sage_tutorial

To go to the next line in a cell, type 'Enter'. To execute a cell, type 'Shift-Enter'.

(Note: Anytime you reload a Sage worksheet, you have to reevaluate any cell that defines something you might use later.)

In the first cell, we define the letters 'a' and 'b' to be the numbers 5 and 3. Then we ask sage to perform some simple arithmetic. Click on the cell and evaluate it using 'Shift-Enter'.

(Note: Sage ignores any 'white space' between characters. I will often include spaces for readability purposes, but more or less spacing does not affect the output.)

```
a = 5
b = 3
print a + b
print a * b
print a / b
print a^b
```

```
8
15
5/3
125
```

We can use the Python 2 print command to print extra information if we would like. In the next cell, we want sage to print that '5 + 3 = 8'. We will use 'a' for 5, 'b' for 8, and let python do the arithmetic. In order to print the '+' and '=' signs, since they have special meaning used for actual addition and definition of variables, we need to put them in quotation marks to make them strings. In the print command, we separate different things to print using commas. I am doing this a lot in the tutorial so that you can see what is happening. However, the printing is unnecessary for the actual computations.

```
print a, '+', b, '=' , a + b
```

```
5 + 3 = 8
```

The equals sign '=' is used to define an object. The double equals sign '==' is used to check equality.

(Note: to put a non-executable line in your code, include a hashtag symbol # at the beginning of the line.)

```
# first, we define d to be equal to 2.
d = 2

# next, we check to see if d is equal to 2 using 'd == 2'.
print 'Is d equal to 2?', d == 2

# finally, we check to see if d is equal to 10 using 'd == 10'.
```
print 'Is d equal to 10?', d == 10

    Is d equal to 2? True
    Is d equal to 10? False

Sage has many functions, for example, it can do calculus, plot graphs, etc. By default, 'x' is a variable whenever you load Sage unless you define it to be something else. If you are going to use another variable, you have to declare it to be a variable. It may be good practice to always declare variables before you use them just in case they were used for other purposes before.

In the following cell, we define a function in terms of a variable 't'. Then we evaluate the function, differentiate the function, integrate the function, and graph the function.

```python
# first we tell Sage that 't' is a variable.
t = var( 't' )

# then we define a function 'f' in terms of t.
f = t^2

# we can evaluate the function at the point 't=5' using 'f.subs(t==5)'.
print 'evaluation of', f, 'at t=5:'
print f.subs( t==5 )
print ''
# this just prints a blank line for readability

# we can differentiate the function using 'diff(f,t)'.
print 'derivative of', f, ':
print diff(f,t)
print ''

# we can evaluate the derivative at the point 't=5'.
print 'derivative of', f, 'at t=5:'
print diff(f,t).subs( t==5 )
print ''

# we can integrate the function symbolically using 'integrate(f,t)'.
print 'integral of', f, ':'
print integrate(f,t)
print ''

# we can integrate the function from 't=1' to 't=3' using 'integrate(f,t,1,3)'.
print 'integral of', f, 'from t=1 to t=3:'
print integrate(f,t,1,3)
print ''

# we can plot the graph of the function from 't=-1' to 't=4'.
print 'graph of', f, 'from t=-1 to t=4:'
plot( f, [t,-1,4] )

    evaluation of t^2 at t=5:
    25

derivative of t^2 :
    2*t

derivative of t^2 at t=5:
    10

integral of t^2 :
    1/3*t^3

integral of t^2 from t=1 to t=3:
    26/3

graph of t^2 from t=-1 to t=4:
Now, we move onto the linear algebra. First we will learn how to define, manipulate, and plot vectors.

(Note: when we put 'QQ' in the first argument when defining a vector, this is telling Sage we are using rational numbers (fractions). We could use 'RR' instead if we are using real numbers (decimals).)

```python
# we define two vectors
u = vector( QQ, [-1,5] )
v = vector( QQ, [2,3] )

# we print the vectors to see how sage prints a vector.
print 'u:', u
print 'v:', v

# note that the first part of the print command is just identifying which vector is which.
# we could easily have just typed "print u" and "print v" to see the vectors.

