PREP Tutorials

Release 10.4

Rob Beezer, Karl-Dieter Crisman, and Jason Grout

Jul 23, 2024

CONTENTS

This is a set of tutorials developed for the MAA PREP workshops "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by the National Science Foundation under grant DUE 0817071) in the summers of 2010-2012.

The original audience for these tutorials was mathematics faculty at undergraduate institutions with little or no experience with programming or computer mathematics software. Because the computer experience required is quite minimal, this should be useful to anyone coming with little such experience to Sage.

Although any mathematics the reader is not familiar with can simply be skipped at first, we do assume throughout that the reader is familiar with the concept of a function and different kinds of numbers. We also make liberal use of basic examples from calculus, linear algebra, and other areas. In the *Quickstart tutorials*, we assume familiarity with the topics at the level of a student who has just completed a course in the subject, or of a faculty member who is about to teach it.

Contents:

CHAPTER

ONE

LOGGING ON AND MAKING A WORKSHEET

This Sage worksheet is from a series of tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071).

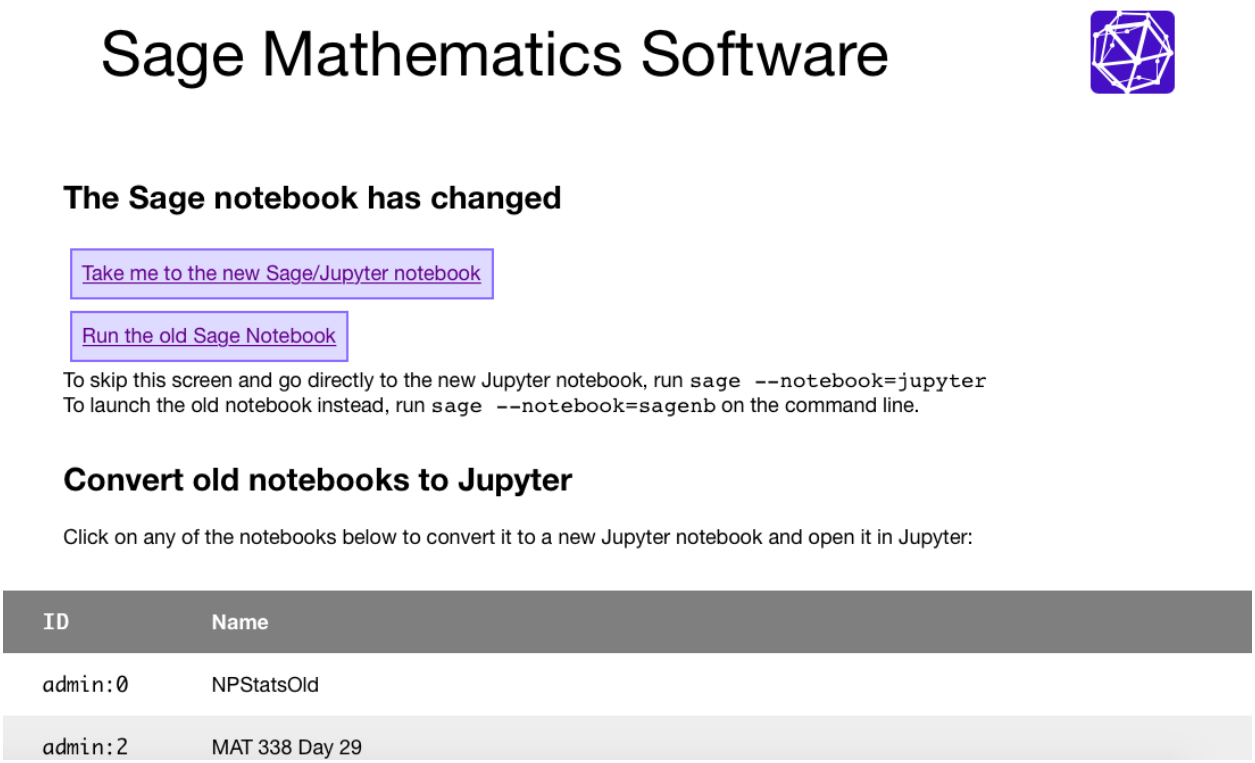
This document describes how to get into a Sage worksheet in the first place. If you already feel comfortable with this process, or at least comfortable enough to see how to actually use Sage, the main content of the tutorials begins with *the introductory tutorial*.

There are three main types of worksheets for Sage, all of which have somewhat similar behavior.

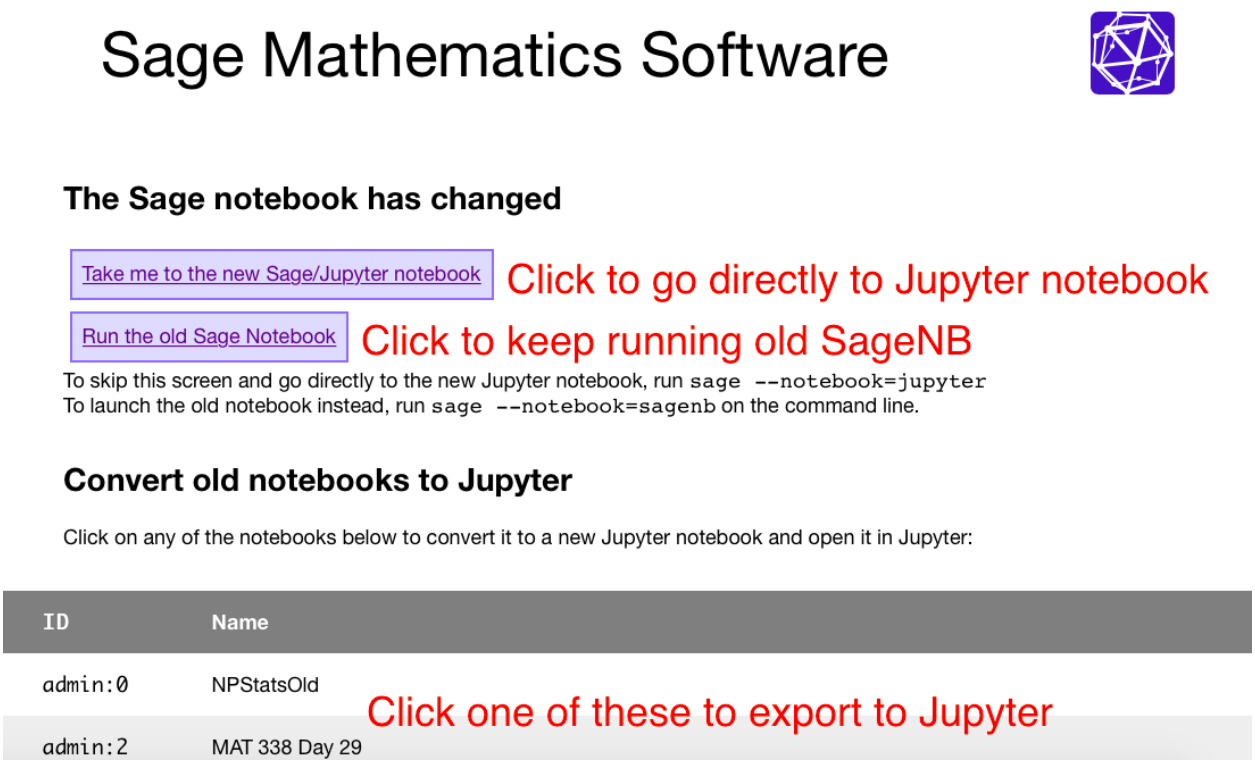
- If you are using the Jupyter notebook or starting Sage from the command line, you may see some screens about *exporting*. We have basic information about this.
- If you are using the CoCalc SageMath worksheets, you will want to contact them or read some of their documentation for further assistance.

1.1 The Export screen and Jupyter notebook

Starting in Sage 8.0, the default is to provide the Jupyter notebook for your worksheet experience via an export screen. When you start Sage you may see a screen like this.

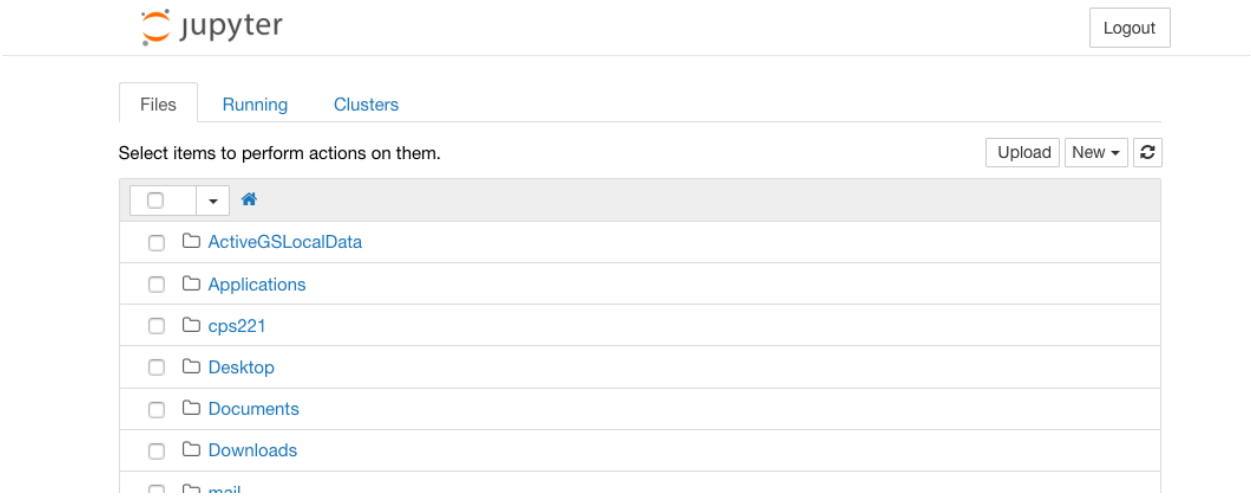


There are three actions you can take, each of which is highlighted in the next picture. Note that if you have no previous worksheets, the third option to "export" them will not make sense.

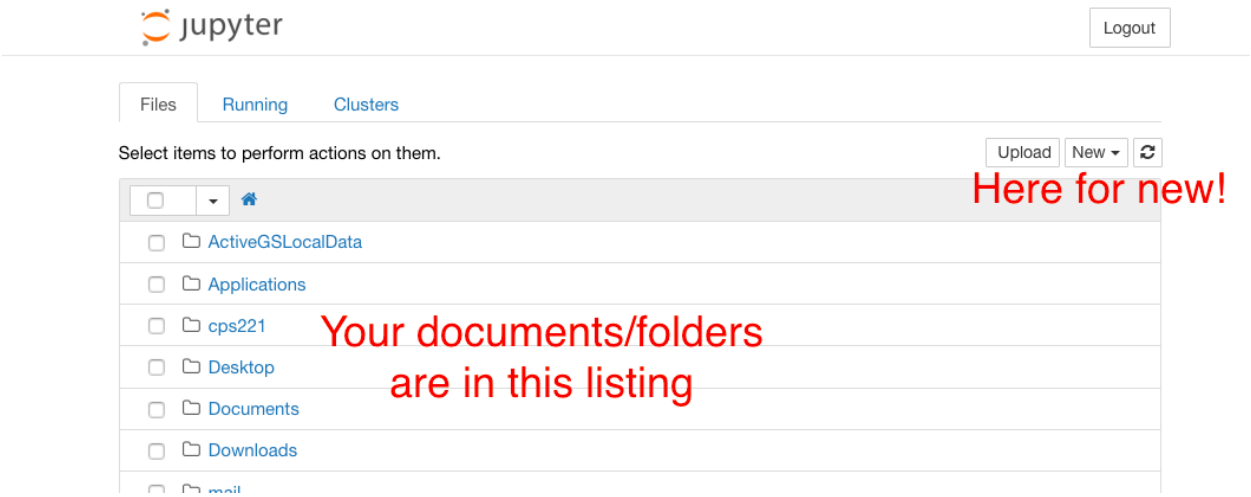


The legacy SageNB has been retired in Sage 9.1. Please use the Jupyter notebook. Jupyter will bring you to a screen that

is simply a listing of files in whatever folder Sage has opened in.

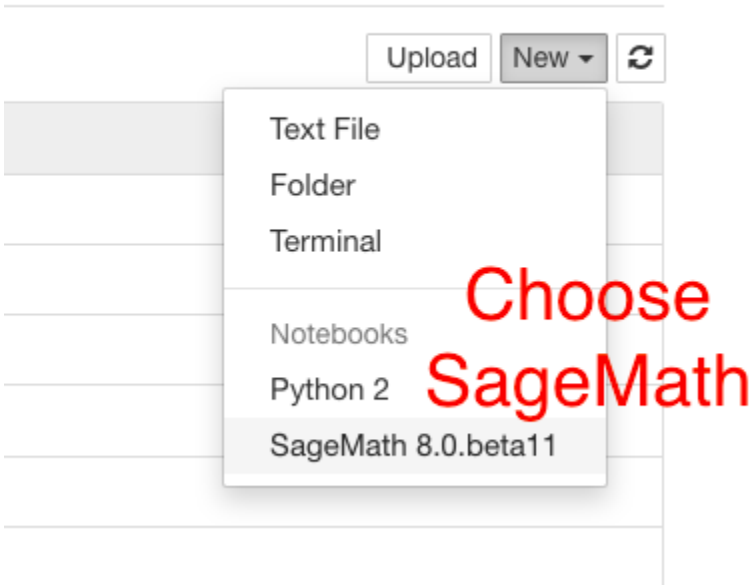


If you want to start a worksheet, you will look at the upper right corner and ask for a new worksheet:

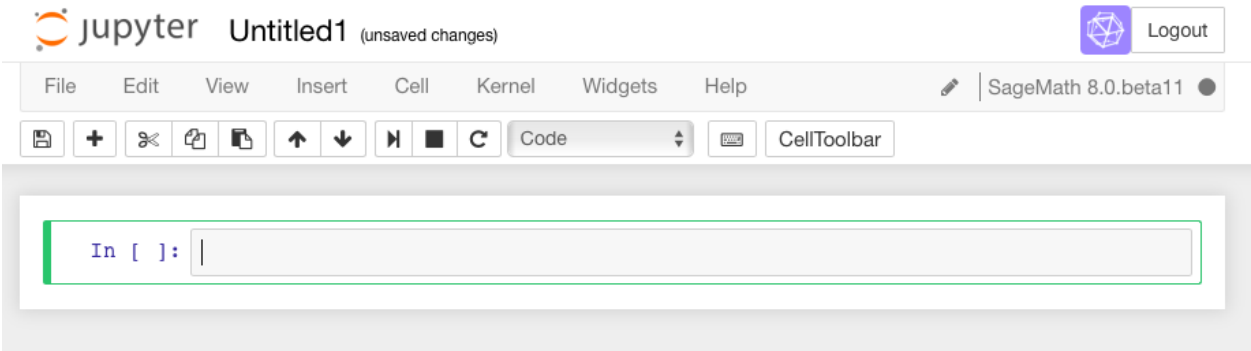


Note: The Jupyter notebook saves your files locally in your normal filesystem, as normal file names. So if you start the notebook from a different location than usual, you may have to navigate a bit to find your worksheet.

Jupyter will allow you many types of files to open. To use SageMath directly, just choose the Sage type; this will ensure that Jupyter runs using Sage and not pure Python or some other language.



You should now have a worksheet that looks more or less like this.



Now you are ready to begin to *evaluate Sage commands*!

CHAPTER

TWO

INTRODUCTORY SAGE TUTORIAL

This Sage document is the first in a series of tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

If you are unsure how to log on to a Sage server, start using a local installation, or to create a new worksheet, you might find the *prelude on logging in* helpful.

Otherwise, you can continue with this tutorial, which has the following sections:

- *Evaluating Sage Commands*
  - See *Evaluating in the Jupyter notebook* for the Jupyter notebook
- *Functions in Sage*
- *Help inside Sage*
- *Annotating with Sage*
  - See *Jupyter Annotation* for the Jupyter notebook

This tutorial only introduces the most basic level of functionality. Later tutorials address topics such as calculus, advanced plotting, and a wide variety of specific mathematical topics.

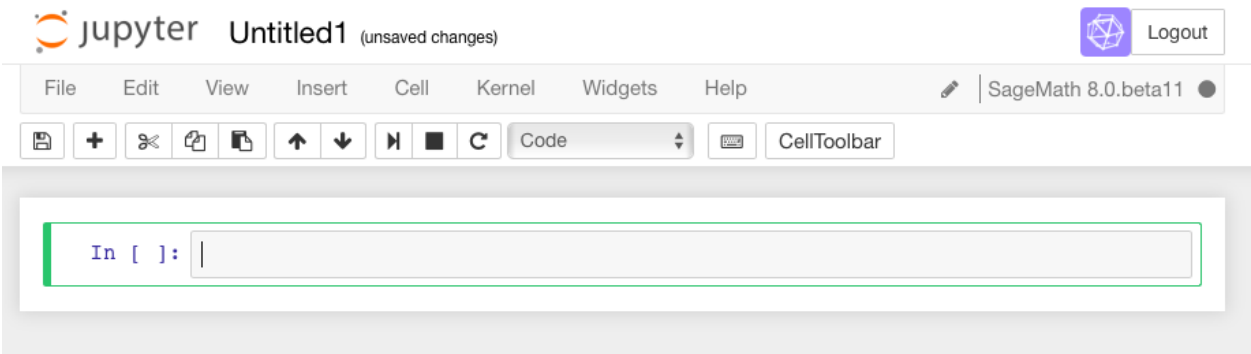
2.1 Evaluating Sage Commands

Or, How do I get Sage to do some math?

- See *Evaluating in the Jupyter notebook* for the Jupyter notebook

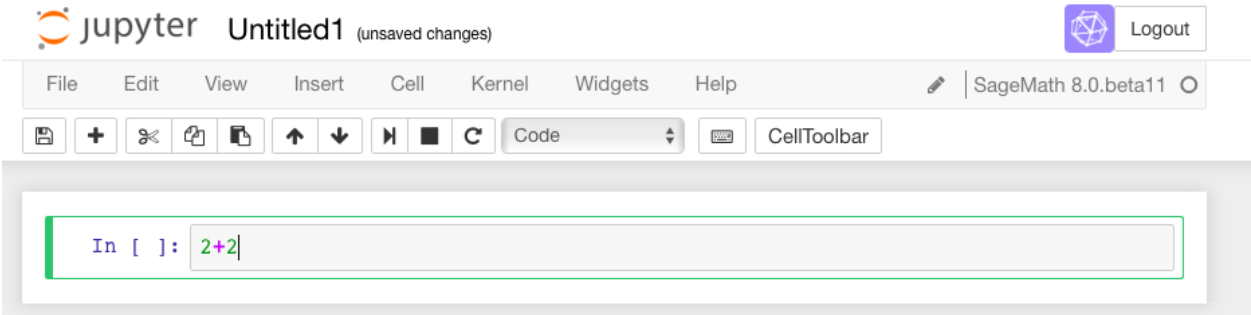
2.1.1 Evaluating in the Jupyter notebook

In a Jupyter worksheet, there are little boxes called *input cells* or *code cells*. They should be about the width of your browser.

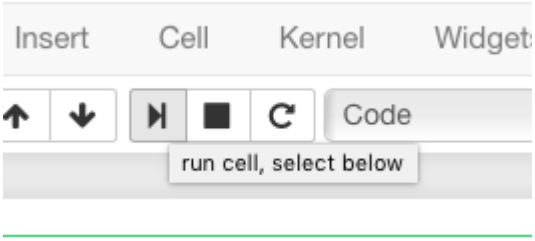


To do math in a Jupyter cell, one must do two things.

- First, click inside the cell so that the cell is active (i.e., has a bright green border). This was already the case in the first cell above. (If it is blue, the Jupyter notebook is in "command mode"). Type some math in it.

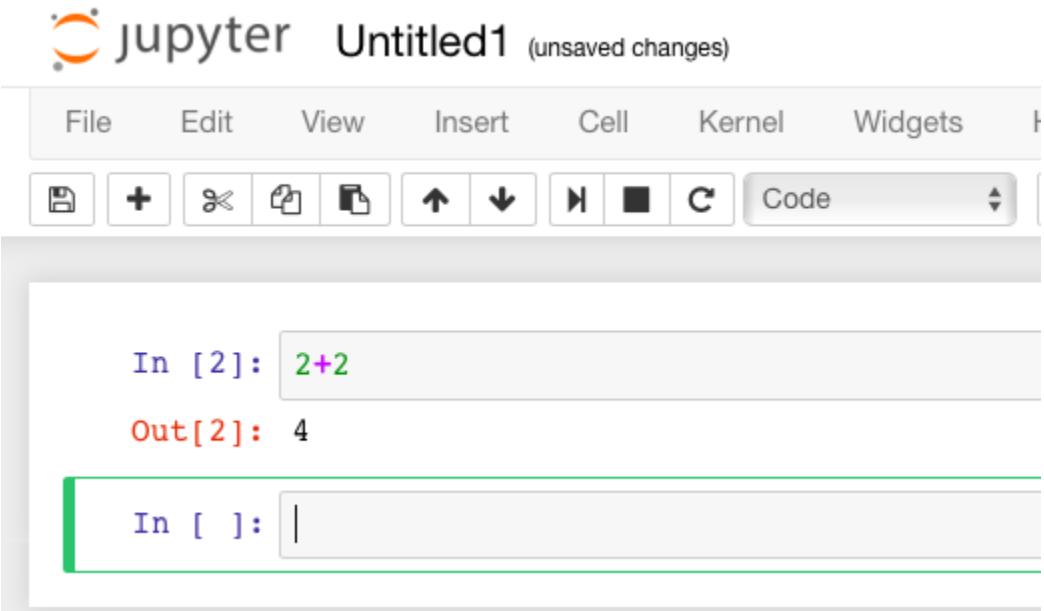


- Then, there are two options. A not-very-evident icon that looks like the "play" symbol on a recording device can be clicked:



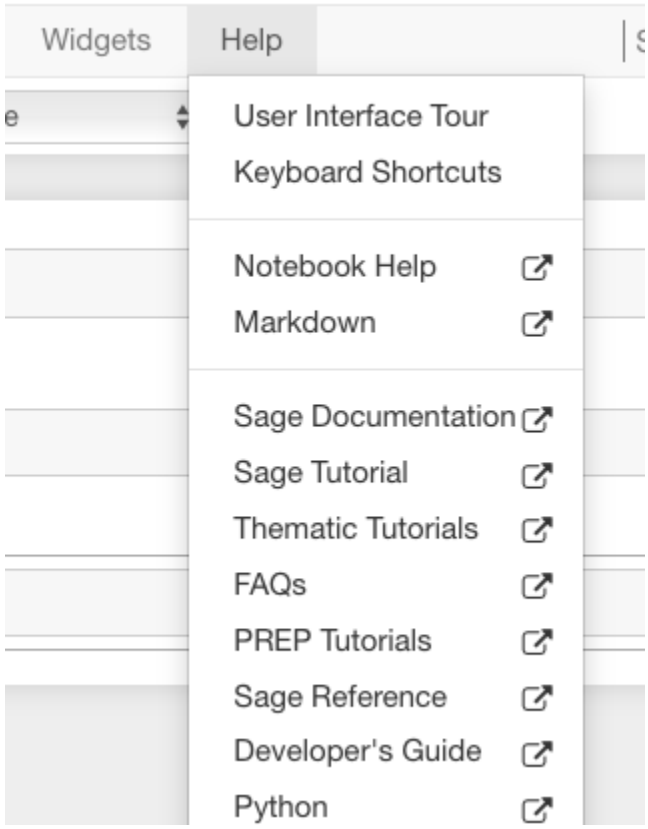
  Or one can use the keyboard shortcut of holding down the `Shift` key while you press the `Enter` key. We call this `Shift` + `Enter`.

Sage prints out its response just below the cell (that's the 4 below, so Sage confirms that $2 + 2 = 4$). Note also that Sage has automatically made a new cell, and made it active, after you evaluated your first cell.



To do more mathematics, just do the same thing with more cells!

One has to learn a variety of keyboard shortcuts or click on various menu items to manipulate cells. There is a help menu to get you started on this; the Jupyter developers also maintain an example notebook which may assist you.



2.2 Functions in Sage

To start out, let's explore how to define and use functions in Sage.

For a typical mathematical function, it's pretty straightforward to define it. Below, we define a function.

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

```
sage: f(x)=x^2
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(2)).function(x)
```

Since all we wanted was to create the function $f(x)$, Sage just does this and doesn't print anything out back to us.

We can check the definition by asking Sage what `f(x)` is:

```
sage: f(x)
x^2
```

```
>>> from sage.all import *
>>> f(x)
x^2
```

If we just ask Sage what `f` is (as opposed to `f(x)`), Sage prints out the standard mathematical notation for a function that maps a variable x to the value x^2 (with the "maps to" arrow $\mapsto$ as `|-->`).

```
sage: f
x |--> x^2
```

```
>>> from sage.all import *
>>> f
x |--> x^2
```

We can evaluate f at various values.

```
sage: f(3)
9
```

```
>>> from sage.all import *
>>> f(Integer(3))
9
```

```
sage: f(3.1)
9.61000000000000
```

```
>>> from sage.all import *
>>> f(RealNumber('3.1'))
9.61000000000000
```

```
sage: f(31/10)
961/100
```

```
>>> from sage.all import *
>>> f(Integer(31)/Integer(10))
961/100
```

Notice that the output type changes depending on whether the input had a decimal; we'll see that again below.

Naturally, we are not restricted to x as a variable. In the next cell, we define the function $g(y) = 2y - 1$.

```
sage: g(y)=2*y-1
```

```
>>> from sage.all import *
>>> __tmp__=var("y"); g = symbolic_expression(Integer(2)*y-Integer(1)).function(y)
```

However, we need to make sure we do define a function if we use a new variable. In the next cell, we see what happens if we try to use a random input by itself.

```
sage: z^2
Traceback (most recent call last):
...
NameError: name 'z' is not defined
```

```
>>> from sage.all import *
>>> z**Integer(2)
Traceback (most recent call last):
...
NameError: name 'z' is not defined
```

This is explained in some detail in following tutorials. At this point, it suffices to know using the function notation (like `g(y)`) tells Sage you are serious about `y` being a variable.

One can also do this with the `var('z')` notation below.

```
sage: var('z')
z
sage: z^2
z^2
```

```
>>> from sage.all import *
>>> var('z')
z
>>> z**Integer(2)
z^2
```

This also demonstrates that we can put several commands in one cell, each on a separate line. The output of the last command (if any) is printed as the output of the cell.

