(i) for i in x]

# Use the derivative function to get the k'(x) values
yd = [kd(i) for i in x]

# Use the 2-derivative function to get the k''(x)
y2d = [k2d(i) for i in x]

# Set up the graph
plt.xlabel('x (time in seconds)')
plt.ylabel('k(x) (height in feet)')
plt.xticks(range(0,15, 1))
plt.yticks(range(-200, 500, 20))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot  k'(x)
plt.plot(x,yd, color='purple')

# Plot k''(x)
plt.plot(x,y2d, color='magenta')

plt.show()
```



Let's take the same approach for the flower bed problem. Here's the function:

$$w(x) = x^2 + 2x + 7$$

Using the power rule and constant rule, gives us the prime derivative function:

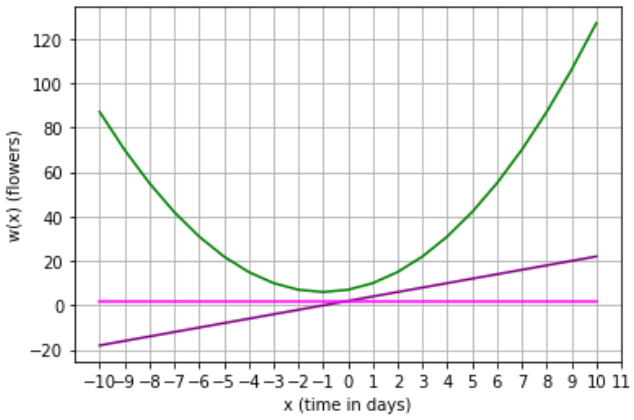
$$w'(x) = 2x + 2$$

Applying the power rule and constant rule to the prime derivative function gives us the second derivative function:

$$w''(x) = 2$$

Note that this time, the second derivative is a positive constant, so the prime derivative (which is the slope of the function) is increasing linearly. The point where the prime derivative crosses 0 must therefore be a minimum. Let's run the code below to check:

In [7]:
```
%matplotlib inline

# Create function w
def w(x):
    return (x**2) + (2*x) + 7

def wd(x):
    return 2*x + 2

def w2d(x):
    return 2

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(-10, 11))

# Use the function to get the y values
y = [w(i) for i in x]

# Use the derivative function to get the w'(x) values
yd = [wd(i) for i in x]

# Use the 2-derivative function to get the w''(x) values
y2d = [w2d(i) for i in x]

# Set up the graph
plt.xlabel('x (time in days)')
plt.ylabel('w(x) (flowers)')
plt.xticks(range(-10,15, 1))
plt.yticks(range(-200, 500, 20))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot w'(x)
plt.plot(x,yd, color='purple')

# Plot w''(x)
plt.plot(x,y2d, color='magenta')

plt.show()
```



Critical Points that are *Not* Maxima or Minima

Of course, it's possible for a function to form a "saddle" where the prime derivative is zero at a point that is not a minimum or maximum. Here's an example of a function like this:

$$v(x) = x^3 - 6x^2 + 12x + 2$$

And here's its prime derivative:

$$v'(x) = 3x^2 - 12x + 12$$

Let's find a critical point where v'(x) = 0

$$3x^2 - 12x + 12 = 0$$

Factor the x-terms

$$3x(x - 4) = 12$$

Divide both sides by 3:

$$x(x - 4) = 4$$

Factor the x terms back again

$$x^2 - 4x = 4$$

Complete the square, step 1

$$x^2 - 4x + 4 = 0$$

Complete the square, step 2

$$(x - 2)^2 = 0$$

Find the square root:

$$x - 2 = \pm\sqrt{0}$$

$$x - 2 = +\sqrt{0} = 0, -\sqrt{0} = 0$$

v'(2) = 0 (only touches 0 once)

Is it a maximum or minimum? Let's find the second derivative:

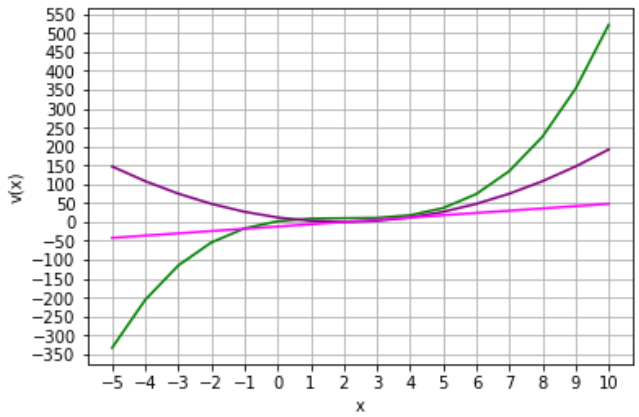
$$v''(x) = 6x - 12$$

So

$$v''(2) = 0$$

So it's neither negative or positive, so it's not a maximum or minimum.

In [8]:
```
%matplotlib inline

# Create function v
def v(x):
    return (x**3) - (6*(x**2)) + (12*x) + 2

def vd(x):
    return (3*(x**2)) - (12*x) + 12

def v2d(x):
    return (3*(2*x)) - 12

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(-5, 11))

# Use the function to get the y values
y = [v(i) for i in x]

# Use the derivative function to get the derivative values
yd = [vd(i) for i in x]

# Use the derivative function to get the derivative values
y2d = [v2d(i) for i in x]

# Set up the graph
plt.xlabel('x')
plt.ylabel('v(x)')
plt.xticks(range(-10,15, 1))
plt.yticks(range(-2000, 2000, 50))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

# Plot the derivative
plt.plot(x,yd, color='purple')

# Plot the derivative
plt.plot(x,y2d, color='magenta')

plt.show()

print ("v(2) = " + str(v(2)))

print ("v'(2) = " + str(vd(2)))

print ("v''(2) = " + str(v2d(2)))
```




```
v(2) = 10
v'(2) = 0
```

Optimization

The ability to use derivatives to find minima and maxima of a function makes it a useful tool for scenarios where you need to optimize a function for a specific variable.

Defining Functions to be Optimized

For example, suppose you have decided to build an online video service that is based on a subscription model. You plan to charge a monthly subscription fee, and you want to make the most revenue possible. The problem is that customers are price-sensitive, so if you set the monthly fee too high, you'll deter some customers from signing up. Conversely, if you set the fee too low, you may get more customers, but at the cost of reduced revenue.

What you need is some kind of function that will tell you how many subscriptions you might expect to get based on a given fee. So you've done some research, and found a formula to indicate that the expected subscription volume (in thousands) can be calculated as 5-times the monthly fee subtracted from 100; or expressed as a function:

$$s(x) = -5x + 100$$

What you actually want to optimize is monthly revenue, which is simply the number of subscribers multiplied by the fee:

$$r(x) = s(x) \cdot x$$

We can combine ***s(x)*** into ***r(x)*** like this:

$$r(x) = -5x^2 + 100x$$

Finding the Prime Derivative

The function ***r(x)*** will return the expected monthly revenue (in thousands) for any proposed fee (*x*). What we need to do now is to find the fee that yields the maximum revenue. Fortunately, we can use a derivative to do that.

First, we need to determine the prime derivative of ***r(x)***, and we can do that easily using the power rule:

$$r'(x) = 2 \cdot -5x + 100$$

Which is:

$$r'(x) = -10x + 100$$

Find Critical Points

Now we need to find any critical points where the derivative is 0, as this could indicate a maximum:

$$-10x + 100 = 0$$

Let's isolate the *x* term:

$$-10x = -100$$

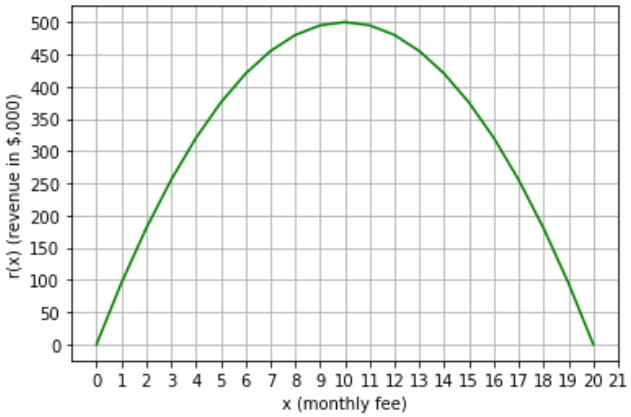
Both sides are negative, so we can mulitply both by -1 to make them positive without affecting the equation:

$$10x = 100$$

Now we can divide both sides by 10 to isolate *x*:

$$x = \frac{100}{10}$$

In [9]:
```
%matplotlib inline

# Create function s
def s(x):
    return (-5*x) + 100

# Create function r
def r(x):
    return s(x) * x

from matplotlib import pyplot as plt

# Create an array of x values to plot
x = list(range(0, 21))

# Use the function to get the y values
y = [r(i) for i in x]

# Set up the graph
plt.xlabel('x (monthly fee)')
plt.ylabel('r(x) (revenue in $,000)')
plt.xticks(range(0,22, 1))
plt.yticks(range(0, 600, 50))
plt.grid()

# Plot the function
plt.plot(x,y, color='green')

plt.show()
```



Partial Derivatives

Until now, we've considered derivatives of functions that operate on a single variable. How do we take the derivatives of a function like the following?

$$f(x, y) = x^2 + y^2$$

We can take a derivative of the changes in the function with respect to either x or y. We call these derivatives with respect to one variable partial derivatives. Let's give this a try by taking the derivative of $f(x, y)$ with respect to ***x***. We write this partial derivative as follows.

$$\frac{\partial f(x, y)}{\partial x} = \frac{\partial (x^2 + y^2)}{\partial x}$$

Just as ordinary derivatives give us a way to compute the rate of change of a function, partial derivatives give us a way to compute the rate of change of a function of many variables with respect to one of those variables.

Since $f(x, y)$ is the sum of several simpler functions we need to take the partial derivative of each of these and sum the result. The first two parts are easy.

$$\frac{\partial x^2}{\partial x} = 2x$$

Notice that we are following the usual rules of differentiation for any function of ***x*** here.

Now we need to take the partial derivative of the last part of $f(x, y)$, which does not depend on ***x*** at all. In these care we get the following.

$$\frac{\partial y^2}{\partial x} = 0$$

Now we can add up the parts to get the complete partail derivative of $f(x, y)$.

$$\frac{\partial f(x, y)}{\partial x} = 2x + 0 = 2x$$

We can also take the partial derivative of $f(x, y)$ with respect to ***y***. The process proceeds in the following manner.

$$\frac{\partial f(x, y)}{\partial y} = 0 + 2y = 2y$$

Computing a Gradient

At this point, you may well ask what is the point of computing partial derivatives? Yes, they are a nifty math trick, but what are they good for? It turns out that partial derivatives are important if you want to find the analog of the slope for multi-dimensonal surfaces. We call this quantity the **gradient**.

Recall that you can find minimum and maximum of curves using derivatives. In the same way, you can find the minimum and maximum of surfaces by following the gradiennt and finding the points were the gradient is zero in all directions.

You have already examined the partial derivatives of the function, $f(x, y) = x^2 + y^2$. These partial derivatives are:

$$\frac{\partial f(x, y)}{\partial x} = 2x$$
$$\frac{\partial f(x, y)}{\partial y} = 2y$$

In this case, the gradient is a 2-dimensional vector of the change of the function in the x direction and the change in the function in the y direction. This vector can be written as follows:

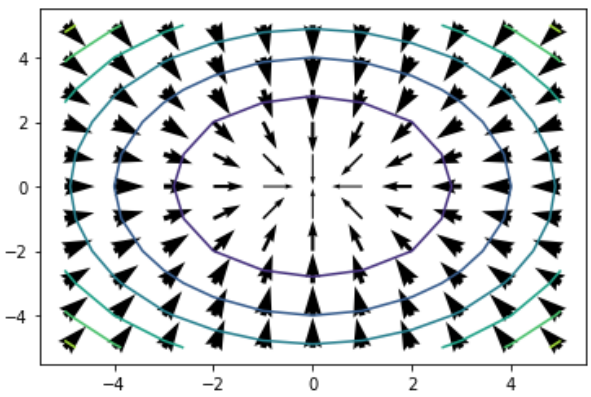
$$grad(f(x, y)) = \vec{g}(x, y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

Plotting the Gradient

A plot will help you get feel for the meaning of the gradient. The code below plots the gradient of the function $f(x, y) = x^2 + y^2$ along with contours of the value of the function. Run this code and examine the plot.

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

## Create a uniform grid
el = np.arange(-5,6)
nx, ny = np.meshgrid(el, el, sparse=False, indexing='ij')

## flatten the gird to 1-d and compute the value of the function z
x_coord = []
y_coord = []
z = []
for i in range(11):
    for j in range(11):
        x_coord.append(float(-nx[i,j]))
        y_coord.append(float(-ny[i,j]))
        z.append(nx[i,j]**2 + ny[i,j]**2)

## perform vector arithmetic to get the x and y gradients
x_grad = [-2 * x for x in x_coord]
y_grad = [-2 * y for y in y_coord]

## Plot the arrows using  width for gradient
plt.xlim(-5.5,5.5)
plt.ylim(-5.5,5.5)
for x, y, xg, yg in zip(list(x_coord), list(y_coord), list(x_grad), list(y_grad
    if x != 0.0 or y != 0.0: ## Avoid the zero divide when scaling the arrow
        l = math.sqrt(xg**2 + yg**2)/2.0
        plt.quiver(x, y, xg, yg, width = l, units = 'dots')

## Plot the countours of the function surface
z = np.array(z).reshape(11,11)
plt.contour(el, el, z)
```

Out[1]:
```
<matplotlib.contour.QuadContourSet at 0x7f8ed4381550>
```



Notice the following properties of this plot.

- The arrows in the plot point in the direction of the gradient.
- The width of the arrows is proportional to the value of the gradient. The width of the arrows and the **gradient decreases as function gets closer to the minimum**. If this is the case everywhere, you can say that a function is **convex**. It is always much easier to find minimum of convex functions.
- The **direction of the gradient is always perpendicular to the contours**. This is an important property of multivariate functions.

Using the gradient

So, what is all this good for? Say that you want to find the minimum of the function $f(x, y) = x^2 + y^2$. It is easy to see that the minimum of this function is at $x = 0$ and $y = 0$. But, what if you did not know this solution? Then you could do the following:

1. Take some starting guess.
2. Compute the gradient.
3. take a small step in the direction of the gradient.
4. Determine if the gradient is close to zero. If so, then stop, since the gradient will be zero at the minimum.
5. Repeate steps 2, 3 and 4.

The algorithm outlined above is called the **gradient decent method**. It is the basis of many real-world minimization algorithms.

Introduction to Integration

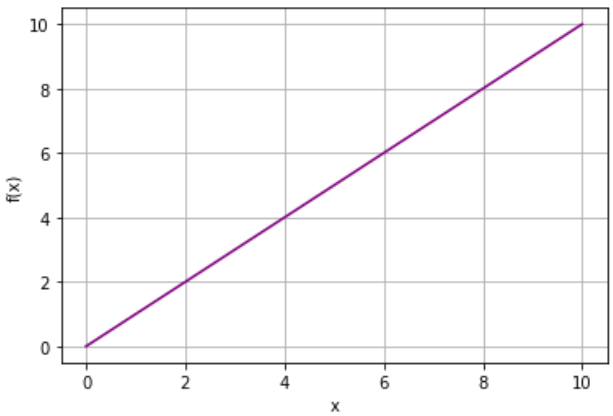
Integrals are the inverses of derivatives. More importantly, using integration provides a way to compute the area under the curve of most any function. There are many applications for integration. For example, if you need to compute a probability of some occurance between limits (which we'll discuss later in this course), then you will use an integral.

Let's start with a simple function:

$$f(x) = x$$

We can plot this function as a line. Run the code below to plot the function for the range 0 to 10:

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

# Define function f
def f(x):
    return x

# Create an array of x values from 0 to 10
x = range(0, 11)

# Get the corresponding y values from the function
y = [f(a) for a in x]

# Set up the plot
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot x against f(x)
plt.plot(x,y, color='purple')

plt.show()
```



Performing Integration

The *integral* of a function is the area under it - in this case, the area under the purple diagonal line down to the x-axis.

So how do you find the integral of a function? well, for our simple function $f(x) = x$, the formula for an integral is written as follows:

$$\int f(x)\ dx$$

The $\int$ symbol shows that this formula is an integral. The dx indicates that the integration is with respect to the x variable. Note that since $f(x) = x$, we could also write this integral formula as $\int x\ dx$

So, what is the integral of $x\ dx$? To answer this question, we need the *antiderivative* of ***f*** - in other words we need to find a function which has a derivative matching the output of f, which is just x. Using the power rule in reverse, a function that has the derivative x would be $\frac{1}{2}x^2$

So, the *unbound* integral formula for f with respect to x can be written as:

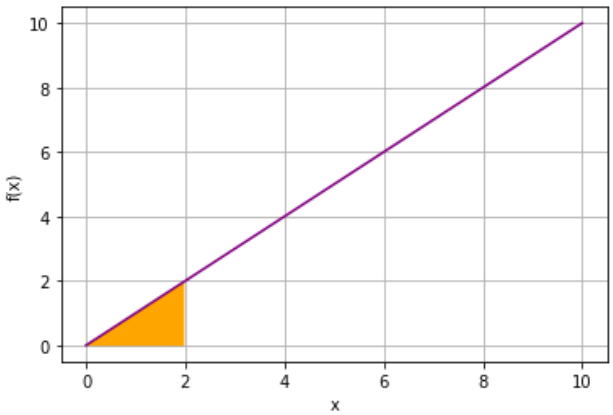
$$\int f(x)\ dx = \frac{1}{2}x^2$$

Integration between Limits

Now that we have the unbound integral formula, we can use it to find the integral between specific start and end points. Let's suppose we want to find the area under the function between the x values 0 and 2. In other words, the *integral* of f for the range 0 to 2 with respect to x.

Run the following code to re-plot the function and show the area we're interested in:

In [2]:
```
# Set up the plot
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot x against f(x)
plt.plot(x,y, color='purple')

# show area for integral
section = np.arange(0, 2, 1/20)
plt.fill_between(section,f(section), color='orange')

plt.show()
```



We call the start and end point the **limits** of the integral. The lower limit is placed as a subscript of the integral sign. The upper limit is placed as a superscript of the integral sign. Using this notation the integral of $f(x)$ from 0 to 2 is written as follows:

$$\int_0^2 f(x)\ dx$$

The integral is evaluated by subtracting the value of the integrand at the lower limit from the integrand at the upper limit; and since we know the formula based on our antiderivative function, the integral can be evaluated in the following manner.

$$\begin{aligned}
&\int_0^2 f(x)\ dx \\
&= \frac{1}{2}x^2 \Big|_0^2 \\
&= \frac{1}{2}2^2 - \frac{1}{2}0^2 \\
&= \frac{4}{2} - \frac{0}{2}x^2 \\
&= 2
\end{aligned}$$

Execute the code in the cell below and verify that the result returned by the ***scipy.integrate.quad*** function in Python is approximately the same as we computed analytically.

In [3]:
```
import scipy.integrate as integrate
i, e = integrate.quad(lambda x: f(x), 0, 2)
print (i)
```

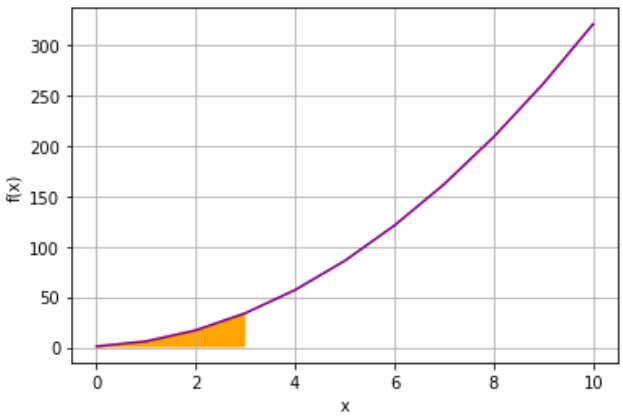
```
2.0
```

Another Integral

Here is another example for a slightly more complex function. What is the area under the curve of the function $3x^2 + 2x + 1$ between 0 and 3?

let's look at that function and the area in question:

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

# Define function g
def g(x):
    return 3 * x**2 + 2 * x + 1

# Create an array of x values from 0 to 10
x = range(0, 11)

# Get the corresponding y values from the function
y = [g(a) for a in x]

# Set up the plot
fig, ax = plt.subplots()
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()

# Plot x against g(x)
plt.plot(x,y, color='purple')

# Make the shaded region
ix = np.linspace(0, 3)
iy = g(ix)
verts = [(0, 0)] + list(zip(ix, iy)) + [(3, 0)]
poly = Polygon(verts, facecolor='orange')
ax.add_patch(poly)

plt.show()
```



We can evaluate this integral just as before, this time using function:

$$\int_0^3 3x^2 + 2x + 1 \; dx$$

We can calculate the antiderivative of $3x^2 + 2x + 1 \; dx$ as $\frac{3}{3}x^3 + \frac{2}{2}x^2 + x$, so:

$$\int_0^3 = \frac{3}{3}x^3 + \frac{2}{2}x^2 + x \Big|_0^3$$
$$= \frac{3}{3}3^3 + \frac{2}{2}3^2 + 3 - \frac{3}{3}0^3 - \frac{2}{2}0^2 + 0$$
$$= 27 + 9 + 3 + 0 + 0 + 0$$
$$= 39$$

Now, execute the code in the cell below to verify the result:

In [5]:
```
i, e = integrate.quad(lambda x: 3 * x**2 + 2 * x + 1, 0, 3)
print(i)
```

```
38.99999999999999
```

Note that the result from the ***scipy.integrate.quad*** function is approximate - the function actually returns an estimated integral (*i* in this case) and also a measure of absolute error (*e*). Run the following code to see what the absolute error was in this case:

In [6]:
```
print(e)
```

```
4.3298697960381095e-13
```

The absolute error in this case is extremely small (around 4.3×10^{-13}).