# we plot the vectors
```

![Graph showing vectors u and v]

print 'graphs of u and v:'
show( plot( u, color='black' ) + plot( v, color='blue' ) )

# note that the '+' sign in between the two 'plot' commands tells Sage
to plot them on the same graph.

u: (-1, 5)
v: (2, 3)

graphs of u and v:
# next we scale the vectors and add them.

print '-2u:', -2*u
print '3v:', 3*v
print '-2u + 3v:', -2*u + 3*v

-2u: (2, -10)
3v: (6, 9)
-2u + 3v: (8, -1)

# finally, we plot all of them on the same graph.
# when i'm doing lots of plots together, i like to given them labels to refer to them easier.

pu = plot( u, color='black' )
pv = plot( v, color='black' )
p1 = plot( -2*u, color='blue' )
p2 = plot( 3*v, color='red' )
p3 = plot( -2*u + 3*v, color='purple' )

show( p1 + p2 + p3 + pu + pv )

# note that the '+' sign in between each just tells Sage to include the five plots in the same graph.
# if we want to make the previous graph a little fancier,
# we could plot the lines to complete the parallelogram between the
vectors $0$, $-2u$, $3v$, and $-2u+3v$
# to do this, we need a dashed blue line parallel to $-2u$ from $3v$ to 
# $-2u+3v$

```python
L1 = plot( line( [ 3*v, -2*u + 3*v ], linestyle='dashed', color='blue' ) )
```

# and we need a dashed red line parallel to $3v$ from $-2u$ to $-2u+3v$

```python
L2 = plot( line( [ -2*u, -2*u + 3*v ], linestyle='dashed', color='red' ) )
```

# then we plot them all together

```python
L1 + L2 + p1 + p2 + p3 + pu + pv
```
Now we move onto matrices. The definition of a matrix is similar to that of a vector, but we put all the entries of a ROW in each set of brackets and we separate each row with a comma. You don't need to include the line breaks, but the line breaks are helpful to see what the matrix should look like.

```python
# first we define a matrix A
A = matrix( QQ, [
    [ 1,  2,  5 ],
    [ -2,  3,  4 ],
    [  4, -5, -6 ]
] )

# now we print the matrix to see that it worked
print A
[[1 2 5]
 [ -2 3 4]
 [ 4 -5 -6]]

# for this class, one of the best Sage functions is '.rref()',
# which finds the reduced row echelon form of a matrix.
A.rref()
[[1 0 1]
 [0 1 2]
 [0 0 0]]

# another nice function is '.pivots()',
# which lists the columns that are pivots.
# in this example above, we can see that the first and second columns are pivots;
# however, Sage (and python) start counting from 0, so 0 is the first
We want to be able to solve matrix equations \( Ax = b \), where \( A \) is a matrix, \( b \) is a vector, and \( x \) is an unknown vector. I will show you two methods. The first method finds one solution if possible. However, it does not tell us whether it is a unique solution or if there are infinitely many solutions. The second method is more familiar and gives us more information.

We keep our same matrix \( A \) from the previous and we define two vectors \( b_1 \) and \( b_2 \).

\[
\begin{align*}
b_1 &= \text{vector}( \text{QQ}, [1, 5, -9] ) \\
b_2 &= \text{vector}( \text{QQ}, [4, 4, 4] )
\end{align*}
\]

We solve \( Ax = b_1 \) in two ways.

# First, we find a solution directly (if possible) by

\[
A \ b_1
\]

\((-1, 1, 0)\)

# Next, we find all solutions by augmenting the matrix \( A \) with the vector \( b_1 \), and then doing row reduction.

\[
A.\text{augment}(b_1).\text{rref}()
\]

\[
\begin{bmatrix}
1 & 0 & 1 & -1 \\
0 & 1 & 2 & 1 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

Notice how the first method was deficient in this case because it only gave us one solution without telling us whether there are infinitely many or not.

# We try to solve \( Ax = b_2 \) using the first method

\[
A \ b_2
\]

Traceback (click to the left of this block for traceback)

...  