Sage knows various common mathematical constants, like π (`pi`) and e.

```
sage: f(pi)
pi^2
```

```
>>> from sage.all import *
>>> f(pi)
pi^2
```

```
sage: f(e^-1)
e^(-2)
```

```
>>> from sage.all import *
>>> f(e**-Integer(1))
e^(-2)
```

In order to see a numeric approximation for an expression, just type the expression inside the parentheses of `N()`.

```
sage: N(f(pi))
9.86960440108936
```

```
>>> from sage.all import *
>>> N(f(pi))
9.86960440108936
```

Another option, often more useful in practice, is having the expression immediately followed by `.n()` (note the dot).

```
sage: f(pi).n()
9.86960440108936
```

```
>>> from sage.all import *
>>> f(pi).n()
9.86960440108936
```

For now, we won't go in great depth explaining the reasons behind this syntax, which may be new to you. For those who are interested, Sage often uses this type of syntax (known as "object-oriented") because…

- Sage uses the Python programming language, which uses this syntax, 'under the hood', and
- Because it makes it easier to distinguish among
- The mathematical object,

- The thing you are doing to it, and
- Any ancillary arguments.

For example, the following numerically evaluates (`n`) the constant π (`pi`) to twenty digits (`digits=20`).

```
sage: pi.n(digits=20)
3.1415926535897932385
```

```
>>> from sage.all import *
>>> pi.n(digits=Integer(20))
3.1415926535897932385
```

Sage has lots of common mathematical functions built in, like $\sqrt{x}$ (`sqrt(x)`) and $\ln(x)$ (`ln(x)` or `log(x)`).

```
sage: log(3)
log(3)
```

```
>>> from sage.all import *
>>> log(Integer(3))
log(3)
```

Notice that there is no reason to numerically evaluate $\log(3)$, so Sage keeps it symbolic. The same is true in the next cell - $2\log(3) = \log(9)$, but there isn't any reason to do that; after all, depending on what you want, $\log(9)$ may be simpler or less simple than you need.

```
sage: log(3)+log(3)
2*log(3)
```

```
>>> from sage.all import *
>>> log(Integer(3))+log(Integer(3))
2*log(3)
```

```
sage: log(3).n()
1.09861228866811
```

```
>>> from sage.all import *
>>> log(Integer(3)).n()
1.09861228866811
```

Notice again that Sage tries to respect the type of input as much as possible; adding the decimal tells Sage that we have approximate input and want a more approximate answer. (Full details are a little too complicated for this introduction.)

```
sage: log(3.)
1.09861228866811
```

```
>>> from sage.all import *
>>> log(RealNumber('3.'))
1.09861228866811
```

```
sage: sqrt(2)
sqrt(2)
```

```
>>> from sage.all import *
>>> sqrt(Integer(2))
sqrt(2)
```

If we want this to look nicer, we can use the `show` command. We'll see more of this sort of thing below.

```
sage: show(sqrt(2))
```

```
>>> from sage.all import *
>>> show(sqrt(Integer(2)))
```

$$\sqrt{2}$$

```
sage: sqrt(2).n()
1.41421356237310
```

```
>>> from sage.all import *
>>> sqrt(Integer(2)).n()
1.41421356237310
```

Do you remember what f does?

```
sage: f(sqrt(2))
2
```

```
>>> from sage.all import *
>>> f(sqrt(Integer(2)))
2
```

We can also plot functions easily.

```
sage: plot(f, (x,-3,3))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(f, (x,-Integer(3),Integer(3)))
Graphics object consisting of 1 graphics primitive
```

In another tutorial, we will go more in depth with plotting. Here, note that the preferred syntax has the variable and endpoints for the plotting domain in parentheses, separated by commas.

If you are feeling bold, plot the `sqrt` function in the next cell between 0 and 100.

2.3 Help inside Sage

There are various ways to get help for doing things in Sage. Here are several common ways to get help as you are working in a Sage worksheet.

2.3.1 Documentation

Sage includes extensive documentation covering thousands of functions, with many examples, tutorials, and other helps.

- One way to access these is to click the "Help" link at the top right of any worksheet, then click your preferred option at the top of the help page.
- They are also available any time online at the Sage website, which has many other links, like video introductions.
- The Quick Reference cards are another useful tool once you get more familiar with Sage.

Our main focus in this tutorial, though, is help you can immediately access from within a worksheet, where you don't have to do *any* of those things.

2.3.2 Tab completion

The most useful help available in the notebook is "tab completion". The idea is that even if you aren't one hundred percent sure of the name of a command, the first few letters should still be enough to help find it. Here's an example.

- Suppose you want to do a specific type of plot - maybe a slope field plot - but aren't quite sure what will do it.
- Still, it seems reasonable that the command might start with `pl`.
- Then one can type `pl` in an input cell, and then press the `Tab` key to see all the commands that start with the letters `pl`.

Try tabbing after the `pl` in the following cell to see all the commands that start with the letters `pl`. You should see that `plot_slope_field` is one of them.

```
sage: pl
```

```
>>> from sage.all import *
>>> pl
```

To pick one, just click on it; to stop viewing them, press the `Escape` key.

You can also use this to see what you can do to an expression or mathematical object.

- Assuming your expression has a name, type it;
- Then type a period after it,
- Then press tab.

You will see a list pop up of all the things you can do to the expression.

To try this, evaluate the following cell, just to make sure f is defined.

```
sage: f(x)=x^2
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(2)).function(x)
```

Now put your cursor after the period and press your `Tab` key.

```
sage: f.
```

```
>>> from sage.all import *
>>> f.
```

Again, `Escape` should remove the list.

One of the things in that list above was `integrate`. Let's try it.

```
sage: f.integrate(x)
x |--> 1/3*x^3
```

```
>>> from sage.all import *
>>> f.integrate(x)
x |--> 1/3*x^3
```

2.3.3 Finding documentation

Or, Why all the question marks?

In the previous example, you might have wondered why I needed to put `f.integrate(x)` rather than just `f.integrate()`, by analogy with `sqrt(2).n()`.

To find out, there is another help tool one can use from right inside the notebook. Almost all documentation in Sage has extensive examples that can illustrate how to use the function.

- As with tab completion, type the expression, period, and the name of the function.
- Then type a question mark.
- Press tab *or* evaluate to see the documentation.

To see how this help works, move your cursor after the question mark below and press `Tab`.

```
sage: f.integrate?
```

```
>>> from sage.all import *
>>> f.integrate?
```

The examples illustrate that the syntax requires `f.integrate(x)` and not just `f.integrate()`. (After all, the latter could be ambiguous if several variables had already been defined).

To stop viewing the documentation after pressing `Tab`, you can press the `Escape` key, just like with the completion of options.

If you would like the documentation to be visible longer-term, you can *evaluate* a command with the question mark (like below) to access the documentation, rather than just tabbing. Then it will stay there until you remove the input cell.

```
sage: binomial?
```

```
>>> from sage.all import *
>>> binomial?
```

Try this with another function!

2.3.4 Finding the source

There is one more source of help you may find useful in the long run, though perhaps not immediately.

- One can use *two* question marks after a function name to pull up the documentation *and* the source code for the function.
- Again, to see this help, you can either evaluate a cell like below, or just move your cursor after the question mark and press tab.

The ability to see the code (the underlying instructions to the computer) is one of Sage's great strengths. You can see *all* the code to *everything* .

This means:

- *You* can see what Sage is doing.
- Your curious students can see what is going on.
- And if you find a better way to do something, then you can see how to change it!

```
sage: binomial??
```

```
>>> from sage.all import *
>>> binomial??
```

2.4 Annotating with Sage

Whether one uses Sage in the classroom or in research, it is usually helpful to describe to the reader what is being done, such as in the description you are now reading.

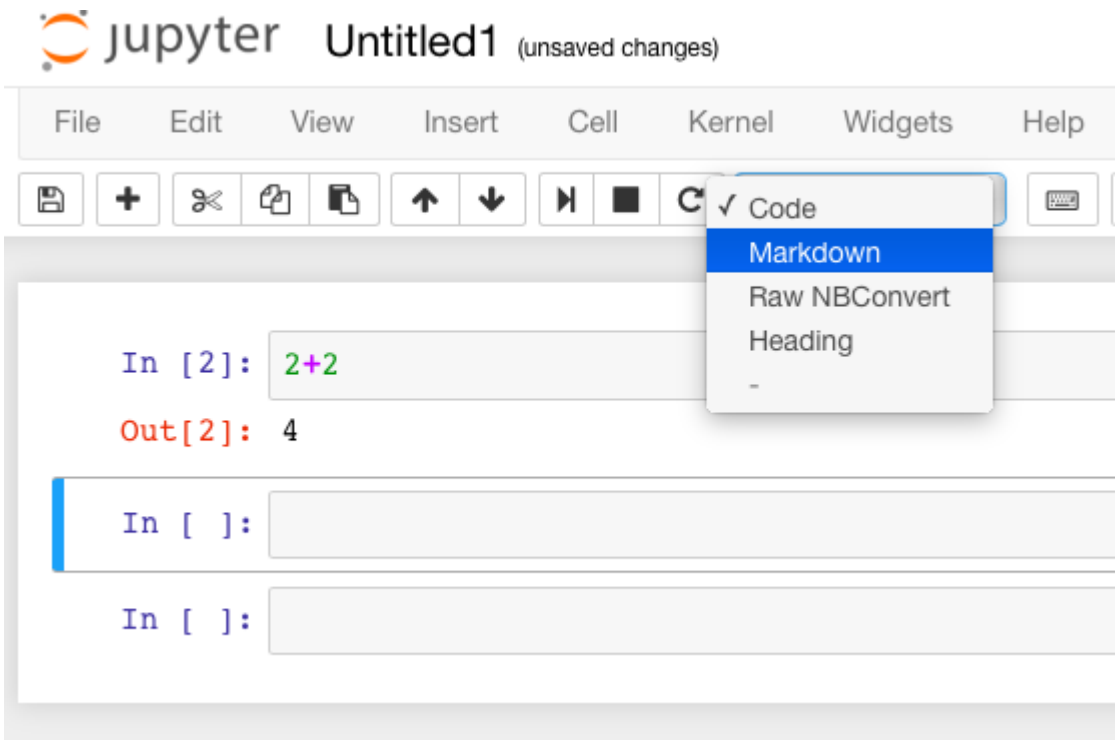
- *Jupyter Annotation*

2.4.1 Jupyter Annotation

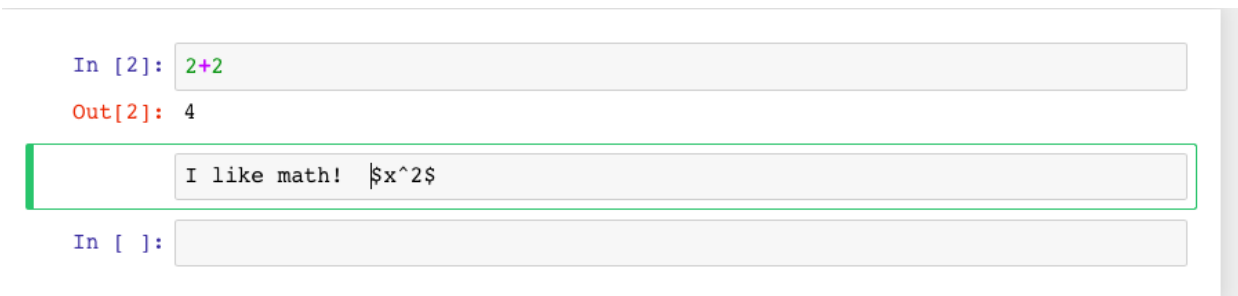
Thanks to a styling language called Markdown and the TeX rendering engine called MathJax, you can type much more in Sage than just Sage commands. This math-aware setup makes Sage perfect for annotating computations.

Jupyter notebook can function as a word processor. To use this functionality, we create a *Markdown cell* (as opposed to a *input cell* that contains Sage commands that Sage evaluates).

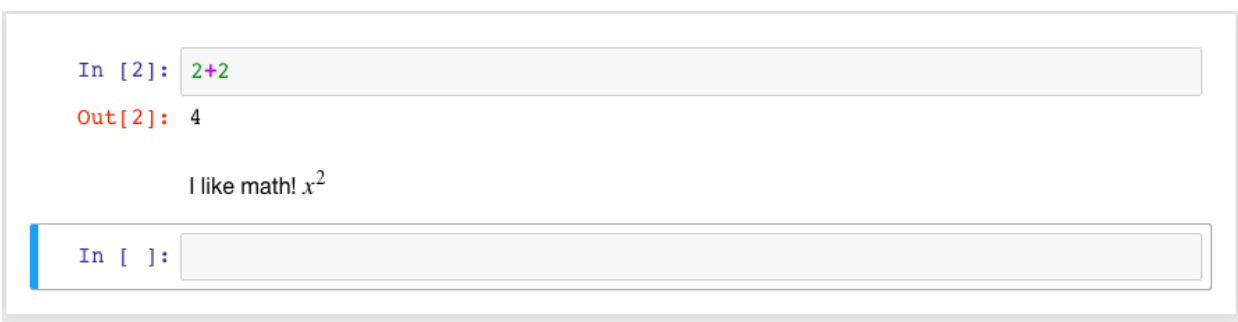
To do this without the keyboard shortcut, there is a menu for each cell; select "Markdown".



Now you can type in whatever you want, including mathematics using LaTeX.



Then evaluate the cell (for instance, with "Shift-Enter"):



Markdown supports a fair amount of basic formatting, such as bold, underline, basic lists, and so forth.

It can be fun to type in fairly complicated math, like this:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_p \left(\frac{1}{1 - p^{-s}} \right) .$$

One just types things like:

```
$$\zeta(s)=\sum_{n=1}^{\infty}\frac{1}{n^s}=\prod_p \left(\frac{1}{1-p^{-s}}\right)$$
```

in a Markdown cell.

```
In [2]: 2+2
Out[2]: 4
```

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_p \left(\frac{1}{1-p^{-s}}\right)$$

```
In [ ]:
```

Of course, one can do much more, since Sage can execute arbitrary commands in the Python programming language, as well as output nicely formatted HTML, and so on. If you have enough programming experience to do things like this, go for it!

```
sage: html("Sage is <a style='text-decoration:line-through'>somewhat</a> <b>really</b>
↪ cool! <p style='color:red'>(It even does HTML.)</p>")
```

```
>>> from sage.all import *
>>> html("Sage is <a style='text-decoration:line-through'>somewhat</a> <b>really</b>␣
↪cool! <p style='color:red'>(It even does HTML.)</p>")
```

2.5 Conclusion

This concludes the introductory tutorial. Our hope is that now you can try finding and using simple commands and functions in Sage. Remember, help is as close as the notebook, or at the Sage website.

CHAPTER

THREE

TUTORIAL FOR SYMBOLICS AND PLOTTING

This Sage document is one of the tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

This tutorial has the following sections:

- *Symbolic Expressions*
- *Basic 2D Plotting*
- *Basic 3D Plotting*

It assumes that one is familiar with the absolute basics of functions and evaluation in Sage. We provide a (very) brief refresher.

1. Make sure the syntax below for defining a function and getting a value makes sense.
2. Then evaluate the cell by clicking the "evaluate" link, or by pressing `Shift` + `Enter` (hold down `Shift` while pressing the `Enter` key).

```
sage: f(x)=x^3+1
sage: f(2)
9
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(3)+Integer(1)).function(x)
>>> f(Integer(2))
9
```

3.1 Symbolic Expressions

In the first tutorial, we defined *functions* using notation similar to that one would use in (say) a calculus course.

There is a useful variant on this - defining *expressions* involving variables. This will give us the opportunity to point out several important, and sometimes subtle, things.

In the cell below, we define an expression FV which is the future value of an investment of \$100, compounded continuously. We then substitute in values for r and t which calculate the future value for $t = 5$ years and $r = 5\%$ nominal interest.

```
sage: var('r,t')
(r, t)
sage: FV=100*e^(r*t)
```

```
>>> from sage.all import *
>>> var('r,t')
(r, t)
>>> FV=Integer(100)*e**(r*t)
```

```
sage: FV(r=.05,t=5)
128.402541668774
```

```
>>> from sage.all import *
>>> FV(r=RealNumber('.05'),t=Integer(5))
128.402541668774
```

The previous cells point out several things to remember when working with symbolic expressions. Some are fairly standard.

- An asterisk (`*`) signifies multiplication. This should be how you always do multiplication.
- Although it is possible to allow implicit multiplication, this can easily lead to ambiguity.
- We can access the most important constants; for instance, e stands for the constant $2.71828\ldots$. Likewise, `pi` (or π) and I (think complex numbers) are also defined.
- Of course, if you redefine e to be something else, all bets are off!

However, two others may be unfamiliar, especially if you have not used much mathematical software before.

- You must tell Sage what the variables are before using them in a symbolic expression.
- We did that above by typing `var('r,t')`.
- This is automatically done with the $f(x)$ notation, but without that it is necessary, so that Sage knows one intends `t` (for instance) is a symbolic variable and not a number or something else.
- If you then wish to substitute some values into the expression, you must explicitly tell Sage which variables are being assigned which values.
- For instance, above we used `FV(r=.05,t=5)` to indicate the precise values of r and t.

Notice that when we define a function, we don't need to specify which variable has which value. In the function defined below, we have already specified an order.

```
sage: FV2(r,t)=100*e^(r*t)
sage: FV2(.05,5)
128.402541668774
```

```
>>> from sage.all import *
>>> __tmp__=var("r,t"); FV2 = symbolic_expression(Integer(100)*e**(r*t)).function(r,t)
>>> FV2(RealNumber('.05'),Integer(5))
128.402541668774
```

In this case it is clear that r is first and t is second.

But with `FV=100*e^(r*t)`, there is no particular reason r or t should be first.

```
sage: FV(r=.05,t=5); FV(t=5,r=.05)
128.402541668774
128.402541668774
```

```
>>> from sage.all import *
>>> FV(r=RealNumber('.05'),t=Integer(5)); FV(t=Integer(5),r=RealNumber('.05'))
```

(continues on next page)

(continued from previous page)

```
128.402541668774
128.402541668774
```

Also remember that when we don't use function notation, we'll need to define our variables.

One of the great things we can do with expressions is manipulate them. Let's make a typical expression.

```
sage: z = (x+1)^3
```

```
>>> from sage.all import *
>>> z = (x+Integer(1))**Integer(3)
```

In the cells below, you'll notice something new: the character `#`. In Sage (and in Python), anything on a single line after the number/pound sign (the octothorp) is ignored. We say that `#` is a comment character. We use it below to mention alternative ways to do the same thing.

```
sage: expand(z) # or z.expand()
x^3 + 3*x^2 + 3*x + 1
```

```
>>> from sage.all import *
>>> expand(z) # or z.expand()
x^3 + 3*x^2 + 3*x + 1
```

```
sage: y = expand(z)
sage: y.factor() # or factor(y)
(x + 1)^3
```

```
>>> from sage.all import *
>>> y = expand(z)
>>> y.factor() # or factor(y)
(x + 1)^3
```

In the previous cell, we *assigned* the expression which is the expansion of z to the variable y with the first line. After that, anything we want to do to the expansion of z can be done by doing it to y.

There are more commands like this as well. Notice that z will no longer be $(x+1)^3$ after this cell is evaluated, since we've assigned z to a (much more complex) expression.

```
sage: z = ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1))
sage: z.simplify_full()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

```
>>> from sage.all import *
>>> z = ((x - Integer(1))**(Integer(3)/Integer(2)) - (x + Integer(1))*sqrt(x -␣
↪Integer(1)))/sqrt((x - Integer(1))*(x + Integer(1)))
>>> z.simplify_full()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

This is a good place for a few reminders of basic help.

- You can see various methods for simplifying an expression by using tab completion. Put your cursor at the end of the next cell (after the `simplify`) and press tab to see lots of different methods.
- Also remember that you can use the question mark (e.g., `z.simplify_rational?`) to get help about a particular method.

```
sage: z.simplify
<built-in method simplify of sage.symbolic.expression.Expression object at ...>
```

```
>>> from sage.all import *
>>> z.simplify
<built-in method simplify of sage.symbolic.expression.Expression object at ...>
```

Finally, recall that you can get nicely typeset versions of the output in several ways.

- One option is to click the 'Typeset' button at the top.
- Another - which does not require scrolling! - is to use the `show` command.

```
sage: show(z.simplify_rational())
```

```
>>> from sage.all import *
>>> show(z.simplify_rational())
```

$$-\frac{2\sqrt{x-1}}{\sqrt{x^2-1}}$$

Another Sage command that is useful in this context is `solve`.

Here, we solve the simple equation $x^2 = -1$.

```
sage: solve(x^2==-1,x) # solve x^2==-1 for x
[x == -I, x == I]
```

```
>>> from sage.all import *
>>> solve(x**Integer(2)==-Integer(1),x) # solve x^2==-1 for x
[x == -I, x == I]
```

- In the `solve` command, one types an equals sign in the equation as *two* equal signs.
- This is because the single equals sign means assignment to a variable, as we've done above, so Sage (along with Python) uses the double equals sign for symbolic equality.
- We also include the variable we'd like to solve for after the comma.

It's also possible to solve more than one expression simultaneously.

```
sage: solve([x^2==1,x^3==1],x)
[[x == 1]]
```

```
>>> from sage.all import *
>>> solve([x**Integer(2)==Integer(1),x**Integer(3)==Integer(1)],x)
[[x == 1]]
```

3.2 Basic 2D Plotting

One of the other basic uses of mathematics software is easy plotting. Here, we include a brief introduction to the sorts of plotting which will prepare us to use Sage in calculus. (There will be a separate tutorial for more advanced plotting techniques.)

Recall that we can generate a plot using fairly simple syntax. Here, we define a function of x and plot it between -1 and 1.

```
sage: f(x)=x^3+1
sage: plot(f,(x,-1,1))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(3)+Integer(1)).function(x)
>>> plot(f,(x,-Integer(1),Integer(1)))
Graphics object consisting of 1 graphics primitive
```

We can give the plot a name, so that if we want to do something with the plot later, we don't have to type out the entire plot command. Remember, this is called *assigning* the plot to the name/variable.

In the next cell, we give the plot the name P.

```
sage: P=plot(f,(x,-1,1))
```

```
>>> from sage.all import *
>>> P=plot(f,(x,-Integer(1),Integer(1)))
```

One plot is nice, but one might want to superimpose plots as well. For instance, the tangent line to f at $x = 0$ is just the line $y = 1$, and we might want to show this together with the plot.

So let's plot this line in a different color, and with a different style for the line, but over the same interval.

```
sage: Q=plot(1,(x,-1,1),color="red", linestyle="--")
sage: Q
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> Q=plot(Integer(1),(x,-Integer(1),Integer(1)),color="red", linestyle="--")
>>> Q
Graphics object consisting of 1 graphics primitive
```

Because we put Q in a line by itself at the end, it shows. We were able to use just one cell to define Q and show it, by putting each command in a separate line in the same input cell.

Now to show the plots superimposed on each other, we simply add them.

```
sage: P+Q
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> P+Q
Graphics object consisting of 2 graphics primitives
```

Suppose we wanted to view a detail of this.

- We could create another plot with different endpoints.

- Another way is to keep the currently created plots, but to set the viewing window using the `show` command, as below.

```
sage: (P+Q).show(xmin=-.1,xmax=.1,ymin=.99,ymax=1.01)
```

```
>>> from sage.all import *
>>> (P+Q).show(xmin=-RealNumber('.1'),xmax=RealNumber('.1'),ymin=RealNumber('.99'),
↪ymax=RealNumber('1.01'))
```

Since the axes no longer cross in the frame of reference, Sage shows a short gap between the horizontal and vertical axes.

There are many options one can pass in for various purposes.

- Some require quotes around the values.
- Such as the `color` option when we made a `"red"` line.
- Some do not.
- Such as the `xmin` in the previous plot, where the minimum x value was just $-.1$

Usually (though not always) quotes are required for option values which are words or strings of characters, and not required for numerical values.

Two of the most useful of these options help in labeling graphs.

- The `axes_labels` option labels the axes.
- As with the word processor, we can use dollar signs (like in LaTeX) to make the labels typeset nicely.
- Here we need both quotes and brackets for proper syntax, since there are two axes to label and the labels are not actually numbers.

```
sage: plot(f,(x,-1,1),axes_labels=['$x$','$y$'],legend_label='$f(x)$',show_
↪legend=True)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(f,(x,-Integer(1),Integer(1)),axes_labels=['$x$','$y$'],legend_label='$f(x)$',
↪show_legend=True)
Graphics object consisting of 1 graphics primitive
```

- The `legend_label` option is especially useful with multiple plots.
- LaTeX notation works here too.
- In the graphic above, we needed to explicitly ask to show the label. With multiple graphs this should not be necessary.

```
sage: P1 = plot(f,(x,-1,1),axes_labels=['$x$','$y$'],legend_label='$f(x)$')
sage: P2 = plot(sin,(x,-1,1),axes_labels=['$x$','$y$'],legend_label=r'$\sin(x)$',
↪color='red')
sage: P1+P2
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> P1 = plot(f,(x,-Integer(1),Integer(1)),axes_labels=['$x$','$y$'],legend_label='
↪$f(x)$')
>>> P2 = plot(sin,(x,-Integer(1),Integer(1)),axes_labels=['$x$','$y$'],legend_label=r'
↪$\sin(x)$',color='red')
```

(continues on next page)

(continued from previous page)

```
>>> P1+P2
Graphics object consisting of 2 graphics primitives
```

One additional useful note is that plots of functions with vertical asymptotes may need their vertical viewing range set manually; otherwise the asymptote may really go to infinity!

```
sage: plot(1/x^2,(x,-10,10),ymax=10)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(Integer(1)/x**Integer(2),(x,-Integer(10),Integer(10)),ymax=Integer(10))
Graphics object consisting of 1 graphics primitive
```

Remember, you can use the command `plot?` to find out about most of the options demonstrated above.

Below, you can experiment with several of the plotting options.