Infinite limits

In many cases the limits of an integral can be $+/-\infty$. Perhaps suprisingly, this situation is not really a problem if the function being integrated converges to 0 at the infinite limit.

Here is an example. The function $e^{-5x} \rightarrow 0$ as $x \rightarrow \infty$. Therefore, the integral of this function from some limit to ∞. This integral can be written as follows:

$$\int_0^\infty e^{-5x} dx$$

The code in the cell below computes this integral numerically.

In [7]:
```
import numpy as np
i, e = integrate.quad(lambda x: np.exp(-x*5), 0, np.inf)

print('Integral: ' + str(i))
print('Absolute Error: ' + str(e))
```

```
Integral: 0.20000000000000004
Absolute Error: 1.5606666951908062e-11
```

This integral converges to a small number with a much smaller error estimate.

Here is another example that illustrates why having infinite integration limits is so useful. When computing probabilities it is often necessary to have infinite limits. Don't worry too much about the details of probability theory. This is covered in a later lesson.

A Normal distribution with zero mean and a standard deviation of 1 has the following density function:

$$\frac{1}{2\pi} e^{\frac{-x^2}{\sqrt{(2\pi)}}}$$

It makes sense that the integral of this probability density function from $-\infty$ to ∞ must be 1.0. In other words the probability of a Normally distributed event ocurring at all possible values must be 1.0.

The code in the cell below computes the following integral:

$$\int_{-\infty}^{\infty} \frac{1}{2\pi} e^{\frac{-x^2}{\sqrt{(2\pi)}}} dx$$

Execute this code and verify that the result is approximately 1.0.

In [8]:
```
import numpy as np
norms = lambda x: np.exp(-x**2/2.0)/np.sqrt(2.0 * 3.14159)
i, e = integrate.quad(norms, -np.inf, np.inf)

print('Integral: ' + str(i))
print('Absolute Error: ' + str(e))
```

```
Integral: 1.0000004223321999
Absolute Error: 1.017819568483304e-08
```

Vectors

Vectors, and vector spaces, are fundamental to *linear algebra*, and they're used in many machine learning models. Vectors describe spatial lines and planes, enabling you to perform calculations that explore relationships in multi-dimensional space.

What is a Vector

At its simplest, a vector is a numeric element that has both *magnitude* and *direction*. The magnitude represents a distance (for example, "2 miles") and the direction indicates which way the vector is headed (for example, "East"). Vectors are defined by an n-dimensional coordinate that describe a point in space that can be connected by a line from an arbitrary origin.

That all seems a bit complicated, so let's start with a simple, two-dimensional example. In this case, we'll have a vector that is defined by a point in a two-dimensional plane: A two dimensional coordinate consists of an *x* and a *y* value, and in this case we'll use **2** for *x* and **1** for *y*.

Our vector can be written as **v**=(2,1), but more formally we would use the following notation, in which the dimensional coordinate values for the vector are shown as a matrix:

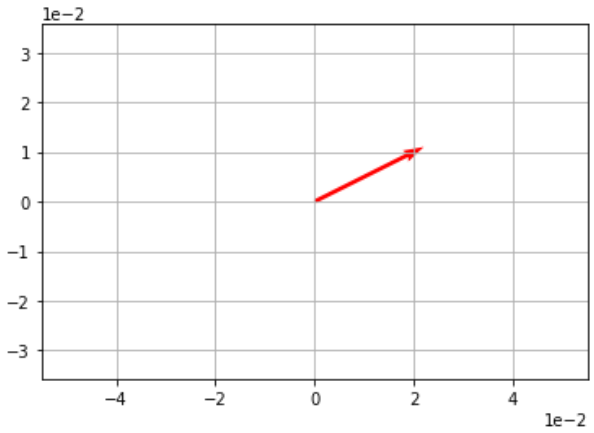
$$\vec{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

So what exactly does that mean? Well, the coordinate is two-dimensional, and describes the movements required to get to the end point (of *head*) of the vector - in this case, we need to move 2 units in the *x* dimension, and 1 unit in the *y* dimension. Note that we don't specify a starting point for the vector - we're simply describing a destination coordinate that encapsulate the magnitide and direction of the vector. Think about it as the directions you need to follow to get to *there* from *here*, without specifying where *here* actually is!

It can help to visualize the vector, and with a two-dimensional vector, that's pretty straightforward. We just define a two-dimensional plane, choose a starting point, and plot the coordinate described by the vector relative to the starting point.

Run the code in the following cell to visualize the vector **v** (which remember is described by the coordinate (2,1)).

In [1]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

# We'll use a numpy array for our vector
v = np.array([2,1])

# and we'll use a quiver plot to visualize it.
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, *v, scale=10, color='r')
plt.show()
```



Note that we can use a numpy array to define the vector in Python; so to create our (2,1) vector, we simply create a numpy array with the elements [2,1]. We've then used a quiver plot to visualize the vector, using the point 0,0 as the starting point (or *origin*). Our vector of (2,1) is shown as an arrow that starts at 0,0 and moves 2 units along the *x* axis (to the right) and 1 unit along the *y* axis (up).

Calculating Vector Magnitude and Direction

We tend to work with vectors by expressing their components as *cartesian coordinates*; that is, *x* and *y* (and other dimension) values that define the number of units travelled along each dimension. So the coordinates of our (2,1) vector indicate that we must travel 2 units along the *x* axis, and *1* unit along the *y* axis.

However, you can also work with verctors in terms of their *polar coordinates*; that is coordinates that describe the magnitude and direction of the vector. The magnitude is the overall distance of the vector from tail to head, and the direction is the angle at which the vector is oriented.

Calculating Magnitude

Calculating the magnitude of the vector from its cartesian coordinates requires measuring the distance between the arbitrary starting point and the vector head point. For a two-dimensional vector, we're actually just calculating the length of the hypotenuse in a right-angled triangle - so we could simply invoke Pythagorean theorum and calculate the square root of the sum of the squares of it's components, like this:

$$\|\vec{v}\| = \sqrt{v_1{}^2 + v_2{}^2}$$

The notation for a vector's magnitude is to surround the vector name with vertical bars - you can use single bars (for example, |**v**|) or double bars (||**v**||). Double-bars are often used to avoid confusion with absolute values. Note that the components of the vector are indicated by subscript indices (v_1, v_2,...v_n),

In this case, the vector **v** has two components with values **2** and **1**, so our magnitude calculation is:

$$\|\vec{v}\| = \sqrt{2^2 + 1^2}$$

Which is:

$$\|\vec{v}\| = \sqrt{4 + 1}$$

So:

$$\|\vec{v}\| = \sqrt{5} \approx 2.24$$

You can run the following Python code to get a more precise result (note that the elements of a numpy array are zero-based)

In [2]:
```
import math

vMag = math.sqrt(v[0]**2 + v[1]**2)
print (vMag)
```

```
2.23606797749979
```

This calculation works for vectors of any dimensionality - you just take the square root of the sum of the squared components:

$$\|\vec{v}\| = \sqrt{v_1{}^2 + v_2{}^2 \ldots + v_n{}^2}$$

In Python, *numpy* provides a linear algebra library named **linalg** that makes it easier to work with vectors - you can use the **norm** function in the following code to calculate the magnitude of a vector:

In [3]:
```
import numpy as np

vMag = np.linalg.norm(v)
print (vMag)
```

```
2.23606797749979
```

Calculating Direction

To calculate the direction, or *amplitude*, of a vector from its cartesian coordinates, you must employ a little trigonometry. We can get the angle of the vector by calculating the *inverse tangent*; sometimes known as the *arctan* (the *tangent* calculates an angle as a ratio - the inverse tangent, or **tan^{-1}**, expresses this in degrees).

In any right-angled triangle, the tangent is calculated as the *opposite* over the *adjacent*. In a two dimensional vector, this is the *y* value over the *x* value, so for our **v** vector (2,1):

$$tan(\theta) = \frac{1}{2}$$

This produces the result ***0.5***, from which we can use a calculator to calculate the inverse tangent to get the angle in degrees:

$$\theta = tan^{-1}(0.5) \approx 26.57^{o}$$

Note that the direction angle is indicated as ***θ***.

Run the following Python code to confirm this:

In [4]:
```
import math
import numpy as np

v = np.array([2,1])
vTan = v[1] / v[0]
print ('tan = ' + str(vTan))
vAtan = math.atan(vTan)
# atan returns the angle in radians, so convert to degrees
print('inverse-tan = ' + str(math.degrees(vAtan)))
```

```
tan = 0.5
inverse-tan = 26.56505117707799
```

There is an added complication however, because if the value for *x* or *y* (or both) is negative, the orientation of the vector is not standard, and a calculator can give you the wrong tan^{-1} value. To ensure you get the correct direction for your vector, use the following rules:

- Both *x* and *y* are positive: Use the tan^{-1} value.
- *x* is negative, *y* is positive: Add 180 to the tan^{-1} value.
- Both *x* and *y* are negative: Add 180 to the tan^{-1} value.
- *x* is positive, *y* is negative: Add 360 to the tan^{-1} value.

To understand why we need to do this, think of it this way. A vector can be pointing in any direction through a 360 degree arc. Let's break that circle into four quadrants with the x and y axis through the center. Angles can be measured from the x axis in both the positive (counter-clockwise) and negative (clockwise) directions. We'll number the quadrants in the positive (counter-clockwise) direction (which is how we measure the *positive* angle) like this:

```
2 | 1
- o -
3 | 4
```

OK, let's look at 4 example vectors

1. Vector [2,4] has positive values for both x and y. The line for this vector travels through the point 0,0 from quadrant 3 to quadrant 1. Tan^{-1} of 4/2 is around 63.4 degrees, which is the positive angle from the x axis to the vector line - so this is the direction of the vector.
2. Vector [-2,4] has a negative x and positive y. The line for this vector travels through point 0,0 from quadrant 4 to quadrant 2. Tan^{-1} of 4/-2 is around -64.4 degrees, which is the *negative* angle from x to the vector line; but in the wrong direction (as if the vector was travelling from quadrant 2 towards quadrant 4). So we need the opposite direction, which we get by adding 180.
3. Vector [-2,-4] has negative x and y. The line for the vector travels through 0,0 from quadrant 1 to quadrant 3. Tan^{-1} of -4/-2 is around 63.4 degrees, which is the angle between the x axis and the line, but again in the opposite direction, from quadrant 3 to quadrant 1; we need to go a further 180 degrees to reflect the correct direction.
4. Vector [2,-4] has positive x and negative y. It travels through 0,0 from quadrant 2 to quadrant 4. Tan^{-1} of -4/2 is around -64.4 degrees, which is the *negative* angle from the x axis to the vector line. Technically it's correct, the line is travelleing down and to the right at an angle of -63.4 degrees; but we want to express the *positive* (counter-clockwise) angle, so we add 360.

In the previous Python code, we used the *math.**atan*** *function to calculate the inverse tangent from a numeric tangent. The *numpy* library includes a similar ***arctan*** function. When working with numpy arrays, you can also use the *numpy.**arctan2**** *function to return the inverse tangent of an array-based vector in *radians*, and you can use the *numpy.**degrees**** *function to convert this to degrees. The **arctan2** function automatically makes the necessary adjustment for negative *x* and *y* values.

In [5]:
```
import numpy as np

v = np.array([2,1])
print ('v: ' + str(np.degrees(np.arctan2(v[1], v[0]))))

s = np.array([-3,2])
print ('s: ' + str(np.degrees(np.arctan2(s[1], s[0]))))
```

```
v: 26.56505117707799
s: 146.30993247402023
```

Vector Addition

So far, we've worked with one vector at a time. What happens when you need to add two vectors.

Let's take a look at an example, we already have a vector named **v**, as defined here:

$$\vec{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

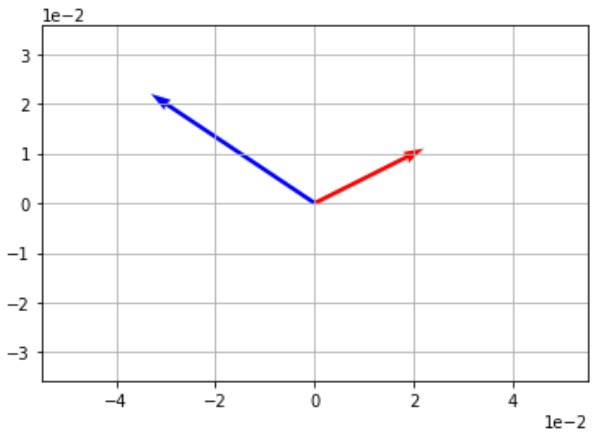
Now let's create a second vector, and called **s** like this:

$$\vec{s} = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$$

Run the cell below to create **s** and plot it together with **v**:

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

v = np.array([2,1])
s = np.array([-3,2])
print (s)

# Plot v and s
vecs = np.array([v,s])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['r', 'b'], scale=10)
plt.show()
```

```
[-3  2]
```



You can see in the plot that the two vectors have different directions and magnitudes. So what happens when we add them together?

Here's the formula:

$$\vec{z} = \vec{v} + \vec{s}$$

In terms of our vector matrices, this looks like this:

$$\vec{z} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} -3 \\ 2 \end{bmatrix}$$

Which gives the following result:

$$\vec{z} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + \begin{bmatrix} -3 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix}$$

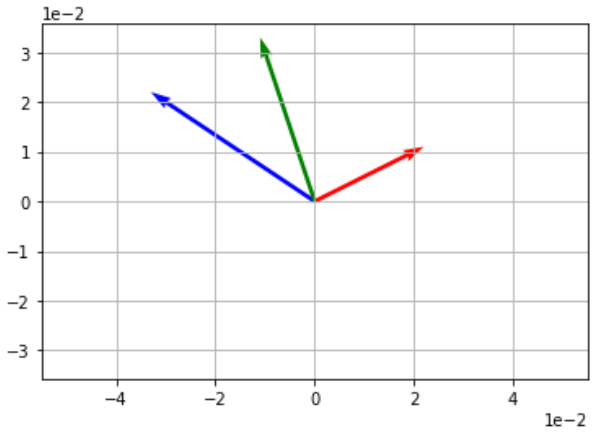
Let's verify that Python gives the same result:

In [7]:
```
z = v + s
print(z)
```

```
[-1  3]
```

So what does that look like on our plot?

In [8]:
```
vecs = np.array([v,s,z])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['r', 'b', 'g'], scale=10)
plt.show()
```



So what's going on here? Well, we added the dimensions of **s** to the dimensions of **v** to describe a new vector **z**. Let's break that down:

- The dimensions of **v** are (2,1), so from our starting point we move 2 units in the *x* dimension (across to the right) and 1 unit in the *y* dimension (up). In the plot, if you start at the (0,0) position, this is shown as the red arrow.
- Then we're adding **s**, which has dimension values (-3, 2), so we move -3 units in the *x* dimension (across to the left, because it's a negative number) and then 2 units in the *y* dimension (up). On the plot, if you start at the head of the red arrow and make these moves, you'll end up at the head of the green arrow, which represents **z**.

The same is true if you perform the addition operation the other way around and add **v** to **s**, the steps to create **s** are described by the blue arrow, and if you use that as the starting point for **v**, you'll end up at the head of the green arrow, which represents **z**.

Note on the plot that if you simply moved the tail of the blue arrow so that it started at the head of red arrow, its head would end up in the same place as the head of the green arrow; and the same would be true if you moved tail of the red arrow to the head of the blue arrow.

Vector Multiplication

Vector multiplication can be performed in three ways:

- Scalar Multiplication
- Dot Product Multiplication
- Cross Product Multiplication

Scalar Multiplication

Let's start with *scalar* multiplication - in other words, multiplying a vector by a single numeric value.

