# 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:

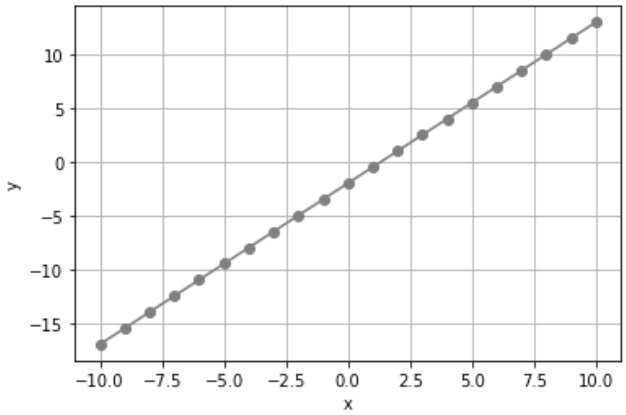
In [1]:
```
import pandas as pd

# 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)

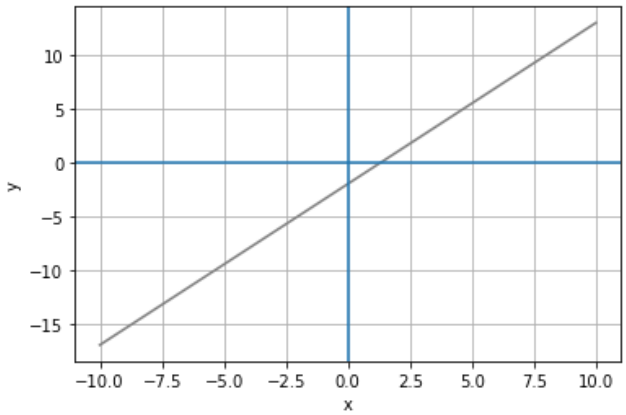
The cool thing about linear equations is that we can plot the points for some specific ordered pair solutions to create the line, and then interpolate the x value for any y value (or vice-versa) along the line.

## 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:

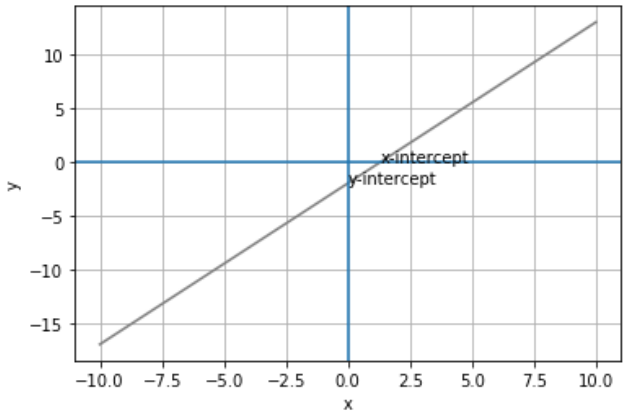
$$y = \frac{-4}{2}$$

-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:

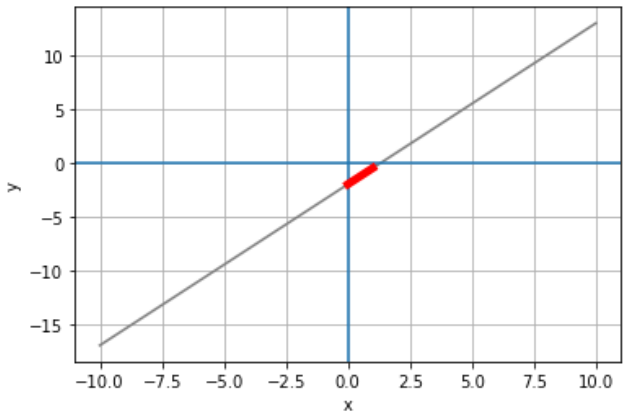
$$m = \frac{7 + 2}{6 - 0}$$

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:

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

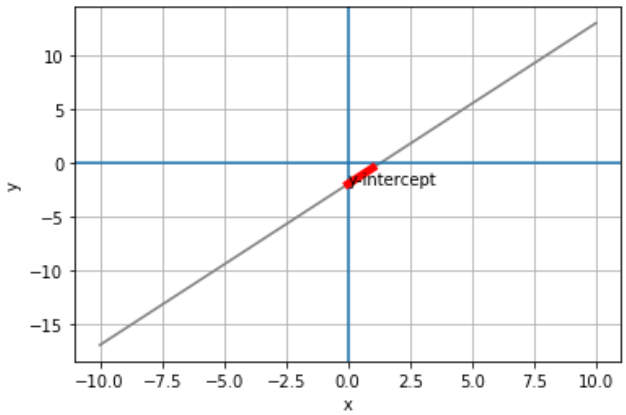
Now that we know the slope and y-intercept for the line that this equation defines, we can rewrite the equation as:

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

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.