- Just evaluate the cell and play with the sliders, buttons, color picker, etc., to change the plot options.
- You can access low-level options like the initial number of plotted points, or high-level ones like whether axes are shown or not.
- This uses a feature of Sage called "interacts", which is a very powerful way to engage students in exploring a problem.

```
sage: x = var('x')
sage: @interact
sage: def plot_example(f=sin(x^2),r=range_slider(-5,5,step_size=1/4,default=(-3,3)),
....:                  color=color_selector(widget='colorpicker'),
....:                  thickness=(3,(1..10)),
....:                  adaptive_recursion=(5,(0..10)), adaptive_tolerance=(0.01,(0.
↪001,1)),
....:                  plot_points=(20,(1..100)),
....:                  linestyle=['-','--','-.',':'],
....:                  gridlines=False, fill=False,
....:                  frame=False, axes=True
....:                  ):
....:     show(plot(f, (x,r[0],r[1]), color=color, thickness=thickness,
....:                  adaptive_recursion=adaptive_recursion,
....:                  adaptive_tolerance=adaptive_tolerance, plot_points=plot_points,
....:                  linestyle=linestyle, fill=fill if fill else None),
....:                  gridlines=gridlines, frame=frame, axes=axes)
```

```
>>> from sage.all import *
>>> x = var('x')
>>> @interact
>>> def plot_example(f=sin(x**Integer(2)),r=range_slider(-Integer(5),Integer(5),step_
↪size=Integer(1)/Integer(4),default=(-Integer(3),Integer(3))),
...                  color=color_selector(widget='colorpicker'),
...                  thickness=(Integer(3),(ellipsis_iter(Integer(1),Ellipsis,
↪Integer(10)))),
...                  adaptive_recursion=(Integer(5),(ellipsis_iter(Integer(0),
↪Ellipsis,Integer(10)))), adaptive_tolerance=(RealNumber('0.01'),(RealNumber('0.001
↪'),Integer(1))),
...                  plot_points=(Integer(20),(ellipsis_iter(Integer(1),Ellipsis,
↪Integer(100)))),
...                  linestyle=['-','--','-.',':'],
```

(continues on next page)

(continued from previous page)

```
...                     gridlines=False, fill=False,
...                     frame=False, axes=True
...                     ):
...     show(plot(f, (x,r[Integer(0)],r[Integer(1)]), color=color,␣
↪thickness=thickness,
...                     adaptive_recursion=adaptive_recursion,
...                     adaptive_tolerance=adaptive_tolerance, plot_points=plot_points,
...                     linestyle=linestyle, fill=fill if fill else None),
...                     gridlines=gridlines, frame=frame, axes=axes)
```

3.3 Basic 3D Plotting

There are several mechanisms for viewing three-dimensional plots in Sage, but we will stick to the default option in the notebook interface, which is via javascript applets from the program Jmol/JSmol .

Plotting a 3D plot is similar to plotting a 2D plot, but we need to specify ranges for two variables instead of one.

```
sage: g(x,y)=sin(x^2+y^2)
sage: plot3d(g,(x,-5,5),(y,-5,5))
Graphics3d Object
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); g = symbolic_expression(sin(x**Integer(2)+y**Integer(2))).
↪function(x,y)
>>> plot3d(g,(x,-Integer(5),Integer(5)),(y,-Integer(5),Integer(5)))
Graphics3d Object
```

There is a lot you can do with the 3D plots.

- Try rotating the plot above by clicking and dragging the mouse inside of the plot.
- Also, right-click (Control-click if you have only one mouse button) just to the right of the plot to see other options in a menu.
- If you have a wheel on your mouse or a multi-touch trackpad, you can scroll to zoom.
- You can also right-click to see other options, such as
- spinning the plot,
- changing various colors,
- and even making the plot suitable for viewing through 3D glasses (under the "style", then "stereographic" submenus),

When using the `plot3d` command, the first variable range specified is plotted along the usual "x" axis, while the second range specified is plotted along the usual "y" axis.

The plot above is somewhat crude because the function is not sampled enough times - this is fairly rapidly changing function, after all. We can make the plot smoother by telling Sage to sample the function using a grid of 300 by 300 points. Sage then samples the function at 90,000 points!

```
sage: plot3d(g,(x,-5,5),(y,-5,5),plot_points=300)
Graphics3d Object
```

```
>>> from sage.all import *
>>> plot3d(g,(x,-Integer(5),Integer(5)),(y,-Integer(5),Integer(5)),plot_
```

(continues on next page)

(continued from previous page)

```
↪points=Integer(300))
Graphics3d Object
```

As with 2D plots, we can superimpose 3D plots by adding them together.

Note that in this one, we do not define the functions, but only use expressions (see the first set of topics in this tutorial), so it is wisest to define the variables ahead of time.

```
sage: var('x,y')
(x, y)
sage: b = 2.2
sage: P=plot3d(sin(x^2-y^2),(x,-b,b),(y,-b,b), opacity=.7)
sage: Q=plot3d(0, (x,-b,b), (y,-b,b), color='red')
sage: P+Q
Graphics3d Object
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> b = RealNumber('2.2')
>>> P=plot3d(sin(x**Integer(2)-y**Integer(2)),(x,-b,b),(y,-b,b), opacity=RealNumber('.
↪7'))
>>> Q=plot3d(Integer(0), (x,-b,b), (y,-b,b), color='red')
>>> P+Q
Graphics3d Object
```

As usual, only the last command shows up in the notebook, though clearly all are evaluated. This also demonstrates that many of the same options work for 3D plots as for 2D plots.

We close this tutorial with a cool plot that we define *implicitly* as a 3D contour plot.

```
sage: var('x,y,z')
(x, y, z)
sage: T = golden_ratio
sage: p = 2 - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z) + cos(y - T*z) + cos(z -␣
↪T*x) + cos(z + T*x))
sage: r = 4.78
sage: implicit_plot3d(p, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=50, color=
↪'yellow')
Graphics3d Object
```

```
>>> from sage.all import *
>>> var('x,y,z')
(x, y, z)
>>> T = golden_ratio
>>> p = Integer(2) - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z) + cos(y - T*z) +␣
↪cos(z - T*x) + cos(z + T*x))
>>> r = RealNumber('4.78')
>>> implicit_plot3d(p, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=Integer(50),␣
↪color='yellow')
Graphics3d Object
```

The next tutorial will use all that you have learned about Sage basics, symbolics, and plotting in a specific mathematical venue - the *calculus sequence*!

CHAPTER

FOUR

TUTORIAL FOR CALCULUS

This Sage document is one of the tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

This tutorial has the following sections. The first three are based on the topics encountered in a typical three-semester calculus sequence in the United States; the final section is a checkpoint of sorts.

- *Calculus 1*
- *Calculus 2*
- *Calculus 3*
- *'Exam'*

The tutorial assumes that one is familiar with the basics of Sage, such as outlined in the previous tutorials.

For a refresher, make sure the syntax below for defining a function and getting a value makes sense; then evaluate the cell by clicking the "evaluate" link, or by pressing `Shift` + `Enter` (hold down `Shift` while pressing the `Enter` key).

```
sage: f(x)=x^3+1
sage: f(2)
9
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(3)+Integer(1)).function(x)
>>> f(Integer(2))
9
```

We'll use this function several times in this tutorial, so it's important that it is evaluated!

4.1 Calculus 1

The calculus is amazing not so much because it solves problems of tangency or area - individual solutions to these problems have been known before. It is amazing because it gives a remarkably comprehensive set of rules for symbolic manipulation for solving such problems in great generality!

Thus, the most typical use of a computer system like Sage in this context is simply to help check (or make less tedious) basic symbolic manipulation of things like derivatives. Let's first show what our function f is again, then do things with it.

```
sage: show(f)
```

```
>>> from sage.all import *
>>> show(f)
```

$$x \mapsto x^3 + 1$$

Something most (but not all) curricula for first-semester calculus do is spend a fair amount of time with limits.

What is the limit of this function as x approaches 1?

```
sage: lim(f,x=1)
x |--> 2
```

```
>>> from sage.all import *
>>> lim(f,x=Integer(1))
x |--> 2
```

Sage gives the answer we expect. The syntax for limits is pretty straightforward, though it may differ slightly from that of other systems.

Since a limit may exist even if the function is not defined, Sage uses the syntax `x=1` to show the input value approached in all situations.

```
sage: lim((x^2-1)/(x-1),x=1)
2
```

```
>>> from sage.all import *
>>> lim((x**Integer(2)-Integer(1))/(x-Integer(1)),x=Integer(1))
2
```

Next, we'll return to the original function, and check the two directional limits.

The syntax uses the extra *keyword* `dir`.

- Notice that we use `dir='right'`, not `dir=right`.
- It's also okay to use `dir='+'`, like we use `dir='-'` below.
- This is basically because otherwise we might have done something like let `right=55` earlier, so instead Sage - and Python - requires us to put `'right'` and `'-'` in quotes to make it clear they are not variables.

```
sage: lim(f,x=1,dir='-'); lim(f,x=1,dir='right'); f(1)
x |--> 2
x |--> 2
2
```

```
>>> from sage.all import *
>>> lim(f,x=Integer(1),dir='-'); lim(f,x=Integer(1),dir='right'); f(Integer(1))
x |--> 2
x |--> 2
2
```

By comparing with $f(1)$, we see that $f(x)$ is continuous at $x = 1$.

This cell also reminds us of something else:

- By separating several commands with semicolons, we can tell Sage to evaluate all of them while still seeing all their outputs.

What we spend the most time on in Calculus 1 is derivatives, and Sage is fully-featured here.

For example, here are three ways to get the basic, single-variable derivative of $f(x) = x^3 + 1$.

```
sage: diff(f,x); derivative(f,x); f.derivative(x)
x |--> 3*x^2
x |--> 3*x^2
x |--> 3*x^2
```

```
>>> from sage.all import *
>>> diff(f,x); derivative(f,x); f.derivative(x)
x |--> 3*x^2
x |--> 3*x^2
x |--> 3*x^2
```

Naturally, Sage knows all of the derivatives you want.

```
sage: derivative(sinh(x^2+sqrt(x-1)),x)
1/2*(4*x + 1/sqrt(x - 1))*cosh(x^2 + sqrt(x - 1))
```

```
>>> from sage.all import *
>>> derivative(sinh(x**Integer(2)+sqrt(x-Integer(1))),x)
1/2*(4*x + 1/sqrt(x - 1))*cosh(x^2 + sqrt(x - 1))
```

And maybe even knows those you do not want. In this case, we put the computation inside `show()` since the output is so long.

```
sage: show(derivative(sinh(x^2+sqrt(x-1)),x,3))
```

```
>>> from sage.all import *
>>> show(derivative(sinh(x**Integer(2)+sqrt(x-Integer(1))),x,Integer(3)))
```

$$\frac{1}{8}\left(4\,x+\frac{1}{\sqrt{x-1}}\right)^{3}\cosh\left(\sqrt{x-1}+x^{2}\right)-\frac{3}{8}\left(\frac{1}{(x-1)^{\frac{3}{2}}}-8\right)\left(4\,x+\frac{1}{\sqrt{x-1}}\right)\sinh\left(\sqrt{x-1}+x^{2}\right)+\frac{3\,\cosh\left(\sqrt{x-1}+x^{2}\right)}{8\,(x-1)^{\left(\frac{5}{2}\right)}}$$

A common question is why Sage might not check automatically if there is some "simpler" version. But simplifying an expression, or indeed, even defining what a 'simple' expression is, turns out to be a very hard technical and mathematical problem in general. Computers will not solve every problem!

As a brief interlude, let's consider an application of our ability to do some basic differential calculus.

First, as a reminder of plotting from the previous tutorial, try to plot the function $f(x)$, together with its tangent line at $x = 1$, in the empty cells below. (If you can, do it without looking at the previous tutorial for a reminder.)

Did you get it?

Of course, in general we might want to see tangent lines at lots of different points - for instance, for a class demonstration.

So in the following cell, there are several auxiliary elements:

- We define the plot `P` of the original function.
- There is a parameter `c=1/3`, which is the x-value where we want a tangent line.
- We let `fprime` be the derivative function simply by declaring it equal to the derivative.
- We let `L` be the tangent line defined by the point-slope formula at $x = c$.
- We make `Q` be the plot of this line.

Finally, we plot everything together in the last line by adding $P + Q$.

```
sage: P=plot(f,(x,-1,1))
sage: c=1/3
sage: fprime=derivative(f,x)
sage: L(x)=fprime(c)*(x-c)+f(c)
sage: Q=plot(L,(x,-1,1), color="red", linestyle="--")
sage: P+Q
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> P=plot(f,(x,-Integer(1),Integer(1)))
>>> c=Integer(1)/Integer(3)
>>> fprime=derivative(f,x)
>>> __tmp__=var("x"); L = symbolic_expression(fprime(c)*(x-c)+f(c)).function(x)
>>> Q=plot(L,(x,-Integer(1),Integer(1)), color="red", linestyle="--")
>>> P+Q
Graphics object consisting of 2 graphics primitives
```

You may want to experiment by

- Changing `c` to some other value, or
- Changing the function `f`, or
- Changing the colors, or
- Changing something else (like the `linestyle` used for the tangent line).

Ideally, it would be *extremely* easy to change that parameter c. In the cell below, we show our second example of a Sage "interact" (or "Sagelet").

In this one, dragging a slider will show the tangent line moving.

- Future tutorials will explain this process in more detail; here it's just as an example.
- However, the reader will note how very similar the code for the two cells is.

```
sage: %auto
sage: f(x)=x^3+1
sage: @interact
sage: def _(c=(1/3,(-1,1))):
....:     P=plot(f,(x,-1,1))
....:     fprime=derivative(f,x)
....:     L(x)=fprime(c)*(x-c)+f(c)
....:     Q=plot(L,(x,-1,1),color="red", linestyle="--")
....:     show(P+Q+point((c,f(c)), pointsize=40, color='red'),ymin=0,ymax=2)
```

```
>>> from sage.all import *
>>> %auto
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(3)+Integer(1)).function(x)
>>> @interact
>>> def _(c=(Integer(1)/Integer(3),(-Integer(1),Integer(1)))):
...     P=plot(f,(x,-Integer(1),Integer(1)))
...     fprime=derivative(f,x)
...     __tmp__=var("x"); L = symbolic_expression(fprime(c)*(x-c)+f(c)).function(x)
...     Q=plot(L,(x,-Integer(1),Integer(1)),color="red", linestyle="--")
...     show(P+Q+point((c,f(c)), pointsize=Integer(40), color='red'),ymin=Integer(0),
↪ymax=Integer(2))
```

A very sharp-eyed reader will also have noticed that the previous cell had `%auto` at the very top, and that it was not necessary to evaluate the cell to use it.

- The command `%auto` allows us to have a cell, especially an interactive one, all loaded up as soon as we start - particularly convenient for a classroom situation.
- Such instructions are called *percent directives* . Most are documented in the notebook help one can access at the top of any worksheet.

A final topic in Calculus 1 usually is basic integration. The syntax for indefinite integration is similar to that for differentiation.

```
sage: integral(cos(x),x)
sin(x)
```

```
>>> from sage.all import *
>>> integral(cos(x),x)
sin(x)
```

We do not get the whole indefinite integral, just a convenient antiderivative.

- (If you were to get a different answer 'by hand', remember that being an antiderivative means the answer is correct *up to a constant* - and deciding whether this is the case with two expressions has the same problems as the issue of "simplification" above.)

Definite integration has similar syntax to plotting.

```
sage: integral(cos(x),(x,0,pi/2))
1
```

```
>>> from sage.all import *
>>> integral(cos(x),(x,Integer(0),pi/Integer(2)))
1
```

4.2 Calculus 2

Second-semester calculus is typically more challenging.

- One reason for that is that the computational problems are not so straightforward as computing derivatives and basic integrals.
- Another reason is that the second semester is usually where the harder versions of problems from the first semester show up.

Nonetheless, Sage can handle this as well.

Sage includes a large number of indefinite integrals (via Maxima), though not all the ones you will find in a comprehensive table.

```
sage: h(x)=sec(x)
sage: h.integrate(x)
x |--> log(sec(x) + tan(x))
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); h = symbolic_expression(sec(x)).function(x)
>>> h.integrate(x)
x |--> log(sec(x) + tan(x))
```

Since I defined `h` as a function, the answer I get is also a function. If I just want an expression as the answer, I can do the following.

```
sage: integrate(sec(x),x)
log(sec(x) + tan(x))
```

```
>>> from sage.all import *
>>> integrate(sec(x),x)
log(sec(x) + tan(x))
```

Here is another (longer) example. Do you remember what command would help it look nicer in the browser?

```
sage: integrate(1/(1+x^5),x)
1/5*sqrt(5)*(sqrt(5) + 1)*arctan((4*x + sqrt(5) - 1)/sqrt(2*sqrt(5) + 10))/
↪sqrt(2*sqrt(5) + 10)
+ 1/5*sqrt(5)*(sqrt(5) - 1)*arctan((4*x - sqrt(5) - 1)/sqrt(-2*sqrt(5) + 10))/sqrt(-
↪2*sqrt(5) + 10)
- 1/10*(sqrt(5) + 3)*log(2*x^2 - x*(sqrt(5) + 1) + 2)/(sqrt(5) + 1)
- 1/10*(sqrt(5) - 3)*log(2*x^2 + x*(sqrt(5) - 1) + 2)/(sqrt(5) - 1)
+ 1/5*log(x + 1)
```

```
>>> from sage.all import *
>>> integrate(Integer(1)/(Integer(1)+x**Integer(5)),x)
1/5*sqrt(5)*(sqrt(5) + 1)*arctan((4*x + sqrt(5) - 1)/sqrt(2*sqrt(5) + 10))/
↪sqrt(2*sqrt(5) + 10)
+ 1/5*sqrt(5)*(sqrt(5) - 1)*arctan((4*x - sqrt(5) - 1)/sqrt(-2*sqrt(5) + 10))/sqrt(-
↪2*sqrt(5) + 10)
- 1/10*(sqrt(5) + 3)*log(2*x^2 - x*(sqrt(5) + 1) + 2)/(sqrt(5) + 1)
- 1/10*(sqrt(5) - 3)*log(2*x^2 + x*(sqrt(5) - 1) + 2)/(sqrt(5) - 1)
+ 1/5*log(x + 1)
```

Some integrals are a little tricky, of course. Sage tries hard to integrate using Maxima, Giac and Sympy:

```
sage: integral(1/(1+x^10),x)
...1/20*(sqrt(5) + 1)*arctan((4*x + sqrt(-2*sqrt(5) + 10))/(sqrt(5) + 1))
+ 1/20*(sqrt(5) + 1)*arctan((4*x - sqrt(-2*sqrt(5) + 10))/(sqrt(5) + 1))
+ 1/20*(sqrt(5) - 1)*arctan((4*x + sqrt(2*sqrt(5) + 10))/(sqrt(5) - 1))
+ 1/20*(sqrt(5) - 1)*arctan((4*x - sqrt(2*sqrt(5) + 10))/(sqrt(5) - 1))
+ 1/40*sqrt(2*sqrt(5) + 10)*log(x^2 + 1/2*x*sqrt(2*sqrt(5) + 10) + 1)
- 1/40*sqrt(2*sqrt(5) + 10)*log(x^2 - 1/2*x*sqrt(2*sqrt(5) + 10) + 1)
+ 1/40*sqrt(-2*sqrt(5) + 10)*log(x^2 + 1/2*x*sqrt(-2*sqrt(5) + 10) + 1)
- 1/40*sqrt(-2*sqrt(5) + 10)*log(x^2 - 1/2*x*sqrt(-2*sqrt(5) + 10) + 1)
+ 1/5*arctan(x)
```

```
>>> from sage.all import *
>>> integral(Integer(1)/(Integer(1)+x**Integer(10)),x)
...1/20*(sqrt(5) + 1)*arctan((4*x + sqrt(-2*sqrt(5) + 10))/(sqrt(5) + 1))
+ 1/20*(sqrt(5) + 1)*arctan((4*x - sqrt(-2*sqrt(5) + 10))/(sqrt(5) + 1))
+ 1/20*(sqrt(5) - 1)*arctan((4*x + sqrt(2*sqrt(5) + 10))/(sqrt(5) - 1))
+ 1/20*(sqrt(5) - 1)*arctan((4*x - sqrt(2*sqrt(5) + 10))/(sqrt(5) - 1))
+ 1/40*sqrt(2*sqrt(5) + 10)*log(x^2 + 1/2*x*sqrt(2*sqrt(5) + 10) + 1)
- 1/40*sqrt(2*sqrt(5) + 10)*log(x^2 - 1/2*x*sqrt(2*sqrt(5) + 10) + 1)
+ 1/40*sqrt(-2*sqrt(5) + 10)*log(x^2 + 1/2*x*sqrt(-2*sqrt(5) + 10) + 1)
- 1/40*sqrt(-2*sqrt(5) + 10)*log(x^2 - 1/2*x*sqrt(-2*sqrt(5) + 10) + 1)
+ 1/5*arctan(x)
```

If you ask for Maxima specifically, the result can be partial:

```
sage: integral(1/(1+x^10),x, algorithm='maxima')
1/5*arctan(x)
- 1/5*integrate((x^6 - 2*x^4 + 3*x^2 - 4)/(x^8 - x^6 + x^4 - x^2 + 1), x)
```

```
>>> from sage.all import *
>>> integral(Integer(1)/(Integer(1)+x**Integer(10)),x, algorithm='maxima')
1/5*arctan(x)
- 1/5*integrate((x^6 - 2*x^4 + 3*x^2 - 4)/(x^8 - x^6 + x^4 - x^2 + 1), x)
```

If no antiderivative is found, the result is just the input:

```
sage: result = integral(sinh(x^2+sqrt(x-1)),x)  # long time (15s on sage.math, 2012)
...
sage: result  # long time
integrate(sinh(x^2 + sqrt(x - 1)), x)
```

```
>>> from sage.all import *
>>> result = integral(sinh(x**Integer(2)+sqrt(x-Integer(1))),x)  # long time (15s on␣
↪sage.math, 2012)
...
>>> result  # long time
integrate(sinh(x^2 + sqrt(x - 1)), x)
```

This last one stumps other systems too.

However, if there is a special function which helps compute the integral, Sage will look for it. In the following case there is no elementary antiderivative, but the `erf` function helps us out.

```
sage: integral(e^(-x^2),x)
1/2*sqrt(pi)*erf(x)
```

```
>>> from sage.all import *
>>> integral(e**(-x**Integer(2)),x)
1/2*sqrt(pi)*erf(x)
```

Do not forget, if this function is unfamiliar to you (as it might be to students trying this integral), Sage's contextual help system comes to the rescue.

```
sage: erf?
```

```
>>> from sage.all import *
>>> erf?
```

There are several ways to do definite integrals in Sage.

The most obvious one is simply turning

$$\int f(x)dx$$

into

$$\int_a^b f(x)dx \ ,$$

as indicated in the Calculus I section.

```
sage: integral(cos(x),(x,0,pi/2))
1
```

```
>>> from sage.all import *
>>> integral(cos(x),(x,Integer(0),pi/Integer(2)))
1
```

The preferred syntax puts the variable and endpoints together in parentheses.

Just like with derivatives, we can visualize this integral using some of the plotting options from the plotting tutorial.

```
sage: plot(cos(x),(x,0,pi/2),fill=True,ticks=[[0,pi/4,pi/2],None],tick_formatter=pi)
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> plot(cos(x),(x,Integer(0),pi/Integer(2)),fill=True,ticks=[[Integer(0),pi/
↪Integer(4),pi/Integer(2)],None],tick_formatter=pi)
Graphics object consisting of 2 graphics primitives
```

It is possible to be completely symbolic in doing integration. If you do this, you'll have to make sure you define anything that's a symbolic variable - which includes constants, naturally.

```
sage: var('a,b')
(a, b)
sage: integral(cos(x),(x,a,b))
-sin(a) + sin(b)
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> integral(cos(x),(x,a,b))
-sin(a) + sin(b)
```

On the numerical side, sometimes the answer one gets from the Fundamental Theorem of Calculus is not entirely helpful. Recall that h is the secant function.

```
sage: integral(h,(x,0,pi/7))
1/2*log(sin(1/7*pi) + 1) - 1/2*log(-sin(1/7*pi) + 1)
```

```
>>> from sage.all import *
>>> integral(h,(x,Integer(0),pi/Integer(7)))
1/2*log(sin(1/7*pi) + 1) - 1/2*log(-sin(1/7*pi) + 1)
```

Here, just a number might be more helpful. Sage has several ways of numerical evaluating integrals.

- Doing a definite integral symbolically, then approximating it numerically
- The `numerical_integral` function
- The `.nintegrate` method

The first one, using the n or N function for numerical approximation, was also mentioned in the introductory tutorial.

```
sage: N(integral(h,(x,0,pi/8)))
0.403199719161511
```

```
>>> from sage.all import *
>>> N(integral(h,(x,Integer(0),pi/Integer(8))))
0.403199719161511
```

The second function, `numerical_integral`, uses a powerful numerical program (the GNU Scientific Library).

- Unfortunately, the syntax for this function is not yet consistent with the rest of Sage.
- Helpfully, the output has two elements - the answer you desire, and its error tolerance.

```
sage: numerical_integral(h,0,pi/8)
(0.4031997191615114, 4.476416117355069e-15)
```

```
>>> from sage.all import *
>>> numerical_integral(h,Integer(0),pi/Integer(8))
(0.4031997191615114, 4.476416117355069e-15)
```

To access just the number, one asks for the 'zeroth' element of this sequence of items. This is done with the following bracket notation.

```
sage: numerical_integral(h,0,pi/8)[0]
0.4031997191615114
```

```
>>> from sage.all import *
>>> numerical_integral(h,Integer(0),pi/Integer(8))[Integer(0)]
0.4031997191615114
```

Notice that we began counting at zero. This is fairly typical in computer programs (though certainly not universal).

To aid readability (more important than one might think), we often assign the numerical integral to a variable, and then take the zeroth element of that.

```
sage: ni = numerical_integral(h,0,pi/8)
sage: ni[0]
0.4031997191615114
```

```
>>> from sage.all import *
>>> ni = numerical_integral(h,Integer(0),pi/Integer(8))
>>> ni[Integer(0)]
0.4031997191615114
```

Finally, the `.nintegrate()` method from Maxima gives even more extra information.

- Notice again the period/dot needed to use this.
- It is only possible to use `h(x)`; doing `h.nintegrate()` raises an error.

```
sage: h(x).nintegrate(x,0,pi/8)
(0.4031997191615114, 4.47641611735507e-15, 21, 0)
```

```
>>> from sage.all import *
>>> h(x).nintegrate(x,Integer(0),pi/Integer(8))
(0.4031997191615114, 4.47641611735507e-15, 21, 0)
```

Second-semester calculus usually also covers various topics in summation. Sage can sum many abstract series; the notation is similar to plotting and integration.

```
sage: var('n') # Do not forget to declare your variables
n
sage: sum((1/3)^n,n,0,oo)
3/2
```

```
>>> from sage.all import *
>>> var('n') # Do not forget to declare your variables
n
>>> sum((Integer(1)/Integer(3))**n,n,Integer(0),oo)
3/2
```

This is the geometric series, of course.

The next one is the famous result that a row of Pascal's triangle is a power of 2 -

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n-1} + \binom{n}{n} = 2^n \ ,$$

which has many pleasing combinatorial interpretations.

```
sage: k = var('k') # We already declared n, so now we just need k
sage: sum(binomial(n,k), k, 0, n)
2^n
```

```
>>> from sage.all import *
>>> k = var('k') # We already declared n, so now we just need k
>>> sum(binomial(n,k), k, Integer(0), n)
2^n
```

Do you remember what to do to see how we typed the nice sum in the text above? That's right, we can double-click the text area/cell to see this.