Suppose I want to multiply my vector by 2, which I could write like this:

$$\vec{w} = 2\vec{v}$$

Note that the result of this calculation is a new vector named **w**. So how would we calculate this? Recall that **v** is defined like this:

$$\vec{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

To calculate 2v, we simply need to apply the operation to each dimension value in the vector matrix, like this:

$$\vec{w} = \begin{bmatrix} 2 \cdot 2 \\ 2 \cdot 1 \end{bmatrix}$$

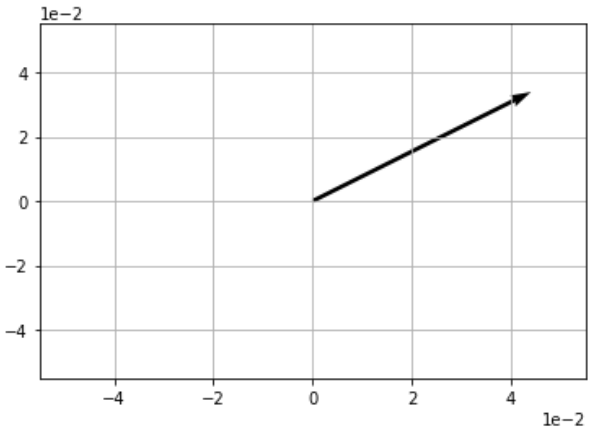
Which gives us the following result:

$$\vec{w} = \begin{bmatrix} 2 \cdot 2 \\ 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

In Python, you can apply these sort of matrix operations directly to numpy arrays, so we can simply calculate **w** like this:

In [1]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import math

v = np.array([2,1])

w = 2 * v
print(w)

# Plot w
origin = [0], [0]
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, *w, scale=10)
plt.show()
```

```
[4 2]
```



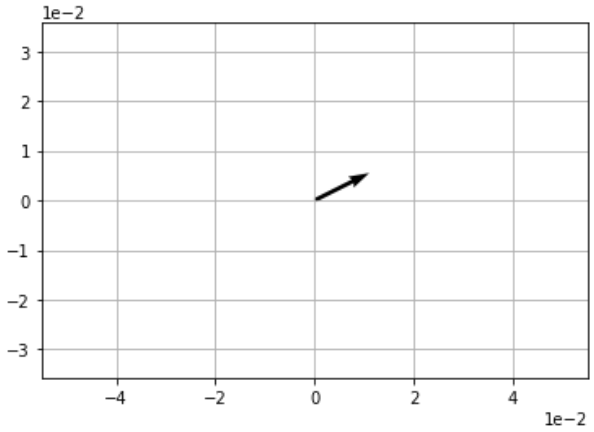
The same approach is taken for scalar division.

Try it for yourself - use the cell below to calculate a new vector named **b** based on the following definition:

$$\vec{b} = \frac{\vec{v}}{2}$$

In [2]:
```
b = v / 2
print(b)

# Plot b
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, *b, scale=10)
plt.show()
```

```
[1.  0.5]
```



Dot Product Multiplication

So we've seen how to multiply a vector by a scalar. How about multiplying two vectors together? There are actually two ways to do this depending on whether you want the result to be a *scalar product* (in other words, a number) or a *vector product* (a vector).

To get a scalar product, we calculate the *dot product*. This takes a similar approach to multiplying a vector by a scalar, except that it multiplies each component pair of the vectors and sums the results. To indicate that we are performing a dot product operation, we use the • operator:

$$\vec{v} \cdot \vec{s} = (v_1 \cdot s_1) + (v_2 \cdot s_2) \ldots + (v_n \cdot s_n)$$

So for our vectors **v** (2,1) and **s** (-3,2), our calculation looks like this:

$$\vec{v} \cdot \vec{s} = (2 \cdot -3) + (1 \cdot 2) = -6 + 2 = -4$$

So the dot product, or scalar product, of **v • s** is **-4**.

In Python, you can use the *numpy.**dot*** function to calculate the dot product of two vector arrays:

In [3]:
```
import numpy as np

v = np.array([2,1])
s = np.array([-3,2])
d = np.dot(v,s)
print (d)
```

```
-4
```

In Python 3.5 and later, you can also use the @ operator to calculate the dot product:

In [4]:
```
import numpy as np

v = np.array([2,1])
s = np.array([-3,2])
d = v @ s
print (d)
```

```
-4
```

The Cosine Rule

An useful property of vector dot product multiplication is that we can use it to calculate the cosine of the angle between two vectors. We could write the dot products as:

$$\vec{v} \cdot \vec{s} = \|\vec{v}\|\|\vec{s}\| \cos(\theta)$$

Which we can rearrange as:

$$\cos(\theta) = \frac{\vec{v} \cdot \vec{s}}{\|\vec{v}\|\|\vec{s}\|}$$

So for our vectors **v** (2,1) and **s** (-3,2), our calculation looks like this:

$$\cos(\theta) = \frac{(2 \cdot -3) + (-3 \cdot 2)}{\sqrt{2^2 + 1^2} \times \sqrt{-3^2 + 2^2}}$$

So:

$$\cos(\theta) = \frac{-4}{8.0622577483}$$

Which calculates to:

$$\cos(\theta) = -0.496138938357$$

So:

$$\theta \approx 119.74$$

Here's that calculation in Python:

In [5]:
```
import math
import numpy as np

# define our vectors
v = np.array([2,1])
s = np.array([-3,2])

# get the magnitudes
vMag = np.linalg.norm(v)
sMag = np.linalg.norm(s)

# calculate the cosine of theta
cos = (v @ s) / (vMag * sMag)

# so theta (in degrees) is:
theta = math.degrees(math.acos(cos))

print(theta)
```

```
119.74488129694222
```

Cross Product Multiplication

To get the *vector product* of multipying two vectors together, you must calculate the *cross product*. The result of this is a new vector that is at right angles to both the other vectors in 3D Euclidean space. This means that the cross-product only really makes sense when working with vectors that contain three components.

For example, let's suppose we have the following vectors:

$$\vec{p} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \quad \vec{q} = \begin{bmatrix} 1 \\ 2 \\ -2 \end{bmatrix}$$

To calculate the cross product of these vectors, written as **p** x **q**, we need to create a new vector (let's call it **r**) with three components (r_1, r_2, and r_3). The values for these components are calculated like this:

$$\begin{aligned} r_1 &= p_2 q_3 - p_3 q_2 \\ r_2 &= p_3 q_1 - p_1 q_3 \\ r_3 &= p_1 q_2 - p_2 q_1 \end{aligned}$$

So in our case:

$$\vec{r} = \vec{p} \times \vec{q} = \begin{bmatrix} (3 \cdot -2) - (1 \cdot 2) \\ (1 \cdot 1) - (2 \cdot -2) \\ (2 \cdot 2) - (3 \cdot 1) \end{bmatrix} = \begin{bmatrix} -6 - 2 \\ 1 - -4 \\ 4 - 3 \end{bmatrix} = \begin{bmatrix} -8 \\ 5 \\ 1 \end{bmatrix}$$

In Python, you can use the *numpy.**cross*** function to calculate the cross product of two vector arrays:

In [6]:
```
import numpy as np

p = np.array([2,3,1])
q = np.array([1,2,-2])
r = np.cross(p,q)
print (r)
```

```
[-8  5  1]
```

Introduction to Matrices

In general terms, a matrix is an array of numbers that are arranged into rows and columns.

Matrices and Matrix Notation

A matrix arranges numbers into rows and columns, like this:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Note that matrices are generally named as a capital letter. We refer to the *elements* of the matrix using the lower case equivalent with a subscript row and column indicator, like this:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

In Python, you can define a matrix as a 2-dimensional *numpy.*array***, like this:

In [1]:
```
import numpy as np

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

```
[[1 2 3]
 [4 5 6]]
```

You can also use the *numpy.*matrix**type, which is a specialist subclass of *array***:

In [2]:
```
import numpy as np

M = np.matrix([[1,2,3],
               [4,5,6]])
print (M)
```

```
[[1 2 3]
 [4 5 6]]
```

There are some differences in behavior between ***array*** and ***matrix*** types - particularly with regards to multiplication (which we'll explore later). You can use either, but most experienced Python programmers who need to work with both vectors and matrices tend to prefer the ***array*** type for consistency.

Matrix Operations

Matrices support common arithmetic operations.

Adding Matrices

To add two matrices of the same size together, just add the corresponding elements in each matrix:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}6 & 5 & 4\\3 & 2 & 1\end{bmatrix} = \begin{bmatrix}7 & 7 & 7\\7 & 7 & 7\end{bmatrix}$$

In this example, we're adding two matrices (let's call them ***A*** and ***B***). Each matrix has two rows of three columns (so we describe them as 2x3 matrices). Adding these will create a new matrix of the same dimensions with the values $a_{1,1} + b_{1,1}$, $a_{1,2} + b_{1,2}$, $a_{1,3} + b_{1,3}$,$a_{2,1} + b_{2,1}$, $a_{2,2} + b_{2,2}$, and $a_{2,3} + b_{2,3}$. In this instance, each pair of corresponding elements(1 and 6, 2, and 5, 3 and 4, etc.) adds up to 7.

Let's try that with Python:

In [3]:
```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
B = np.array([[6,5,4],
              [3,2,1]])
print(A + B)
```

```
[[7 7 7]
 [7 7 7]]
```

Subtracting Matrices

Matrix subtraction works similarly to matrix addition:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} - \begin{bmatrix}6 & 5 & 4\\3 & 2 & 1\end{bmatrix} = \begin{bmatrix}-5 & -3 & -1\\1 & 3 & 5\end{bmatrix}$$

Here's the Python code to do this:

In [4]:
```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
B = np.array([[6,5,4],
              [3,2,1]])
print (A - B)
```

```
[[-5 -3 -1]
 [ 1  3  5]]
```

Conformability

In the previous examples, we were able to add and subtract the matrices, because the *operands* (the matrices we are operating on) are ***conformable*** for the specific operation (in this case, addition or subtraction). To be conformable for addition and subtraction, the operands must have the same number of rows and columns. There are different conformability requirements for other operations, such as multiplication; which we'll explore later.

Negative Matrices

The nagative of a matrix, is just a matrix with the sign of each element reversed:

$$C = \begin{bmatrix} -5 & -3 & -1 \\ 1 & 3 & 5 \end{bmatrix}$$

$$-C = \begin{bmatrix} 5 & 3 & 1 \\ -1 & -3 & -5 \end{bmatrix}$$

Let's see that with Python:

In [5]:
```
import numpy as np

C = np.array([[-5,-3,-1],
              [1,3,5]])
print (C)
print (-C)
```

```
[[-5 -3 -1]
 [ 1  3  5]]
[[ 5  3  1]
 [-1 -3 -5]]
```

Matrix Transposition

You can *transpose* a matrix, that is switch the orientation of its rows and columns. You indicate this with a superscript **T**, like this:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

In Python, both *numpy.*array*** and *numpy.**matrix*** have a **T** function:

In [6]:
```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
print(A.T)
```

```
[[1 4]
 [2 5]
 [3 6]]
```

More Matrices

This notebook continues your exploration of matrices.

Matrix Multiplication

Multiplying matrices is a little more complex than the operations we've seen so far. There are two cases to consider, *scalar multiplication* (multiplying a matrix by a single number), and *dot product matrix multiplication* (multiplying a matrix by another matrix.

Scalar Multiplication

To multiply a matrix by a scalar value, you just multiply each element by the scalar to produce a new matrix:

$$2 \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix}$$

In Python, you perform this calculation using the ***** operator:

In [1]:
```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
print(2 * A)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

Dot Product Matrix Multiplication

To mulitply two matrices together, you need to calculate the *dot product* of rows and columns. This means multiplying each of the elements in each row of the first matrix by each of the elements in each column of the second matrix and adding the results. We perform this operation by applying the *RC* rule - always multiplying ***R***ows by ***C***olumns. For this to work, the number of ***columns*** in the first matrix must be the same as the number of ***rows*** in the second matrix so that the matrices are *conformable* for the dot product operation.

Sounds confusing, right?

Let's look at an example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 9 & 8 \\ 7 & 6 \\ 5 & 4 \end{bmatrix}$$

Note that the first matrix is 2x3, and the second matrix is 3x2. The important thing here is that the first matrix has two rows, and the second matrix has two columns. To perform the multiplication, we first take the dot product of the first ***row*** of the first matrix (1,2,3) and the first ***column*** of the second matrix (9,7,5):

$$(1, 2, 3) \cdot (9, 7, 5) = (1 \times 9) + (2 \times 7) + (3 \times 5) = 38$$

In our resulting matrix (which will always have the same number of ***rows*** as the first matrix, and the same number of ***columns*** as the second matrix), we can enter this into the first row and first column element:

$$\begin{bmatrix} 38 & ? \\ ? & ? \end{bmatrix}$$

Now we can take the dot product of the first row of the first matrix and the second column of the second matrix:

$$(1, 2, 3) \cdot (8, 6, 4) = (1 \times 8) + (2 \times 6) + (3 \times 4) = 32$$

Let's add that to our resulting matrix in the first row and second column element:

$$\begin{bmatrix} 38 & 32 \\ ? & ? \end{bmatrix}$$

Now we can repeat this process for the second row of the first matrix and the first column of the second matrix:

$$(4, 5, 6) \cdot (9, 7, 5) = (4 \times 9) + (5 \times 7) + (6 \times 5) = 101$$

Which fills in the next element in the result:

$$\begin{bmatrix} 38 & 32 \\ 101 & ? \end{bmatrix}$$

Finally, we get the dot product for the second row of the first matrix and the second column of the second matrix:

$$(4, 5, 6) \cdot (8, 6, 4) = (4 \times 8) + (5 \times 6) + (6 \times 4) = 86$$

Giving us:

$$\begin{bmatrix} 38 & 32 \\ 101 & 86 \end{bmatrix}$$

In Python, you can use the *numpy.**dot*** function or the ****@ operator to multiply matrices and two-dimensional arrays:

In [2]:
```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6]])
B = np.array([[9,8],
              [7,6],
              [5,4]])
print(np.dot(A,B))
print(A @ B)
```

```
[[ 38  32]
 [101  86]]
[[ 38  32]
 [101  86]]
```

This is one case where there is a difference in behavior between *numpy.*array*** and *numpy.**matrix***, You can also use a regular multiplication (***) operator with a matrix, but not with an array:

In [3]:
```
import numpy as np

A = np.matrix([[1,2,3]
               ,[4,5,6]])
B = np.matrix([[9,8],
               [7,6],
               [5,4]])
print(A * B)
```

```
[[ 38  32]
 [101  86]]
```

Note that, unlike with multiplication of regular scalar numbers, the order of the operands in a multiplication operation is significant. For scalar numbers, the *commmutative law* of multiplication applies, so for example:

$$2 \times 4 = 4 \times 2$$

With matrix multiplication, things are different, for example:

$$\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix} \neq \begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Run the following Python code to test this:

In [4]:
```
import numpy as np

A = np.array([[2,4],
              [6,8]])
B = np.array([[1,3],
              [5,7]])
print(A @ B)
print(B @ A)
```

```
[[22 34]
 [46 74]]
[[20 28]
 [52 76]]
```

Identity Matrices

An *identity* matrix (usually indicated by a capital **I**) is the equivalent in matrix terms of the number **1**. It always has the same number of rows as columns, and it has the value **1** in the diagonal element positions $I_{1,1}$, $I_{2,2}$, etc; and 0 in all other element positions. Here's an example of a 3x3 identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying any matrix by an identity matrix is the same as multiplying a number by 1; the result is the same as the original value:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

If you doubt me, try the following Python code!

In [5]:
```
import numpy as np

A = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
B = np.array([[1,0,0],
              [0,1,0],
              [0,0,1]])
print(A @ B)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Matrix Division

You can't actually divide by a matrix; but when you want to divide matrices, you can take advantage of the fact that division by a given number is the same as multiplication by the reciprocal of that number. For example:

$$6 \div 3 = \frac{1}{3} \times 6$$

In this case, $^1/_3$ is the reciprocal of 3 (which as a fraction is $^3/_1$ - we "flip" the numerator and denominator to get the reciprocal). You can also write $^1/_3$ as 3^{-1}.

Inverse of a Matrix

For matrix division, we use a related idea; we multiply by the *inverse* of a matrix:

$$A \div B = A \cdot B^{-1}$$

The inverse of B is B^{-1} as long as the following equation is true:

$$B \cdot B^{-1} = B^{-1} \cdot B = I$$

I, you may recall, is an *identity* matrix; the matrix equivalent of 1.

So how do you calculate the inverse of a matrix? For a 2x2 matrix, you can follow this formula:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

What happened there?

- We swapped the positions of *a* and *d*
- We changed the signs of *b* and *c*
- We multiplied the resulting matrix by 1 over the *determinant* of the matrix (*ad-bc*)

Let's try with some actual numbers:

$$\begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}^{-1} = \frac{1}{(6 \times 2) - (2 \times 1)} \begin{bmatrix} 2 & -2 \\ -1 & 6 \end{bmatrix}$$

So:

$$\begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}^{-1} = \frac{1}{10} \begin{bmatrix} 2 & -2 \\ -1 & 6 \end{bmatrix}$$

Which gives us the result:

$$\begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}^{-1} = \begin{bmatrix} 0.2 & -0.2 \\ -0.1 & 0.6 \end{bmatrix}$$

To check this, we can multiply the original matrix by its inverse to see if we get an identity matrix. This makes sense if you think about it; in the same way that 3 x $^1/_3$ = 1, a matrix multiplied by its inverse results in an identity matrix:

$$\begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0.2 & -0.2 \\ -0.1 & 0.6 \end{bmatrix}$$

$$= \begin{bmatrix} (6 \times 0.2) + (2 \times -0.1) & (6 \times -0.2) + (2 \times 0.6) \\ (1 \times 0.2) + (2 \times -0.1) & (1 \times -0.2) + (2 \times 0.6) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

In [6]:
```
import numpy as np

B = np.array([[6,2],
              [1,2]])

print(np.linalg.inv(B))
```

```
[[ 0.2 -0.2]
 [-0.1  0.6]]
```

Additionally, the *matrix* type has an ***I*** method that returns the inverse matrix:

In [7]:
```
import numpy as np

B = np.matrix([[6,2],
               [1,2]])

print(B.I)
```

```
[[ 0.2 -0.2]
 [-0.1  0.6]]
```

For larger matrices, the process to calculate the inverse is more complex. Let's explore an example based on the following matrix:

$$\begin{bmatrix} 4 & 2 & 2 \\ 6 & 2 & 4 \\ 2 & 2 & 8 \end{bmatrix}$$

The process to find the inverse consists of the following steps:

1: Create a matrix of *minors* by calculating the *determinant* for each element in the matrix based on the elements that are not in the same row or column; like this:

$$\begin{bmatrix} \color{blue}{4} & 2 & 2 \\ 6 & \color{red}{2} & \color{red}{4} \\ 2 & \color{red}{2} & \color{red}{8} \end{bmatrix} \quad (2 \times 8) - (4 \times 2) = 8 \quad \begin{bmatrix} 8 & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} 4 & \color{blue}{2} & 2 \\ \color{red}{6} & 2 & \color{red}{4} \\ \color{red}{2} & 2 & \color{red}{8} \end{bmatrix} \quad (6 \times 8) - (4 \times 2) = 40 \quad \begin{bmatrix} 8 & 40 & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} 4 & 2 & \color{blue}{2} \\ \color{red}{6} & \color{red}{2} & 4 \\ \color{red}{2} & \color{red}{2} & 8 \end{bmatrix} \quad (6 \times 2) - (2 \times 2) = 8 \quad \begin{bmatrix} 8 & 40 & 8 \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} 4 & \color{red}{2} & \color{red}{2} \\ \color{blue}{6} & 2 & 4 \\ 2 & \color{red}{2} & \color{red}{8} \end{bmatrix} \quad (2 \times 8) - (2 \times 2) = 12 \quad \begin{bmatrix} 8 & 40 & 8 \\ 12 & ? & ? \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} \color{red}{4} & 2 & \color{red}{2} \\ 6 & \color{blue}{2} & 4 \\ \color{red}{2} & 2 & \color{red}{8} \end{bmatrix} \quad (4 \times 8) - (2 \times 2) = 28 \quad \begin{bmatrix} 8 & 40 & 8 \\ 12 & 28 & ? \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} \color{red}{4} & \color{red}{2} & 2 \\ 6 & 2 & \color{blue}{4} \\ \color{red}{2} & \color{red}{2} & 8 \end{bmatrix} \quad (4 \times 2) - (2 \times 2) = 4 \quad \begin{bmatrix} 8 & 40 & 8 \\ 12 & 28 & 4 \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} \color{red}{4} & 2 & \color{red}{2} \\ 6 & \color{red}{2} & 4 \\ \color{blue}{2} & 2 & 8 \end{bmatrix} \quad (2 \times 4) - (2 \times 2) = 4 \quad \begin{bmatrix} 8 & 40 & 8 \\ 12 & 28 & 4 \\ 4 & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} \color{red}{4} & 2 & \color{red}{2} \\ \color{red}{6} & 2 & \color{red}{4} \\ 2 & \color{blue}{2} & 8 \end{bmatrix} \quad (4 \times 4) - (2 \times 6) = 4 \quad \begin{bmatrix} 8 & 40 & 8 \\ 12 & 28 & 4 \\ 4 & 4 & ? \end{bmatrix}$$

$$\begin{bmatrix} \color{red}{4} & \color{red}{2} & 2 \\ \color{red}{6} & 2 & \color{blue}{4} \\ 2 & 2 & 8 \end{bmatrix} \quad (4 \times 2) - (2 \times 6) = -4 \quad \begin{bmatrix} 8 & 40 & 8 \\ 12 & 28 & 4 \\ 4 & 4 & -4 \end{bmatrix}$$