ValueError: matrix equation has no solutions

# The second approach here is again MUCH better  
# because we can see why there is no solution by looking at the reduced row echelon form.
A.augment(b2).rref()

    [1 0 1 0]
    [0 1 2 0]
    [0 0 0 1]

# We can include the line which subdivides the matrix from that augmented vector if we want.
# To do this, we include the argument 'subdivide=True' in the '.augment()' command.

print 'augmented reduction for Ax = b1:
print A.augment(b1, subdivide=True).rref()

print ''

print 'augmented reduction for Ax = b2:
print A.augment(b2, subdivide=True).rref()

    augmented reduction for Ax = b1:
        [ 1  0  1|-1]
        [ 0  1  2| 1]
        [ 0  0  0| 0]

    augmented reduction for Ax = b2:
        [1 0 1|0]
        [0 1 2|0]
        [0 0 0|1]

Let's put this all together for a few examples.

Example 1.

A = matrix( QQ, [
    [  1,  5 ],
    [ -3,  2 ]
] )

b = vector( QQ, [17, 0] )

print A.augment(b, subdivide=True).rref()

    [1 0|2]
    [0 1|3]

# Example 1 (continued).

# We see that the unique solution to Ax = b is the vector x = (2, 3)

# First, we interpret this as the unique intersection point of two lines: the line x + 5y = 17 and the line -3x + 2y = 0.
we set 'x' and 'y' to be variables, then we use 'implicit_plot' to graph the two lines.  
we will make the first line blue and the second red.  also, we include the axes for reference (optional).  
if we have done everything correctly, we will see the lines intersect at the point (2, 3)

# define the variables
x, y = var( 'x', 'y' )

# define the equations
eq1 = x + 5*y == 17
eq2 = -3*x + 2*y == 0

# define the plots
p1 = implicit_plot( eq1, (x,-3,7), (y,-2,8), axes=True, color='blue' )
p2 = implicit_plot( eq2, (x,-3,7), (y,-2,8), axes=True, color='red' )

#display the plots
p1 + p2
# Example 1 (continued).

# Next, we interpret the solution as a solution to the linear combination problem
# $x_1 \cdot a_1 + x_2 \cdot a_2 = b$, where $x_1 = 2$, $x_2 = 3$, and $a_1$, $a_2$ are the columns of the matrix $A$.
# In other words, we found that
# $2 \cdot (1,-3) + 3 \cdot (5,2) = (17,0)$.

# We will illustrate this by defining vectors and plotting the linear combinations as before.

# define the column vectors

a1 = vector( QQ, [1,-3] )
a2 = vector( QQ, [5,2] )
# define the vector plots
p1 = plot( a1, color='black' )
p2 = plot( a2, color='black' )

pp1 = plot( 2*a1, color='red' )
pp2 = plot( 3*a2, color='blue' )

pb = plot( b, color='purple' )

# define the line plots:
# the blue line parallel to 3*a2 between 2*a1 and b,
# and the red line parallel to 2*a1 between 3*a2 and b
L1 = plot( line( [ 2*a1, b], linestyle='dashed', color='blue' ) )
L2 = plot( line( [ 3*a2, b], linestyle='dashed', color='red' ) )

# display the plots
L1 + L2 + pp1 + pp2 + p1 + p2 + pb
Example 2.

\[
A = \begin{bmatrix}
1 & 3 \\
-2 & 2 \\
4 & -1 \\
\end{bmatrix}
\]

\[
b = \begin{bmatrix}
4 \\
0 \\
9 \\
\end{bmatrix}
\]

\[
\text{print } A\text{.augment}(b, \text{subdivide=True}).\text{rref()}
\]

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

# Example 2 (continued).

# We see that there is no solution to \( Ax = b \).

# We interpret this as the fact that there is no common intersection point between
# the line \( x + 3y = 4 \), the line \(-2x + 2y = 0\), and the line \( 4x - y = 9 \).

# We illustrate this by plotting the three lines.

\[
x, y = \text{var}( 'x', 'y' )
\]

\[
eq_1 = x + 3*y == 4
\]

\[
eq_2 = -2*x + 2*y == 0
\]

\[
eq_3 = 4*x - y == 9
\]

\[
p1 = \text{implicit_plot( eq1, (x,-3,5), (y,-3,5), axes=True, color='black' )}
p2 = \text{implicit_plot( eq2, (x,-3,5), (y,-3,5), axes=True, color='blue' )}
p3 = \text{implicit_plot( eq3, (x,-3,5), (y,-3,5), axes=True, color='red' )}
\]

\[
p1 + p2 + p3
\]
# Example 2 (continued).