Sage also can compute Taylor polynomials.

Taylor expansions depend on a lot of things. Whenever there are several inputs, keeping syntax straight is important. Here we have as inputs:

- the function,
- the variable,
- the point around which we are expanding the function, and
- the degree.

In the next cell, we call $g(x)$ the Taylor polynomial in question.

```
sage: g(x)=taylor(log(x),x,1,6); g(x)
-1/6*(x - 1)^6 + 1/5*(x - 1)^5 - 1/4*(x - 1)^4 + 1/3*(x - 1)^3 - 1/2*(x - 1)^2 + x - 1
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); g = symbolic_expression(taylor(log(x),x,Integer(1),Integer(6))).
↪function(x); g(x)
-1/6*(x - 1)^6 + 1/5*(x - 1)^5 - 1/4*(x - 1)^4 + 1/3*(x - 1)^3 - 1/2*(x - 1)^2 + x - 1
```

Notice how close the approximation is to the function on this interval!

```
sage: plot(g,(x,0,2))+plot(log(x),(x,0,2),color='red')
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> plot(g,(x,Integer(0),Integer(2)))+plot(log(x),(x,Integer(0),Integer(2)),color='red
↪')
Graphics object consisting of 2 graphics primitives
```

4.3 Calculus 3

We have already seen three-dimensional plotting, so it is not surprising that Sage has support for a variety of multivariable calculus problems.

Warning: We will often need to define all variables other than x.

```
sage: var('y')
y
sage: f(x,y)=3*sin(x)-2*cos(y)-x*y
```

```
>>> from sage.all import *
>>> var('y')
y
>>> __tmp__=var("x,y"); f = symbolic_expression(Integer(3)*sin(x)-Integer(2)*cos(y)-
↪x*y).function(x,y)
```

Above, we have defined a typical function of two variables.

Below, we use the separating semicolons to demonstrate several things one might do with such a function, including:

- The gradient vector of all $\frac{\partial f}{\partial x_i}$
- The Hessian of all possible second derivatives
- A double partial derivative of f with respect to x, then y (that is, $\frac{\partial f}{\partial y \partial x}$)

```
sage: f.gradient(); f.hessian(); f.diff(x,y)
(x, y) |--> (-y + 3*cos(x), -x + 2*sin(y))
[(x, y) |--> -3*sin(x)         (x, y) |--> -1]
[        (x, y) |--> -1   (x, y) |--> 2*cos(y)]
(x, y) |--> -1
```

```
>>> from sage.all import *
>>> f.gradient(); f.hessian(); f.diff(x,y)
(x, y) |--> (-y + 3*cos(x), -x + 2*sin(y))
[(x, y) |--> -3*sin(x)         (x, y) |--> -1]
[        (x, y) |--> -1   (x, y) |--> 2*cos(y)]
(x, y) |--> -1
```

In an effort to make the syntax simpler, the gradient and Hessian are also available by asking for a total derivative. We also ask for nicer output again.

```
sage: show(f.diff()); show(f.diff(2))
```

```
>>> from sage.all import *
>>> show(f.diff()); show(f.diff(Integer(2)))
```

$$(x, y) \mapsto (-y + 3\cos(x), -x + 2\sin(y))$$

$$\begin{pmatrix} (x, y) \mapsto -3\sin(x) & (x, y) \mapsto -1 \\ (x, y) \mapsto -1 & (x, y) \mapsto 2\cos(y) \end{pmatrix}$$

If we take the determinant of the Hessian, we get something useful for evaluating (the two-dimensional) critical points of f.

```
sage: show(f.diff(2).det())
```

```
>>> from sage.all import *
>>> show(f.diff(Integer(2)).det())
```

$$(x, y) \mapsto -6\sin(x)\cos(y) - 1$$

These ideas are particularly helpful if one wants to plot a vector field.

The following example is of the gradient. The vector plotted in the cell below is the unit vector in the direction $(1, 2)$.

```
sage: P=plot_vector_field(f.diff(), (x,-3,3), (y,-3,3))
sage: u=vector([1,2])
sage: Q=plot(u/u.norm())
sage: P+Q
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> P=plot_vector_field(f.diff(), (x,-Integer(3),Integer(3)), (y,-Integer(3),
↪Integer(3)))
>>> u=vector([Integer(1),Integer(2)])
>>> Q=plot(u/u.norm())
>>> P+Q
Graphics object consisting of 2 graphics primitives
```

Rather than actually figure out the unit vector in that direction, it's easier to let Sage compute it by dividing the vector by its norm.

The directional derivative itself (in that direction, at the origin) can also be computed in this way.

```
sage: (f.diff()*u/u.norm())(0,0)
3/5*sqrt(5)
```

```
>>> from sage.all import *
>>> (f.diff()*u/u.norm())(Integer(0),Integer(0))
3/5*sqrt(5)
```

Another useful type of plot in these situations is a contour plot.

Notice that the one below uses several options. Try to correlate the options with features of the graphic.

```
sage: y = var('y')
sage: contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi),\
....:    contours=[-8,-4,0,4,8], colorbar=True, labels=True, label_colors='red')
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> y = var('y')
>>> contour_plot(y**Integer(2) + Integer(1) - x**Integer(3) - x, (x,-pi,pi), (y,-pi,
↪pi),   contours=[-Integer(8),-Integer(4),Integer(0),Integer(4),Integer(8)],␣
↪colorbar=True, labels=True, label_colors='red')
Graphics object consisting of 1 graphics primitive
```

In this one, we have used options to:

- Explicitly list the contours we want to show,
- Label these contours,
- Place a color bar on the side to show the different levels.

(Incidentally, the `True` and `False` valued options are some of the few non-numerical ones that do *not* need quotes.)

This is another good time to remind us we must explicitly ask for y to be a variable here, as will be the case a few more times.

As you gain experience in Sage, we will slowly explain less and less of the syntax of commands in these tutorials. You can think of places where not everything is explained as a mini-quiz.

For example, the next example shows how one currently does a multiple integral. What have we done here?

```
sage: integrate(integrate(f,(x,0,pi)),(y,0,pi))
6*pi - 1/4*pi^4
```

```
>>> from sage.all import *
>>> integrate(integrate(f,(x,Integer(0),pi)),(y,Integer(0),pi))
6*pi - 1/4*pi^4
```

Answer: notice that `integrate(f,(x,0,pi))` has been itself placed as the function inside `integrate(...,(y,0,pi))`.

We could use a 3D plot to help visualize this; these were already mentioned in the symbolics and plotting tutorial.

```
sage: plot3d(f,(x,0,pi),(y,0,pi),color='red')+plot3d(0,(x,0,pi),(y,0,pi))
Graphics3d Object
```

```
>>> from sage.all import *
>>> plot3d(f,(x,Integer(0),pi),(y,Integer(0),pi),color='red')+plot3d(Integer(0),(x,
↪Integer(0),pi),(y,Integer(0),pi))
Graphics3d Object
```

In addition to multivariate calculus, Calculus 3 often covers parametric calculus of a single variable. Sage can do arbitrary parametric plots, with fairly natural syntax.

This plot shows the tangent line to the most basic Lissajous curve at $t = 1$. The commands should be strongly reminiscent of the ones at the beginning of this tutorial.

```
sage: t = var('t')
sage: my_curve(t)=(sin(t), sin(2*t))
sage: PP=parametric_plot( my_curve, (t, 0, 2*pi), color="purple" )
sage: my_prime=my_curve.diff(t)
sage: L=my_prime(1)*t+my_curve(1) # tangent line at t=1
sage: parametric_plot(L, (t,-2,2))+PP
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> t = var('t')
>>> __tmp__=var("t"); my_curve = symbolic_expression((sin(t), sin(Integer(2)*t))).
↪function(t)
>>> PP=parametric_plot( my_curve, (t, Integer(0), Integer(2)*pi), color="purple" )
>>> my_prime=my_curve.diff(t)
>>> L=my_prime(Integer(1))*t+my_curve(Integer(1)) # tangent line at t=1
>>> parametric_plot(L, (t,-Integer(2),Integer(2)))+PP
Graphics object consisting of 2 graphics primitives
```

Tip:

- After a while, you'll find that giving things names other than `f` and `g` becomes quite helpful in distinguishing things from each other. Use descriptive names! We have tried to do so here.
- If you are adventurous, try turning this into an interactive cell along the lines of the single variable example earlier!

4.4 'Exam'

Before moving out of the calculus world, it is good to have a sort of miniature exam.

In the cell below, we have plotted and given:

- A slope field for a differential equation,
- A solution to an initial value problem,
- And a symbolic formula for that solution.

We assume you have never seen several of the commands before. Can you nonetheless figure out which commands are doing each piece, and what their syntax is? How would you look for help to find out more?

```
sage: y = var('y')
sage: Plot1=plot_slope_field(2-y,(x,0,3),(y,0,20))
sage: y = function('y',x) # declare y to be a function of x
sage: h = desolve(diff(y,x) + y - 2, y, ics=[0,7])
sage: Plot2=plot(h,0,3)
sage: show(expand(h)); show(Plot1+Plot2)
```

```
>>> from sage.all import *
>>> y = var('y')
>>> Plot1=plot_slope_field(Integer(2)-y,(x,Integer(0),Integer(3)),(y,Integer(0),
↪Integer(20)))
>>> y = function('y',x) # declare y to be a function of x
>>> h = desolve(diff(y,x) + y - Integer(2), y, ics=[Integer(0),Integer(7)])
>>> Plot2=plot(h,Integer(0),Integer(3))
>>> show(expand(h)); show(Plot1+Plot2)
```

$$5\,e^{(-x)} + 2$$

Ready to see the answers? Do not peek until you have really tried it.

In this cell we do the following:

- Make sure that `y` is indeed a variable for the first plot.
- Create a slope field for the DE for appropriate inputs of x and y, and give the plot the name `Plot1`.
- Use the formalism of the function command to get ready for the DE.
- Notice we have here once again used `#` to indicate a comment.
- In this case, in order to use common terminology, we now have told Sage `y` is no longer a variable, but instead a function (abstract) of the variable `x`.
- Use the differential equation solving command, with **I**nitial **C**ondition **S** of 2 and 2.
- Plot the solution and give it the name `Plot2`.
- Show a simplification of the symbolic version of the solution (which we did not know ahead of time!) as well as the sum of the two graphs - the solution against the slope field.

As you gain experience, you will see how to glean what *you* are looking for from examples in the documentation like this - which is one of the real goals of these tutorials.

Congratulations! You are now armed with the basics of deploying Sage in the calculus sequence.

CHAPTER

FIVE

SAGE INTRODUCTORY PROGRAMMING TUTORIAL

This Sage document is one of the tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

This tutorial will cover the following topics (and various others throughout):

- *Methods and Dot Notation*
- *Lists, Loops, and Set Builders*
- *Defining Functions*
- *Gotchas from names and copies*
- *Appendix: Advanced Introductory Topics*

We will motivate our examples using basic matrices and situations one might want to handle in everyday use of matrices in the classroom.

5.1 Methods and Dot Notation

Making a new matrix is not too hard in Sage.

```
sage: A = matrix([[1,2],[3,4]])
```

```
>>> from sage.all import *
>>> A = matrix([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
```

As we can see, this gives the matrix with rows given in the two bracketed sets.

```
sage: A
[1 2]
[3 4]
```

```
>>> from sage.all import *
>>> A
[1 2]
[3 4]
```

Some commands are available right off the bat, like derivatives are. This is the determinant.

```
sage: det(A)
-2
```

```
>>> from sage.all import *
>>> det(A)
-2
```

But some things are not available this way - for instance a row-reduced echelon form. We can 'tab' after this to make sure.

```
sage: r
```

```
>>> from sage.all import *
>>> r
```

So, as we've already seen in previous tutorials, *many* of the commands in Sage are "methods" of objects.

That is, we access them by typing:

- the name of the mathematical object,
- a dot/period,
- the name of the method, and
- parentheses (possibly with an argument).

This is a huge advantage, once you get familiar with it, because it allows you to do *only* the things that are possible, and *all* such things.

First, let's do the determinant again.

```
sage: A.det()
-2
```

```
>>> from sage.all import *
>>> A.det()
-2
```

Then we do the row-reduced echelon form.

```
sage: A.rref()
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> A.rref()
[1 0]
[0 1]
```

It is very important to keep in the parentheses.

Note: Things that would be legal without them would be called 'attributes', but Sage prefers stylistically to hide them, since math is made of functions and not elements of sets. Or so a category-theorist would say.

```
sage: A.det # Won't work
<built-in method det of sage.matrix.matrix_integer_dense.Matrix_integer_dense object␣
↪at ...>
```

```
>>> from sage.all import *
>>> A.det # Won't work
```

(continues on next page)

(continued from previous page)

```
<built-in method det of sage.matrix.matrix_integer_dense.Matrix_integer_dense object␣
↪at ...>
```

This is so useful because we can use the 'tab' key, remember!

```
sage: A.
```

```
>>> from sage.all import *
>>> A.
```

Sometimes you will have surprises. Subtle changes in an object can affect what commands are available, or what their outcomes are.

```
sage: A.echelon_form()
[1 0]
[0 2]
```

```
>>> from sage.all import *
>>> A.echelon_form()
[1 0]
[0 2]
```

This is because our original matrix had only integer coefficients, and you can't make the last entry one via elementary operations unless you multiply by a rational number!

```
sage: B = A.change_ring(QQ); B.echelon_form()
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> B = A.change_ring(QQ); B.echelon_form()
[1 0]
[0 1]
```

Another question is whether one needs an argument. Remember, it's easy to just read the documentation!

Below, let's see whether we need an argument to get a column.

```
sage: A.column?
```

```
>>> from sage.all import *
>>> A.column?
```

It looks like we do. Let's input `1`.

```
sage: A.column(1)
(2, 4)
```

```
>>> from sage.all import *
>>> A.column(Integer(1))
(2, 4)
```

Notice that this gives the SECOND column!

```
sage: A
[1 2]
[3 4]
```

```
>>> from sage.all import *
>>> A
[1 2]
[3 4]
```

What is that about?

5.2 Lists, Loops, and Set Builders

(Especially List Comprehensions!)

In the previous example, we saw that the `1` choice for the column of a matrix gives the *second* column.

```
sage: matrix([[1,2],[3,4]]).column(1)
(2, 4)
```

```
>>> from sage.all import *
>>> matrix([[Integer(1),Integer(2)],[Integer(3),Integer(4)]]).column(Integer(1))
(2, 4)
```

You might have thought that the would give the first column, but Sage (along with the Python programming language) begins numbering of anything that is like a sequence at zero. We've mentioned this once before, but it's very important to remember.

To reinforce this, let's formally introduce a fundamental object we've seen once or twice before, called a *list*.

You should think of a list as an ordered set, where the elements of the set can be pretty much anything - including other lists.

```
sage: my_list=[2,'Grover',[3,2,1]]; my_list
[2, 'Grover', [3, 2, 1]]
```

```
>>> from sage.all import *
>>> my_list=[Integer(2),'Grover',[Integer(3),Integer(2),Integer(1)]]; my_list
[2, 'Grover', [3, 2, 1]]
```

You can access any elements of such a list quite easily using square brackets. Just remember that the counting starts at zero.

```
sage: my_list[0]; my_list[2]
2
[3, 2, 1]
```

```
>>> from sage.all import *
>>> my_list[Integer(0)]; my_list[Integer(2)]
2
[3, 2, 1]
```

There are lots of advanced things one can do with lists.

```
sage: my_list[0:2]
[2, 'Grover']
```

```
>>> from sage.all import *
>>> my_list[Integer(0):Integer(2)]
[2, 'Grover']
```

However, our main reason for introducing this is more practical, as we'll now see.

- One of the best uses of the computer in the classroom is to quickly show tedious things.
- One of the most tedious things to do by hand in linear algebra is taking powers of matrices.
- Here we make the first four powers of our matrix 'by hand'.

```
sage: A = matrix([[1,2],[3,4]])
sage: A^0; A^1; A^2; A^3; A^4
[1 0]
[0 1]
[1 2]
[3 4]
[ 7 10]
[15 22]
[ 37  54]
[ 81 118]
[199 290]
[435 634]
```

```
>>> from sage.all import *
>>> A = matrix([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> A**Integer(0); A**Integer(1); A**Integer(2); A**Integer(3); A**Integer(4)
[1 0]
[0 1]
[1 2]
[3 4]
[ 7 10]
[15 22]
[ 37  54]
[ 81 118]
[199 290]
[435 634]
```

This is not terrible, but it's not exactly nice either, particularly if you might want to do something *with* these new matrices.

Instead, we can do what is known as a *loop* construction. See the notation below; it's at least vaguely mathematical.

```
sage: for i in [0,1,2,3,4]:
....:     A^i
[1 0]
[0 1]
[1 2]
[3 4]
[ 7 10]
[15 22]
[ 37  54]
[ 81 118]
[199 290]
[435 634]
```

```
>>> from sage.all import *
>>> for i in [Integer(0),Integer(1),Integer(2),Integer(3),Integer(4)]:
...     A**i
[1 0]
[0 1]
[1 2]
[3 4]
[ 7 10]
[15 22]
[ 37  54]
[ 81 118]
[199 290]
[435 634]
```

What did we do?

- For each i in the set $\{0, 1, 2, 3, 4\}$, return A^i.

Yeah, that makes sense. The square brackets created a list, and the powers of the original matrix come in the same order as the list.

(The colon in the first line and the indentation in the second line are **extremely** important; they are the basic syntactical structure of Python.)

For the curious: this is better, but still not perfect. It would be best to find a quicker way to write the possible values for i. There are two ways to do this in Sage.

```
sage: for i in [0..4]:
....:     det(A^i)
1
-2
4
-8
16
```

```
>>> from sage.all import *
>>> for i in (ellipsis_range(Integer(0),Ellipsis,Integer(4))):
...     det(A**i)
1
-2
4
-8
16
```

```
sage: for i in range(5):
....:     det(A^i)
1
-2
4
-8
16
```

```
>>> from sage.all import *
>>> for i in range(Integer(5)):
...     det(A**i)
1
-2
```

(continues on next page)

(continued from previous page)

```
4
-8
16
```

These ways of constructing lists are very useful - and demonstrate that, like many Sage/Python things, that counting begins at zero and ends at one less than the "end" in things like `range`.

Below, we show that one can get step sizes other than one as well.

```
sage: list(range(3, 23, 2))
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
sage: [3,5..21]
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```
>>> from sage.all import *
>>> list(range(Integer(3), Integer(23), Integer(2)))
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
>>> (ellipsis_range(Integer(3),Integer(5),Ellipsis,Integer(21)))
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Note: It is also important to emphasize that the `range` command does *not* include its last value! For a quick quiz, confirm this in the examples above.

This all works well. However, after a short time this will seem tedious as well (you may have to trust us on this). It turns out that there is a very powerful way to create such lists in a way that very strongly resembles the so-called set builder notation, called a *list comprehension* .

We start with a relatively easy example:

$$\{n^2 \mid n \in \mathbf{Z}, 3 \leq n \leq 12\}$$

Who hasn't written something like this at some point in a course?

This is a natural for the list comprehension, and can be very powerful when used in Sage.

```
sage: [n^2 for n in [3..12]]
[9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

```
>>> from sage.all import *
>>> [n**Integer(2) for n in (ellipsis_range(Integer(3),Ellipsis,Integer(12)))]
[9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

That's it. This sort of turns the loop around.

- The notation is easiest if you think of it mathematically; "The set of n^2, for (all) n in the range between 3 and 13."

This is phenomenally useful. Here is a nice plotting example:

```
sage: plot([x^n for n in [2..6]],(x,0,1))
Graphics object consisting of 5 graphics primitives
```

```
>>> from sage.all import *
>>> plot([x**n for n in (ellipsis_range(Integer(2),Ellipsis,Integer(6)))],(x,
↪Integer(0),Integer(1)))
Graphics object consisting of 5 graphics primitives
```

Now we apply it to the example we were doing in the first place. Notice we now have a nice concise description of all determinants of these matrices, without the syntax of colon and indentation:

```
sage: [det(A^i) for i in [0..4]]
[1, -2, 4, -8, 16]
```

```
>>> from sage.all import *
>>> [det(A**i) for i in (ellipsis_range(Integer(0),Ellipsis,Integer(4)))]
[1, -2, 4, -8, 16]
```

5.2.1 Tables

Finally, getting away from strictly programming, here is a useful tip.

Some of you may be familiar with a way to take such data and put it in tabular form from other programs. The `table` command does this for us:

```
sage: table( [ (i,det(A^i)) for i in [0..4] ] )
  0   1
  1   -2
  2   4
  3   -8
  4   16
```

```
>>> from sage.all import *
>>> table( [ (i,det(A**i)) for i in (ellipsis_range(Integer(0),Ellipsis,Integer(4)))␣
↪] )
  0   1
  1   -2
  2   4
  3   -8
  4   16
```

Notice that each element of *this* list is two items in parentheses (a so-called *tuple*).

Even better, we can put a header line on it to make it really clear what we are doing, by adding lists. We've seen keywords like `header=True` when doing some of our plotting and limits. What do you think will happen if you put dollar signs around the labels in the header?

```
sage: table( [('i', 'det(A^i)')] + [ (i,det(A^i)) for i in [0..4] ], header_row=True)
    i   det(A^i)
  ├───┼──────────┤
    0   1
    1   -2
    2   4
    3   -8
    4   16
```

```
>>> from sage.all import *
>>> table( [('i', 'det(A^i)')] + [ (i,det(A**i)) for i in (ellipsis_range(Integer(0),
↪Ellipsis,Integer(4))) ], header_row=True)
    i   det(A^i)
  ├───┼──────────┤
    0   1
    1   -2
```

(continues on next page)

(continued from previous page)

```
    2   4
    3   -8
    4   16
```

5.3 Defining Functions

Or, Extending Sage

It is often the case that Sage can do something, but doesn't have a simple command for it. For instance, you might want to take a matrix and output the square of that matrix minus the original matrix.

```
sage: A = matrix([[1,2],[3,4]])
sage: A^2-A
[ 6  8]
[12 18]
```

```
>>> from sage.all import *
>>> A = matrix([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> A**Integer(2)-A
[ 6  8]
[12 18]
```

How might one do this for other matrices? Of course, you could just always do $A^2 - A$ again and again. But this would be tedious and hard to follow, as with so many things that motivate a little programming. Here is how we solve this problem.

```
sage: def square_and_subtract(mymatrix):
....:     return mymatrix^2-mymatrix
```

```
>>> from sage.all import *
>>> def square_and_subtract(mymatrix):
...     return mymatrix**Integer(2)-mymatrix
```

The `def` command has created a new function called `square_and_subtract`. It should even be available using tab-completion.

Here are things to note about its construction:

- The input is inside the parentheses.
- The indentation and colon are crucial, as above.
- There will usually be a return value, given by `return`. This is what Sage will give below the input cell.

```
sage: square_and_subtract(A)
[ 6  8]
[12 18]
```

```
>>> from sage.all import *
>>> square_and_subtract(A)
[ 6  8]
[12 18]
```

```
sage: square_and_subtract(matrix([[1.5,0],[0,2]]))
[0.750000000000000 0.000000000000000]
[0.000000000000000  2.00000000000000]
```

```
>>> from sage.all import *
>>> square_and_subtract(matrix([[RealNumber('1.5'),Integer(0)],[Integer(0),
↪Integer(2)]]))
[0.750000000000000 0.000000000000000]
[0.000000000000000  2.00000000000000]
```

We can get a documentation string available by putting it in triple quotes `"""`.

```
sage: def square_and_subtract(mymatrix):
....:     """
....:     Return `A^2-A`
....:     """
....:     return mymatrix^2-mymatrix
```

```
>>> from sage.all import *
>>> def square_and_subtract(mymatrix):
...     """
...     Return `A^2-A`
...     """
...     return mymatrix**Integer(2)-mymatrix
```

```
sage: square_and_subtract?
```

```
>>> from sage.all import *
>>> square_and_subtract?
```

Pretty cool! And potentially quite helpful to students - and you - especially if the function is complicated. The A typesets properly because we put it in backticks (see above).

A very careful reader *may* have noticed that there is nothing that requires the input `mymatrix` to be a matrix. Sage will just try to square whatever you give it and subtract the original thing.

```
sage: square_and_subtract(sqrt(5))
-sqrt(5) + 5
```

```
>>> from sage.all import *
>>> square_and_subtract(sqrt(Integer(5)))
-sqrt(5) + 5
```

This is a typical thing to watch out for; just because you define something doesn't mean it's useful (though in this case it was).

Try to define a function which inputs a matrix and returns the determinant of the cube of the matrix. (There are a few ways to do this, of course!)

5.4 Gotchas from names and copies

Or, What's in a Name

Before we finish the tutorial, we want to point out a few programming-related things that often trip people up.

The first 'gotcha' is that it's possible to clobber constants!

```
sage: i
4
```

```
>>> from sage.all import *
>>> i
4
```

Can you figure out why `i=4`? Look carefully above to see when this happened.

- This gives a valuable lesson; *any time* you use a name there is potential for renaming.

This may seem quite bad, but could be quite logical to do - for instance, if you are only dealing with real matrices. It is definitely is something a Sage user needs to know, though.

Luckily, it's possible to restore symbolic constants.

```
sage: reset('i')
sage: i; i^2
I
-1
```

```
>>> from sage.all import *
>>> reset('i')
>>> i; i**Integer(2)
I
-1
```

```
sage: type(e)
<class 'sage.symbolic.expression.E'>
```

```
>>> from sage.all import *
>>> type(e)
<class 'sage.symbolic.expression.E'>
```

```
sage: type(pi)
<class 'sage.symbolic.expression.Expression'>
```

```
>>> from sage.all import *
>>> type(pi)
<class 'sage.symbolic.expression.Expression'>
```

Variables are another thing to keep in mind. As mentioned briefly in earlier tutorials, in order to maintain maximum flexibility while not allowing things to happen which shouldn't, only `x` is predefined, nothing else.

```
sage: type(x)
<class 'sage.symbolic.expression.Expression'>
```

```
>>> from sage.all import *
>>> type(x)
<class 'sage.symbolic.expression.Expression'>
```

```
sage: type(y)
Traceback (most recent call last):
...
NameError: name 'y' is not defined
```

```
>>> from sage.all import *
>>> type(y)
Traceback (most recent call last):
...
NameError: name 'y' is not defined
```

Warning: There *is* a way to get around this, but it unleashes a horde of potential misuse. See the cells below if you are interested in this.