2: Apply *cofactors* to the matrix by switching the sign of every alternate element in the matrix of minors:

$$\begin{bmatrix} 8 & -40 & 8 \\ -12 & 28 & -4 \\ 4 & -4 & -4 \end{bmatrix}$$

3: Adjugate by transposing elements diagonally

In [8]:
```
import numpy as np

B = np.array([[4,2,2],
              [6,2,4],
              [2,2,8]])

print(np.linalg.inv(B))
```

```
[[-0.25   0.375 -0.125]
 [ 1.25  -0.875  0.125]
 [-0.25   0.125  0.125]]
```

Multiplying by an Inverse Matrix

Now that you know how to calculate an inverse matrix, you can use that knowledge to multiply the inverse of a matrix by another matrix as an alternative to division:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 6 & 2 \\ 1 & 2 \end{bmatrix}^{-1}$$

$$= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 0.2 & -0.2 \\ -0.1 & 0.6 \end{bmatrix}$$

$$= \begin{bmatrix} (1 \times 0.2) + (2 \times -0.1) & (1 \times -0.2) + (2 \times 0.6) \\ (3 \times 0.2) + (4 \times -0.1) & (3 \times -0.2) + (4 \times 0.6) \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 \\ 0.2 & 1.8 \end{bmatrix}$$

Here's the Python code to calculate this:

In [9]:
```
import numpy as np

A = np.array([[1,2],
              [3,4]])

B = np.array([[6,2],
              [1,2]])


C = A @ np.linalg.inv(B)

print(C)
```

```
[[0.  1. ]
 [0.2 1.8]]
```

Solving Systems of Equations with Matrices

One of the great things about matrices, is that they can help us solve systems of equations. For example, consider the following system of equations:

$$2x + 4y = 18$$
$$6x + 2y = 34$$

We can write this in matrix form, like this:

$$\begin{bmatrix} 2 & 4 \\ 6 & 2 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 18 \\ 34 \end{bmatrix}$$

Note that the variables (***x*** and ***y***) are arranged as a column in one matrix, which is multiplied by a matrix containing the coefficients to produce as matrix containing the results. If you calculate the dot product on the left side, you can see clearly that this represents the original equations:

$$\begin{bmatrix} 2x + 4y \\ 6x + 2y \end{bmatrix} = \begin{bmatrix} 18 \\ 34 \end{bmatrix}$$

Now. let's name our matrices so we can better understand what comes next:

$$A = \begin{bmatrix} 2 & 4 \\ 6 & 2 \end{bmatrix} \quad X = \begin{bmatrix} x \\ y \end{bmatrix} \quad B = \begin{bmatrix} 18 \\ 34 \end{bmatrix}$$

We already know that ***A*** • ***X*** = ***B***, which arithmetically means that ***X*** = ***B*** ÷ ***A***. Since we can't actually divide by a matrix, we need to multiply by the inverse; so we can find the values for our variables (*X*) like this: ***X*** = ***A*$^{-1}$** • ***B***

So, first we need the inverse of A:

$$\begin{bmatrix} 2 & 4 \\ 6 & 2 \end{bmatrix}^{-1} = \frac{1}{(2 \times 2) - (4 \times 6)} \begin{bmatrix} 2 & -4 \\ -6 & 2 \end{bmatrix}$$
$$= \frac{1}{-20} \begin{bmatrix} 2 & -4 \\ -6 & 2 \end{bmatrix}$$
$$= \begin{bmatrix} -0.1 & 0.2 \\ 0.3 & -0.1 \end{bmatrix}$$

Then we just multiply this with B:

$$X = \begin{bmatrix} -0.1 & 0.2 \\ 0.3 & -0.1 \end{bmatrix} \cdot \begin{bmatrix} 18 \\ 34 \end{bmatrix}$$
$$X = \begin{bmatrix} (-0.1 \times 18) + (0.2 \times 34) \\ (0.3 \times 18) + (-0.1 \times 34) \end{bmatrix}$$
$$X = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

The resulting matrix (*X*) contains the values for our *x* and *y* variables, and we can check these by plugging them into the original equations:

$$(2 \times 5) + (4 \times 2) = 18$$
$$(6 \times 5) + (2 \times 2) = 34$$

These of course simplify to:

$$10 + 8 = 18$$

In [10]:
```
import numpy as np

A = np.array([[2,4],
              [6,2]])

B = np.array([[18],
              [34]])

C = np.linalg.inv(A) @ B

print(C)
```

```
[[5.]
 [2.]]
```

Transformations, Eigenvectors, and Eigenvalues

Matrices and vectors are used together to manipulate spatial dimensions. This has a lot of applications, including the mathematical generation of 3D computer graphics, geometric modeling, and the training and optimization of machine learning algorithms. We're not going to cover the subject exhaustively here; but we'll focus on a few key concepts that are useful to know when you plan to work with machine learning.

Linear Transformations

You can manipulate a vector by multiplying it with a matrix. The matrix acts a function that operates on an input vector to produce a vector output. Specifically, matrix multiplications of vectors are *linear transformations* that transform the input vector into the output vector.

For example, consider this matrix ***A*** and vector ***v***:

$$A = \begin{bmatrix}2 & 3\\5 & 2\end{bmatrix} \quad \vec{v} = \begin{bmatrix}1\\2\end{bmatrix}$$

We can define a transformation ***T*** like this:

$$T(\vec{v}) = A\vec{v}$$

To perform this transformation, we simply calculate the dot product by applying the *RC* rule; multiplying each row of the matrix by the single column of the vector:

$$\begin{bmatrix}2 & 3\\5 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}8\\9\end{bmatrix}$$

Here's the calculation in Python:

In [1]:
```
import numpy as np

v = np.array([1,2])
A = np.array([[2,3],
              [5,2]])

t = A@v
print (t)
```

```
[8 9]
```

In this case, both the input vector and the output vector have 2 components - in other words, the transformation takes a 2-dimensional vector and produces a new 2-dimensional vector; which we can indicate like this:

$$T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

Note that the output vector may have a different number of dimensions from the input vector; so the matrix function might transform the vector from one space to another - or in notation, $\mathbb{R}^n$ -> $\mathbb{R}^m$.

For example, let's redefine matrix ***A***, while retaining our original definition of vector ***v***:

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 2 \\ 1 & 1 \end{bmatrix} \quad \vec{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Now if we once again define ***T*** like this:

$$T(\vec{v}) = A\vec{v}$$

We apply the transformation like this:

$$\begin{bmatrix} 2 & 3 \\ 5 & 2 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 8 \\ 9 \\ 3 \end{bmatrix}$$

So now, our transformation transforms the vector from 2-dimensional space to 3-dimensional space:

$$T : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

Here it is in Python:

In [2]:
```
import numpy as np
v = np.array([1,2])
A = np.array([[2,3],
              [5,2],
              [1,1]])

t = A@v
print (t)
```

```
[8 9 3]
```

In [3]:
```
import numpy as np
v = np.array([1,2])
A = np.array([[1,2],
              [2,1]])

t = A@v
print (t)
```

```
[5 4]
```

Transformations of Magnitude and Amplitude

When you multiply a vector by a matrix, you transform it in at least one of the following two ways:

- Scale the length (*magnitude*) of the matrix to make it longer or shorter
- Change the direction (*amplitude*) of the matrix

For example consider the following matrix and vector:

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \vec{v} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

As before, we transform the vector ***v*** by multiplying it with the matrix ***A***:

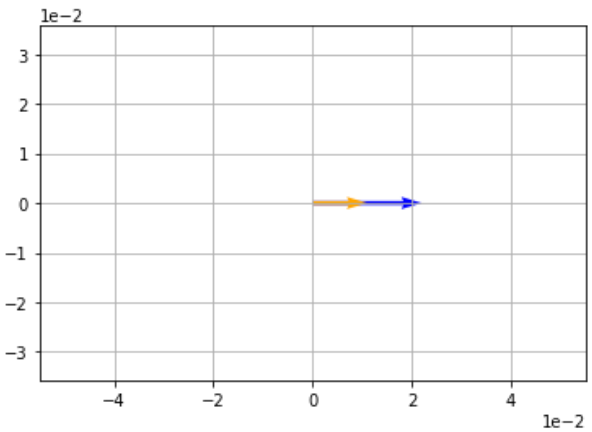
$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

In this case, the resulting vector has changed in length (*magnitude*), but has not changed its direction (*amplitude*).

Let's visualize that in Python:

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

v = np.array([1,0])
A = np.array([[2,0],
              [0,2]])

t = A@v
print (t)

# Plot v and t
vecs = np.array([t,v])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
```

```
[2 0]
```



The original vector ***v*** is shown in orange, and the transformed vector ***t*** is shown in blue - note that ***t*** has the same direction (*amplitude*) as ***v*** but a greater length (*magnitude*).

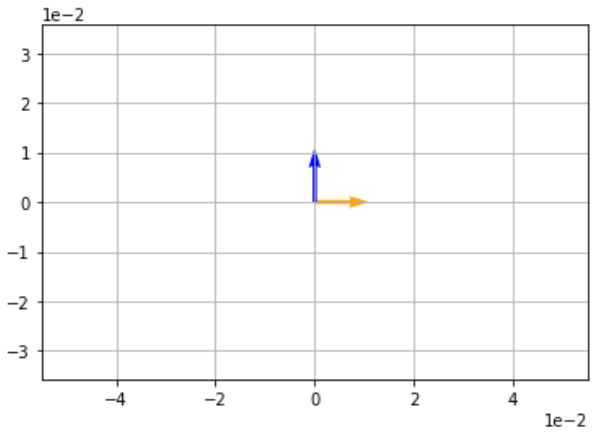
Now let's use a different matrix to transform the vector ***v***:

$$\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$$

This time, the resulting vector has been changed to a different amplitude, but has the same magnitude.

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

v = np.array([1,0])
A = np.array([[0,-1],
              [1,0]])

t = A@v
print (t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'], scale=10)
plt.show()
```

```
[0 1]
```



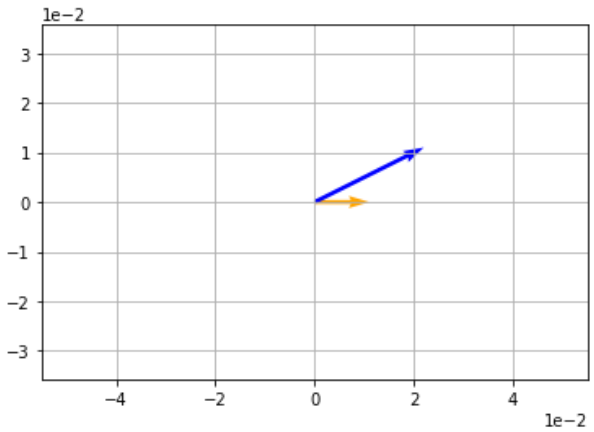
Now let's see change the matrix one more time:

$$\begin{bmatrix}2 & 1\\1 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\1\end{bmatrix}$$

Now our resulting vector has been transformed to a new amplitude *and* magnitude - the transformation has affected both direction and scale.

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

v = np.array([1,0])
A = np.array([[2,1],
              [1,2]])

t = A@v
print (t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'], scale=10)
plt.show()
```

```
[2 1]
```



Afine Transformations

An Afine transformation multiplies a vector by a matrix and adds an offset vector, sometimes referred to as *bias*; like this:

$$T(\vec{v}) = A\vec{v} + \vec{b}$$

For example:

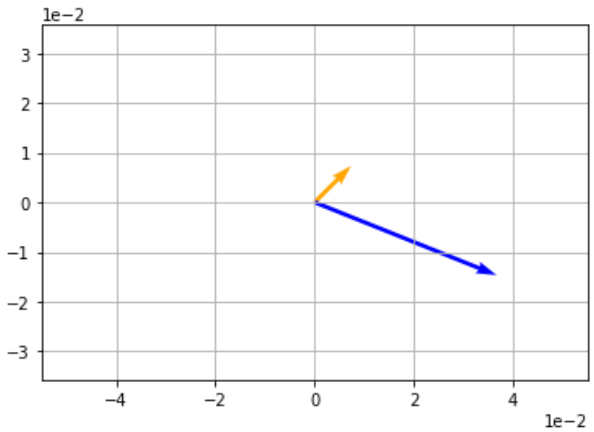
$$\begin{bmatrix} 5 & 2 \\ 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ -6 \end{bmatrix} = \begin{bmatrix} 5 \\ -2 \end{bmatrix}$$

This kind of transformation is actually the basis of linear regression, which is a core foundation for machine learning. The matrix defines the *features*, the first vector is the *coefficients*, and the bias vector is the *intercept*.

here's an example of an Afine transformation in Python:

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

v = np.array([1,1])
A = np.array([[5,2],
              [3,1]])
b = np.array([-2,-6])

t = A@v + b
print (t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'blue'], scale=15)
plt.show()
```

```
[ 5 -2]
```



Eigenvectors and Eigenvalues

So we can see that when you transform a vector using a matrix, we change its direction, length, or both. When the transformation only affects scale (in other words, the output vector has a different magnitude but the same amplitude as the input vector), the matrix multiplication for the transformation is the equivalent operation as some scalar multiplication of the vector.

For example, earlier we examined the following transformation that dot-mulitplies a vector by a matrix:

$$\begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

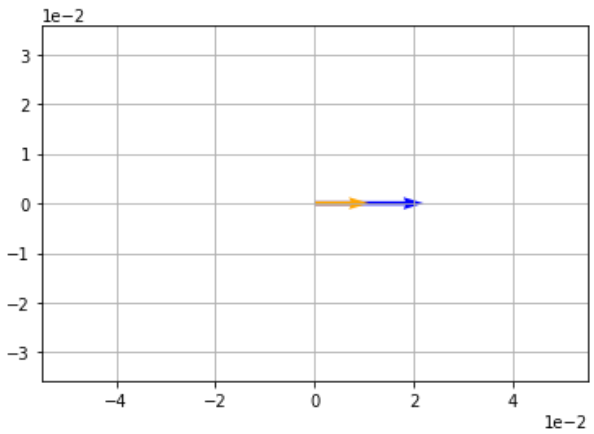
You can achieve the same result by mulitplying the vector by the scalar value ***2***:

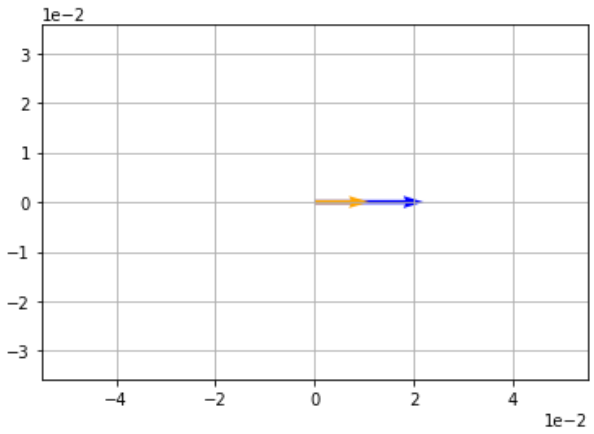
$$2 \times \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

The following python performs both of these calculation and shows the results, which are identical.

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

v = np.array([1,0])
A = np.array([[2,0],
              [0,2]])

t1 = A@v
print (t1)
t2 = 2*v
print (t2)

fig = plt.figure()
a=fig.add_subplot(1,1,1)
# Plot v and t1
vecs = np.array([t1,v])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
a=fig.add_subplot(1,2,1)
# Plot v and t2
vecs = np.array([t2,v])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
```

```
[2 0]
[2 0]
```





In cases like these, where a matrix transformation is the equivelent of a scalar-vector multiplication, the scalar-vector pairs that correspond to the matrix are known respectively as eigenvalues and eigenvectors. We generally indicate eigenvalues using the Greek letter lambda (λ), and the formula that defines eigenvalues and eigenvectors with respect to a transformation is:

$$T(\vec{v}) = \lambda\vec{v}$$

Where the vector ***v*** is an eigenvector and the value ***λ*** is an eigenvalue for transformation ***T***.

When the transformation ***T*** is represented as a matrix multiplication, as in this case where the transformation is represented by matrix ***A***:

$$T(\vec{v}) = A\vec{v} = \lambda\vec{v}$$

Then ***v*** is an eigenvector and ***λ*** is an eigenvalue of ***A***.

A matrix can have multiple eigenvector-eigenvalue pairs, and you can calculate them manually. However, it's generally easier to use a tool or programming language. For example, in Python you can use the ***linalg.eig*** function, which returns an array of eigenvalues and a matrix of the corresponding eigenvectors for the specified matrix.

Here's an example that returns the eigenvalue and eigenvector pairs for the following matrix:

$$A = \begin{bmatrix}2 & 0\\0 & 3\end{bmatrix}$$

In [9]:
```
import numpy as np
A = np.array([[2,0],
              [0,3]])
eVals, eVecs = np.linalg.eig(A)
print(eVals)
print(eVecs)
```

```
[2. 3.]
[[1. 0.]
 [0. 1.]]
```

So there are two eigenvalue-eigenvector pairs for this matrix, as shown here:

$$\lambda_1 = 2, \vec{v_1} = \begin{bmatrix}1\\0\end{bmatrix} \qquad \lambda_2 = 3, \vec{v_2} = \begin{bmatrix}0\\1\end{bmatrix}$$

Let's verify that multiplying each eigenvalue-eigenvector pair corresponds to the dot-product of the eigenvector and the matrix. Here's the first pair:

$$2 \times \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix} \quad and \quad \begin{bmatrix}2 & 0\\0 & 3\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

So far so good. Now let's check the second pair:

$$3 \times \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\3\end{bmatrix} \quad and \quad \begin{bmatrix}2 & 0\\0 & 3\end{bmatrix} \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\3\end{bmatrix}$$

So our eigenvalue-eigenvector scalar multiplications do indeed correspond to our matrix-eigenvector dot-product transformations.

Here's the equivalent code in Python, using the ***eVals*** and ***eVecs*** variables you generated in the previous code cell:

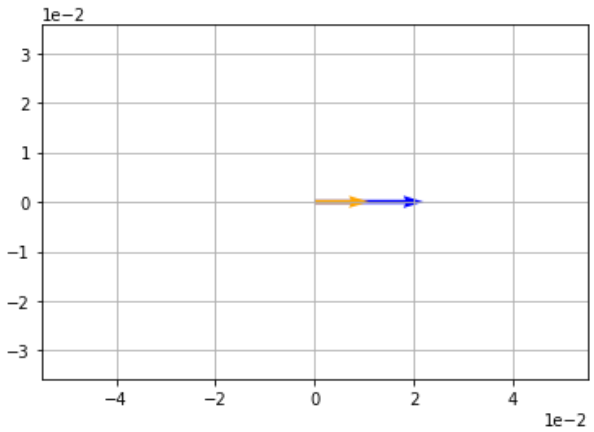
In [10]:
```
vec1 = eVecs[:,0]
lam1 = eVals[0]

print('Matrix A:')
print(A)
print('-------')

print('lam1: ' + str(lam1))
print ('v1: ' + str(vec1))
print ('Av1: ' + str(A@vec1))
print ('lam1 x v1: ' + str(lam1*vec1))

print('-------')

vec2 = eVecs[:,1]
lam2 = eVals[1]

print('lam2: ' + str(lam2))
print ('v2: ' + str(vec2))
print ('Av2: ' + str(A@vec2))
print ('lam2 x v2: ' + str(lam2*vec2))
```

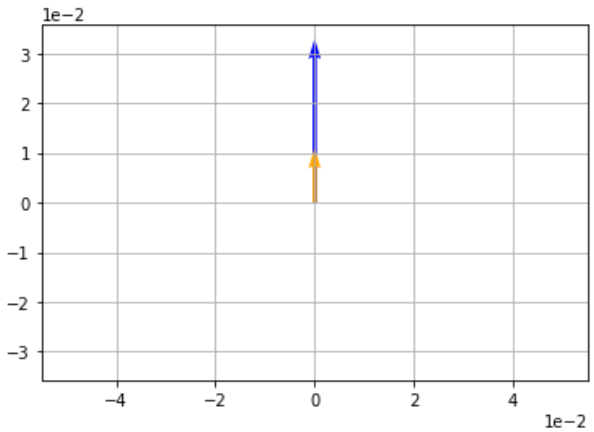
```
Matrix A:
[[2 0]
 [0 3]]
-------
lam1: 2.0
v1: [1. 0.]
Av1: [2. 0.]
lam1 x v1: [2. 0.]
-------
lam2: 3.0
v2: [0. 1.]
Av2: [0. 3.]
lam2 x v2: [0. 3.]
```

You can use the following code to visualize these transformations:

In [11]:
```
t1 = lam1*vec1
print (t1)
t2 = lam2*vec2
print (t2)

fig = plt.figure()
a=fig.add_subplot(1,1,1)
# Plot v and t1
vecs = np.array([t1,vec1])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
a=fig.add_subplot(1,2,1)
# Plot v and t2
vecs = np.array([t2,vec2])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
```