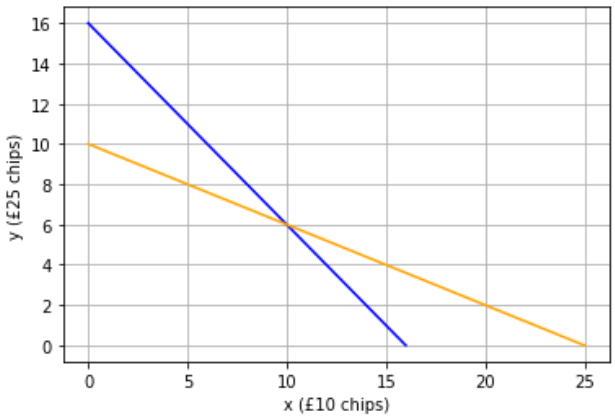
A collection of 16 chips could be made up of 16 £10 chips and no £25 chips, no £10 chips and 16 £25 chips, or any combination between these.

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 $6x^3$:

$$2y = 6x^3$$

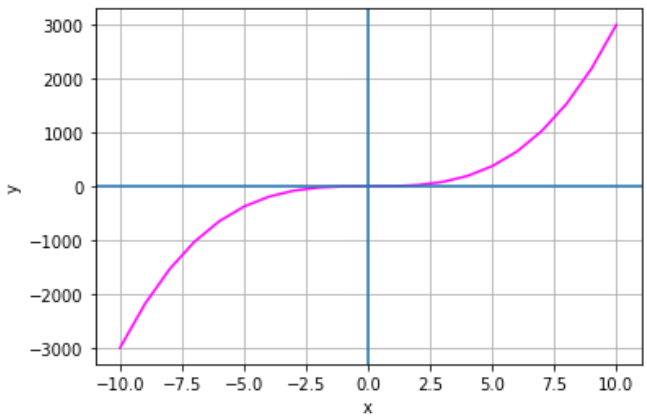
We're in the home stretch now, we just need to isolate y on the left side, and we can do that by dividing both sides by 2. Note that we're not dividing by an exponential, we simply need to divide the whole $6x^3$ term by two; and half of 6 times $x^3$ is just 3 times $x^3$:

$$y = 3x^3$$

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*.

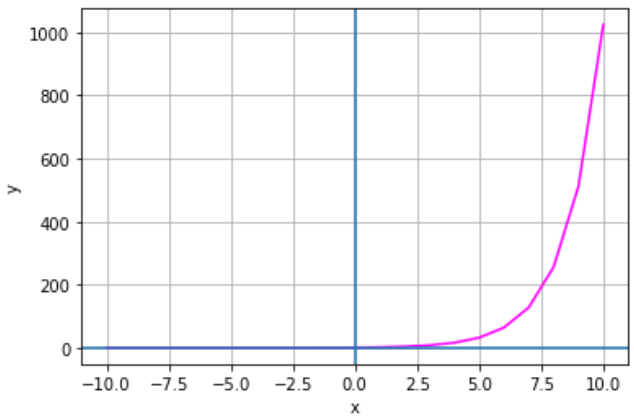
Let's look at an example in which x is the exponential, not the base:

$$y = 2^x$$

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:

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

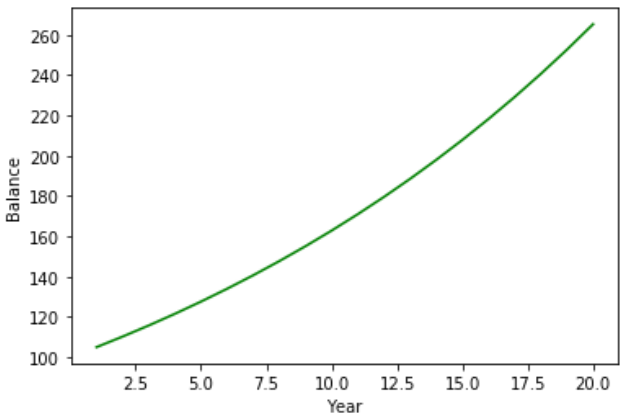
It turns out, if we just use the year as the exponent, we can easily calculate the growth after twenty years like this:

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

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:

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

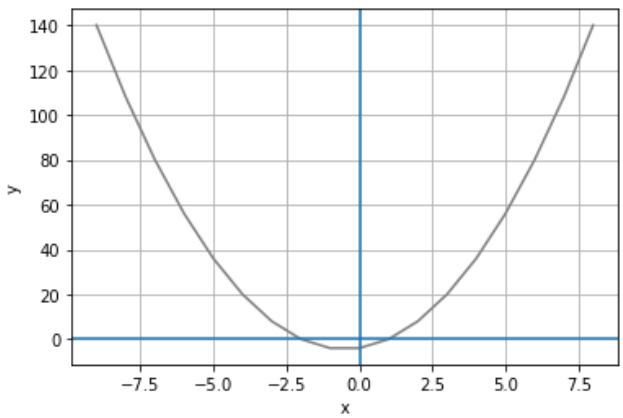
If you multiply out the factored ***x*** expressions, this equates to:

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

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.