```
sage: automatic_names(True) # not tested
sage: trig_expand((2*x + 4*y + sin(2*theta))^2) # not tested
4*(sin(theta)*cos(theta) + x + 2*y)^2
```

```
>>> from sage.all import *
>>> automatic_names(True) # not tested
>>> trig_expand((Integer(2)*x + Integer(4)*y + sin(Integer(2)*theta))**Integer(2))
↪# not tested
4*(sin(theta)*cos(theta) + x + 2*y)^2
```

This only works in the notebook. Now we'll turn it off.

```
sage: automatic_names(False) # not tested
```

```
>>> from sage.all import *
>>> automatic_names(False) # not tested
```

Another related issue is that a few names are "reserved" by Python/Sage, and which aren't allowed as variable names.

It's not surprising that 'for' is not allowed, but neither is 'lambda' (λ)! People often request a workaround for that.

```
sage: var('lambda')
Traceback (most recent call last):
...
ValueError: The name "lambda" is not a valid Python identifier.
```

```
>>> from sage.all import *
>>> var('lambda')
Traceback (most recent call last):
...
ValueError: The name "lambda" is not a valid Python identifier.
```

There are lots of ways to get around this. One popular, though annoying, way is this.

```
sage: var('lambda_')
lambda_
```

```
>>> from sage.all import *
>>> var('lambda_')
lambda_
```

```
sage: lambda_^2-1
lambda_^2 - 1
```

```
>>> from sage.all import *
>>> lambda_**Integer(2)-Integer(1)
lambda_^2 - 1
```

Still, in this one case, showing the expression still shows the Greek letter.

```
sage: show(lambda_^2-1)
```

```
>>> from sage.all import *
>>> show(lambda_**Integer(2)-Integer(1))
```

$$\lambda^2 - 1$$

Finally, there is another thing that can happen if you rename things too loosely.

```
sage: A = matrix(QQ,[[1,2],[3,4]])
sage: B = A
sage: C = copy(A)
```

```
>>> from sage.all import *
>>> A = matrix(QQ,[[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> B = A
>>> C = copy(A)
```

This actually has just made B and A refer to the same matrix. B isn't like A, it *is* A. The `copy` command gets around this (though not always).

```
sage: A[0,0]=987
```

```
>>> from sage.all import *
>>> A[Integer(0),Integer(0)]=Integer(987)
```

```
sage: show([A,B,C])
```

```
>>> from sage.all import *
>>> show([A,B,C])
```

$$\left[\begin{pmatrix} 987 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 987 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\right]$$

This is very subtle if you've never programmed before. Suffice it to say that it is safest to let each = sign stand for one thing, and to avoid redundant equals (unlike students at times).

5.5 Appendix: Advanced Introductory Topics

There are several things which are useful to know about, but which are not always introduced immediately in programming. We give a few examples here, but they are mainly here to make sure you have seen them so that they are not completely surprising when they come up again.

We saw the "block" structure of Python earlier, with the indentation. This gives the opportunity to introduce conditional statements and comparisons. Here, we just give an example for those who have seen conditionals ("if" clauses) before.

```
sage: B = matrix([[0,1,0,0],[0,0,1,0],[0,0,0,1],[0,0,0,0]])
sage: for i in range(5): # all integers from 0 to 4, remember
....:     if B^i==0: # We ask if the power is the zero matrix
....:         print(i)
4
```

```
>>> from sage.all import *
>>> B = matrix([[Integer(0),Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(0),Integer(0),Integer(0)]])
>>> for i in range(Integer(5)): # all integers from 0 to 4, remember
...     if B**i==Integer(0): # We ask if the power is the zero matrix
...         print(i)
4
```

We use the double equals sign to test for equality, because = assigns something to a variable name. Notice again that colons and indentation are the primary way that Sage/Python indicate syntax, just as commas and spaces do in English.

Another useful concept is that of a *dictionary* . This can be thought of as a mathematical mapping from "keys" to "values". The order is *not* important and *not* guaranteed. A dictionary is delimited by curly brackets and correspondence is indicated by colons.

Again, we will just give a small example to illustrate the idea.

What if one wants to specify a matrix using just the nonzero entries? A dictionary is a great way to do this.

This one puts 3 as an entry in the $(2, 3)$ spot, for example (remember, this is the *third* row and *fourth* column, since we start with zero).

```
sage: D = {(2,3):3, (4,5):6, (6,0):-3}
sage: C = matrix(D)
sage: C
[ 0  0  0  0  0  0]
[ 0  0  0  0  0  0]
[ 0  0  0  3  0  0]
[ 0  0  0  0  0  0]
[ 0  0  0  0  0  6]
[ 0  0  0  0  0  0]
[-3  0  0  0  0  0]
```

```
>>> from sage.all import *
>>> D = {(Integer(2),Integer(3)):Integer(3), (Integer(4),Integer(5)):Integer(6),␣
↪(Integer(6),Integer(0)):-Integer(3)}
>>> C = matrix(D)
>>> C
[ 0  0  0  0  0  0]
[ 0  0  0  0  0  0]
[ 0  0  0  3  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  0  0  0]
[ 0  0  0  0  0  6]
[ 0  0  0  0  0  0]
[-3  0  0  0  0  0]
```

That was a lot easier than inputting the whole matrix!

Finally, although Sage tries to anticipate what you want, sometimes it does matter how you define a given element in Sage.

- We saw this above with matrices over the rationals versus integers, for instance.

Here's an example with straight-up numbers.

```
sage: a = 2
sage: b = 2/1
sage: c = 2.0
sage: d = 2 + 0*I
sage: e = 2.0 + 0.0*I
```

```
>>> from sage.all import *
>>> a = Integer(2)
>>> b = Integer(2)/Integer(1)
>>> c = RealNumber('2.0')
>>> d = Integer(2) + Integer(0)*I
>>> e = RealNumber('2.0') + RealNumber('0.0')*I
```

We will not go in great depth about this, either, but it is worth knowing about. Notice that each of these types of numbers has or does not have $I = \sqrt{-1}$, decimal points, or division.

```
sage: parent(a)
Integer Ring
sage: parent(b)
Rational Field
sage: parent(c)
Real Field with 53 bits of precision
sage: parent(d)
Number Field in I with defining polynomial x^2 + 1 with I = 1*I
sage: parent(e)
Complex Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> parent(a)
Integer Ring
>>> parent(b)
Rational Field
>>> parent(c)
Real Field with 53 bits of precision
>>> parent(d)
Number Field in I with defining polynomial x^2 + 1 with I = 1*I
>>> parent(e)
Complex Field with 53 bits of precision
```

CHAPTER

SIX

TUTORIAL FOR ADVANCED 2D PLOTTING

This Sage document is one of the tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

Thanks to Sage's integration of projects like matplotlib, Sage has comprehensive two-dimensional plotting capabilities. This worksheet consists of the following sections:

This tutorial assumes that one is familiar with the basics of Sage, such as evaluating a cell by clicking the "evaluate" link, or by pressing `Shift` + `Enter` (hold down `Shift` while pressing the `Enter` key).

6.1 Graphing Functions and Plotting Curves

6.1.1 Cartesian Plots

A simple quadratic is easy.

```
sage: plot(x^2, (x,-2,2))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(x**Integer(2), (x,-Integer(2),Integer(2)))
Graphics object consisting of 1 graphics primitive
```

You can combine "plot objects" by adding them.

```
sage: regular = plot(x^2, (x,-2,2), color= 'purple')
sage: skinny = plot(4*x^2, (x,-2,2), color = 'green')
sage: regular + skinny
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> regular = plot(x**Integer(2), (x,-Integer(2),Integer(2)), color= 'purple')
>>> skinny = plot(Integer(4)*x**Integer(2), (x,-Integer(2),Integer(2)), color = 'green
↪')
>>> regular + skinny
Graphics object consisting of 2 graphics primitives
```

Problem : Plot a green $y = \sin(x)$ together with a red $y = 2\ \cos(x)$. (Hint: you can use `pi` as part of your range.)

Boundaries of a plot can be specified, in addition to the overall size.

```
sage: plot(1+e^(-x^2), xmin=-2, xmax=2, ymin=0, ymax=2.5, figsize=10)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(Integer(1)+e**(-x**Integer(2)), xmin=-Integer(2), xmax=Integer(2),␣
↪ymin=Integer(0), ymax=RealNumber('2.5'), figsize=Integer(10))
Graphics object consisting of 1 graphics primitive
```

Problem : Plot $y = 5 + 3\ \sin(4x)$ with suitable boundaries.

You can add lots of extra information.

```
sage: exponential = plot(1+e^(-x^2), xmin=-2, xmax=2, ymin=0, ymax=2.5)
sage: max_line = plot(2, xmin=-2, xmax=2, linestyle='-.', color = 'red')
sage: min_line = plot(1, xmin=-2, xmax=2, linestyle=':', color = 'red')
sage: exponential + max_line + min_line
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> exponential = plot(Integer(1)+e**(-x**Integer(2)), xmin=-Integer(2),␣
↪xmax=Integer(2), ymin=Integer(0), ymax=RealNumber('2.5'))
>>> max_line = plot(Integer(2), xmin=-Integer(2), xmax=Integer(2), linestyle='-.',␣
↪color = 'red')
>>> min_line = plot(Integer(1), xmin=-Integer(2), xmax=Integer(2), linestyle=':',␣
↪color = 'red')
>>> exponential + max_line + min_line
Graphics object consisting of 3 graphics primitives
```

You can fill regions with transparent color, and thicken the curve. This example uses several options to fine-tune our graphic.

```
sage: exponential = plot(1+e^(-x^2), xmin=-2, xmax=2, ymin=0, ymax=2.5, fill=0.5,␣
↪fillcolor='grey', fillalpha=0.3)
sage: min_line = plot(1, xmin=-2, xmax=2, linestyle='-', thickness= 6, color = 'red')
sage: exponential + min_line
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> exponential = plot(Integer(1)+e**(-x**Integer(2)), xmin=-Integer(2),␣
↪xmax=Integer(2), ymin=Integer(0), ymax=RealNumber('2.5'), fill=RealNumber('0.5'),␣
```

(continues on next page)

(continued from previous page)

```
↪fillcolor='grey', fillalpha=RealNumber('0.3'))
>>> min_line = plot(Integer(1), xmin=-Integer(2), xmax=Integer(2), linestyle='-',␣
↪thickness= Integer(6), color = 'red')
>>> exponential + min_line
Graphics object consisting of 3 graphics primitives
```

```
sage: sum([plot(x^n,(x,0,1),color=rainbow(5)[n]) for n in [0..4]])
Graphics object consisting of 5 graphics primitives
```

```
>>> from sage.all import *
>>> sum([plot(x**n,(x,Integer(0),Integer(1)),color=rainbow(Integer(5))[n]) for n in␣
↪(ellipsis_range(Integer(0),Ellipsis,Integer(4)))])
Graphics object consisting of 5 graphics primitives
```

Problem : Create a plot showing the cross-section area for the following solid of revolution problem: Consider the area bounded by $y = x^2 - 3x + 6$ and the line $y = 4$. Find the volume created by rotating this area around the line $y = 1$.

6.1.2 Parametric Plots

A parametric plot needs a list of two functions of the parameter; in Sage, we use *square* brackets to delimit the list. Notice also that we must declare `t` as a variable first. Because the graphic is slightly wider than it is tall, we use the `aspect_ratio` option (such options are called *keywords*) to ensure the axes are correct for how we want to view this object.

```
sage: t = var('t')
sage: parametric_plot([cos(t) + 3 * cos(t/9), sin(t) - 3 * sin(t/9)], (t, 0, 18*pi),␣
↪fill = True, aspect_ratio=1)
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> t = var('t')
>>> parametric_plot([cos(t) + Integer(3) * cos(t/Integer(9)), sin(t) - Integer(3) *␣
↪sin(t/Integer(9))], (t, Integer(0), Integer(18)*pi), fill = True, aspect_
↪ratio=Integer(1))
Graphics object consisting of 2 graphics primitives
```

Problem : These parametric equations will create a hypocycloid.

$$x(t) = 17\cos(t) + 3\cos(17t/3)$$

$$y(t) = 17\sin(t) - 3\sin(17t/3)$$

Create this as a parametric plot.

Sage automatically plots a 2d or 3d plot, and a curve or a surface, depending on how many variables and coordinates you specify.

```
sage: t = var('t')
sage: parametric_plot((t^2,sin(t)), (t,0,pi))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> t = var('t')
>>> parametric_plot((t**Integer(2),sin(t)), (t,Integer(0),pi))
Graphics object consisting of 1 graphics primitive
```

```
sage: parametric_plot((t^2,sin(t),cos(t)), (t,0,pi))
Graphics3d Object
```

```
>>> from sage.all import *
>>> parametric_plot((t**Integer(2),sin(t),cos(t)), (t,Integer(0),pi))
Graphics3d Object
```

```
sage: r = var('r')
sage: parametric_plot((t^2,sin(r*t),cos(r*t)), (t,0,pi),(r,-1,1))
Graphics3d Object
```

```
>>> from sage.all import *
>>> r = var('r')
>>> parametric_plot((t**Integer(2),sin(r*t),cos(r*t)), (t,Integer(0),pi),(r,-
↪Integer(1),Integer(1)))
Graphics3d Object
```

6.1.3 Polar Plots

Sage can also do polar plots.

```
sage: polar_plot(2 + 2*cos(x), (x, 0, 2*pi), color=hue(0.5), thickness=4)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> polar_plot(Integer(2) + Integer(2)*cos(x), (x, Integer(0), Integer(2)*pi),␣
↪color=hue(RealNumber('0.5')), thickness=Integer(4))
Graphics object consisting of 1 graphics primitive
```

Although they aren't essential, many of these examples try to demonstrate things like coloring, fills, and shading to give you a sense of the possibilities.

More than one polar curve can be specified in a list (square brackets). Notice the automatic graded shading of the fill color.

```
sage: t = var('t')
sage: polar_plot([cos(4*t) + 1.5,  0.5 * cos(4*t) + 2.5], (t, 0, 2*pi),
....:            color='black', thickness=2, fill=True, fillcolor='orange')
Graphics object consisting of 4 graphics primitives
```

```
>>> from sage.all import *
>>> t = var('t')
>>> polar_plot([cos(Integer(4)*t) + RealNumber('1.5'),  RealNumber('0.5') *␣
↪cos(Integer(4)*t) + RealNumber('2.5')], (t, Integer(0), Integer(2)*pi),
...            color='black', thickness=Integer(2), fill=True, fillcolor='orange')
Graphics object consisting of 4 graphics primitives
```

Problem: Create a plot for the following problem. Find the area that is inside the circle $r = 2$, but outside the cardioid $2 + 2\cos(\theta)$.

6.1.4 Interactive Demonstration

It may be of interest to see all these things put together in a very nice pedagogical graphic. Even though this is fairly advanced, and so you may want to skip the code, it is not as difficult as you might think to put together.

```
sage: html('<h2>Sine and unit circle (by Jurgis Pralgauskis)</h2> inspired by <a href=
↪"http://www.youtube.com/watch?v=Ohp6Okk_tww&feature=related">this video</a>' )
sage: # http://doc.sagemath.org/html/en/reference/sage/plot/plot.html
sage: radius = 100 # scale for radius of "unit" circle
sage: graph_params = dict(xmin = -2*radius,    xmax = 360,
....:                     ymin = -(radius+30), ymax = radius+30,
....:                     aspect_ratio=1,
....:                     axes = False
....:                     )
sage: def sine_and_unit_circle( angle=30, instant_show = True, show_pi=True ):
....:     ccenter_x, ccenter_y = -radius, 0  # center of circle on real coords
....:     sine_x = angle # the big magic to sync both graphs :)
....:     current_y = circle_y = sine_y = radius * sin(angle*pi/180)
....:     circle_x = ccenter_x + radius * cos(angle*pi/180)
....:     graph = Graphics()
....:     # we'll put unit circle and sine function on the same graph
....:     # so there will be some coordinate mangling ;)
....:     # CIRCLE
....:     unit_circle = circle((ccenter_x, ccenter_y), radius, color="#ccc")
....:     # SINE
....:     x = var('x')
....:     sine = plot( radius * sin(x*pi/180) , (x, 0, 360), color="#ccc" )
....:     graph += unit_circle + sine
....:     # CIRCLE axis
....:     # x axis
....:     graph +=  arrow( [-2*radius, 0], [0, 0], color = "#666" )
....:     graph += text("$(1, 0)$",  [-16, 16],  color = "#666")
....:     # circle y axis
....:     graph +=  arrow( [ccenter_x,-radius], [ccenter_x, radius], color = "#666" )
....:     graph += text("$(0, 1)$",  [ccenter_x, radius+15],  color = "#666")
....:     # circle center
....:     graph += text("$(0, 0)$",  [ccenter_x, 0],  color = "#666")
....:     # SINE x axis
....:     graph +=  arrow( [0,0], [360, 0], color = "#000" )
....:     # let's set tics
....:     # or http://aghitza.org/posts/tweak_labels_and_ticks_in_2d_plots_using_
↪matplotlib/
....:     # or wayt for https://github.com/sagemath/sage/issues/1431
....:     # ['$-\pi/3$', '$2\pi/3$', '$5\pi/3$']
....:     for x in range(0, 361, 30):
....:         graph += point( [x, 0] )
....:         angle_label = ".  $%3d^{\circ}$ " % x
....:         if show_pi: angle_label += " $(%s\pi) $"% x/180
....:         graph += text(angle_label,  [x, 0], rotation=-90,
....:         vertical_alignment='top', fontsize=8, color="#000" )
....:     # CURRENT VALUES
....:     # SINE -- y
....:     graph +=  arrow( [sine_x,0], [sine_x, sine_y], width=1, arrowsize=3)
....:     graph +=  arrow( [circle_x,0], [circle_x, circle_y], width=1, arrowsize=3)
....:     graph +=  line(([circle_x, current_y], [sine_x, current_y]), rgbcolor="#0F0
↪", linestyle = "--", alpha=0.5)
....:     # LABEL on sine
....:     graph += text("$(%d^{\circ}, %3.2f)$"%(sine_x, float(current_y)/radius), ␣
```

(continues on next page)

(continued from previous page)

```
↪[sine_x+30, current_y],  color = "#0A0")
....:     # ANGLE -- x
....:     # on sine
....:     graph += arrow( [0,0], [sine_x, 0], width=1, arrowsize=1, color='red')
....:     # on circle
....:     graph += disk( (ccenter_x, ccenter_y), float(radius)/4, (0, angle*pi/180),␣
↪color='red', fill=False, thickness=1)
....:     graph +=  arrow( [ccenter_x, ccenter_y], [circle_x, circle_y],
....:                 rgbcolor="#cccccc", width=1, arrowsize=1)
....:     if instant_show:
....:         show (graph,  **graph_params)
....:     return graph
sage: ####################
sage: # make Interaction
sage: #######################
sage: @interact
sage: def _( angle = slider([0..360], default=30, step_size=5,
....:          label="Pasirinkite kampą:    ", display_value=True) ):
....:     sine_and_unit_circle(angle, show_pi = False)
```

```
>>> from sage.all import *
>>> html('<h2>Sine and unit circle (by Jurgis Pralgauskis)</h2> inspired by <a href=
↪"http://www.youtube.com/watch?v=Ohp6Okk_tww&feature=related">this video</a>' )
>>> # http://doc.sagemath.org/html/en/reference/sage/plot/plot.html
>>> radius = Integer(100) # scale for radius of "unit" circle
>>> graph_params = dict(xmin = -Integer(2)*radius,    xmax = Integer(360),
...                     ymin = -(radius+Integer(30)), ymax = radius+Integer(30),
...                     aspect_ratio=Integer(1),
...                     axes = False
...                     )
>>> def sine_and_unit_circle( angle=Integer(30), instant_show = True, show_pi=True ):
...     ccenter_x, ccenter_y = -radius, Integer(0)  # center of circle on real coords
...     sine_x = angle # the big magic to sync both graphs :)
...     current_y = circle_y = sine_y = radius * sin(angle*pi/Integer(180))
...     circle_x = ccenter_x + radius * cos(angle*pi/Integer(180))
...     graph = Graphics()
...     # we'll put unit circle and sine function on the same graph
...     # so there will be some coordinate mangling ;)
...     # CIRCLE
...     unit_circle = circle((ccenter_x, ccenter_y), radius, color="#ccc")
...     # SINE
...     x = var('x')
...     sine = plot( radius * sin(x*pi/Integer(180)) , (x, Integer(0), Integer(360)),␣
↪color="#ccc" )
...     graph += unit_circle + sine
...     # CIRCLE axis
...     # x axis
...     graph +=  arrow( [-Integer(2)*radius, Integer(0)], [Integer(0), Integer(0)],␣
↪color = "#666" )
...     graph += text("$(1, 0)$",  [-Integer(16), Integer(16)],  color = "#666")
...     # circle y axis
...     graph +=  arrow( [ccenter_x,-radius], [ccenter_x, radius], color = "#666" )
...     graph += text("$(0, 1)$",  [ccenter_x, radius+Integer(15)],  color = "#666")
...     # circle center
...     graph += text("$(0, 0)$",  [ccenter_x, Integer(0)],  color = "#666")
...     # SINE x axis
```

(continues on next page)

(continued from previous page)

```
...     graph +=  arrow( [Integer(0),Integer(0)], [Integer(360), Integer(0)], color =
↪"#000" )
...     # let's set tics
...     # or http://aghitza.org/posts/tweak_labels_and_ticks_in_2d_plots_using_
↪matplotlib/
...     # or wayt for https://github.com/sagemath/sage/issues/1431
...     # ['$-\pi/3$', '$2\pi/3$', '$5\pi/3$']
...     for x in range(Integer(0), Integer(361), Integer(30)):
...         graph += point( [x, Integer(0)] )
...         angle_label = ".  $%3d^{\circ}$ " % x
...         if show_pi: angle_label += " $(%s\pi) $"% x/Integer(180)
...         graph += text(angle_label,  [x, Integer(0)], rotation=-Integer(90),
...         vertical_alignment='top', fontsize=Integer(8), color="#000" )
...     # CURRENT VALUES
...     # SINE -- y
...     graph +=  arrow( [sine_x,Integer(0)], [sine_x, sine_y], width=Integer(1),␣
↪arrowsize=Integer(3))
...     graph +=  arrow( [circle_x,Integer(0)], [circle_x, circle_y],␣
↪width=Integer(1), arrowsize=Integer(3))
...     graph +=  line(([circle_x, current_y], [sine_x, current_y]), rgbcolor="#0F0",␣
↪linestyle = "--", alpha=RealNumber('0.5'))
...     # LABEL on sine
...     graph += text("$(%d^{\circ}, %3.2f)$"%(sine_x, float(current_y)/radius), ␣
↪[sine_x+Integer(30), current_y],  color = "#0A0")
...     # ANGLE -- x
...     # on sine
...     graph += arrow( [Integer(0),Integer(0)], [sine_x, Integer(0)],␣
↪width=Integer(1), arrowsize=Integer(1), color='red')
...     # on circle
...     graph += disk( (ccenter_x, ccenter_y), float(radius)/Integer(4), (Integer(0),␣
↪angle*pi/Integer(180)), color='red', fill=False, thickness=Integer(1))
...     graph +=  arrow( [ccenter_x, ccenter_y], [circle_x, circle_y],
...                 rgbcolor="#cccccc", width=Integer(1), arrowsize=Integer(1))
...     if instant_show:
...         show (graph,  **graph_params)
...     return graph
>>> ####################
>>> # make Interaction
>>> #####################
>>> @interact
>>> def _( angle = slider((ellipsis_range(Integer(0),Ellipsis,Integer(360))),␣
↪default=Integer(30), step_size=Integer(5),
...          label="Pasirinkite kampą:    ", display_value=True) ):
...     sine_and_unit_circle(angle, show_pi = False)
```

6.2 Plotting Data

6.2.1 Plotting Data Points

Sometimes one wishes to simply plot data. Here, we demonstrate several ways of plotting points and data via the simple approximation to the Fibonacci numbers given by

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n ,$$

which is quite good after about $n = 5$.

First, we notice that the Fibonacci numbers are built in.

```
sage: fibonacci_sequence(6)
<generator object fibonacci_sequence at ...>
```

```
>>> from sage.all import *
>>> fibonacci_sequence(Integer(6))
<generator object fibonacci_sequence at ...>
```

```
sage: list(fibonacci_sequence(6))
[0, 1, 1, 2, 3, 5]
```

```
>>> from sage.all import *
>>> list(fibonacci_sequence(Integer(6)))
[0, 1, 1, 2, 3, 5]
```

The `enumerate` command is useful for taking a list and coordinating it with the counting numbers.

```
sage: list(enumerate(fibonacci_sequence(6)))
[(0, 0), (1, 1), (2, 1), (3, 2), (4, 3), (5, 5)]
```

```
>>> from sage.all import *
>>> list(enumerate(fibonacci_sequence(Integer(6))))
[(0, 0), (1, 1), (2, 1), (3, 2), (4, 3), (5, 5)]
```

So we just define the numbers and coordinate pairs we are about to plot.

```
sage: fibonacci = list(enumerate(fibonacci_sequence(6)))
sage: f(n)=(1/sqrt(5))*((1+sqrt(5))/2)^n
sage: asymptotic = [(i, f(i)) for i in range(6)]
sage: fibonacci
[(0, 0), (1, 1), (2, 1), (3, 2), (4, 3), (5, 5)]
sage: asymptotic
[(0, 1/5*sqrt(5)), (1, 1/10*sqrt(5)*(sqrt(5) + 1)), (2, 1/20*sqrt(5)*(sqrt(5) + 1)^2),
↪ (3, 1/40*sqrt(5)*(sqrt(5) + 1)^3), (4, 1/80*sqrt(5)*(sqrt(5) + 1)^4), (5, 1/
↪160*sqrt(5)*(sqrt(5) + 1)^5)]
```

```
>>> from sage.all import *
>>> fibonacci = list(enumerate(fibonacci_sequence(Integer(6))))
>>> __tmp__=var("n"); f = symbolic_expression((Integer(1)/
↪sqrt(Integer(5)))*((Integer(1)+sqrt(Integer(5)))/Integer(2))**n).function(n)
>>> asymptotic = [(i, f(i)) for i in range(Integer(6))]
>>> fibonacci
[(0, 0), (1, 1), (2, 1), (3, 2), (4, 3), (5, 5)]
>>> asymptotic
[(0, 1/5*sqrt(5)), (1, 1/10*sqrt(5)*(sqrt(5) + 1)), (2, 1/20*sqrt(5)*(sqrt(5) + 1)^2),
↪ (3, 1/40*sqrt(5)*(sqrt(5) + 1)^3), (4, 1/80*sqrt(5)*(sqrt(5) + 1)^4), (5, 1/
↪160*sqrt(5)*(sqrt(5) + 1)^5)]
```

Now we can plot not just the two sets of points, but also use several of the documented options for plotting points. Those coming from other systems may prefer `list_plot`.