```
[2. 0.]
[0. 3.]
```





Similarly, earlier we examined the following matrix transformation:

$$\begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

And we saw that you can achieve the same result by mulitplying the vector by the scalar value ***2***:

$$2 \times \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

This works because the scalar value 2 and the vector (1,0) are an eigenvalue-eigenvector pair for this matrix.

Let's use Python to determine the eigenvalue-eigenvector pairs for this matrix:

In [12]:
```
import numpy as np
A = np.array([[2,0],
              [0,2]])
eVals, eVecs = np.linalg.eig(A)
print(eVals)
print(eVecs)
```

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

So once again, there are two eigenvalue-eigenvector pairs for this matrix, as shown here:

$$\lambda_1 = 2, \vec{v_1} = \begin{bmatrix}1\\0\end{bmatrix} \qquad \lambda_2 = 2, \vec{v_2} = \begin{bmatrix}0\\1\end{bmatrix}$$

Let's verify that multiplying each eigenvalue-eigenvector pair corresponds to the dot-product of the eigenvector and the matrix. Here's the first pair:

$$2 \times \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix} \quad \textit{and} \quad \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}2\\0\end{bmatrix}$$

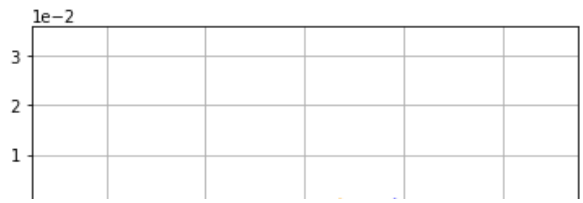
Well, we already knew that. Now let's check the second pair:

$$2 \times \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\2\end{bmatrix} \quad \textit{and} \quad \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix} \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\2\end{bmatrix}$$

Now let's use Pythonto verify and plot these transformations:

In [13]:
```
vec1 = eVecs[:,0]
lam1 = eVals[0]

print('Matrix A:')
print(A)
print('-------')

print('lam1: ' + str(lam1))
print ('v1: ' + str(vec1))
print ('Av1: ' + str(A@vec1))
print ('lam1 x v1: ' + str(lam1*vec1))

print('-------')

vec2 = eVecs[:,1]
lam2 = eVals[1]

print('lam2: ' + str(lam2))
print ('v2: ' + str(vec2))
print ('Av2: ' + str(A@vec2))
print ('lam2 x v2: ' + str(lam2*vec2))


# Plot the resulting vectors
t1 = lam1*vec1
t2 = lam2*vec2

fig = plt.figure()
a=fig.add_subplot(1,1,1)
# Plot v and t1
vecs = np.array([t1,vec1])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
a=fig.add_subplot(1,2,1)
# Plot v and t2
vecs = np.array([t2,vec2])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
```

```
Matrix A:
[[2 0]
 [0 2]]
-------
lam1: 2.0
v1: [1. 0.]
Av1: [2. 0.]
lam1 x v1: [2. 0.]
-------
lam2: 2.0
v2: [0. 1.]
Av2: [0. 2.]
lam2 x v2: [0. 2.]
```



Let's take a look at one more, slightly more complex example. Here's our matrix:

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Let's get the eigenvalue and eigenvector pairs:

In [14]:
```
import numpy as np

A = np.array([[2,1],
              [1,2]])

eVals, eVecs = np.linalg.eig(A)
print(eVals)
print(eVecs)
```

```
[3. 1.]
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

This time the eigenvalue-eigenvector pairs are:

$$\lambda_1 = 3, \vec{v_1} = \begin{bmatrix} 0.70710678 \\ 0.70710678 \end{bmatrix} \qquad \lambda_2 = 1, \vec{v_2} = \begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix}$$

So let's check the first pair:

$$3 \times \begin{bmatrix} 0.70710678 \\ 0.70710678 \end{bmatrix} = \begin{bmatrix} 2.12132034 \\ 2.12132034 \end{bmatrix} \quad and \quad \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0.70710678 \\ 0.70710678 \end{bmatrix} = \begin{bmatrix} 2.12132034 \\ 2.12132034 \end{bmatrix}$$

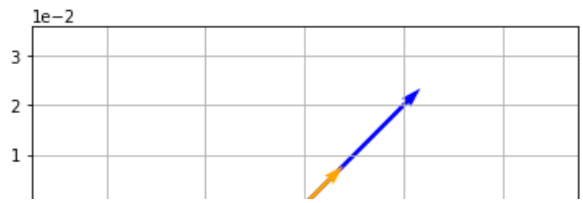
Now let's check the second pair:

$$1 \times \begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix} = \begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix} \quad and \quad \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix} = \begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix}$$

With more complex examples like this, it's generally easier to do it with Python:

In [15]:
```
vec1 = eVecs[:,0]
lam1 = eVals[0]

print('Matrix A:')
print(A)
print('-------')

print('lam1: ' + str(lam1))
print ('v1: ' + str(vec1))
print ('Av1: ' + str(A@vec1))
print ('lam1 x v1: ' + str(lam1*vec1))

print('-------')

vec2 = eVecs[:,1]
lam2 = eVals[1]

print('lam2: ' + str(lam2))
print ('v2: ' + str(vec2))
print ('Av2: ' + str(A@vec2))
print ('lam2 x v2: ' + str(lam2*vec2))


# Plot the results
t1 = lam1*vec1
t2 = lam2*vec2

fig = plt.figure()
a=fig.add_subplot(1,1,1)
# Plot v and t1
vecs = np.array([t1,vec1])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
a=fig.add_subplot(1,2,1)
# Plot v and t2
vecs = np.array([t2,vec2])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['blue', 'orange'], scale=10)
plt.show()
```

```
Matrix A:
[[2 1]
 [1 2]]
-------
lam1: 3.0
v1: [0.70710678 0.70710678]
Av1: [2.12132034 2.12132034]
lam1 x v1: [2.12132034 2.12132034]
-------
lam2: 1.0
v2: [-0.70710678  0.70710678]
Av2: [-0.70710678  0.70710678]
lam2 x v2: [-0.70710678  0.70710678]
```



Eigendecomposition

So we've learned a little about eigenvalues and eigenvectors; but you may be wondering what use they are. Well, one use for them is to help decompose transformation matrices.

Recall that previously we found that a matrix transformation of a vector changes its magnitude, amplitude, or both. Without getting too technical about it, we need to remember that vectors can exist in any spatial orientation, or *basis*; and the same transformation can be applied in different *bases*.

We can decompose a matrix using the following formula:

$$A = Q\Lambda Q^{-1}$$

Where ***A*** is a trasformation that can be applied to a vector in its current base, ***Q*** is a matrix of eigenvectors that defines a change of basis, and ***Λ*** is a matrix with eigenvalues on the diagonal that defines the same linear transformation as ***A*** in the base defined by ***Q***.

Let's look at these in some more detail. Consider this matrix:

$$A = \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}$$

Q is a matrix in which each column is an eigenvector of ***A***; which as we've seen previously, we can calculate using Python:

In [16]:
```
import numpy as np

A = np.array([[3,2],
              [1,0]])

l, Q = np.linalg.eig(A)
print(Q)
```

```
[[ 0.96276969 -0.48963374]
 [ 0.27032301  0.87192821]]
```

So for matrix ***A***, ***Q*** is the following matrix:

$$Q = \begin{bmatrix} 0.96276969 & -0.48963374 \\ 0.27032301 & 0.87192821 \end{bmatrix}$$

Λ is a matrix that contains the eigenvalues for ***A*** on the diagonal, with zeros in all other elements; so for a 2x2 matrix, Λ will look like this:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

In our Python code, we've already used the ***linalg.eig*** function to return the array of eigenvalues for ***A*** into the variable ***l***, so now we just need to format that as a matrix:

In [17]:
```
L = np.diag(l)
print (L)
```

```
[[ 3.56155281  0.        ]
 [ 0.         -0.56155281]]
```

So ***Λ*** is the following matrix:

$$\Lambda = \begin{bmatrix} 3.56155281 & 0 \\ 0 & -0.56155281 \end{bmatrix}$$

Now we just need to find $\boldsymbol{Q^{-1}}$, which is the inverse of ***Q***:

In [18]:
```
Qinv = np.linalg.inv(Q)
print(Qinv)
```

```
[[ 0.89720673  0.50382896]
 [-0.27816009  0.99068183]]
```

The inverse of ***Q*** then, is:

$$Q^{-1} = \begin{bmatrix} 0.89720673 & 0.50382896 \\ -0.27816009 & 0.99068183 \end{bmatrix}$$

So what does that mean? Well, it means that we can decompose the transformation of *any* vector multiplied by matrix ***A*** into the separate operations $\boldsymbol{Q\Lambda Q^{-1}}$:

$$A\vec{v} = Q\Lambda Q^{-1}\vec{v}$$

To prove this, let's take vector ***v***:

$$\vec{v} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

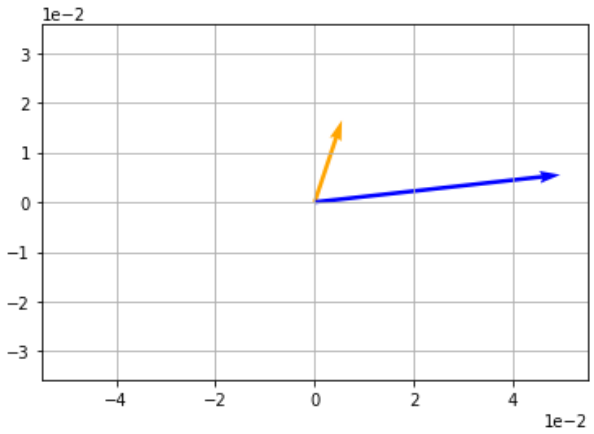
Our matrix transformation using ***A*** is:

$$\begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

So let's show the results of that using Python:

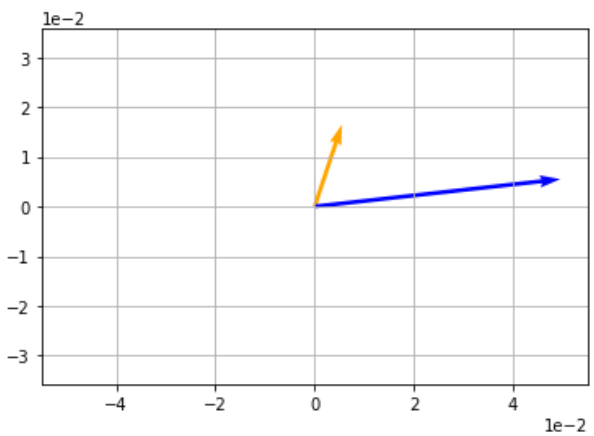
In [19]:
```
v = np.array([1,3])
t = A@v

print(t)

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'b'], scale=20)
plt.show()
```

```
[9 1]
```



And now, let's do the same thing using the $QΛQ^{-1}$ sequence of operations:

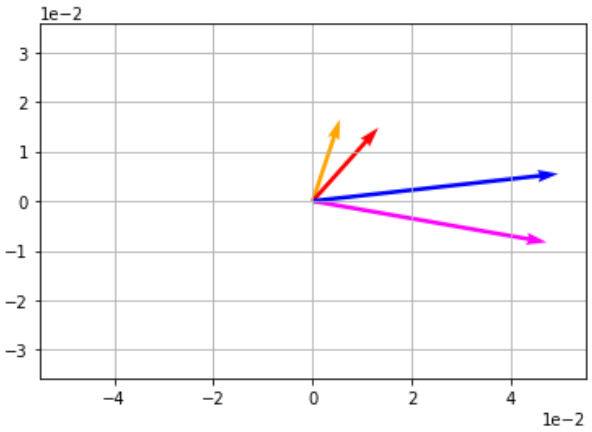
In [20]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

t = (Q@(L@(Qinv)))@v

# Plot v and t
vecs = np.array([v,t])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'b'], scale=20)
plt.show()
```



So ***A*** and $QΛQ^{-1}$ are equivalent.

If we view the intermediary stages of the decomposed transformation, you can see the transformation using ***A*** in the original base for ***v*** (orange to blue) and the transformation using ***Λ*** in the change of basis decribed by ***Q*** (red to magenta):

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

t1 = Qinv@v
t2 = L@t1
t3 = Q@t2

# Plot the transformations
vecs = np.array([v,t1, t2, t3])
origin = [0], [0]
plt.axis('equal')
plt.grid()
plt.ticklabel_format(style='sci', axis='both', scilimits=(0,0))
plt.quiver(*origin, vecs[:,0], vecs[:,1], color=['orange', 'red', 'magenta', 'b
plt.show()
```



So from this visualization, it should be apparent that the transformation ***Av*** can be performed by changing the basis for ***v*** using ***Q*** (from orange to red in the above plot) applying the equivalent linear transformation in that base using ***Λ*** (red to magenta), and switching back to the original base using Q^{-1} (magenta to blue).

Rank of a Matrix

The **rank** of a square matrix is the number of non-zero eigenvalues of the matrix. A **full rank** matrix has the same number of non-zero eigenvalues as the dimension of the matrix. A **rank-deficient** matrix has fewer non-zero eigenvalues as dimensions. The inverse of a rank deficient matrix is singular and so does not exist (this is why in a previous notebook we noted that some matrices have no inverse).

Consider the following matrix ***A***:

$$A = \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$$

Let's find its eigenvalues (***Λ***):

In [22]:
```
import numpy as np
A = np.array([[1,2],
              [4,3]])
l, Q = np.linalg.eig(A)
L = np.diag(l)
print(L)
```

```
[[-1.  0.]
 [ 0.  5.]]
```

$$\Lambda = \begin{bmatrix} -1 & 0 \\ 0 & 5 \end{bmatrix}$$

This matrix has full rank. The dimensions of the matrix is 2. There are two non-zero eigenvalues.

Now consider this matrix:

$$B = \begin{bmatrix} 3 & -3 & 6 \\ 2 & -2 & 4 \\ 1 & -1 & 2 \end{bmatrix}$$

Note that the second and third columns are just scalar multiples of the first column.

Let's examine it's eigenvalues:

In [23]:
```
B = np.array([[3,-3,6],
              [2,-2,4],
              [1,-1,2]])
lb, Qb = np.linalg.eig(B)
Lb = np.diag(lb)
print(Lb)
```

```
[[3.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 5.23364153e-16 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

$$\Lambda = \begin{bmatrix} 3 & 0 & 0 \\ 0 & -6 \times 10^{-17} & 0 \\ 0 & 0 & 3.6 \times 10^{-16} \end{bmatrix}$$

Note that matrix has only 1 non-zero eigenvalue. The other two eigenvalues are so extremely small as to be effectively zero. This is an example of a rank-deficient matrix; and as such, it has no inverse.

Inverse of a Square Full Rank Matrix

You can calculate the inverse of a square full rank matrix by using the following formula:

$$A^{-1} = Q\Lambda^{-1}Q^{-1}$$

Let's apply this to matrix ***A***:

$$A = \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$$

Let's find the matrices for ***Q***, $\boldsymbol{\Lambda^{-1}}$, and $\boldsymbol{Q^{-1}}$:

In [24]:
```
import numpy as np
A = np.array([[1,2],
              [4,3]])

l, Q = np.linalg.eig(A)
L = np.diag(l)
print(Q)
Linv = np.linalg.inv(L)
Qinv = np.linalg.inv(Q)
print(Linv)
print(Qinv)
```

```
[[-0.70710678 -0.4472136 ]
 [ 0.70710678 -0.89442719]]
[[-1.  -0. ]
 [ 0.   0.2]]
[[-0.94280904  0.47140452]
 [-0.74535599 -0.74535599]]
```

So:

$$A^{-1} = \begin{bmatrix} -0.70710678 & -0.4472136 \\ 0.70710678 & -0.89442719 \end{bmatrix} \cdot \begin{bmatrix} -1 & -0 \\ 0 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} -0.94280904 & 0.47140452 \\ -0.74535599 & -0.74535599 \end{bmatrix}$$

Let's calculate that in Python:

In [25]:
```
Ainv = (Q@(Linv@(Qinv)))
print(Ainv)
```

```
[[-0.6  0.4]
 [ 0.8 -0.2]]
```

That gives us the result:

$$A^{-1} = \begin{bmatrix} -0.6 & 0.4 \\ 0.8 & -0.2 \end{bmatrix}$$

We can apply the ***np.linalg.inv*** function directly to ***A*** to verify this:

In [26]:
```
print(np.linalg.inv(A))
```

```
[[-0.6  0.4]
 [ 0.8 -0.2]]
```

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 thing 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 x mother) ÷ 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 desending 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 feaures 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 it 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 heirarchy, 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* quantative 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* quantative 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.

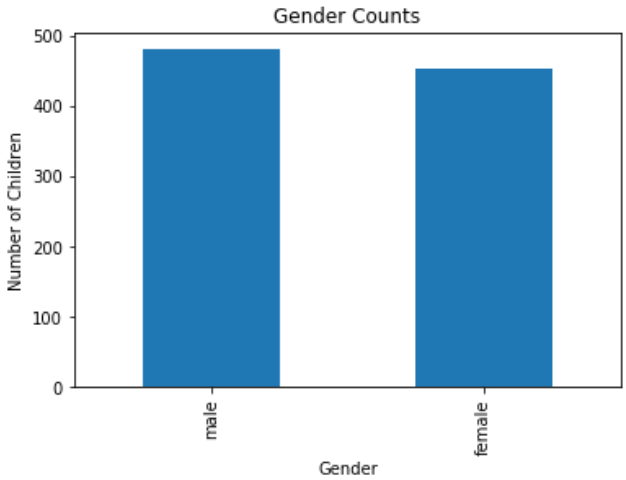
Let's examine some common kinds of data visualization:

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()
```



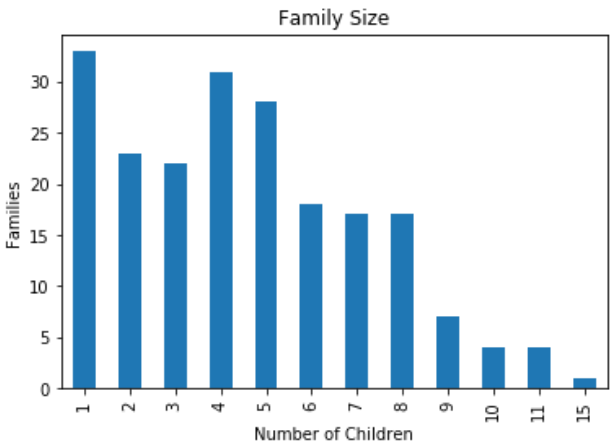
From this chart, you can see that there are slightly more male children than female children; but the data is reasonably evenly split between the two genders.