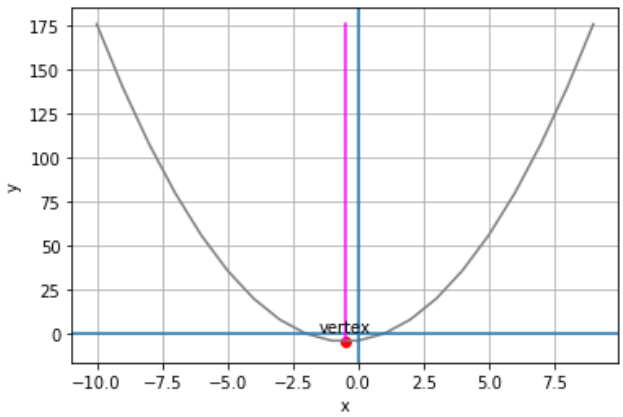
Let's start by examining the parabolas generated by quadratic equations in more detail.

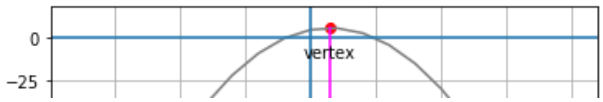
## Parabola Vertex and Line of Symmetry

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**.

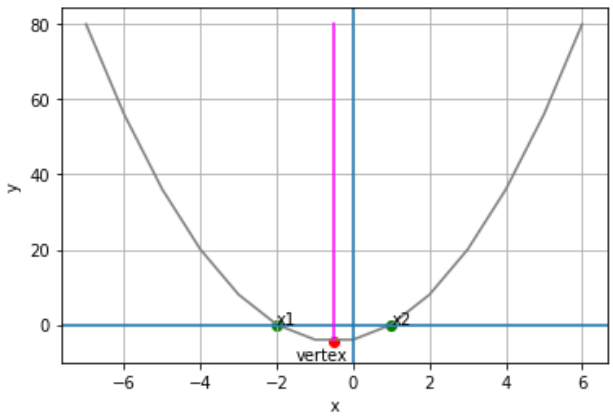
Now let's try the other expression:

$$x + 2 = 0$$

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 $x^2$:

$$x^2 = 4$$

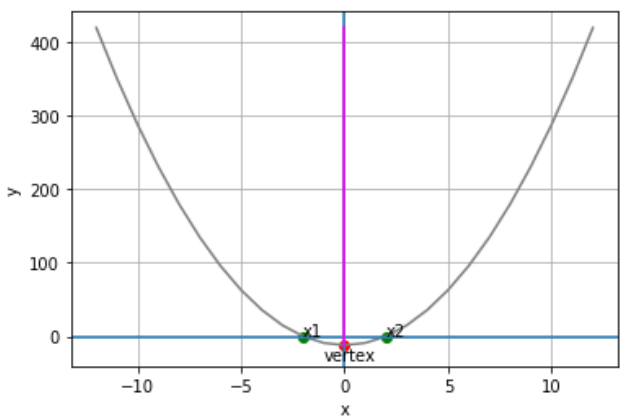
No we can isolate ***x*** by taking the square root of both sides. However, there's an additional consideration because this is a quadratic equation. The ***x*** variable can have two possibe values, so we must calculate the *principle* and *negative* square roots of the expression on the right:

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

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:

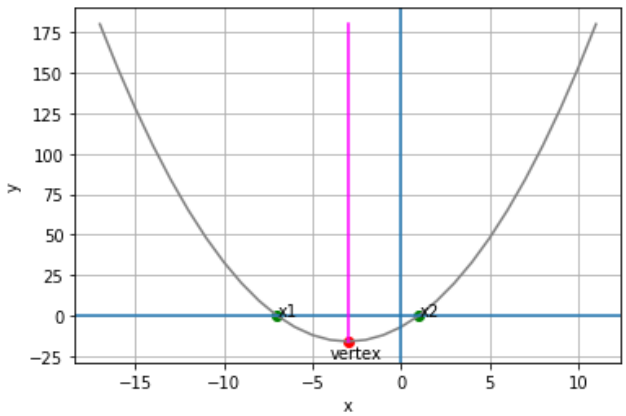
$$x + 3 = \pm\sqrt{16}$$

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**:

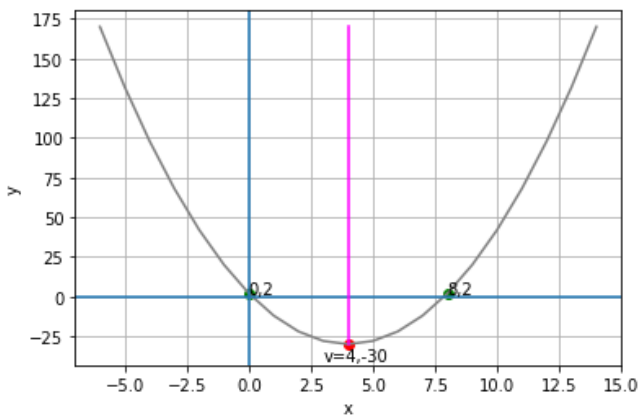
$$y = 2(0 - 4)^2 - 30$$