```
sage: fib_plot=list_plot(fibonacci, color='red', pointsize=30)
sage: asy_plot = list_plot(asymptotic, marker='D',color='black',thickness=2,
```

(continues on next page)

(continued from previous page)

```
↪plotjoined=True)
sage: show(fib_plot+asy_plot, aspect_ratio=1)
```

```
>>> from sage.all import *
>>> fib_plot=list_plot(fibonacci, color='red', pointsize=Integer(30))
>>> asy_plot = list_plot(asymptotic, marker='D',color='black',thickness=Integer(2),
↪plotjoined=True)
>>> show(fib_plot+asy_plot, aspect_ratio=Integer(1))
```

Other options include `line`, `points`, and `scatter_plot`. Having the choice of markers for different data is particularly helpful for generating publishable graphics.

```
sage: fib_plot=scatter_plot(fibonacci, facecolor='red', marker='o',markersize=40)
sage: asy_plot = line(asymptotic, marker='D',color='black',thickness=2)
sage: show(fib_plot+asy_plot, aspect_ratio=1)
```

```
>>> from sage.all import *
>>> fib_plot=scatter_plot(fibonacci, facecolor='red', marker='o',
↪markersize=Integer(40))
>>> asy_plot = line(asymptotic, marker='D',color='black',thickness=Integer(2))
>>> show(fib_plot+asy_plot, aspect_ratio=Integer(1))
```

6.3 Contour-type Plots

6.3.1 Contour Plots

Contour plotting can be very useful when trying to get a handle on multivariable functions, as well as modeling. The basic syntax is essentially the same as for 3D plotting - simply an extension of the 2D plotting syntax.

```
sage: f(x,y)=y^2+1-x^3-x
sage: contour_plot(f, (x,-pi,pi), (y,-pi,pi))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(y**Integer(2)+Integer(1)-
↪x**Integer(3)-x).function(x,y)
>>> contour_plot(f, (x,-pi,pi), (y,-pi,pi))
Graphics object consisting of 1 graphics primitive
```

We can change colors, specify contours, label curves, and many other things. When there are many levels, the `colorbar` keyword becomes quite useful for keeping track of them. Notice that, as opposed to many other options, it can only be `True` or `False` (corresponding to whether it appears or does not appear).

```
sage: contour_plot(f, (x,-pi,pi), (y,-pi,pi),colorbar=True,labels=True)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> contour_plot(f, (x,-pi,pi), (y,-pi,pi),colorbar=True,labels=True)
Graphics object consisting of 1 graphics primitive
```

This example is fairly self-explanatory, but demonstrates the power of formatting, labeling, and the wide variety of built-in color gradations (colormaps or `cmap`). The strange-looking construction corresponding to `label_fmt` is a Sage/Python

data type called a *dictionary* , and turns out to be useful for more advanced Sage use; it consists of pairs connected by a colon, all inside curly braces.

```
sage: contour_plot(f, (x,-pi,pi), (y,-pi,pi), contours=[-4,0,4], fill=False,
....:     cmap='cool', labels=True, label_inline=True,
....:     label_fmt={-4:"low", 0:"medium", 4: "hi"}, label_colors='black')
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> contour_plot(f, (x,-pi,pi), (y,-pi,pi), contours=[-Integer(4),Integer(0),
↪Integer(4)], fill=False,
...     cmap='cool', labels=True, label_inline=True,
...     label_fmt={-Integer(4):"low", Integer(0):"medium", Integer(4): "hi"}, label_
↪colors='black')
Graphics object consisting of 1 graphics primitive
```

Implicit plots are a special type of contour plot (they just plot the zero contour).

```
sage: f(x,y)
-x^3 + y^2 - x + 1
```

```
>>> from sage.all import *
>>> f(x,y)
-x^3 + y^2 - x + 1
```

```
sage: implicit_plot(f(x,y)==0,(x,-pi,pi),(y,-pi,pi))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> implicit_plot(f(x,y)==Integer(0),(x,-pi,pi),(y,-pi,pi))
Graphics object consisting of 1 graphics primitive
```

A density plot is like a contour plot, but without discrete levels.

```
sage: density_plot(f, (x, -2, 2), (y, -2, 2))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> density_plot(f, (x, -Integer(2), Integer(2)), (y, -Integer(2), Integer(2)))
Graphics object consisting of 1 graphics primitive
```

Sometimes contour plots can be a little misleading (which makes for a *great* classroom discussion about the problems of ignorantly relying on technology). Here we combine a density plot and contour plot to show even better what is happening with the function.

```
sage: density_plot(f,(x,-2,2),(y,-2,2))+contour_plot(f,(x,-2,2),(y,-2,2),fill=False,
↪labels=True,label_inline=True,cmap='jet')
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> density_plot(f,(x,-Integer(2),Integer(2)),(y,-Integer(2),Integer(2)))+contour_
↪plot(f,(x,-Integer(2),Integer(2)),(y,-Integer(2),Integer(2)),fill=False,labels=True,
↪label_inline=True,cmap='jet')
Graphics object consisting of 2 graphics primitives
```

It can be worth getting familiar with the various options for different plots, especially if you will be doing a lot of them in a given worksheet or pedagogical situation.

Here are the options for contour plots.

- They are given as an "attribute" - no parentheses - of the `contour_plot` object.
- They are given as a dictionary (see *the programming tutorial*).

```
sage: contour_plot.options
{'aspect_ratio': 1,
 'axes': False,
 'colorbar': False,
 'contours': None,
 'fill': True,
 'frame': True,
 'labels': False,
 'legend_label': None,
 'linestyles': None,
 'linewidths': None,
 'plot_points': 100,
 'region': None}
```

```
>>> from sage.all import *
>>> contour_plot.options
{'aspect_ratio': 1,
 'axes': False,
 'colorbar': False,
 'contours': None,
 'fill': True,
 'frame': True,
 'labels': False,
 'legend_label': None,
 'linestyles': None,
 'linewidths': None,
 'plot_points': 100,
 'region': None}
```

Let's change it so that all future contour plots don't have the fill. That's how some of us might use them in a class. We'll also check that the change happened.

```
sage: contour_plot.options["fill"]=False
sage: contour_plot.options
{'aspect_ratio': 1,
 'axes': False,
 'colorbar': False,
 'contours': None,
 'fill': False,
 'frame': True,
 'labels': False,
 'legend_label': None,
 'linestyles': None,
 'linewidths': None,
 'plot_points': 100,
 'region': None}
```

```
>>> from sage.all import *
>>> contour_plot.options["fill"]=False
```

(continues on next page)

(continued from previous page)

```
>>> contour_plot.options
{'aspect_ratio': 1,
 'axes': False,
 'colorbar': False,
 'contours': None,
 'fill': False,
 'frame': True,
 'labels': False,
 'legend_label': None,
 'linestyles': None,
 'linewidths': None,
 'plot_points': 100,
 'region': None}
```

And it works!

```
sage: contour_plot(f,(x,-2,2),(y,-2,2))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> contour_plot(f,(x,-Integer(2),Integer(2)),(y,-Integer(2),Integer(2)))
Graphics object consisting of 1 graphics primitive
```

We can always access the default options, of course, to remind us.

```
sage: contour_plot.defaults()
{'aspect_ratio': 1,
 'axes': False,
 'colorbar': False,
 'contours': None,
 'fill': True,
 'frame': True,
 'labels': False,
 'legend_label': None,
 'linestyles': None,
 'linewidths': None,
 'plot_points': 100,
 'region': None}
```

```
>>> from sage.all import *
>>> contour_plot.defaults()
{'aspect_ratio': 1,
 'axes': False,
 'colorbar': False,
 'contours': None,
 'fill': True,
 'frame': True,
 'labels': False,
 'legend_label': None,
 'linestyles': None,
 'linewidths': None,
 'plot_points': 100,
 'region': None}
```

6.3.2 Vector fields

The syntax for vector fields is very similar to other multivariate constructions. Notice that the arrows are scaled appropriately, and colored by length in the 3D case.

```
sage: var('x,y')
(x, y)
sage: plot_vector_field((-y+x,y*x),(x,-3,3),(y,-3,3))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> plot_vector_field((-y+x,y*x),(x,-Integer(3),Integer(3)),(y,-Integer(3),
↪Integer(3)))
Graphics object consisting of 1 graphics primitive
```

```
sage: var('x,y,z')
(x, y, z)
sage: plot_vector_field3d((-y,-z,x), (x,-3,3),(y,-3,3),(z,-3,3))
Graphics3d Object
```

```
>>> from sage.all import *
>>> var('x,y,z')
(x, y, z)
>>> plot_vector_field3d((-y,-z,x), (x,-Integer(3),Integer(3)),(y,-Integer(3),
↪Integer(3)),(z,-Integer(3),Integer(3)))
Graphics3d Object
```

3d vector field plots are ideally viewed with 3d glasses (right-click on the plot and select "Style" and "Stereographic")

6.3.3 Complex Plots

We can plot functions of complex variables, where the magnitude is indicated by the brightness (black is zero magnitude) and the argument is indicated by the hue (red is a positive real number).

```
sage: f(z) = exp(z) #z^5 + z - 1 + 1/z
sage: complex_plot(f, (-5,5),(-5,5))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("z"); f = symbolic_expression(exp(z) ).function(z)#z^5 + z - 1 + 1/z
>>> complex_plot(f, (-Integer(5),Integer(5)),(-Integer(5),Integer(5)))
Graphics object consisting of 1 graphics primitive
```

6.3.4 Region plots

These plot where an expression is true, and are useful for plotting inequalities.

```
sage: region_plot(cos(x^2+y^2) <= 0, (x, -3, 3), (y, -3, 3),aspect_ratio=1)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> region_plot(cos(x**Integer(2)+y**Integer(2)) <= Integer(0), (x, -Integer(3),␣
↪Integer(3)), (y, -Integer(3), Integer(3)),aspect_ratio=Integer(1))
Graphics object consisting of 1 graphics primitive
```

We can get fancier options as well.

```
sage: region_plot(sin(x)*sin(y) >= 1/4, (x,-10,10), (y,-10,10), incol='yellow',␣
↪bordercol='black', borderstyle='dashed', plot_points=250,aspect_ratio=1)
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> region_plot(sin(x)*sin(y) >= Integer(1)/Integer(4), (x,-Integer(10),Integer(10)),␣
↪(y,-Integer(10),Integer(10)), incol='yellow', bordercol='black', borderstyle='dashed
↪', plot_points=Integer(250),aspect_ratio=Integer(1))
Graphics object consisting of 2 graphics primitives
```

Remember, what command would give full information about the syntax, options, and examples?

6.4 Miscellaneous Plot Information

6.4.1 Builtin Graphics Objects

Sage includes a variety of built-in graphics objects. These are particularly useful for adding to one's plot certain objects which are difficult to describe with equations, but which are basic geometric objects nonetheless. In this section we will try to demonstrate the syntax of some of the most useful of them; for most of them the contextual (remember, append `?`) help will give more details.

Points

To make one point, a coordinate pair suffices.

```
sage: point((3,5))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> point((Integer(3),Integer(5)))
Graphics object consisting of 1 graphics primitive
```

It doesn't matter how multiple point are generated; they must go in as input via a list (square brackets). Here, we demonstrate the hard (but naive) and easy (but a little more sophisticated) way to do this.

```
sage: f(x)=x^2
sage: points([(0,f(0)), (1,f(1)), (2,f(2)), (3,f(3)), (4,f(4))])
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(2)).function(x)
>>> points([(Integer(0),f(Integer(0))), (Integer(1),f(Integer(1))), (Integer(2),
↪f(Integer(2))), (Integer(3),f(Integer(3))), (Integer(4),f(Integer(4)))])
Graphics object consisting of 1 graphics primitive
```

```
sage: points([(x,f(x)) for x in range(5)])
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> points([(x,f(x)) for x in range(Integer(5))])
Graphics object consisting of 1 graphics primitive
```

Sage tries to tell how many dimensions you are working in automatically.

```
sage: f(x,y)=x^2-y^2
sage: points([(x,y,f(x,y)) for x in range(5) for y in range(5)])
Graphics3d Object
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x**Integer(2)-y**Integer(2)).
↪function(x,y)
>>> points([(x,y,f(x,y)) for x in range(Integer(5)) for y in range(Integer(5))])
Graphics3d Object
```

Lines

The syntax for lines is the same as that for points, but you get… well, you get connecting lines too!

```
sage: f(x)=x^2
sage: line([(x,f(x)) for x in range(5)])
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(2)).function(x)
>>> line([(x,f(x)) for x in range(Integer(5))])
Graphics object consisting of 1 graphics primitive
```

Balls

Sage has disks and spheres of various types available. Generally the center and radius are all that is needed, but other options are possible.

```
sage: circle((0,1),1,aspect_ratio=1)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> circle((Integer(0),Integer(1)),Integer(1),aspect_ratio=Integer(1))
Graphics object consisting of 1 graphics primitive
```

```
sage: disk((0,0), 1, (pi, 3*pi/2), color='yellow',aspect_ratio=1)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> disk((Integer(0),Integer(0)), Integer(1), (pi, Integer(3)*pi/Integer(2)), color=
↪'yellow',aspect_ratio=Integer(1))
Graphics object consisting of 1 graphics primitive
```

There are also ellipses and various arcs; see the full plot documentation.

Arrows

```
sage: arrow((0,0), (1,1))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> arrow((Integer(0),Integer(0)), (Integer(1),Integer(1)))
Graphics object consisting of 1 graphics primitive
```

Polygons

Polygons will try to complete themselves and fill in the interior; otherwise the syntax is fairly self-evident.

```
sage: polygon([[0,0],[1,1],[1,2]])
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> polygon([[Integer(0),Integer(0)],[Integer(1),Integer(1)],[Integer(1),Integer(2)]])
Graphics object consisting of 1 graphics primitive
```

Text

In 2d, one can typeset mathematics using the `text` command. This can be used to fine-tune certain types of labels. Unfortunately, in 3D the text is just text.

```
sage: text(r'$\int_0^2 x^2\, dx$', (0.5,2))+plot(x^2,(x,0,2),fill=True)
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> text(r'$\int_0^2 x^2\, dx$', (RealNumber('0.5'),Integer(2)))+plot(x**Integer(2),
↪(x,Integer(0),Integer(2)),fill=True)
Graphics object consisting of 3 graphics primitives
```

6.4.2 Saving Plots

We can save 2d plots to many different formats. Sage can determine the format based on the filename for the image.

```
sage: p=plot(x^2,(x,-1,1))
sage: p
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> p=plot(x**Integer(2),(x,-Integer(1),Integer(1)))
>>> p
Graphics object consisting of 1 graphics primitive
```

For testing purposes, we use the Sage standard temporary filename; however, you could use any string for a name that you wanted, like `"my_plot.png"`.

```
sage: name = tmp_filename() # this is a string
sage: png_savename = name+'.png'
sage: p.save(png_savename)
```

```
>>> from sage.all import *
>>> name = tmp_filename() # this is a string
>>> png_savename = name+'.png'
>>> p.save(png_savename)
```

In the notebook, these are usually ready for downloading in little links by the cells.

```
sage: pdf_savename = name+'.pdf'
sage: p.save(pdf_savename)
```

```
>>> from sage.all import *
>>> pdf_savename = name+'.pdf'
>>> p.save(pdf_savename)
```

Notably, we can export in formats ready for inclusion in web pages.

```
sage: svg_savename = name+'.svg'
sage: p.save(svg_savename)
```

```
>>> from sage.all import *
>>> svg_savename = name+'.svg'
>>> p.save(svg_savename)
```

CHAPTER

SEVEN

PREP QUICKSTART TUTORIALS

Sometimes all one needs to start using Sage in a field is a little help. These "Quickstart" tutorials are designed to get you up and running in one of these fields with a minimum of fuss and muss.

The assumed background in most of these is the first several tutorials in this set of documents. Similarly, keep in mind that these documents are *concise*; the next step from each of them is really the Sage reference manual.

Finally, the last tutorial is a special one introducing the reader to creation of interactive material in Sage. It's highly recommended that you study the *Programming tutorial* before tackling this one.

Contents:

7.1 Sage Quickstart for Abstract Algebra

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071).

As computers are discrete and finite, anything with a discrete, finite set of generators is natural to implement and explore.

7.1.1 Group Theory

Many common groups are pre-defined, usually as permutation groups: that is, explicitly described as subgroups of symmetric groups.

- Every group of order 15 or less is available as a permutation group.
- Sometimes they are available under special names, though.

```
sage: G = QuaternionGroup()
sage: G
Quaternion group of order 8 as a permutation group
```

```
>>> from sage.all import *
>>> G = QuaternionGroup()
>>> G
Quaternion group of order 8 as a permutation group
```

```
sage: H = AlternatingGroup(5)
sage: H
Alternating group of order 5!/2 as a permutation group
```

```
>>> from sage.all import *
>>> H = AlternatingGroup(Integer(5))
>>> H
Alternating group of order 5!/2 as a permutation group
```

```
sage: H.is_simple()
True
```

```
>>> from sage.all import *
>>> H.is_simple()
True
```

```
sage: D = DihedralGroup(8)
sage: D
Dihedral group of order 16 as a permutation group
```

```
>>> from sage.all import *
>>> D = DihedralGroup(Integer(8))
>>> D
Dihedral group of order 16 as a permutation group
```

We can access a lot of information about groups, such as:

- A list of subgroups up to conjugacy,
- or a stabilizer,
- or other things demonstrated below.

```
sage: for K in D.conjugacy_classes_subgroups():
....:     print(K)
Subgroup generated by [()] of (Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8)] of (Dihedral group of order 16 as a␣
↪permutation group)
Subgroup generated by [(2,8)(3,7)(4,6)] of (Dihedral group of order 16 as a␣
↪permutation group)
Subgroup generated by [(1,2)(3,8)(4,7)(5,6)] of (Dihedral group of order 16 as a␣
↪permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8)] of (Dihedral group␣
↪of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (2,8)(3,7)(4,6)] of (Dihedral group of␣
↪order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,2)(3,8)(4,7)(5,6)] of (Dihedral group␣
↪of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (2,8)(3,7)(4,6)] of␣
↪(Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (1,2,3,4,5,6,7,8)]␣
↪of (Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (1,2)(3,8)(4,7)(5,
↪6)] of (Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (2,8)(3,7)(4,6), (1,
↪2,3,4,5,6,7,8)] of (Dihedral group of order 16 as a permutation group)
```

```
>>> from sage.all import *
>>> for K in D.conjugacy_classes_subgroups():
...     print(K)
```

(continues on next page)

(continued from previous page)

```
Subgroup generated by [()] of (Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8)] of (Dihedral group of order 16 as a␣
↪permutation group)
Subgroup generated by [(2,8)(3,7)(4,6)] of (Dihedral group of order 16 as a␣
↪permutation group)
Subgroup generated by [(1,2)(3,8)(4,7)(5,6)] of (Dihedral group of order 16 as a␣
↪permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8)] of (Dihedral group␣
↪of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (2,8)(3,7)(4,6)] of (Dihedral group of␣
↪order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,2)(3,8)(4,7)(5,6)] of (Dihedral group␣
↪of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (2,8)(3,7)(4,6)] of␣
↪(Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (1,2,3,4,5,6,7,8)]␣
↪of (Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (1,2)(3,8)(4,7)(5,
↪6)] of (Dihedral group of order 16 as a permutation group)
Subgroup generated by [(1,5)(2,6)(3,7)(4,8), (1,3,5,7)(2,4,6,8), (2,8)(3,7)(4,6), (1,
↪2,3,4,5,6,7,8)] of (Dihedral group of order 16 as a permutation group)
```

In the previous cell we once again did a for loop over a set of objects rather than just a list of numbers. This can be very powerful.

```
sage: D.stabilizer(3)
Subgroup generated by [(1,5)(2,4)(6,8)] of (Dihedral group of order 16 as a␣
↪permutation group)
```

```
>>> from sage.all import *
>>> D.stabilizer(Integer(3))
Subgroup generated by [(1,5)(2,4)(6,8)] of (Dihedral group of order 16 as a␣
↪permutation group)
```

```
sage: len(D.normal_subgroups())
7
sage: for K in sorted(D.normal_subgroups()):
....:     print(K)
Subgroup generated by [()] of (Dihedral group of order 16 as a permutation group)
...
Subgroup generated by [(1,2,3,4,5,6,7,8), (1,8)(2,7)(3,6)(4,5)] of (Dihedral group of␣
↪order 16 as a permutation group)
```

```
>>> from sage.all import *
>>> len(D.normal_subgroups())
7
>>> for K in sorted(D.normal_subgroups()):
...     print(K)
Subgroup generated by [()] of (Dihedral group of order 16 as a permutation group)
...
Subgroup generated by [(1,2,3,4,5,6,7,8), (1,8)(2,7)(3,6)(4,5)] of (Dihedral group of␣
↪order 16 as a permutation group)
```

We can access specific subgroups if we know the generators as a permutation group.

```
sage: L = D.subgroup(["(1,3,5,7)(2,4,6,8)"])
```

```
>>> from sage.all import *
>>> L = D.subgroup(["(1,3,5,7)(2,4,6,8)"])
```

```
sage: L.is_normal(D)
True
```

```
>>> from sage.all import *
>>> L.is_normal(D)
True
```

```
sage: Q=D.quotient(L)
sage: Q
Permutation Group with generators [(1,2)(3,4), (1,3)(2,4)]
```

```
>>> from sage.all import *
>>> Q=D.quotient(L)
>>> Q
Permutation Group with generators [(1,2)(3,4), (1,3)(2,4)]
```

```
sage: Q.is_isomorphic(KleinFourGroup())
True
```

```
>>> from sage.all import *
>>> Q.is_isomorphic(KleinFourGroup())
True
```

There are some matrix groups as well, both finite and infinite.

```
sage: S = SL(2, GF(3))
sage: S
Special Linear Group of degree 2 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> S = SL(Integer(2), GF(Integer(3)))
>>> S
Special Linear Group of degree 2 over Finite Field of size 3
```

We can print out *all* of the elements of this group.

```
sage: for a in S:
....:     print(a)
[1 0]
[0 1]
...
[2 2]
[2 1]
```

```
>>> from sage.all import *
>>> for a in S:
...     print(a)
[1 0]
[0 1]
```

(continues on next page)

(continued from previous page)

```
...
[2 2]
[2 1]
```

```
sage: SS = SL(2, ZZ)
```

```
>>> from sage.all import *
>>> SS = SL(Integer(2), ZZ)
```

Of course, you have to be careful what you try to do!

```
sage: SS.list()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
>>> from sage.all import *
>>> SS.list()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

```
sage: for a in SS.gens():
....:     print(a)
[ 0  1]
[-1  0]
...
```

```
>>> from sage.all import *
>>> for a in SS.gens():
...     print(a)
[ 0  1]
[-1  0]
...
```

7.1.2 Rings

Sage has many pre-defined rings to experiment with. Here is how one would access $\mathbf{Z}/12\mathbf{Z}$, for instance.

```
sage: twelve = Integers(12)
sage: twelve
Ring of integers modulo 12
```

```
>>> from sage.all import *
>>> twelve = Integers(Integer(12))
>>> twelve
Ring of integers modulo 12
```

```
sage: twelve.is_field()
False
```

```
>>> from sage.all import *
>>> twelve.is_field()
False
```

```
sage: twelve.is_integral_domain()
False
```

```
>>> from sage.all import *
>>> twelve.is_integral_domain()
False
```

Quaternions, and generalizations

We can define generalized quaternion algebras, where $i^2 = a$, $j^2 = b$, and $k = i \cdot j$, all over $\mathbf{Q}$:

```
sage: quat = QuaternionAlgebra(-1, -1)
sage: quat
Quaternion Algebra (-1, -1) with base ring Rational Field
```

```
>>> from sage.all import *
>>> quat = QuaternionAlgebra(-Integer(1), -Integer(1))
>>> quat
Quaternion Algebra (-1, -1) with base ring Rational Field
```

```
sage: quat.is_field()
False
```

```
>>> from sage.all import *
>>> quat.is_field()
False
```

```
sage: quat.is_commutative()
False
```

```
>>> from sage.all import *
>>> quat.is_commutative()
False
```

```
sage: quat.is_division_algebra()
True
```

```
>>> from sage.all import *
>>> quat.is_division_algebra()
True
```

```
sage: quat2 = QuaternionAlgebra(5, -7)
```

```
>>> from sage.all import *
>>> quat2 = QuaternionAlgebra(Integer(5), -Integer(7))
```

```
sage: quat2.is_division_algebra()
True
```

```
>>> from sage.all import *
>>> quat2.is_division_algebra()
True
```

```
sage: quat2.is_field()
False
```

```
>>> from sage.all import *
>>> quat2.is_field()
False
```

Polynomial Rings

Polynomial arithmetic in Sage is a very important tool.

The first cell brings us back to the symbolic world. This is **not the same thing** as polynomials!

```
sage: reset('x') # This returns x to being a variable
sage: (x^4 + 2*x).parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> reset('x') # This returns x to being a variable
>>> (x**Integer(4) + Integer(2)*x).parent()
Symbolic Ring
```

Now we will turn x into the generator of a polynomial ring. The syntax is a little unusual, but you will see it often.

```
sage: R.<x> = QQ[]
sage: R
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> R
Univariate Polynomial Ring in x over Rational Field
```

```
sage: R.random_element() # random
-5/2*x^2 - 1/4*x - 1
```

```
>>> from sage.all import *
>>> R.random_element() # random
-5/2*x^2 - 1/4*x - 1
```

```
sage: R.is_integral_domain()
True
```

```
>>> from sage.all import *
>>> R.is_integral_domain()
True
```

```
sage: (x^4 + 2*x).parent()
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> (x**Integer(4) + Integer(2)*x).parent()
Univariate Polynomial Ring in x over Rational Field
```

```
sage: (x^2+x+1).is_irreducible()
True
```

```
>>> from sage.all import *
>>> (x**Integer(2)+x+Integer(1)).is_irreducible()
True
```

```
sage: F = GF(5)
sage: P.<y> = F[]
```

```
>>> from sage.all import *
>>> F = GF(Integer(5))
>>> P = F['y']; (y,) = P._first_ngens(1)
```

```
sage: P.random_element() # random
2*y
```

```
>>> from sage.all import *
>>> P.random_element() # random
2*y
```

```
sage: I = P.ideal(y^3+2*y)
sage: I
Principal ideal (y^3 + 2*y) of Univariate Polynomial Ring in y over Finite Field of␣
↪size 5
```

```
>>> from sage.all import *
>>> I = P.ideal(y**Integer(3)+Integer(2)*y)
>>> I
Principal ideal (y^3 + 2*y) of Univariate Polynomial Ring in y over Finite Field of␣
↪size 5
```

```
sage: Q = P.quotient(I)
```

```
>>> from sage.all import *
>>> Q = P.quotient(I)
```

```
sage: Q
Univariate Quotient Polynomial Ring in ybar over Finite Field of size 5 with modulus␣
↪y^3 + 2*y
```

```
>>> from sage.all import *
>>> Q
Univariate Quotient Polynomial Ring in ybar over Finite Field of size 5 with modulus␣
↪y^3 + 2*y
```

7.1.3 Fields

Sage has superb support for finite fields and extensions of the rationals.