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 familes 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()
```



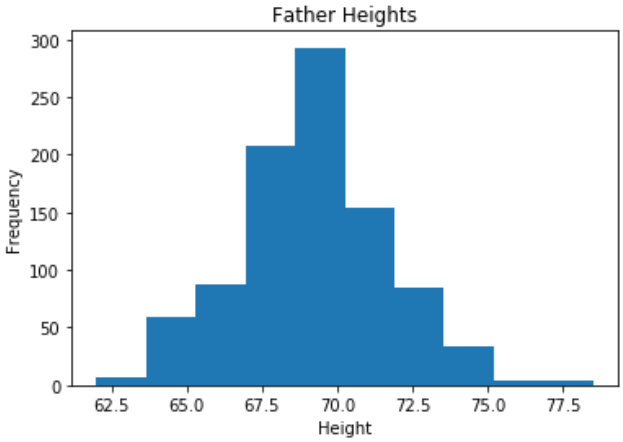
Note that the code sorts the data so that the categories on the *x* axis are in order - attention to this sort of detail can make your charts easier to read. In this case, we can see that the most common number of children per family is 1, followed by 5 and 6. Comparatively fewer families have more than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

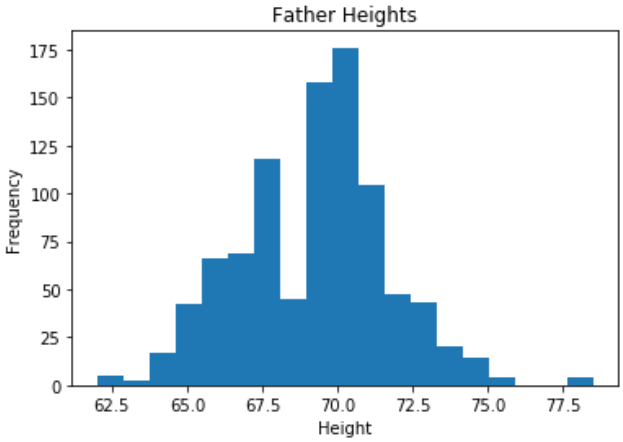
In [7]:
```
import statsmodels.api as sm

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 occuring heights tend to be in the mid-range. There are fewer extremely short or exteremely 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()
```



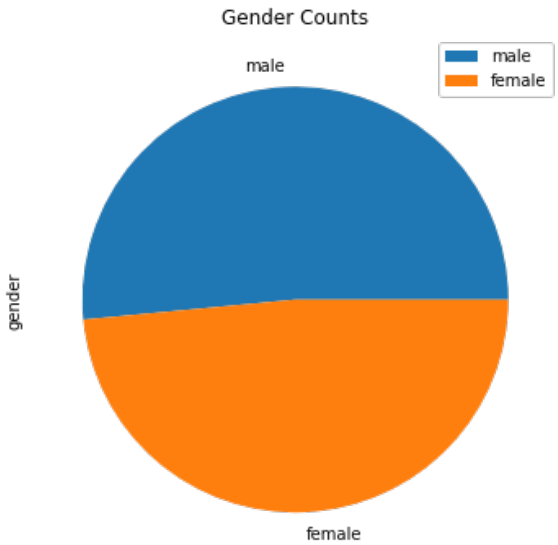
We can still see that the most common heights are in the middle, but there's a notable drop in the number of fathers with a height between 67.5 and 70.

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()
```



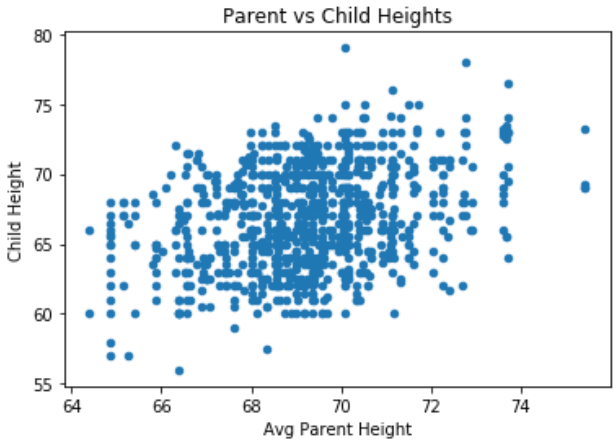
Note that the chart includes a *legend* to make it clear what category each colored area in the pie chart represents. From this chart, you can see that males make up slightly more than half of the overall number of children; with females accounting for the rest.

Scatter Plots

Often you'll want to compare quantative 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='midparen
plt.xlabel('Avg Parent Height')
plt.ylabel('Child Height')
plt.show()
```

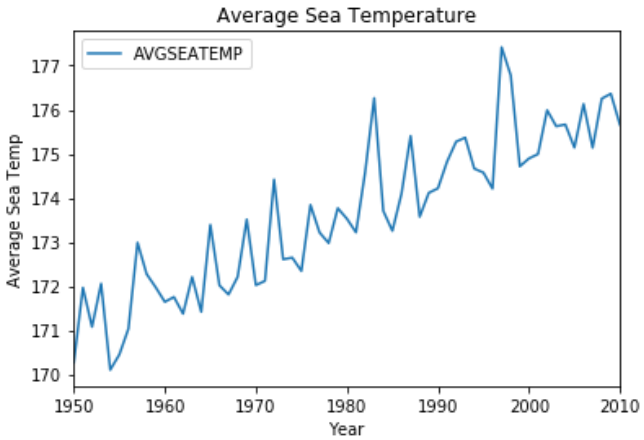


In a scatter plot, each dot marks the intersection point of the two values being plotted. In this chart, most of the heights are clustered around the center; which indicates that most parents and children tend to have a height that is somewhere in the middle of the range of heights observed. At the bottom 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 udnerstanding 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 $\boldsymbol{\mu}$ (*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 $\boldsymbol{\mu}$ 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 tendancy 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:

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

So, for our graduate salaries; first lets sort the dataset:

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

There's an odd number of observation (7), so the median value is at position (7 + 1) ÷ 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:

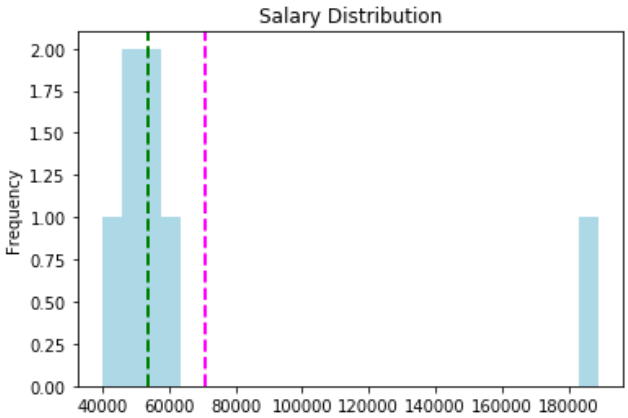
In [5]:
```
import pandas as pd

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:

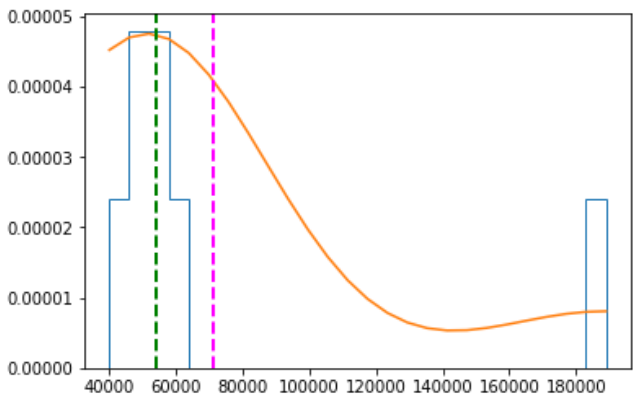
- *Salary* is a continuous data value - graduates could potentially earn any value along the scale, even down to a fraction of cent.
- The number of bins in the histogram determines the size of each salary band for which we're counting frequencies. Fewer bins means merging more individual salaries together 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_l
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.

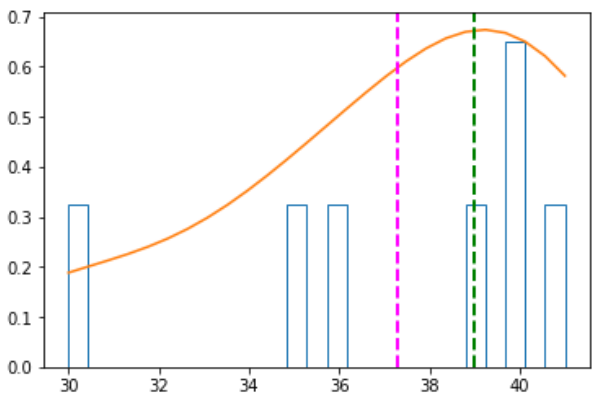
Let's take a look at another set of data. We know how much money our graduates make, but how many hours per week do they need to work to earn their salaries? Here's the data:

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_l
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] == '':
```



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.

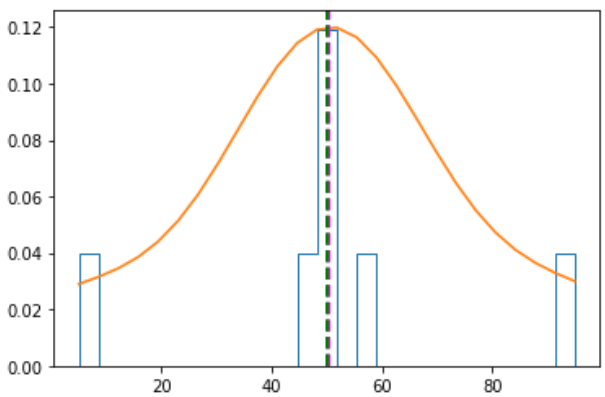
Once again, Rosie seems to be getting the better of the deal. She earns more than her former classmates for working fewer hours. Maybe a look at the test scores the students achieved on their final grade at school might help explain her success:

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_l
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] == '':
```



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.

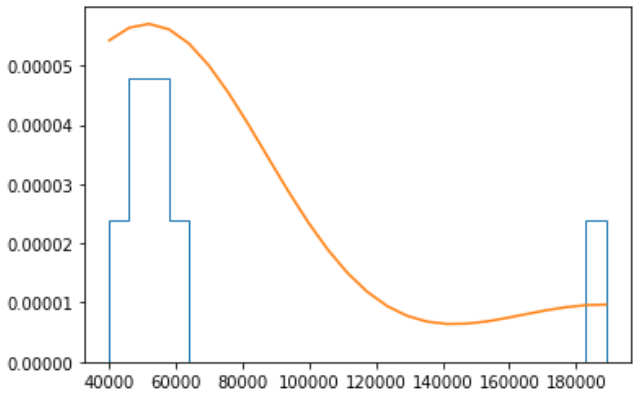
Statisticians call this a *normal* distribution (or sometimes a *Gaussian* distribution), and it occurs quite commonly in many scenarios due to something called the *Central Limit Theorem*, which reflects the way continuous probability works - more about that later.

Skewness and Kurtosis

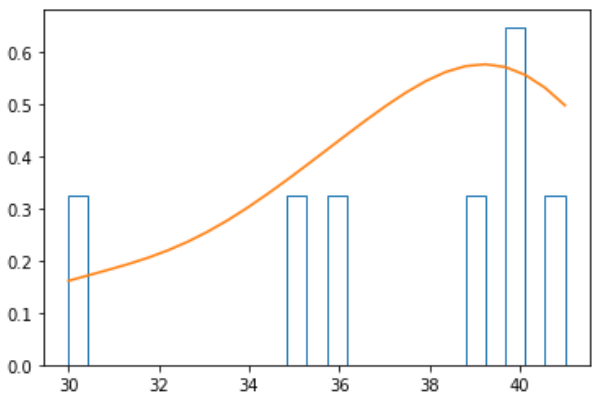
You can measure *skewness* (in which direction the data is skewed and to what degree) and kurtosis (how "peaked" the data is) to get an idea of the shape of the data distribution. In Python, you can use the ***skew*** and ***kurt*** functions to find this:

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_l
auncher.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
```

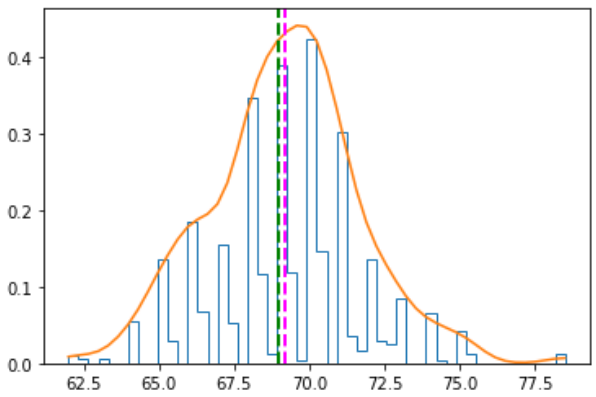



```
Grade skewness: -0.06512433009682762
Grade kurtosis: 2.7484764913773034
```

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 minimim 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 nothing 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:

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

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:

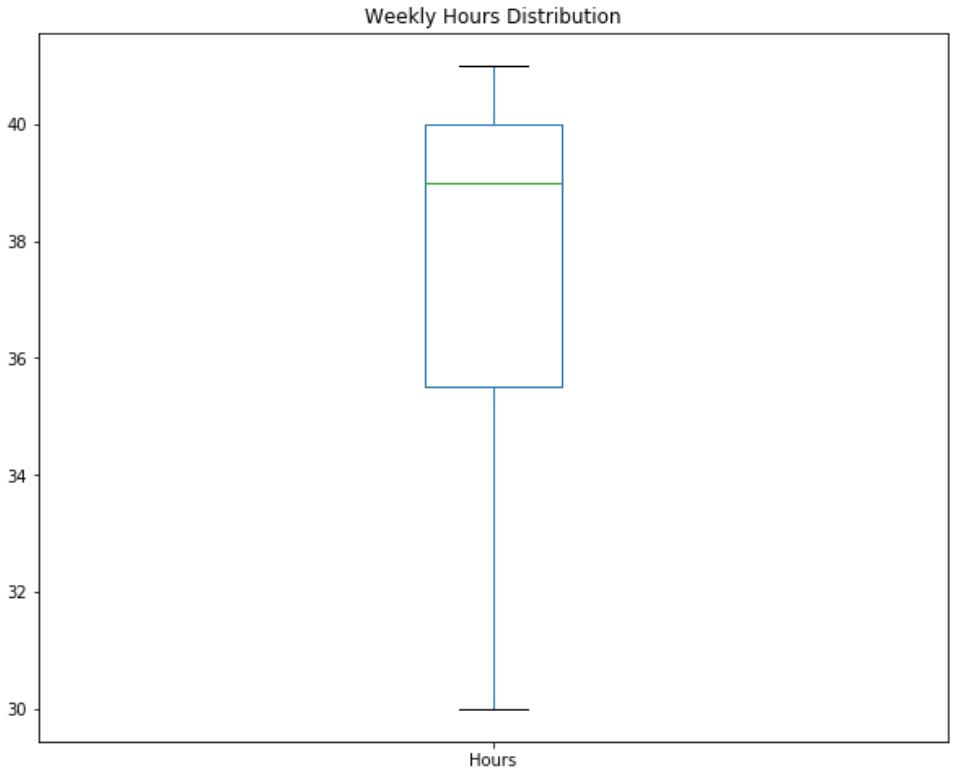
In [16]:
```
# Quartiles
import pandas as pd

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
```

Its 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?

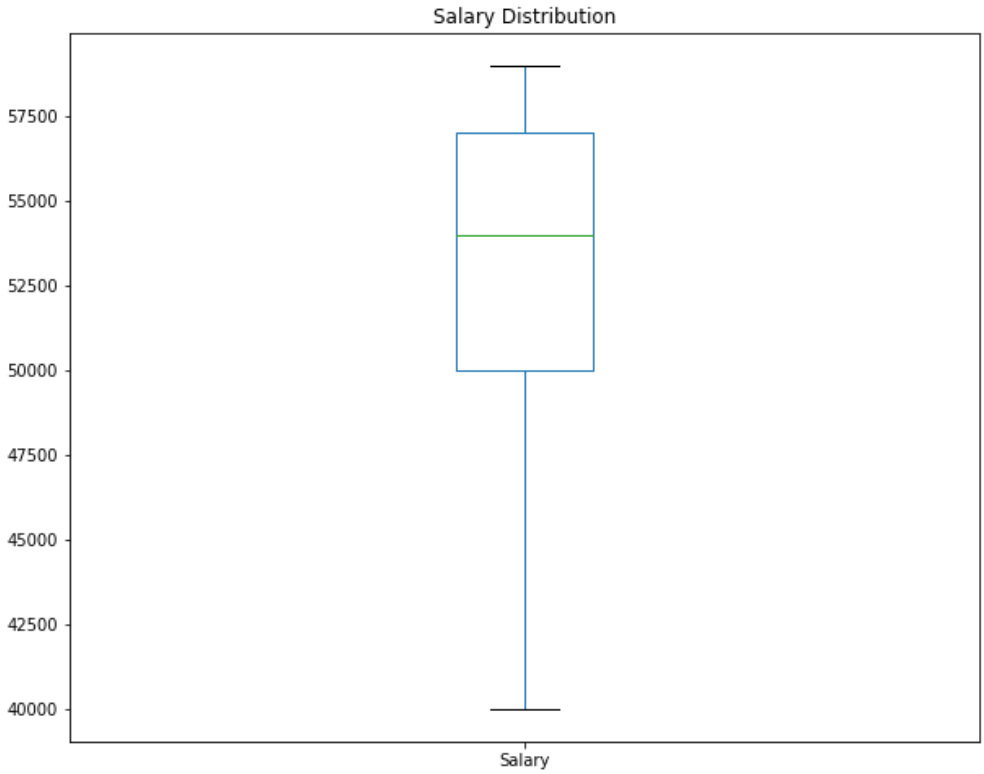
Well, as we've already noticed, Rosie earns significantly more than her former classmates. So much more in fact, that her salary has been identifed as an *outlier*. An outlier is a value that is so far from the center of the distribution compared to other values that it skews the distribution by affecting 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 extremly 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 athelete 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()
```

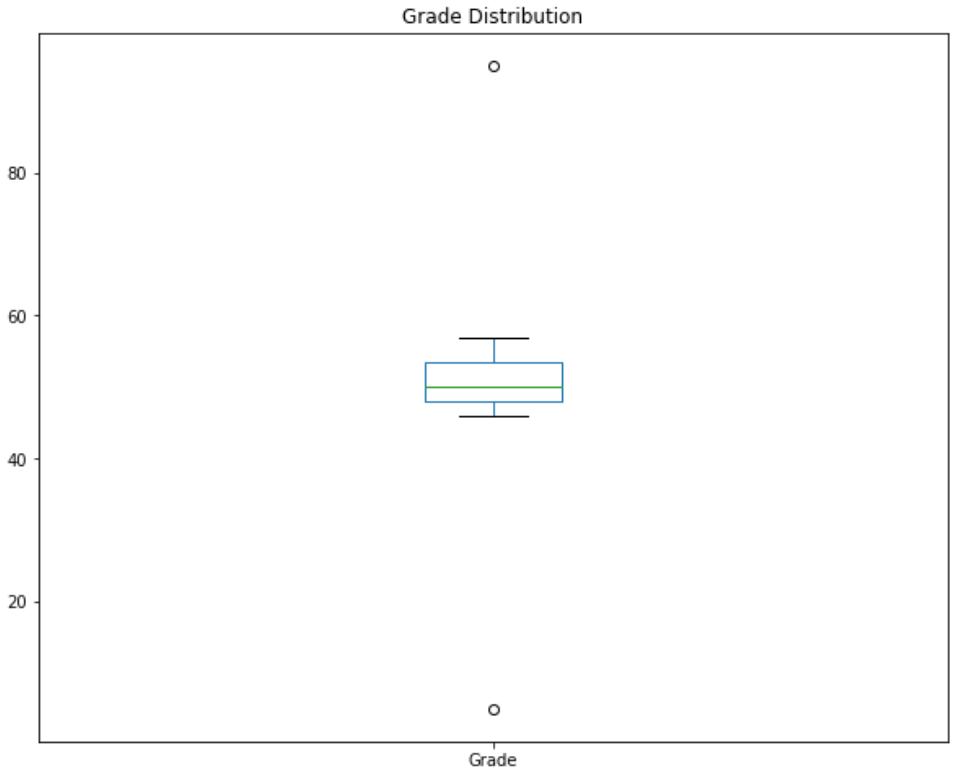


Now it looks like there's a more even distribution of salaries. It's still not quite symmetrical, but there's much less overall variance. There's potentially some cause here to disregard Rosie's salary data when we compare the salaries, as it is tending to skew the analysis.