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:

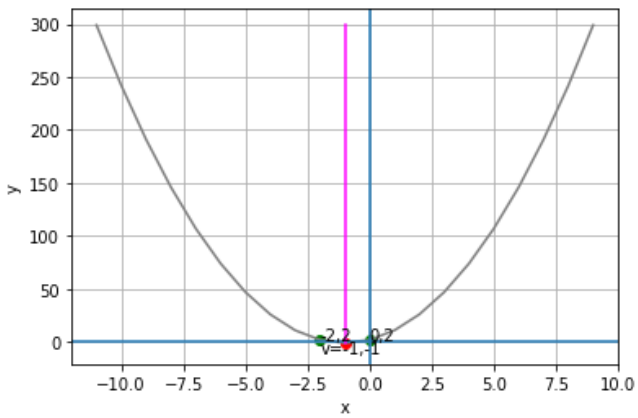
1. Move the constant to the left side:
$$y - 2 = 3x^2 + 6x$$
2. Factor the ***x*** expressions on the right:
$$y - 2 = 3(x^2 + 2x)$$
3. Add the square of half the x coefficient to the right, and the corresponding multiple on the left:
$$y - 2 + 3 = 3(x^2 + 2x + 1)$$
4. Factor out a perfect square binomial: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

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

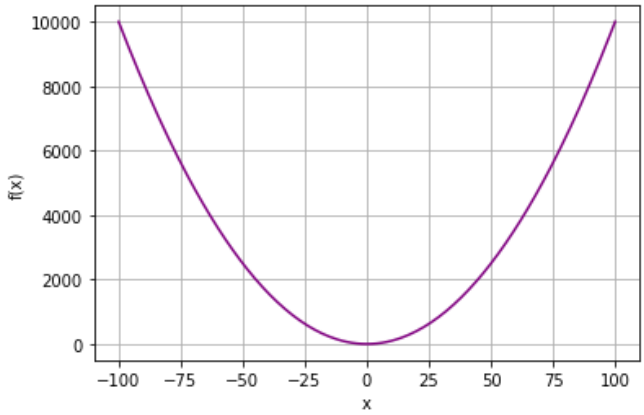
To calculate a value for ***y***, we take the ***f*** of ***x*** and subtract 1. So assuming that ***f*** is defined as previously, given an ***x*** value of 4, this equation returns a ***y*** value of **17** ($f(4)$ returns $4^2 + 2$, so 16 + 2 = 18; and then the equation subtracts 1 to give us 17).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

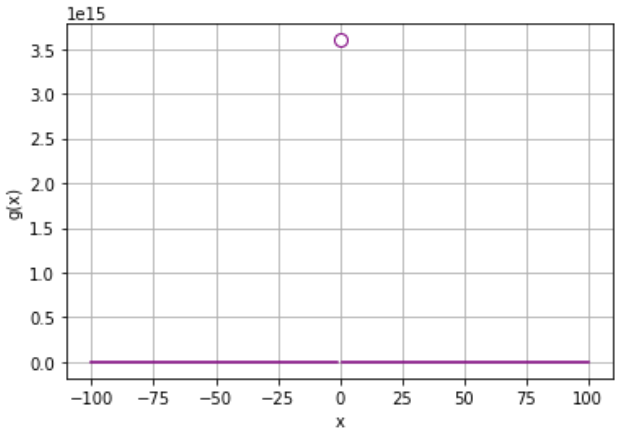
$$g(x) = (\frac{12}{2x})^2, \{x \in \mathbb{R} \mid x \neq 0\}$$

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:

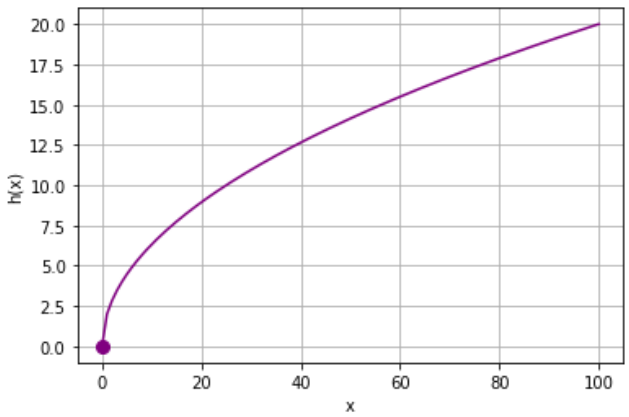
$$h(x) = 2\sqrt{x}, \quad x \geq 0$$

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:

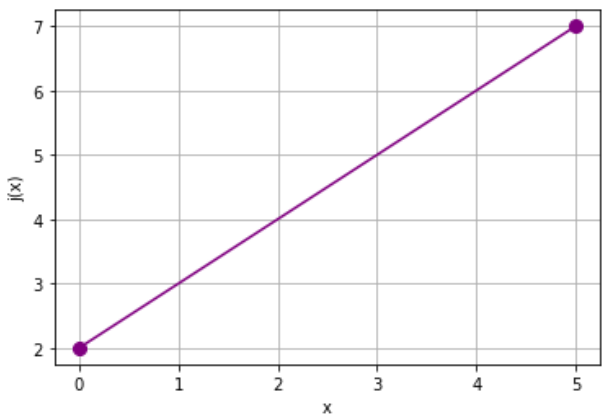
$$\{x \in \mathbb{R} \ | \ [0, 5]\}$$

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:

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

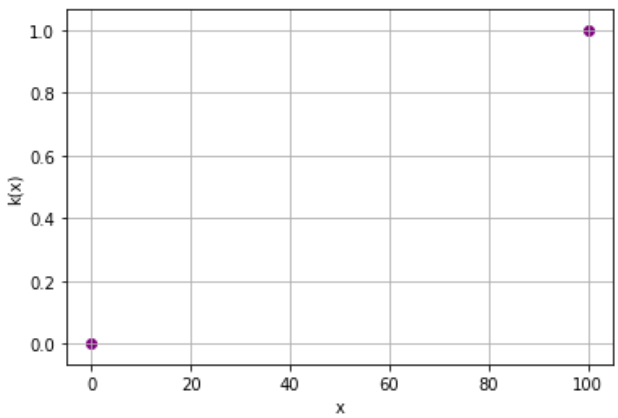
In this case, the function has highly restricted domain; it only returns a defined output for 0 and 100. No output for any other ***x*** value is defined. In this case, the set of the domain is:

$$\{0, 100\}$$

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:

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

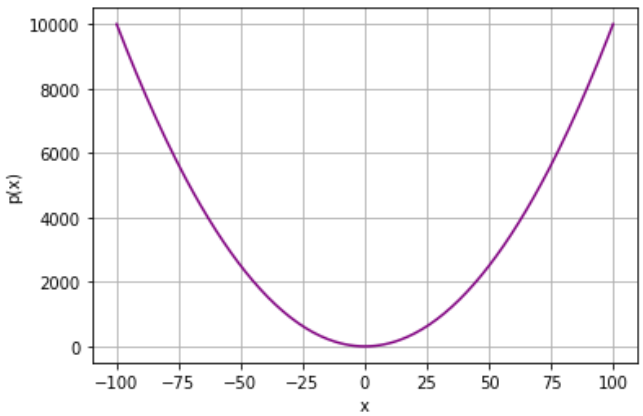
The domain of this function is all real numbers. However, this is a quadratic function, so the output values will form a parabola; and since the function has no negative coefficient or constant, it will be an upward opening parabola with a vertex that has a y value of 1.

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**.