Finite Fields

```
sage: F.<a> = GF(3^4)
sage: F
Finite Field in a of size 3^4
```

```
>>> from sage.all import *
>>> F = GF(Integer(3)**Integer(4), names=('a',)); (a,) = F._first_ngens(1)
>>> F
Finite Field in a of size 3^4
```

The generator satisfies a Conway polynomial, by default, or the polynomial can be specified.

```
sage: F.polynomial()
a^4 + 2*a^3 + 2
```

```
>>> from sage.all import *
>>> F.polynomial()
a^4 + 2*a^3 + 2
```

```
sage: F.list()
[0, a, a^2, a^3, a^3 + 1, a^3 + a + 1, a^3 + a^2 + a + 1, 2*a^3 + a^2 + a + 1, a^2 +␣
↪a + 2, a^3 + a^2 + 2*a, 2*a^3 + 2*a^2 + 1, a^3 + a + 2, a^3 + a^2 + 2*a + 1, 2*a^3␣
↪+ 2*a^2 + a + 1, a^3 + a^2 + a + 2, 2*a^3 + a^2 + 2*a + 1, 2*a^2 + a + 2, 2*a^3 + a^
↪2 + 2*a, 2*a^2 + 2, 2*a^3 + 2*a, 2*a^3 + 2*a^2 + 2, a^3 + 2*a + 2, a^3 + 2*a^2 +␣
↪2*a + 1, 2*a^2 + a + 1, 2*a^3 + a^2 + a, a^2 + 2, a^3 + 2*a, a^3 + 2*a^2 + 1, a + 1,
↪ a^2 + a, a^3 + a^2, 2*a^3 + 1, 2*a^3 + a + 2, 2*a^3 + a^2 + 2*a + 2, 2*a^2 + 2*a +␣
↪2, 2*a^3 + 2*a^2 + 2*a, a^3 + 2*a^2 + 2, 2*a + 1, 2*a^2 + a, 2*a^3 + a^2, 2, 2*a,␣
↪2*a^2, 2*a^3, 2*a^3 + 2, 2*a^3 + 2*a + 2, 2*a^3 + 2*a^2 + 2*a + 2, a^3 + 2*a^2 +␣
↪2*a + 2, 2*a^2 + 2*a + 1, 2*a^3 + 2*a^2 + a, a^3 + a^2 + 2, 2*a^3 + 2*a + 1, 2*a^3␣
↪+ 2*a^2 + a + 2, a^3 + a^2 + 2*a + 2, 2*a^3 + 2*a^2 + 2*a + 1, a^3 + 2*a^2 + a + 2,␣
↪a^2 + 2*a + 1, a^3 + 2*a^2 + a, a^2 + 1, a^3 + a, a^3 + a^2 + 1, 2*a^3 + a + 1, 2*a^
↪3 + a^2 + a + 2, a^2 + 2*a + 2, a^3 + 2*a^2 + 2*a, 2*a^2 + 1, 2*a^3 + a, 2*a^3 + a^
↪2 + 2, 2*a + 2, 2*a^2 + 2*a, 2*a^3 + 2*a^2, a^3 + 2, a^3 + 2*a + 1, a^3 + 2*a^2 + a␣
↪+ 1, a^2 + a + 1, a^3 + a^2 + a, 2*a^3 + a^2 + 1, a + 2, a^2 + 2*a, a^3 + 2*a^2, 1]
```

```
>>> from sage.all import *
>>> F.list()
[0, a, a^2, a^3, a^3 + 1, a^3 + a + 1, a^3 + a^2 + a + 1, 2*a^3 + a^2 + a + 1, a^2 +␣
↪a + 2, a^3 + a^2 + 2*a, 2*a^3 + 2*a^2 + 1, a^3 + a + 2, a^3 + a^2 + 2*a + 1, 2*a^3␣
↪+ 2*a^2 + a + 1, a^3 + a^2 + a + 2, 2*a^3 + a^2 + 2*a + 1, 2*a^2 + a + 2, 2*a^3 + a^
↪2 + 2*a, 2*a^2 + 2, 2*a^3 + 2*a, 2*a^3 + 2*a^2 + 2, a^3 + 2*a + 2, a^3 + 2*a^2 +␣
↪2*a + 1, 2*a^2 + a + 1, 2*a^3 + a^2 + a, a^2 + 2, a^3 + 2*a, a^3 + 2*a^2 + 1, a + 1,
↪ a^2 + a, a^3 + a^2, 2*a^3 + 1, 2*a^3 + a + 2, 2*a^3 + a^2 + 2*a + 2, 2*a^2 + 2*a +␣
↪2, 2*a^3 + 2*a^2 + 2*a, a^3 + 2*a^2 + 2, 2*a + 1, 2*a^2 + a, 2*a^3 + a^2, 2, 2*a,␣
↪2*a^2, 2*a^3, 2*a^3 + 2, 2*a^3 + 2*a + 2, 2*a^3 + 2*a^2 + 2*a + 2, a^3 + 2*a^2 +␣
↪2*a + 2, 2*a^2 + 2*a + 1, 2*a^3 + 2*a^2 + a, a^3 + a^2 + 2, 2*a^3 + 2*a + 1, 2*a^3␣
↪+ 2*a^2 + a + 2, a^3 + a^2 + 2*a + 2, 2*a^3 + 2*a^2 + 2*a + 1, a^3 + 2*a^2 + a + 2,␣
↪a^2 + 2*a + 1, a^3 + 2*a^2 + a, a^2 + 1, a^3 + a, a^3 + a^2 + 1, 2*a^3 + a + 1, 2*a^
↪3 + a^2 + a + 2, a^2 + 2*a + 2, a^3 + 2*a^2 + 2*a, 2*a^2 + 1, 2*a^3 + a, 2*a^3 + a^
```

(continues on next page)

(continued from previous page)

```
↪2 + 2, 2*a + 2, 2*a^2 + 2*a, 2*a^3 + 2*a^2, a^3 + 2, a^3 + 2*a + 1, a^3 + 2*a^2 + a␣
↪+ 1, a^2 + a + 1, a^3 + a^2 + a, 2*a^3 + a^2 + 1, a + 2, a^2 + 2*a, a^3 + 2*a^2, 1]
```

```
sage: (a^3 + 2*a^2 + 2)*(2*a^3 + 2*a + 1)
2*a^3 + a^2 + a + 1
```

```
>>> from sage.all import *
>>> (a**Integer(3) + Integer(2)*a**Integer(2) + Integer(2))*(Integer(2)*a**Integer(3)␣
↪+ Integer(2)*a + Integer(1))
2*a^3 + a^2 + a + 1
```

F should be the splitting field of the polynomial $x^{81} - x$, so it is very good that we get no output from the following cell, which combines a loop and a conditional statement.

```
sage: for a in F:
....:     if not (a^81 - a == 0):
....:         print("Oops!")
```

```
>>> from sage.all import *
>>> for a in F:
...     if not (a**Integer(81) - a == Integer(0)):
...         print("Oops!")
```

Field Extensions, Number Fields

Most things you will need in an undergraduate algebra classroom are already in Sage.

```
sage: N = QQ[sqrt(2)]
sage: N
Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?
```

```
>>> from sage.all import *
>>> N = QQ[sqrt(Integer(2))]
>>> N
Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?
```

```
sage: var('z')
z
sage: M.<a>=NumberField(z^2-2)
sage: M
Number Field in a with defining polynomial z^2 - 2
```

```
>>> from sage.all import *
>>> var('z')
z
>>> M = NumberField(z**Integer(2)-Integer(2), names=('a',)); (a,) = M._first_ngens(1)
>>> M
Number Field in a with defining polynomial z^2 - 2
```

```
sage: M.degree()
2
```

```
>>> from sage.all import *
>>> M.degree()
2
```

```
sage: M.is_galois()
True
```

```
>>> from sage.all import *
>>> M.is_galois()
True
```

```
sage: M.is_isomorphic(N)
True
```

```
>>> from sage.all import *
>>> M.is_isomorphic(N)
True
```

7.2 Sage Quickstart for Differential Equations

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

Solving differential equations is a combination of exact and numerical methods, and hence a great place to explore with the computer. We have already seen one example of this in the calculus tutorial, which is worth reviewing.

7.2.1 Basic Symbolic Techniques

```
sage: y = function('y')(x)
sage: de = diff(y,x) + y -2
sage: h = desolve(de, y)
```

```
>>> from sage.all import *
>>> y = function('y')(x)
>>> de = diff(y,x) + y -Integer(2)
>>> h = desolve(de, y)
```

Forgetting about plotting for the moment, notice that there are three things one needs to solve a differential equation symbolically:

- an abstract function `y`;
- a differential equation, which here we put in a separate line;
- the actual **d**ifferential **e**quation **solve** command (bold for the acronym `desolve`).

```
sage: show(expand(h))
```

```
>>> from sage.all import *
>>> show(expand(h))
```

$$ce^{(-x)} + 2$$

Since we did not specify any initial conditions, Sage (from Maxima) puts in a parameter. If we want to put in an initial condition, we use `ics` (for **i**nitial **c**onditions). For example, we set `ics=[0,3]` to specify that when $x = 0$, $y = 3$.

```
sage: h = desolve(de, y, ics=[0,3]); h
(2*e^x + 1)*e^(-x)
```

```
>>> from sage.all import *
>>> h = desolve(de, y, ics=[Integer(0),Integer(3)]); h
(2*e^x + 1)*e^(-x)
```

And of course we have already noted that we can plot all this with a slope field.

```
sage: y = var('y') # Needed so we can plot
sage: Plot1=plot_slope_field(2-y,(x,0,3),(y,0,5))
sage: Plot2=plot(h,x,0,3)
sage: Plot1+Plot2
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> y = var('y') # Needed so we can plot
>>> Plot1=plot_slope_field(Integer(2)-y,(x,Integer(0),Integer(3)),(y,Integer(0),
↪Integer(5)))
>>> Plot2=plot(h,x,Integer(0),Integer(3))
>>> Plot1+Plot2
Graphics object consisting of 2 graphics primitives
```

Note: Regarding symbolic functions versus symbolic variables:

- If you wanted to make `y` an abstract function again instead of a variable, you'd have to do that separately. A differential equation requires a `function` but plotting requires a `var`.
- Another option is to let `z` be the name of the vertical axis variable.
- Either way something will have to give, since in common speaking about these things we treat `y` as both a variable and a function, which is much trickier to accomplish with a computer.

There are many other differential equation facilities in Sage. We can't cover all the variants of this in a quickstart, but the documentation is good for symbolic solvers.

```
sage: desolvers?
```

```
>>> from sage.all import *
>>> desolvers?
```

For instance, Maxima can do systems, as well as use Laplace transforms, and we include versions of these wrapped for ease of use.

In all differential equation solving routines, it is important to pay attention to the syntax! In the following example, we have placed the differential equation in the body of the command, and had to specify that `f` was the **d**ependent **var**iable (`dvar`), as well as give initial conditions $f(0) = 1$ and $f'(0) = 2$, which gives the last list in the example.

```
sage: f=function('f')(x)
sage: desolve_laplace(diff(f,x,2) == 2*diff(f,x)-f, dvar = f, ics = [0,1,2])
x*e^x + e^x
```

```
>>> from sage.all import *
>>> f=function('f')(x)
>>> desolve_laplace(diff(f,x,Integer(2)) == Integer(2)*diff(f,x)-f, dvar = f, ics =␣
↪[Integer(0),Integer(1),Integer(2)])
x*e^x + e^x
```

```
sage: g(x)=x*e^x+e^x
sage: derivative(g,x,2)-2*derivative(g,x)+g
x |--> 0
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); g = symbolic_expression(x*e**x+e**x).function(x)
>>> derivative(g,x,Integer(2))-Integer(2)*derivative(g,x)+g
x |--> 0
```

7.2.2 Numerical and Power Series Methods

There are also numerical methods.

For instance, one of the options above was `desolve_rk4`. This is a fourth-order Runge-Kutta method, and returns appropriate (numerical) output. Here, we *must* give the dependent variable *and* initial conditions.

```
sage: y = function('y')(x)
sage: de = diff(y,x) + y -2
sage: h = desolve_rk4(de, y, step=.05, ics=[0,3])
```

```
>>> from sage.all import *
>>> y = function('y')(x)
>>> de = diff(y,x) + y -Integer(2)
>>> h = desolve_rk4(de, y, step=RealNumber('.05'), ics=[Integer(0),Integer(3)])
```

It can be fun to compare this with the original, symbolic solution. We use the `points` command from the advanced plotting tutorial.

```
sage: h1 = desolve(de, y, ics=[0,3])
sage: plot(h1,(x,0,5),color='red')+points(h)
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> h1 = desolve(de, y, ics=[Integer(0),Integer(3)])
>>> plot(h1,(x,Integer(0),Integer(5)),color='red')+points(h)
Graphics object consisting of 2 graphics primitives
```

The primary use of numerical routines from here is pedagogical in nature.

For more advanced numerical routines, we primarily use the GNU scientific library. Using this is a little more sophisticated, but gives a wealth of options.

```
sage: ode_solver?
```

```
>>> from sage.all import *
>>> ode_solver?
```

We can even do power series solutions. In order to do this, we must first define a special *power series ring* , including the precision.

```
sage: R.<t> = PowerSeriesRing(QQ, default_prec=10)
sage: a = -1 + 0*t
sage: b = 2 + 0*t
sage: h=a.solve_linear_de(b=b,f0=3,prec=10)
```

```
>>> from sage.all import *
>>> R = PowerSeriesRing(QQ, default_prec=Integer(10), names=('t',)); (t,) = R._first_
↪ngens(1)
>>> a = -Integer(1) + Integer(0)*t
>>> b = Integer(2) + Integer(0)*t
>>> h=a.solve_linear_de(b=b,f0=Integer(3),prec=Integer(10))
```

This power series solution is pretty good for a while!

```
sage: h = h.polynomial()
sage: plot(h,-2,5)+plot(2+e^-x,(x,-2,5),color='red',linestyle=':',thickness=3)
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> h = h.polynomial()
>>> plot(h,-Integer(2),Integer(5))+plot(Integer(2)+e**-x,(x,-Integer(2),Integer(5)),
↪color='red',linestyle=':',thickness=Integer(3))
Graphics object consisting of 2 graphics primitives
```

This was just an introduction; there are a lot of resources for differential equations using Sage elsewhere, including a book by David Joyner, who wrote much of the original code wrapping Maxima for Sage to do just this.

7.3 Sage Quickstart for Graph Theory and Discrete Mathematics

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

As computers are discrete and finite, topics from discrete mathematics are natural to implement and use. We'll start with Graph Theory.

7.3.1 Graph Theory

The pre-defined `graphs` object provides an abundance of examples. Just tab to see!

```
sage: graphs.[tab]
```

```
>>> from sage.all import *
>>> graphs.[tab]
```

Its companion `digraphs` has many built-in examples as well.

Visualizing a graph is similar to plotting functions.

```
sage: G = graphs.HeawoodGraph()
sage: plot(G)
Graphics object consisting of 36 graphics primitives
```

```
>>> from sage.all import *
>>> G = graphs.HeawoodGraph()
>>> plot(G)
Graphics object consisting of 36 graphics primitives
```

Defining your own graph is easy. One way is the following.

- Put a vertex next to a list (recall this concept from the programming tutorial) with a colon, to show its adjacent vertices. For example, to put vertex 4 next to vertices 0 and 2, use `4:[0,2]`.
- Now combine all these in curly braces (in the advanced appendix to the programming tutorial, this is called a *dictionary*).

```
sage: H=Graph({0:[1,2,3], 4:[0,2], 6:[1,2,3,4,5]})
sage: plot(H)
Graphics object consisting of 18 graphics primitives
```

```
>>> from sage.all import *
>>> H=Graph({Integer(0):[Integer(1),Integer(2),Integer(3)], Integer(4):[Integer(0),
↪Integer(2)], Integer(6):[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)]})
>>> plot(H)
Graphics object consisting of 18 graphics primitives
```

Adjacency matrices, other graphs, and similar inputs are also recognized.

Graphs have "position" information for location of vertices. There are several different ways to compute a layout, or you can compute your own. Pre-defined graphs often come with "nice" layouts.

```
sage: H.set_pos(H.layout_circular())
sage: plot(H)
Graphics object consisting of 18 graphics primitives
```

```
>>> from sage.all import *
>>> H.set_pos(H.layout_circular())
>>> plot(H)
Graphics object consisting of 18 graphics primitives
```

Vertices can be lots of things, for example the codewords of an error-correcting code.

Note: Technical caveat: they need to be "immutable", like Python's tuples.

Here we have a matrix over the integers and a matrix of variables as vertices.

```
sage: a=matrix([[1,2],[3,4]])
sage: var('x y z w')
(x, y, z, w)
sage: b=matrix([[x,y],[z,w]])
sage: a.set_immutable()
sage: b.set_immutable()
sage: K=DiGraph({a:[b]})
sage: show(K, vertex_size=800)
```

```
>>> from sage.all import *
>>> a=matrix([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> var('x y z w')
(x, y, z, w)
>>> b=matrix([[x,y],[z,w]])
>>> a.set_immutable()
>>> b.set_immutable()
>>> K=DiGraph({a:[b]})
>>> show(K, vertex_size=Integer(800))
```

Edges can be labeled.

```
sage: L=graphs.CycleGraph(5)
sage: for edge in L.edges(sort=True):
....:     u = edge[0]
....:     v = edge[1]
....:     L.set_edge_label(u, v, u*v)
sage: plot(L, edge_labels=True)
Graphics object consisting of 16 graphics primitives
```

```
>>> from sage.all import *
>>> L=graphs.CycleGraph(Integer(5))
>>> for edge in L.edges(sort=True):
...     u = edge[Integer(0)]
...     v = edge[Integer(1)]
...     L.set_edge_label(u, v, u*v)
>>> plot(L, edge_labels=True)
Graphics object consisting of 16 graphics primitives
```

There are natural connections to other areas of mathematics. Here we compute the automorphism group and eigenvalues of the skeleton of a cube.

```
sage: C = graphs.CubeGraph(3)
sage: plot(C)
Graphics object consisting of 21 graphics primitives
```

```
>>> from sage.all import *
>>> C = graphs.CubeGraph(Integer(3))
>>> plot(C)
Graphics object consisting of 21 graphics primitives
```

```
sage: Aut=C.automorphism_group()
sage: print("Order of automorphism group: {}".format(Aut.order()))
Order of automorphism group:  48
sage: print("Group: \n{}".format(Aut)) # random
Group:
Permutation Group with generators [('010','100')('011','101'), ('001','010')('101',
↪'110'), ('000','001')('010','011')('100','101')('110','111')]
```

```
>>> from sage.all import *
>>> Aut=C.automorphism_group()
>>> print("Order of automorphism group: {}".format(Aut.order()))
Order of automorphism group:  48
>>> print("Group: \n{}".format(Aut)) # random
Group:
```

(continues on next page)

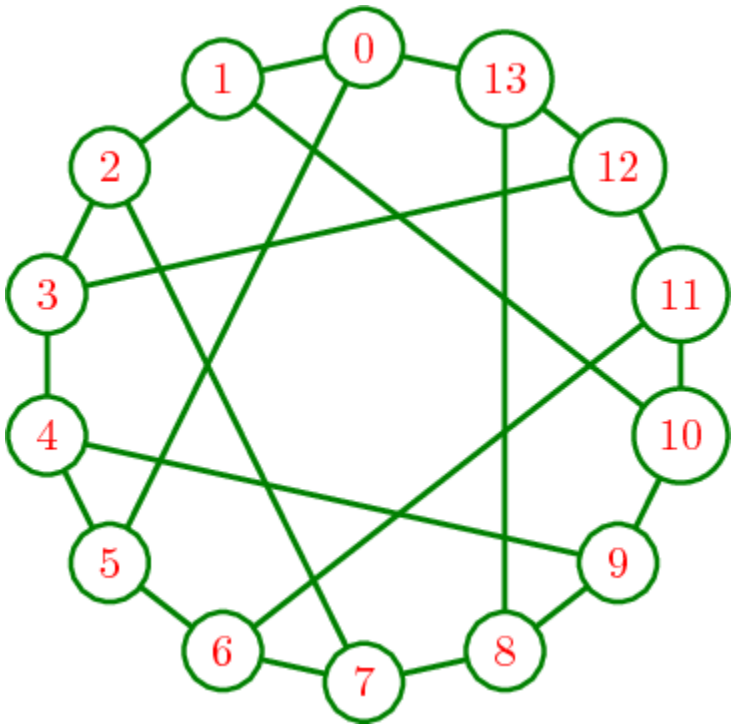
(continued from previous page)

```
Permutation Group with generators [('010','100')('011','101'), ('001','010')('101',
↪'110'), ('000','001')('010','011')('100','101')('110','111')]
```

```
sage: C.spectrum()
[3, 1, 1, 1, -1, -1, -1, -3]
```

```
>>> from sage.all import *
>>> C.spectrum()
[3, 1, 1, 1, -1, -1, -1, -3]
```

There is a huge amount of LaTeX support for graphs. The following graphic shows an example of what can be done; this is the Heawood graph.



Press 'tab' at the next command to see all the available options.

```
sage: sage.graphs.graph_latex.GraphLatex.set_option?
```

```
>>> from sage.all import *
>>> sage.graphs.graph_latex.GraphLatex.set_option?
```

7.3.2 More Discrete Mathematics

Discrete mathematics is a broad area, and Sage has excellent support for much of it. This is largely due to the "sage-combinat" group. These developers previously developed for MuPad (as "mupad-combinat") but switched over to Sage shortly before MuPad was sold.

Simple Combinatorics

Sage can work with basic combinatorial structures like combinations and permutations.

```
sage: pets = ['dog', 'cat', 'snake', 'spider']
sage: C=Combinations(pets)
sage: C.list()
[[], ['dog'], ['cat'], ['snake'], ['spider'], ['dog', 'cat'], ['dog', 'snake'], ['dog
↪', 'spider'], ['cat', 'snake'], ['cat', 'spider'], ['snake', 'spider'], ['dog', 'cat
↪', 'snake'], ['dog', 'cat', 'spider'], ['dog', 'snake', 'spider'], ['cat', 'snake',
↪'spider'], ['dog', 'cat', 'snake', 'spider']]
```

```
>>> from sage.all import *
>>> pets = ['dog', 'cat', 'snake', 'spider']
>>> C=Combinations(pets)
>>> C.list()
[[], ['dog'], ['cat'], ['snake'], ['spider'], ['dog', 'cat'], ['dog', 'snake'], ['dog
↪', 'spider'], ['cat', 'snake'], ['cat', 'spider'], ['snake', 'spider'], ['dog', 'cat
↪', 'snake'], ['dog', 'cat', 'spider'], ['dog', 'snake', 'spider'], ['cat', 'snake',
↪'spider'], ['dog', 'cat', 'snake', 'spider']]
```

```
sage: for a, b in Combinations(pets, 2):
....:     print("The {} chases the {}.".format(a, b))
The dog chases the cat.
The dog chases the snake.
The dog chases the spider.
The cat chases the snake.
The cat chases the spider.
The snake chases the spider.
```

```
>>> from sage.all import *
>>> for a, b in Combinations(pets, Integer(2)):
...     print("The {} chases the {}.".format(a, b))
The dog chases the cat.
The dog chases the snake.
The dog chases the spider.
The cat chases the snake.
The cat chases the spider.
The snake chases the spider.
```

```
sage: for pair in Permutations(pets, 2):
....:     print(pair)
['dog', 'cat']
['dog', 'snake']
['dog', 'spider']
['cat', 'dog']
['cat', 'snake']
['cat', 'spider']
['snake', 'dog']
```

(continues on next page)

(continued from previous page)

```
['snake', 'cat']
['snake', 'spider']
['spider', 'dog']
['spider', 'cat']
['spider', 'snake']
```

```
>>> from sage.all import *
>>> for pair in Permutations(pets, Integer(2)):
...     print(pair)
['dog', 'cat']
['dog', 'snake']
['dog', 'spider']
['cat', 'dog']
['cat', 'snake']
['cat', 'spider']
['snake', 'dog']
['snake', 'cat']
['snake', 'spider']
['spider', 'dog']
['spider', 'cat']
['spider', 'snake']
```

Of course, we often want these for numbers, and these are present as well. Some are familiar:

```
sage: Permutations(5).cardinality()
120
```

```
>>> from sage.all import *
>>> Permutations(Integer(5)).cardinality()
120
```

Others somewhat less so:

```
sage: D = Derangements([1,1,2,2,3,4,5])
sage: D.list()[:5]
[[2, 2, 1, 1, 4, 5, 3], [2, 2, 1, 1, 5, 3, 4], [2, 2, 1, 3, 1, 5, 4], [2, 2, 1, 3, 4,␣
↪5, 1], [2, 2, 1, 3, 5, 1, 4]]
```

```
>>> from sage.all import *
>>> D = Derangements([Integer(1),Integer(1),Integer(2),Integer(2),Integer(3),
↪Integer(4),Integer(5)])
>>> D.list()[:Integer(5)]
[[2, 2, 1, 1, 4, 5, 3], [2, 2, 1, 1, 5, 3, 4], [2, 2, 1, 3, 1, 5, 4], [2, 2, 1, 3, 4,␣
↪5, 1], [2, 2, 1, 3, 5, 1, 4]]
```

And some somewhat more advanced – in this case, symmetric polynomials.

```
sage: s = SymmetricFunctions(QQ).schur()
sage: a = s([2,1])
sage: a.expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
```

```
>>> from sage.all import *
>>> s = SymmetricFunctions(QQ).schur()
>>> a = s([Integer(2),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> a.expand(Integer(3))
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
```

Various functions related to this are available as well.

```
sage: binomial(25,3)
2300
```

```
>>> from sage.all import *
>>> binomial(Integer(25),Integer(3))
2300
```

```
sage: multinomial(24,3,5)
589024800
```

```
>>> from sage.all import *
>>> multinomial(Integer(24),Integer(3),Integer(5))
589024800
```

```
sage: falling_factorial(10,4)
5040
```

```
>>> from sage.all import *
>>> falling_factorial(Integer(10),Integer(4))
5040
```

Do you recognize this famous identity?

```
sage: var('k,n')
(k, n)
sage: sum(binomial(n,k),k,0,n)
2^n
```

```
>>> from sage.all import *
>>> var('k,n')
(k, n)
>>> sum(binomial(n,k),k,Integer(0),n)
2^n
```

Cryptography (for education)

This is also briefly mentioned in the *Number theory quickstart*. Sage has a number of good pedagogical resources for cryptography.

```
sage: # Two objects to make/use encryption scheme
sage: #
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: bin = BinaryStrings()
sage: #
sage: # Convert English to binary
sage: #
```

(continues on next page)

(continued from previous page)

```
sage: P = bin.encoding("Encrypt this using S-DES!")
sage: print("Binary plaintext:  {}\n".format(P))
sage: #
sage: # Choose a random key
sage: #
sage: K = sdes.list_to_string(sdes.random_key())
sage: print("Random key:  {}\n".format(K))
sage: #
sage: # Encrypt with Simplified DES
sage: #
sage: C = sdes(P, K, algorithm="encrypt")
sage: print("Encrypted:  {}\n".format(C))
sage: #
sage: # Decrypt for the round-trip
sage: #
sage: plaintxt = sdes(C, K, algorithm="decrypt")
sage: print("Decrypted:  {}\n".format(plaintxt))
sage: #
sage: # Verify easily
sage: #
sage: print("Verify encryption/decryption: {}".format(P == plaintxt))
Binary plaintext: ␣
↪0100010101101110011000110111001001111001011100000111010000100000011101000110100001101001011100110010

Random key:  0100000011

Encrypted: ␣
↪0010000110000101001100011100011001000001101110101111101110001101111110111111011111001010100001001000

Decrypted: ␣
↪0100010101101110011000110111001001111001011100000111010000100000011101000110100001101001011100110010

Verify encryption/decryption:  True
```

```
>>> from sage.all import *
>>> # Two objects to make/use encryption scheme
>>> #
>>> from sage.crypto.block_cipher.sdes import SimplifiedDES
>>> sdes = SimplifiedDES()
>>> bin = BinaryStrings()
>>> #
>>> # Convert English to binary
>>> #
>>> P = bin.encoding("Encrypt this using S-DES!")
>>> print("Binary plaintext:  {}\n".format(P))
>>> #
>>> # Choose a random key
>>> #
>>> K = sdes.list_to_string(sdes.random_key())
>>> print("Random key:  {}\n".format(K))
>>> #
>>> # Encrypt with Simplified DES
>>> #
>>> C = sdes(P, K, algorithm="encrypt")
>>> print("Encrypted:  {}\n".format(C))
>>> #
```

(continues on next page)

(continued from previous page)

```
>>> # Decrypt for the round-trip
>>> #
>>> plaintxt = sdes(C, K, algorithm="decrypt")
>>> print("Decrypted:  {}\n".format(plaintxt))
>>> #
>>> # Verify easily
>>> #
>>> print("Verify encryption/decryption: {}".format(P == plaintxt))
Binary plaintext: ␣
↪0100010101101110011000110111001001111001011100000111010000100000011101000110100001101001011100110011

Random key:  0100000011

Encrypted: ␣
↪0010000110000101001100011100011001000001101110101111101110001101111110111111011111001010100001001000

Decrypted: ␣
↪0100010101101110011000110111001001111001011100000111010000100000011101000110100001101001011100110011

Verify encryption/decryption:  True
```

Coding Theory

Here is a brief example of a linear binary code (group code).