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

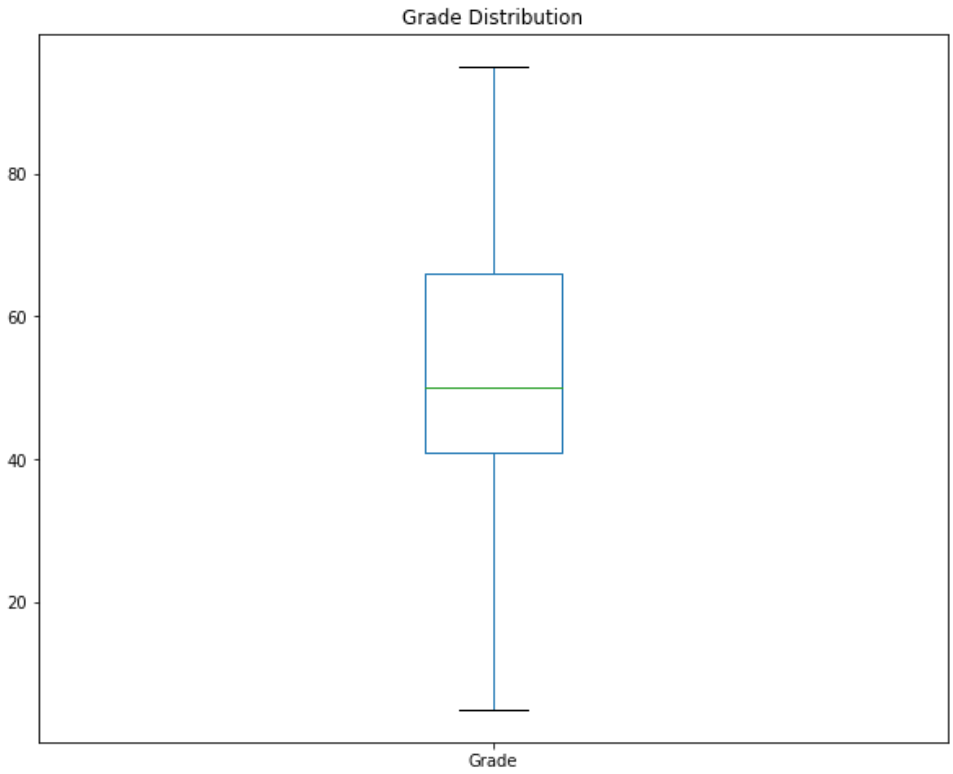
Again, it depends on what you're analyzing. Let's take a look at the distribution of final grades:

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* (***σ²***) and calculated like this:

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

For a sample, it's indicated as ***s²*** 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:

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

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 forumla!

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
```

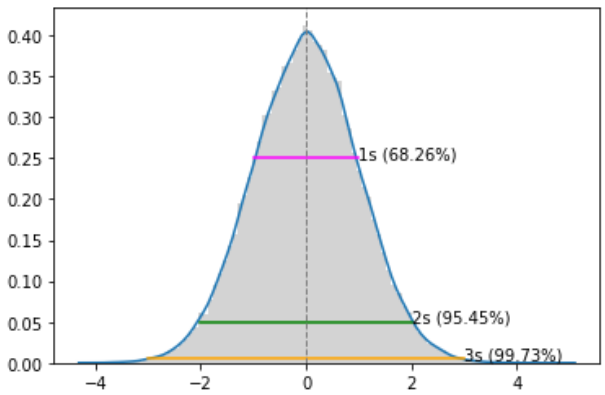
Standard Deviation in a Normal Distribution

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_l
auncher.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

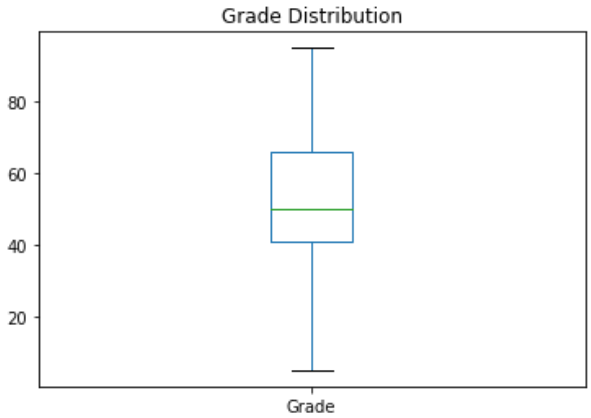
You'll often want to compare data in your dataset, to see if you can discern trends or relationships.

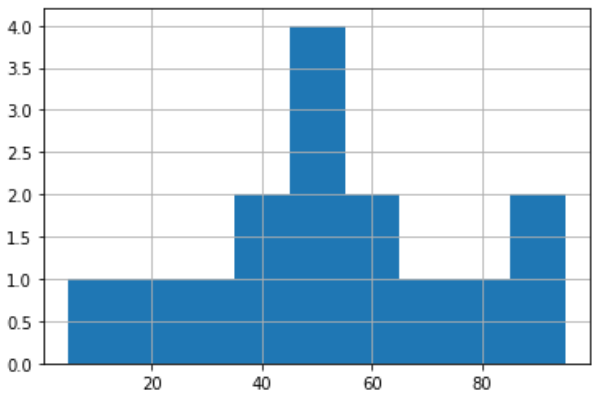
Univariate Data

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:

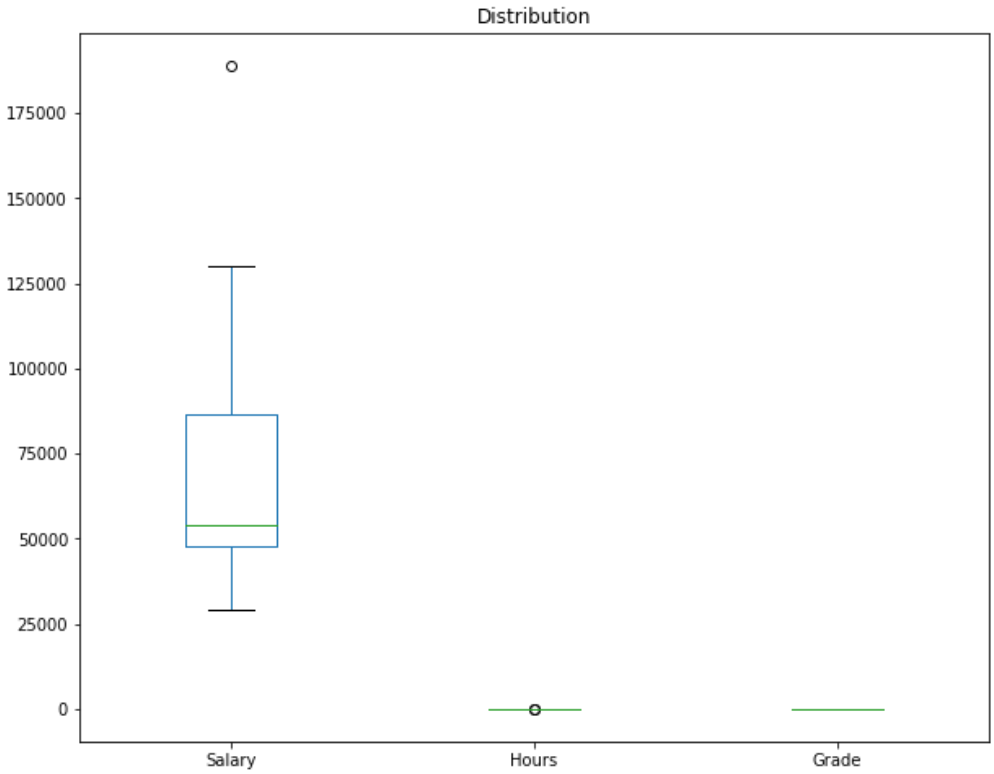
In [10]:
```
import pandas as pd

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()
```



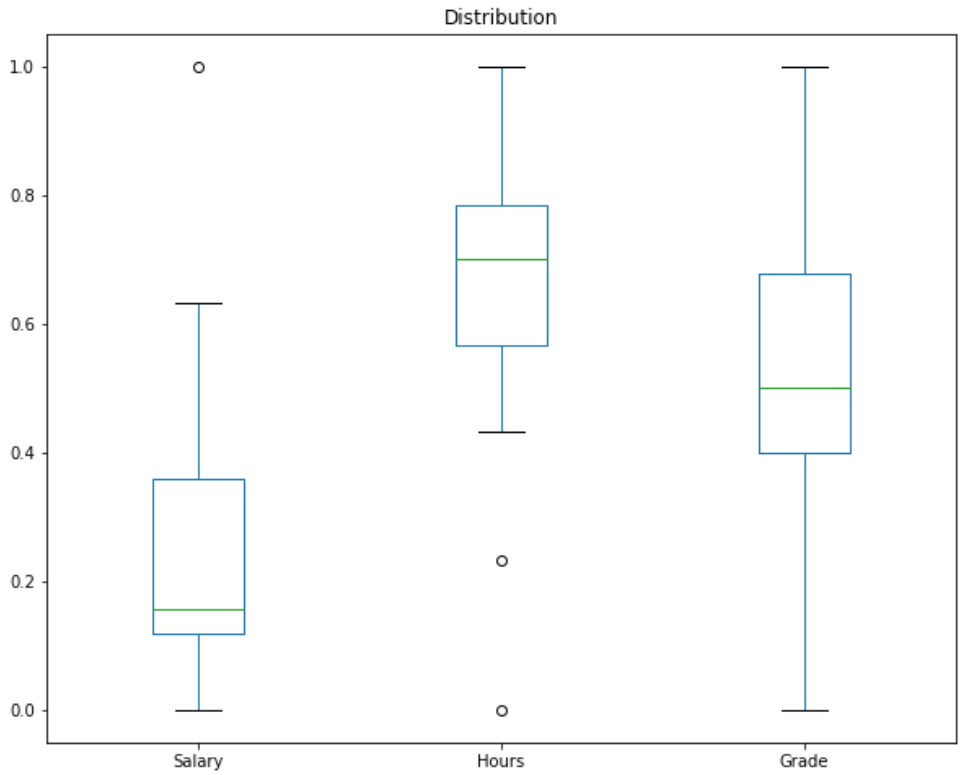
Hmm, that's not particularly useful is it?

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()
```



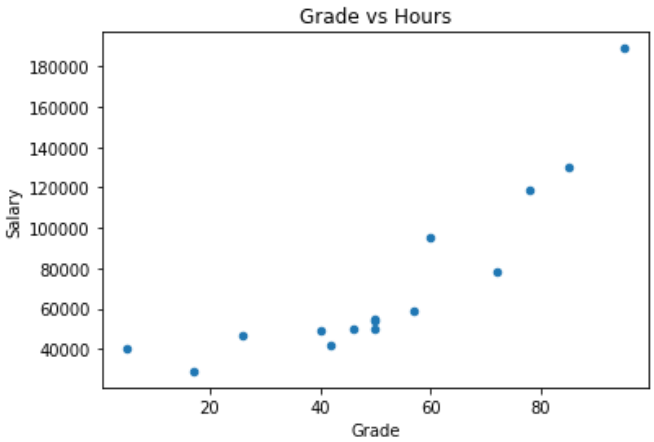
Now the numbers on the y axis aren't particularly meaningful, but they're on a similar scale.

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:

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

$\mathbf{r_{x,\,y}}$ 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']))
```

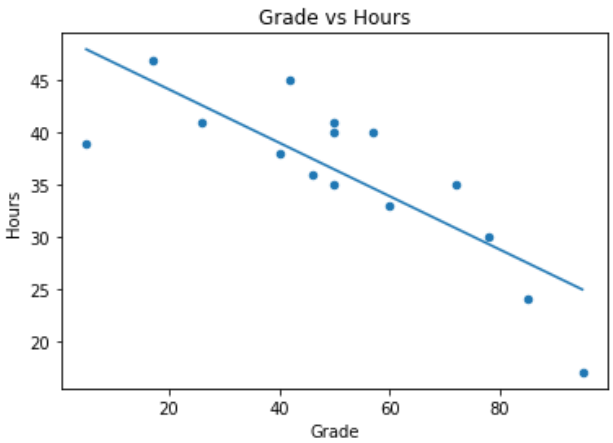
```
0.8149286388911882
```

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.poly1d(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 lookes 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 is to calculate the intercept (*b*) like this:

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

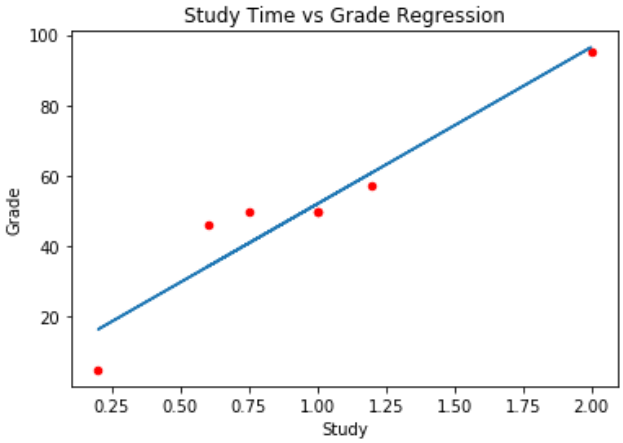
Let's look at a simple example that compares the number of hours of nightly study each student undertook with the final grade the student achieved:

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='G

# 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='Salar

# 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='Salar

# 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='Salar

# 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 oucomes (*heads* and *tails*).
- A ***sample point*** is a single possible outcome - for example, *heads*)
- An ***event*** is a specific outome 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:

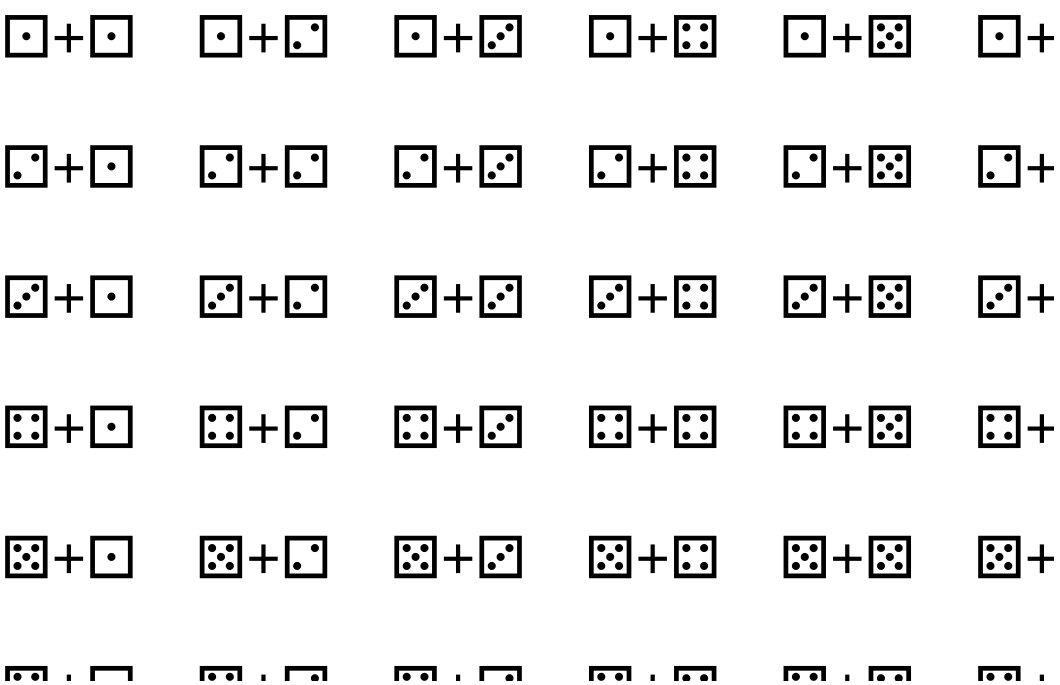
$$\text{probability of an event} = \frac{\text{Number of sample points that produce the event}}{\text{Total number of sample points in the sample space}}$$

For example, the probability of getting *heads* when tossing as coin is $^1/_2$ - 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 outome 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 outomes: heads ( ) or tails ( ).

The probability of getting *heads* is $^1/_2$, and the probability of getting *tails* is also $^1/_2$. Let's toss a coin...

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

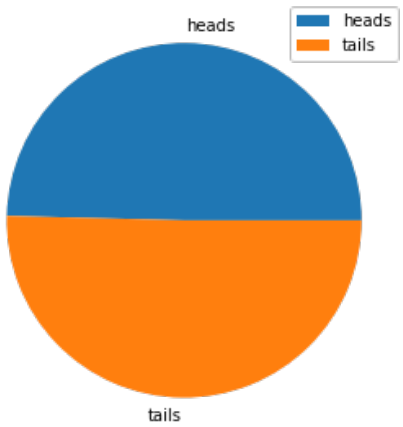
It looks like we got *heads* again. If we were to toss the coin a third time, what's the probability that we'd get *heads*?

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 $^1/_2$, 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 $^1/_2$, 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:

☼ = $^1/_2$

☼☼ = $^1/_2$ x $^1/_2$

☼☼☼ = $^1/_2$ x $^1/_2$ x $^1/_2$

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

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

In [2]:

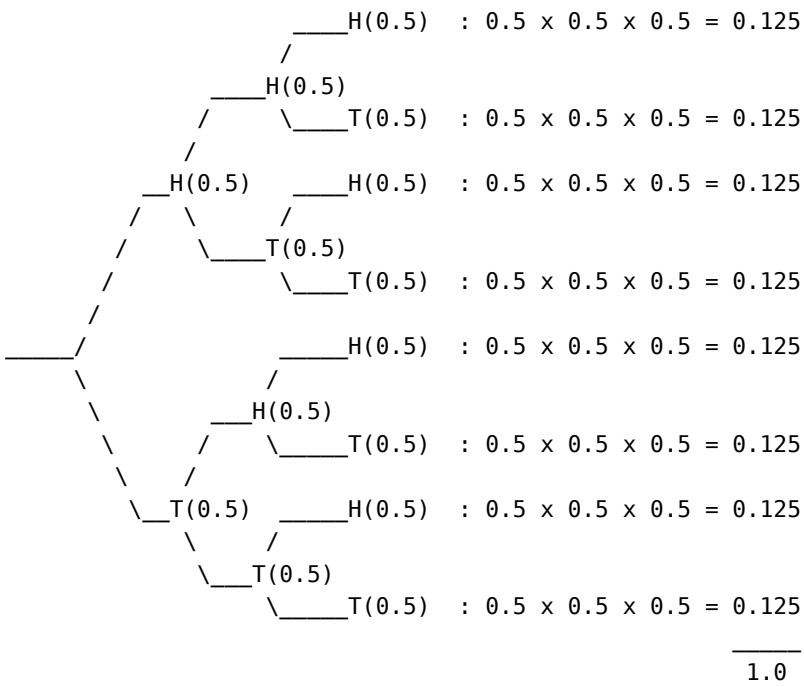
```
import random