# We interpret the matrix equation as a linear combination problem of finding \( b \) in terms of \( a_1 \) and \( a_2 \).
# Since there is no solution, that means that \( b \) is not in the plane spanned by \( a_1 \) and \( a_2 \).

# We plot these 3 vectors in 3-space to see that \( b \) is not in the same plane as \( a_1 \) and \( a_2 \).

\[
a_1 = \text{vector}( \mathbb{Q}, [1, -2, 4] ) \\
a_2 = \text{vector}( \mathbb{Q}, [3, 2, -1] ) \\
p_1 = \text{plot}( a_1, \text{color}='blue' ) \\
p_2 = \text{plot}( a_2, \text{color}='red' ) \\
p_b = \text{plot}( b, \text{color}='purple' )
\]

\( p_1 + p_2 + p_b \)
Example 3.

# We redo Example 2 but we change the vector b to the vector (7, 2, 2).

```python
c = matrix( QQ, [
    [  1,  3 ],
    [ -2,  2 ],
    [  4, -1 ]
  ] )

b = vector( QQ, [7, 2, 2] )

print A.augment(b, subdivide=True).rref()
```

```
[1 0|1]
[0 1|2]
[0 0|0]
```

# Example 3 (continued).

# Now the three lines, which have the same slopes as Example 2, intersect at the point (1,2).
# The new lines are \( x + 3y = 7 \), \( -2x + 2y = 2 \), and \( 4x - y = 2 \).

# We illustrate this by plotting the three lines.
x, y = var('x', 'y')

eq1 = x + 3*y == 7
eq2 = -2*x + 2*y == 2
eq3 = 4*x - y == 2

p1 = implicit_plot(eq1, (x,-3,5), (y,-3,5), axes=True, color='black')
p2 = implicit_plot(eq2, (x,-3,5), (y,-3,5), axes=True, color='blue')
p3 = implicit_plot(eq3, (x,-3,5), (y,-3,5), axes=True, color='red')

p1 + p2 + p3

# Example 3 (continued).
# And now the vector b is in the span of a1 and a2, namely b = a1 + 2*a2;
# that is, (7,2,2) = (1,-2,4) + 2*(3,2,1)
\( a_1 = \text{vector}( \mathbb{Q}, [1, -2, 4] ) \)
\( a_2 = \text{vector}( \mathbb{Q}, [3, 2, -1] ) \)

\( p_1 = \text{plot}( a_1, \text{color}='\text{blue}' ) \)
\( p_2 = \text{plot}( a_2, \text{color}='\text{black}' ) \)
\( pp_2 = \text{plot}( 2*a_2, \text{color}='\text{red}' ) \)
\( pb = \text{plot}( b, \text{color}='\text{purple}' ) \)

\( L_1 = \text{plot}( \text{line}( [2*a_2, b], \text{color}='\text{blue}' ) ) \)
\( L_2 = \text{plot}( \text{line}( [a_1, b], \text{color}='\text{red}' ) ) \)

\( L_1 + L_2 + pp_2 + p_1 + p_2 + pb \)

---

Example 4.

\( A = \text{matrix}( \mathbb{Q}, [ \)
\[ -2, 3, 7 \],
\[ 2, 2, -2 \] \]
\] )

\( b = \text{vector}( \mathbb{Q}, [6, 14] ) \)
A.augment(b, subdivide=True).rref()

\[
\begin{bmatrix}
1 & 0 & -2 & 3 \\
0 & 1 & 1 & 4 \\
\end{bmatrix}
\]

# Example 4 (continued).

# The system has infinitely many solutions, namely
# x1 - 2*x3 = 3
# x2 + x3 = 4
# x3 is free, so x3 = t

# x1 = 3 + 2t
# x2 = 4 - t
# x3 = t

# in other words, the solution is (3 + 2t, 4 - t, t), which is a line in 3-space.

# the corresponding system of equations represents 2 planes in 3 space
# that intersect at the line given.
# first plane: -2x +3y + 7z = 6
# second plane: 2x + 2y - 2z = 14
# intersection line: (3 + 2t, 4 - t, t)

# we'll plot these to see the intersection.

x, y, z = var( 'x', 'y', 'z' )

eq1 = -2*x + 3*y + 7*z == 6
eq2 = 2*x + 2*y - 2*z == 14

p1 = implicit_plot3d( eq1, (x, 0, 6), (y, 0, 6), (z, -3, 3), color='blue' )
p2 = implicit_plot3d( eq2, (x, 0, 6), (y, 0, 6), (z, -3, 3), color='green' )

# we'll plot the line from 't=-1' to 't=1'
# that is, from the point (1, 5, -1) to the point (5, 3, 1)
L = plot( line( [ [1, 5, -1], [5, 3, 1] ], color='red', thickness=5 ) )

p1 + p2 + L
# Example 4 (continued).

# We interpret the matrix equation as a linear combination problem, 
# that is, we want to write \( b \) as a linear combination of the columns of 
# \( A \): \( a_1 \), \( a_2 \), and \( a_3 \). 
# there are infinitely many solutions. i'll pick 3 solutions to display 
# for illustration purposes.

# Solution #1
# if \( t=0 \), we get \( x=(3,4,0) \), so \( b = 3a_1 + 4a_2 \),
# \((6,14) = 3*(-2,2) + 4*(3,2)\)
# which we plot below.

```
a1 = vector( QQ, [-2,2] )
a2 = vector( QQ, [3,2] )
p1 = plot( a1, color='black' )
p2 = plot( a2, color='black' )
pp1 = plot( 3*a1, color='red' )
pp2 = plot( 4*a2, color='blue' )
pb = plot( b, color='purple' )
L1 = plot( line( [4*a2, b], linestyle='dashed', color='red' ) )
L2 = plot( line( [3*a1, b], linestyle='dashed', color='blue' ) )
L1 + L2 + pp1 + pp2 + p1 + p2 + pb
```
# Solution #2
# if t=4, we get x=(11,0,4), so b = 11*a1 + 4*a3,
# (6,14) = 11*(-2,2) + 4*(7,-2),
# which we plot below.

a3 = vector( QQ, [7,-2] )

p3 = plot( a3, color='black' )
pp1 = plot( 11*a1, color='red' )
pp3 = plot( 4*a3, color='blue' )

L1 = plot( line( [4*a3, b], linestyle='dashed', color='red' ) )
L2 = plot( line( [11*a1, b], linestyle='dashed', color='blue' ) )

L1 + L2 + pp1 + pp3 + p1 + p3 + pb
# Solution #3
# t=1, we get x=(5,3,1), so b = 5*a1 + 3*a2 + a3,
# (6,14) = 5*(-2,2) + 3*(3,2) + (7,-2)
# which we plot below.

pp1 = plot( 5*a1, color='red' )
pp2 = plot( 3*a2, color='blue' )

L12 = plot( line( [ 3*a2, 5*a1 + 3*a2 ], linestyle='dashed', color='red' ) )
L21 = plot( line( [ 5*a1, 5*a1 + 3*a2 ], linestyle='dashed', color='blue' ) )
L13 = plot( line( [ a3, 5*a1 + a3 ], linestyle='dashed', color='red' ) )
L31 = plot( line( [ 5*a1, 5*a1 + a3 ], linestyle='dashed', color='black' ) )
L23 = plot( line( [ a3, 3*a2 + a3 ], linestyle='dashed', color='blue' ) )
L32 = plot( line( [ 3*a2, 3*a2 + a3 ], linestyle='dashed', color='black' ) )
L1b = plot( line( [ 3*a2 + a3, b ], linestyle='dashed', color='red' ) )
L2b = plot( line( [ 5*a1 + a3, b ], linestyle='dashed', color='blue' ) )
L3b = plot( line( [ 5*a1 + 3*a2, b ], linestyle='dashed', color='black' ) )

L12 + L21 + L13 + L31 + L23 + L32 + L1b + L2b + L3b + pp1 + pp2 + p1 + p2 + p3 + pb