# 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)
```

```
12.33%
[['T', 'T', 'H'], ['T', 'H', 'T'], ['H', 'H', 'T'], ['H', 'T', 'H'], ['T', 'T
', 'H'], ['H', 'H', 'H'], ['H', 'T', 'T'], ['T', 'T', 'H'], ['H', 'T', 'T'],
['T', 'T', 'H'], ['T', 'T', 'H'], ['T', 'H', 'T'], ['H', 'H', 'T'], ['H', 'T',
'H'], ['H', 'H', 'T'], ['H', 'H', 'T'], ['H', 'T', 'H'], ['T', 'H', 'T'], ['T
', 'H', 'T'], ['T', 'H', 'T'], ['H', 'H', 'H'], ['H', 'H', 'T'], ['H', 'H', 'H
'], ['H', 'H', 'T'], ['T', 'H', 'T'], ['H', 'H', 'T'], ['T', 'T', 'H'], ['H',
'T', 'T'], ['H', 'H', 'H'], ['T', 'T', 'H'], ['T', 'H', 'H'], ['T', 'H', 'T'],
['H', 'T', 'H'], ['T', 'T', 'H'], ['H', 'T', 'T'], ['T', 'T', 'H'], ['T', 'H',
'H'], ['H', 'T', 'H'], ['T', 'T', 'T'], ['H', 'T', 'T'], ['H', 'T', 'T'], ['H
', 'H', 'T'], ['T', 'T', 'T'], ['T', 'T', 'H'], ['H', 'H', 'H'], ['T', 'H', 'H
'], ['T', 'H', 'T'], ['T', 'H', 'H'], ['H', 'H', 'T'], ['H', 'T', 'H'], ['H',
'T', 'T'], ['T', 'H', 'T'], ['T', 'H', 'H'], ['T', 'T', 'H'], ['H', 'H', 'H'],
['H', 'T', 'H'], ['H', 'H', 'H'], ['T', 'H', 'H'], ['H', 'T', 'T'], ['T', 'T',
'T'], ['H', 'H', 'H'], ['T', 'H', 'H'], ['T', 'T', 'T'], ['T', 'H', 'T'], ['T
', 'T', 'H'], ['T', 'T', 'H'], ['H', 'T', 'H'], ['H', 'H', 'H'], ['H', 'T', 'H
'], ['T', 'T', 'H'], ['H', 'H', 'H'], ['H', 'H', 'H'], ['T', 'H', 'T'], ['T',
'T', 'H'], ['T', 'H', 'T'], ['H', 'H', 'T'], ['H', 'H', 'T'], ['H', 'T', 'T'],
['H', 'T', 'H'], ['T', 'T', 'T'], ['T', 'T', 'H'], ['H', 'H', 'T'], ['T', 'T',
```

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.

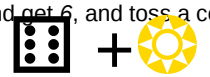
Combined Event Probability Notation

When calculating the probability of combined events, we assign a letter such as **A** or **B** to each event, and we use the *intersection* (∩) 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 excitment 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:

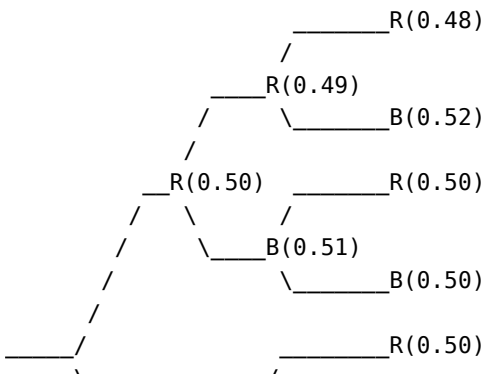
12 x ♥ 13 x ♠ 13 x ♣ 12 x ♦

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 us 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

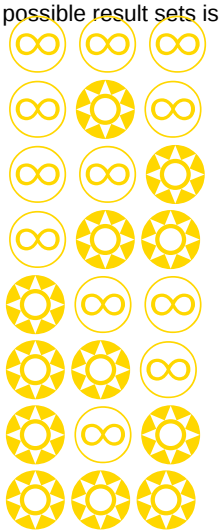
One particular type of random variable that we use in statistics is a *binomial* variable. A binomial variable is used to count how frequently an event occurs in a fixed number of repeated independent experiments. The event in question must have the same probability of occurring in each experiment, and 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 possibile 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:

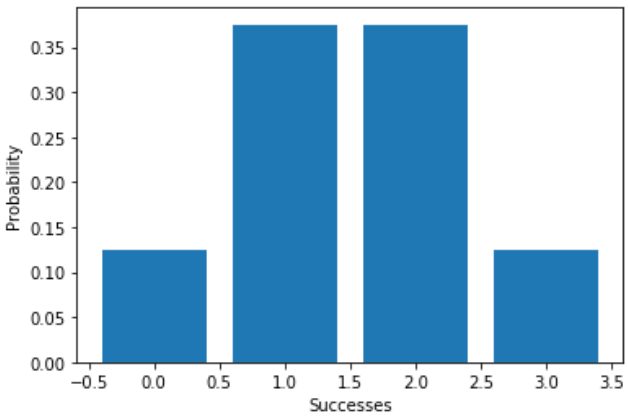
$$X = \{0, 1, 2, 3\}$$

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 though. 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 $^1/_4$ 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 $^3/_4$ 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 x 0.25 x 0.25 which is the same as 0.25^3. Using our generic variables, this is $\mathbf{p^k}$.
- The probability that the rest don't get searched is 0.75 x 0.75, or 0.75^2. In terms of our variables, this is $\mathbf{1\text{-}p^{(n\text{-}k)}}$.
- The combined probability of three searchs and two non-searches is therefore 0.25^3 x 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:

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

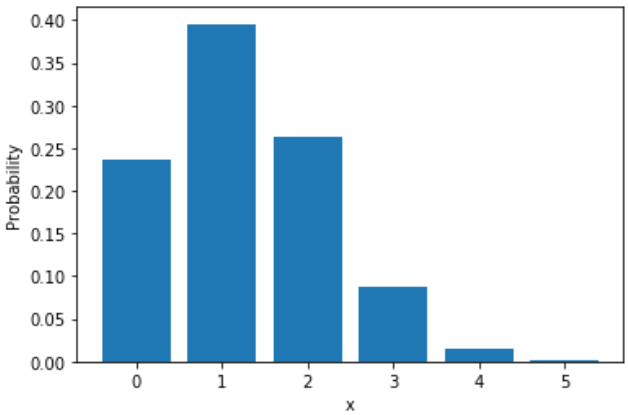
So 5 out of our 32 combinations had 3 searches and 2 non-searches.

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 $^5/_{32}$ x 0.088, which is 0.01375, or 13.75%.

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

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

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.

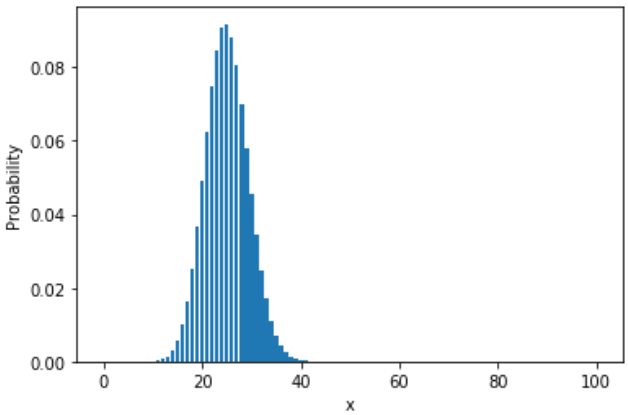
There is an important distinction here - the *normal* distribution applies to *continuous* variables, while the *binomial* distribution applies to *discrete* variables. However, the similarities help in a number of statistical contexts where the number of observations (experiments) is large enough for 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 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 the 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 $^{3}/_{16}$ 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 numberic distribution, we can also calculate statistics like *mean* and *standard deviation*:

To calculate these, you can use the following formulae:

$$\mu_{\hat{p}} = \hat{p}$$

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

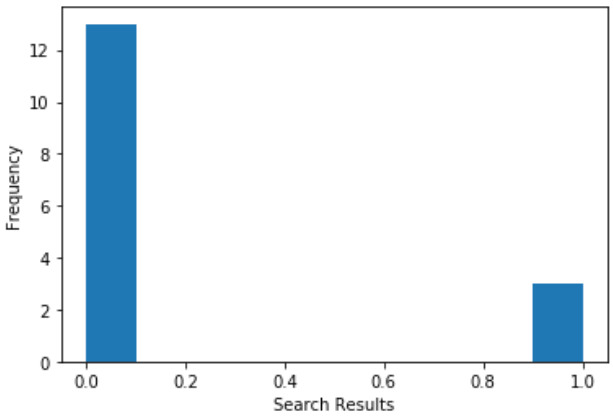
The mean is just the value of **p̂**, so in the case of the passenger search sample it is 0.1875.

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.

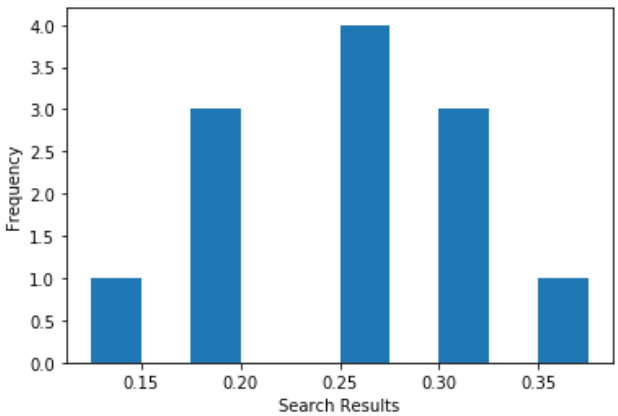
Creating a Sampling Distribution of a Sample Proportion

So, let's collect mulitple 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

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

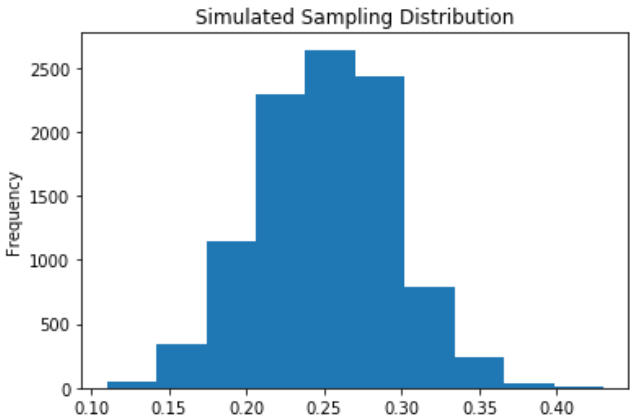


The Central Limit Theorem

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.

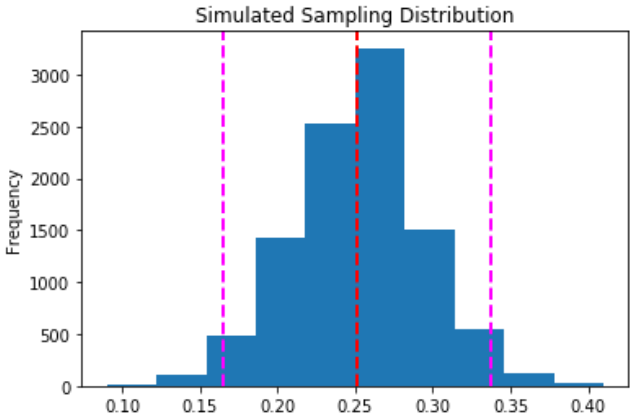
Being able to calculate the mean (or *expected value*) and standard error is useful, because we can apply these to what we know about an approximately normal distribution to estimate probabilities for particular values. For example, we know that in a normal distribution, around 95.4% of the values are 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 x 0.043) of the mean, so the value for 95.4% of our samples is 0.25 ± (*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()
moe1 = 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(moe1, 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 occassions. We might end up with some data like this:

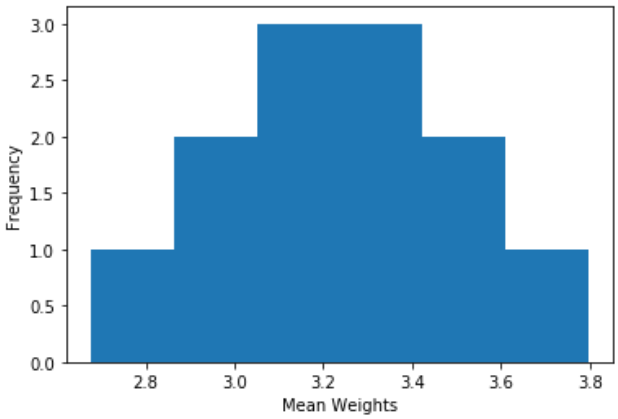
Sample	Weights
1	[4.020992,2.143457,2.260409,2.339641,4.699211]
2	[3.38532,4.438345,3.170228,3.499913,4.489557]
3	[3.338228,1.825221,3.53633,3.507952,2.698669]
4	[2.992756,3.292431,3.38148,3.479455,3.051273]
5	[2.969977,3.869029,4.149342,2.785682,3.03557]
6	[3.138055,2.535442,3.530052,3.029846,2.881217]
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 $\overline{\mathbf{X}}$, 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 distributi
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 **X** is a random variable representing every possible bag weight, then its mean (indicated as $\boldsymbol{\mu_X}$) is the population mean ($\boldsymbol{\mu}$). The mean of the $\boldsymbol{\bar{X}}$ sampling distribution (which is indicated as $\boldsymbol{\mu_{\bar{X}}}$) is considered to have the same value. Or, as an equation:

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

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 ($\boldsymbol{\bar{x}}$), 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:

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

In this formula, $\boldsymbol{\sigma}$ is the population standard deviation and $\boldsymbol{n}$ is the size of each sample.

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

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

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:

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

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 ± (*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 approriate 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:

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

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

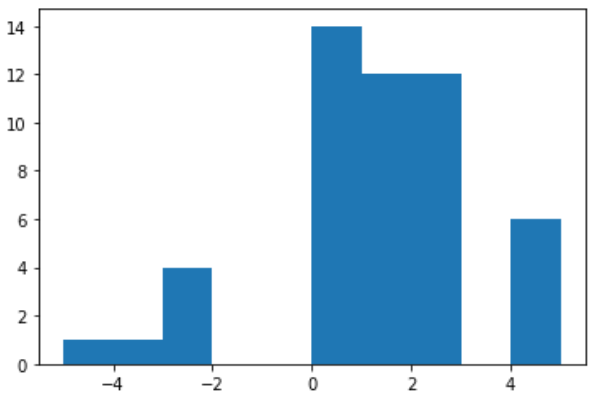
Single-Sample, One-Sided Tests

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 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:

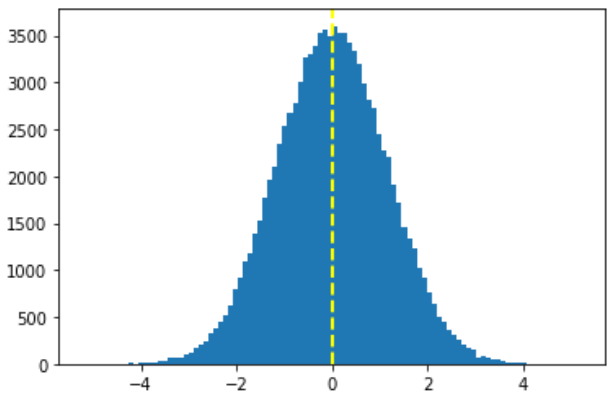
$$H_0 : \mu \leq 0$$
$$H_1 : \mu > 0$$

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:

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

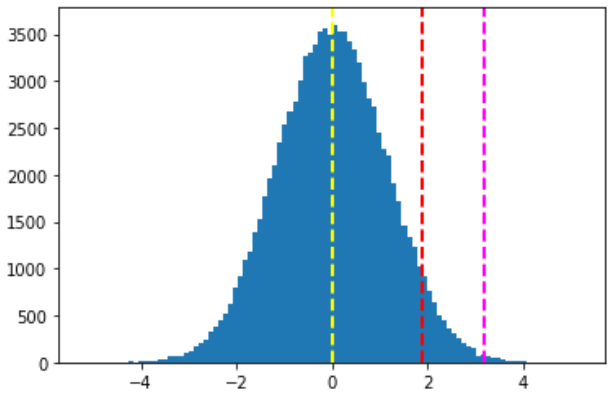
In this formula, $\bar{x}$ is the sample mean, **μ** is the population mean, **s** is the standard deviation, and **n** is the sample size. You can think of the numerator of this equation (the expression at the top of the fraction) as a *signal*, and the denominator (the expression at the bottom of the 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-va
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 propbability is unc
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 o
plt.axvline(pop.mean() + t*pop.std(), color='magenta', linestyle='dashed', line
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 (**H_0**) 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 (**H_1**) is that the population mean is not equal to 0.

We can write these as mutually exclusive expressions like this:

$$H_0 : \mu = 0$$
$$H_1 : \mu \neq 0$$

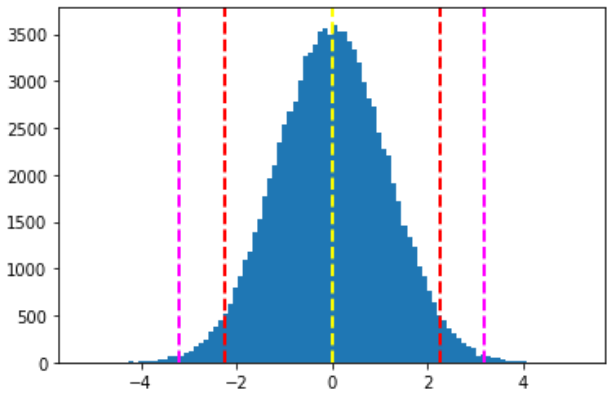
Why would we do this? Well, in the test we performed earlier, we could only reject the null hypothesis if we had really *positive* ratings, but what if our sample data looked really *negative*? It would be a mistake to turn around and run a one-tailed test the other way, for negative ratings. Instead,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

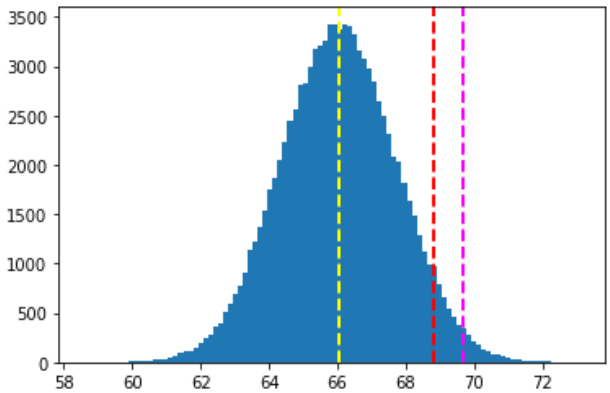
$$H_0 : \mu_1 \leq \mu_2$$
$$H_1 : \mu_1 > \mu_2$$

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 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-valu
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 propbability is und
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 o
plt.axvline(pop.mean() + t*pop.std(), color='magenta', linestyle='dashed', line
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 conduced previously, the samples were independent; in other words there was no relatioship 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.

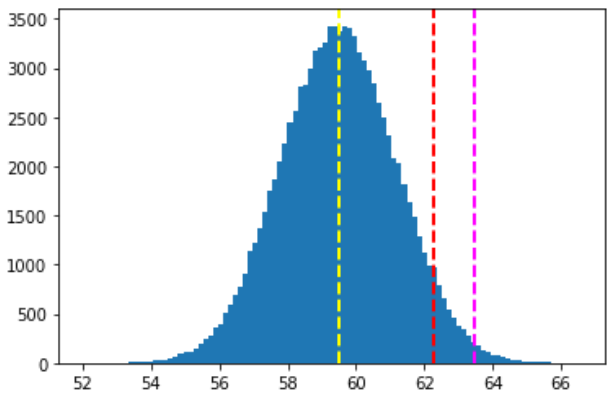
For example, suppose our students took a mid-term exam, and later took and end-of-term exam. You might hypothesise that the students will improve their grades in the end-of-term exam, after they've undertaken additional study. We could test for a general improvement on average across all students with 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-valu
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 propbability is und
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 o
plt.axvline(pop.mean() + t*pop.std(), color='magenta', linestyle='dashed', line
plt.show()
```

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



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