Start with a generator matrix over $\mathbf{Z}/2\mathbf{Z}$.

```
sage: G = matrix(GF(2), [[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,
↪0,0,1]])
sage: C = LinearCode(G)
```

```
>>> from sage.all import *
>>> G = matrix(GF(Integer(2)), [[Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(0),Integer(0)], [Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0)], [Integer(0),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1),Integer(0)], [Integer(1),Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(0),Integer(1)]])
>>> C = LinearCode(G)
```

```
sage: C.is_self_dual()
False
```

```
>>> from sage.all import *
>>> C.is_self_dual()
False
```

```
sage: D = C.dual_code()
sage: D
[7, 3] linear code over GF(2)
```

```
>>> from sage.all import *
>>> D = C.dual_code()
>>> D
[7, 3] linear code over GF(2)
```

```
sage: D.basis()
[
(1, 0, 1, 0, 1, 0, 1),
(0, 1, 1, 0, 0, 1, 1),
(0, 0, 0, 1, 1, 1, 1)
]
```

```
>>> from sage.all import *
>>> D.basis()
[
(1, 0, 1, 0, 1, 0, 1),
(0, 1, 1, 0, 0, 1, 1),
(0, 0, 0, 1, 1, 1, 1)
]
```

```
sage: D.permutation_automorphism_group()
Permutation Group with generators [(4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,
↪2)(5,6)]
```

```
>>> from sage.all import *
>>> D.permutation_automorphism_group()
Permutation Group with generators [(4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,
↪2)(5,6)]
```

7.4 Sage Quickstart for Linear Algebra

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

Linear algebra underpins a lot of Sage's algorithms, so it is fast, robust and comprehensive. We've already seen some basic linear algebra, including matrices, determinants, and the `.rref()` method for row-reduced echelon form in the *Programming Tutorial*, so the content here continues from there to some extent.

7.4.1 Matrices and Vectors

We can make a matrix easily by passing a list of the rows. Don't forget to use tab-completion to see routines that are possible.

```
sage: A = matrix([[1,2,3],[4,5,6]]); A
[1 2 3]
[4 5 6]
```

```
>>> from sage.all import *
>>> A = matrix([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5),
↪Integer(6)]]); A
[1 2 3]
[4 5 6]
```

But there are lots of other ways to make matrices. Each of these shows what is assumed with different input; can you figure out how Sage interprets them before you read the documentation which the command `matrix?` provides?

It's a good idea to get in the habit of telling Sage what ring to make the matrix over. Otherwise, Sage guesses based on the elements, so you may not have a matrix over a field! Here, we tell Sage to make the ring over the rationals.

```
sage: B = matrix(QQ, 3, 2, [1,2,3,4,5,6]); B
[1 2]
[3 4]
[5 6]
```

```
>>> from sage.all import *
>>> B = matrix(QQ, Integer(3), Integer(2), [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(6)]); B
[1 2]
[3 4]
[5 6]
```

```
sage: C = matrix(QQ, 3, [1,2,3,4,5,6]); C
[1 2]
[3 4]
[5 6]
```

```
>>> from sage.all import *
>>> C = matrix(QQ, Integer(3), [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5),Integer(6)]); C
[1 2]
[3 4]
[5 6]
```

```
sage: D = matrix(CC, 20, range(400)); D
20 x 20 dense matrix over Complex Field with 53 bits of precision (use the '.str()'␣
↪method to see the entries)
```

```
>>> from sage.all import *
>>> D = matrix(CC, Integer(20), range(Integer(400))); D
20 x 20 dense matrix over Complex Field with 53 bits of precision (use the '.str()'␣
↪method to see the entries)
```

Don't forget that when viewing this in the notebook, you can click to the left of the matrix in order to cycle between "wrapped", "unwrapped" and "hidden" modes of output.

```
sage: print(D.str())
[0.000000000000000  1.00000000000000  2.00000000000000  3.00000000000000  4.
↪00000000000000  5.00000000000000  6.00000000000000  7.00000000000000  8.
↪00000000000000  9.00000000000000  10.0000000000000  11.0000000000000  12.
↪0000000000000  13.0000000000000  14.0000000000000  15.0000000000000  16.
↪0000000000000  17.0000000000000  18.0000000000000  19.0000000000000]
[ 20.0000000000000  21.0000000000000  22.0000000000000  23.0000000000000  24.
↪0000000000000  25.0000000000000  26.0000000000000  27.0000000000000  28.
↪0000000000000  29.0000000000000  30.0000000000000  31.0000000000000  32.
↪0000000000000  33.0000000000000  34.0000000000000  35.0000000000000  36.
↪0000000000000  37.0000000000000  38.0000000000000  39.0000000000000]
[ 40.0000000000000  41.0000000000000  42.0000000000000  43.0000000000000  44.
↪0000000000000  45.0000000000000  46.0000000000000  47.0000000000000  48.
↪0000000000000  49.0000000000000  50.0000000000000  51.0000000000000  52.
↪0000000000000  53.0000000000000  54.0000000000000  55.0000000000000  56.
↪0000000000000  57.0000000000000  58.0000000000000  59.0000000000000]
[ 60.0000000000000  61.0000000000000  62.0000000000000  63.0000000000000  64.
```

(continues on next page)

(continued from previous page)

```
↪0000000000000  65.0000000000000  66.0000000000000  67.0000000000000  68.
↪0000000000000  69.0000000000000  70.0000000000000  71.0000000000000  72.
↪0000000000000  73.0000000000000  74.0000000000000  75.0000000000000  76.
↪0000000000000  77.0000000000000  78.0000000000000  79.0000000000000]
[ 80.0000000000000  81.0000000000000  82.0000000000000  83.0000000000000  84.
↪0000000000000  85.0000000000000  86.0000000000000  87.0000000000000  88.
↪0000000000000  89.0000000000000  90.0000000000000  91.0000000000000  92.
↪0000000000000  93.0000000000000  94.0000000000000  95.0000000000000  96.
↪0000000000000  97.0000000000000  98.0000000000000  99.0000000000000]
[ 100.000000000000  101.000000000000  102.000000000000  103.000000000000  104.
↪000000000000  105.000000000000  106.000000000000  107.000000000000  108.
↪000000000000  109.000000000000  110.000000000000  111.000000000000  112.
↪000000000000  113.000000000000  114.000000000000  115.000000000000  116.
↪000000000000  117.000000000000  118.000000000000  119.000000000000]
[ 120.000000000000  121.000000000000  122.000000000000  123.000000000000  124.
↪000000000000  125.000000000000  126.000000000000  127.000000000000  128.
↪000000000000  129.000000000000  130.000000000000  131.000000000000  132.
↪000000000000  133.000000000000  134.000000000000  135.000000000000  136.
↪000000000000  137.000000000000  138.000000000000  139.000000000000]
[ 140.000000000000  141.000000000000  142.000000000000  143.000000000000  144.
↪000000000000  145.000000000000  146.000000000000  147.000000000000  148.
↪000000000000  149.000000000000  150.000000000000  151.000000000000  152.
↪000000000000  153.000000000000  154.000000000000  155.000000000000  156.
↪000000000000  157.000000000000  158.000000000000  159.000000000000]
[ 160.000000000000  161.000000000000  162.000000000000  163.000000000000  164.
↪000000000000  165.000000000000  166.000000000000  167.000000000000  168.
↪000000000000  169.000000000000  170.000000000000  171.000000000000  172.
↪000000000000  173.000000000000  174.000000000000  175.000000000000  176.
↪000000000000  177.000000000000  178.000000000000  179.000000000000]
[ 180.000000000000  181.000000000000  182.000000000000  183.000000000000  184.
↪000000000000  185.000000000000  186.000000000000  187.000000000000  188.
↪000000000000  189.000000000000  190.000000000000  191.000000000000  192.
↪000000000000  193.000000000000  194.000000000000  195.000000000000  196.
↪000000000000  197.000000000000  198.000000000000  199.000000000000]
[ 200.000000000000  201.000000000000  202.000000000000  203.000000000000  204.
↪000000000000  205.000000000000  206.000000000000  207.000000000000  208.
↪000000000000  209.000000000000  210.000000000000  211.000000000000  212.
↪000000000000  213.000000000000  214.000000000000  215.000000000000  216.
↪000000000000  217.000000000000  218.000000000000  219.000000000000]
[ 220.000000000000  221.000000000000  222.000000000000  223.000000000000  224.
↪000000000000  225.000000000000  226.000000000000  227.000000000000  228.
↪000000000000  229.000000000000  230.000000000000  231.000000000000  232.
↪000000000000  233.000000000000  234.000000000000  235.000000000000  236.
↪000000000000  237.000000000000  238.000000000000  239.000000000000]
[ 240.000000000000  241.000000000000  242.000000000000  243.000000000000  244.
↪000000000000  245.000000000000  246.000000000000  247.000000000000  248.
↪000000000000  249.000000000000  250.000000000000  251.000000000000  252.
↪000000000000  253.000000000000  254.000000000000  255.000000000000  256.
↪000000000000  257.000000000000  258.000000000000  259.000000000000]
[ 260.000000000000  261.000000000000  262.000000000000  263.000000000000  264.
↪000000000000  265.000000000000  266.000000000000  267.000000000000  268.
↪000000000000  269.000000000000  270.000000000000  271.000000000000  272.
↪000000000000  273.000000000000  274.000000000000  275.000000000000  276.
↪000000000000  277.000000000000  278.000000000000  279.000000000000]
[ 280.000000000000  281.000000000000  282.000000000000  283.000000000000  284.
↪000000000000  285.000000000000  286.000000000000  287.000000000000  288.
↪000000000000  289.000000000000  290.000000000000  291.000000000000  292.
```

(continues on next page)

(continued from previous page)

```
↪000000000000  293.000000000000  294.000000000000  295.000000000000  296.
↪000000000000  297.000000000000  298.000000000000  299.000000000000]
[ 300.000000000000  301.000000000000  302.000000000000  303.000000000000  304.
↪000000000000  305.000000000000  306.000000000000  307.000000000000  308.
↪000000000000  309.000000000000  310.000000000000  311.000000000000  312.
↪000000000000  313.000000000000  314.000000000000  315.000000000000  316.
↪000000000000  317.000000000000  318.000000000000  319.000000000000]
[ 320.000000000000  321.000000000000  322.000000000000  323.000000000000  324.
↪000000000000  325.000000000000  326.000000000000  327.000000000000  328.
↪000000000000  329.000000000000  330.000000000000  331.000000000000  332.
↪000000000000  333.000000000000  334.000000000000  335.000000000000  336.
↪000000000000  337.000000000000  338.000000000000  339.000000000000]
[ 340.000000000000  341.000000000000  342.000000000000  343.000000000000  344.
↪000000000000  345.000000000000  346.000000000000  347.000000000000  348.
↪000000000000  349.000000000000  350.000000000000  351.000000000000  352.
↪000000000000  353.000000000000  354.000000000000  355.000000000000  356.
↪000000000000  357.000000000000  358.000000000000  359.000000000000]
[ 360.000000000000  361.000000000000  362.000000000000  363.000000000000  364.
↪000000000000  365.000000000000  366.000000000000  367.000000000000  368.
↪000000000000  369.000000000000  370.000000000000  371.000000000000  372.
↪000000000000  373.000000000000  374.000000000000  375.000000000000  376.
↪000000000000  377.000000000000  378.000000000000  379.000000000000]
[ 380.000000000000  381.000000000000  382.000000000000  383.000000000000  384.
↪000000000000  385.000000000000  386.000000000000  387.000000000000  388.
↪000000000000  389.000000000000  390.000000000000  391.000000000000  392.
↪000000000000  393.000000000000  394.000000000000  395.000000000000  396.
↪000000000000  397.000000000000  398.000000000000  399.000000000000]
```

```
>>> from sage.all import *
>>> print(D.str())
[0.00000000000000  1.00000000000000  2.00000000000000  3.00000000000000  4.
↪00000000000000  5.00000000000000  6.00000000000000  7.00000000000000  8.
↪00000000000000  9.00000000000000  10.0000000000000  11.0000000000000  12.
↪0000000000000  13.0000000000000  14.0000000000000  15.0000000000000  16.
↪0000000000000  17.0000000000000  18.0000000000000  19.0000000000000]
[ 20.0000000000000  21.0000000000000  22.0000000000000  23.0000000000000  24.
↪0000000000000  25.0000000000000  26.0000000000000  27.0000000000000  28.
↪0000000000000  29.0000000000000  30.0000000000000  31.0000000000000  32.
↪0000000000000  33.0000000000000  34.0000000000000  35.0000000000000  36.
↪0000000000000  37.0000000000000  38.0000000000000  39.0000000000000]
[ 40.0000000000000  41.0000000000000  42.0000000000000  43.0000000000000  44.
↪0000000000000  45.0000000000000  46.0000000000000  47.0000000000000  48.
↪0000000000000  49.0000000000000  50.0000000000000  51.0000000000000  52.
↪0000000000000  53.0000000000000  54.0000000000000  55.0000000000000  56.
↪0000000000000  57.0000000000000  58.0000000000000  59.0000000000000]
[ 60.0000000000000  61.0000000000000  62.0000000000000  63.0000000000000  64.
↪0000000000000  65.0000000000000  66.0000000000000  67.0000000000000  68.
↪0000000000000  69.0000000000000  70.0000000000000  71.0000000000000  72.
↪0000000000000  73.0000000000000  74.0000000000000  75.0000000000000  76.
↪0000000000000  77.0000000000000  78.0000000000000  79.0000000000000]
[ 80.0000000000000  81.0000000000000  82.0000000000000  83.0000000000000  84.
↪0000000000000  85.0000000000000  86.0000000000000  87.0000000000000  88.
↪0000000000000  89.0000000000000  90.0000000000000  91.0000000000000  92.
↪0000000000000  93.0000000000000  94.0000000000000  95.0000000000000  96.
↪0000000000000  97.0000000000000  98.0000000000000  99.0000000000000]
[ 100.000000000000  101.000000000000  102.000000000000  103.000000000000  104.
```

(continues on next page)

(continued from previous page)

```
↪000000000000  105.000000000000  106.000000000000  107.000000000000  108.
↪000000000000  109.000000000000  110.000000000000  111.000000000000  112.
↪000000000000  113.000000000000  114.000000000000  115.000000000000  116.
↪000000000000  117.000000000000  118.000000000000  119.000000000000]
[ 120.000000000000  121.000000000000  122.000000000000  123.000000000000  124.
↪000000000000  125.000000000000  126.000000000000  127.000000000000  128.
↪000000000000  129.000000000000  130.000000000000  131.000000000000  132.
↪000000000000  133.000000000000  134.000000000000  135.000000000000  136.
↪000000000000  137.000000000000  138.000000000000  139.000000000000]
[ 140.000000000000  141.000000000000  142.000000000000  143.000000000000  144.
↪000000000000  145.000000000000  146.000000000000  147.000000000000  148.
↪000000000000  149.000000000000  150.000000000000  151.000000000000  152.
↪000000000000  153.000000000000  154.000000000000  155.000000000000  156.
↪000000000000  157.000000000000  158.000000000000  159.000000000000]
[ 160.000000000000  161.000000000000  162.000000000000  163.000000000000  164.
↪000000000000  165.000000000000  166.000000000000  167.000000000000  168.
↪000000000000  169.000000000000  170.000000000000  171.000000000000  172.
↪000000000000  173.000000000000  174.000000000000  175.000000000000  176.
↪000000000000  177.000000000000  178.000000000000  179.000000000000]
[ 180.000000000000  181.000000000000  182.000000000000  183.000000000000  184.
↪000000000000  185.000000000000  186.000000000000  187.000000000000  188.
↪000000000000  189.000000000000  190.000000000000  191.000000000000  192.
↪000000000000  193.000000000000  194.000000000000  195.000000000000  196.
↪000000000000  197.000000000000  198.000000000000  199.000000000000]
[ 200.000000000000  201.000000000000  202.000000000000  203.000000000000  204.
↪000000000000  205.000000000000  206.000000000000  207.000000000000  208.
↪000000000000  209.000000000000  210.000000000000  211.000000000000  212.
↪000000000000  213.000000000000  214.000000000000  215.000000000000  216.
↪000000000000  217.000000000000  218.000000000000  219.000000000000]
[ 220.000000000000  221.000000000000  222.000000000000  223.000000000000  224.
↪000000000000  225.000000000000  226.000000000000  227.000000000000  228.
↪000000000000  229.000000000000  230.000000000000  231.000000000000  232.
↪000000000000  233.000000000000  234.000000000000  235.000000000000  236.
↪000000000000  237.000000000000  238.000000000000  239.000000000000]
[ 240.000000000000  241.000000000000  242.000000000000  243.000000000000  244.
↪000000000000  245.000000000000  246.000000000000  247.000000000000  248.
↪000000000000  249.000000000000  250.000000000000  251.000000000000  252.
↪000000000000  253.000000000000  254.000000000000  255.000000000000  256.
↪000000000000  257.000000000000  258.000000000000  259.000000000000]
[ 260.000000000000  261.000000000000  262.000000000000  263.000000000000  264.
↪000000000000  265.000000000000  266.000000000000  267.000000000000  268.
↪000000000000  269.000000000000  270.000000000000  271.000000000000  272.
↪000000000000  273.000000000000  274.000000000000  275.000000000000  276.
↪000000000000  277.000000000000  278.000000000000  279.000000000000]
[ 280.000000000000  281.000000000000  282.000000000000  283.000000000000  284.
↪000000000000  285.000000000000  286.000000000000  287.000000000000  288.
↪000000000000  289.000000000000  290.000000000000  291.000000000000  292.
↪000000000000  293.000000000000  294.000000000000  295.000000000000  296.
↪000000000000  297.000000000000  298.000000000000  299.000000000000]
[ 300.000000000000  301.000000000000  302.000000000000  303.000000000000  304.
↪000000000000  305.000000000000  306.000000000000  307.000000000000  308.
↪000000000000  309.000000000000  310.000000000000  311.000000000000  312.
↪000000000000  313.000000000000  314.000000000000  315.000000000000  316.
↪000000000000  317.000000000000  318.000000000000  319.000000000000]
[ 320.000000000000  321.000000000000  322.000000000000  323.000000000000  324.
↪000000000000  325.000000000000  326.000000000000  327.000000000000  328.
↪000000000000  329.000000000000  330.000000000000  331.000000000000  332.
```

(continues on next page)

(continued from previous page)

```
↪000000000000  333.000000000000  334.000000000000  335.000000000000  336.
↪000000000000  337.000000000000  338.000000000000  339.000000000000]
[ 340.000000000000  341.000000000000  342.000000000000  343.000000000000  344.
↪000000000000  345.000000000000  346.000000000000  347.000000000000  348.
↪000000000000  349.000000000000  350.000000000000  351.000000000000  352.
↪000000000000  353.000000000000  354.000000000000  355.000000000000  356.
↪000000000000  357.000000000000  358.000000000000  359.000000000000]
[ 360.000000000000  361.000000000000  362.000000000000  363.000000000000  364.
↪000000000000  365.000000000000  366.000000000000  367.000000000000  368.
↪000000000000  369.000000000000  370.000000000000  371.000000000000  372.
↪000000000000  373.000000000000  374.000000000000  375.000000000000  376.
↪000000000000  377.000000000000  378.000000000000  379.000000000000]
[ 380.000000000000  381.000000000000  382.000000000000  383.000000000000  384.
↪000000000000  385.000000000000  386.000000000000  387.000000000000  388.
↪000000000000  389.000000000000  390.000000000000  391.000000000000  392.
↪000000000000  393.000000000000  394.000000000000  395.000000000000  396.
↪000000000000  397.000000000000  398.000000000000  399.000000000000]
```

```
sage: E = diagonal_matrix( [0..40,step=4] ); E
[ 0  0  0  0  0  0  0  0  0  0  0]
[ 0  4  0  0  0  0  0  0  0  0  0]
[ 0  0  8  0  0  0  0  0  0  0  0]
[ 0  0  0 12  0  0  0  0  0  0  0]
[ 0  0  0  0 16  0  0  0  0  0  0]
[ 0  0  0  0  0 20  0  0  0  0  0]
[ 0  0  0  0  0  0 24  0  0  0  0]
[ 0  0  0  0  0  0  0 28  0  0  0]
[ 0  0  0  0  0  0  0  0 32  0  0]
[ 0  0  0  0  0  0  0  0  0 36  0]
[ 0  0  0  0  0  0  0  0  0  0 40]
```

```
>>> from sage.all import *
>>> E = diagonal_matrix( (ellipsis_range(Integer(0),Ellipsis,Integer(40),
↪step=Integer(4))) ); E
[ 0  0  0  0  0  0  0  0  0  0  0]
[ 0  4  0  0  0  0  0  0  0  0  0]
[ 0  0  8  0  0  0  0  0  0  0  0]
[ 0  0  0 12  0  0  0  0  0  0  0]
[ 0  0  0  0 16  0  0  0  0  0  0]
[ 0  0  0  0  0 20  0  0  0  0  0]
[ 0  0  0  0  0  0 24  0  0  0  0]
[ 0  0  0  0  0  0  0 28  0  0  0]
[ 0  0  0  0  0  0  0  0 32  0  0]
[ 0  0  0  0  0  0  0  0  0 36  0]
[ 0  0  0  0  0  0  0  0  0  0 40]
```

```
sage: column_matrix(QQ,[[1,2,3],[4,5,6],[7,8,9]])
[1 4 7]
[2 5 8]
[3 6 9]
```

```
>>> from sage.all import *
>>> column_matrix(QQ,[[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5),
↪Integer(6)],[Integer(7),Integer(8),Integer(9)]])
[1 4 7]
```

(continues on next page)

(continued from previous page)

```
[2 5 8]
[3 6 9]
```

You can also combine matrices in different ways.

```
sage: F1=matrix(QQ,2,2,[0,1,1,0])
sage: F2=matrix(QQ,2,2,[1,2,3,4])
sage: F3=matrix(QQ,1,2,[3,1])
sage: block_matrix(2,2,[F1,F2,0,F3])
[0 1|1 2]
[1 0|3 4]
[---+---]
[0 0|3 1]
```

```
>>> from sage.all import *
>>> F1=matrix(QQ,Integer(2),Integer(2),[Integer(0),Integer(1),Integer(1),Integer(0)])
>>> F2=matrix(QQ,Integer(2),Integer(2),[Integer(1),Integer(2),Integer(3),Integer(4)])
>>> F3=matrix(QQ,Integer(1),Integer(2),[Integer(3),Integer(1)])
>>> block_matrix(Integer(2),Integer(2),[F1,F2,Integer(0),F3])
[0 1|1 2]
[1 0|3 4]
[---+---]
[0 0|3 1]
```

```
sage: F1.augment(F2)
[0 1 1 2]
[1 0 3 4]
```

```
>>> from sage.all import *
>>> F1.augment(F2)
[0 1 1 2]
[1 0 3 4]
```

```
sage: F1.stack(F2)
[0 1]
[1 0]
[1 2]
[3 4]
```

```
>>> from sage.all import *
>>> F1.stack(F2)
[0 1]
[1 0]
[1 2]
[3 4]
```

```
sage: block_diagonal_matrix([F1,F2])
[0 1|0 0]
[1 0|0 0]
[---+---]
[0 0|1 2]
[0 0|3 4]
```

```
>>> from sage.all import *
>>> block_diagonal_matrix([F1,F2])
[0 1|0 0]
[1 0|0 0]
[---+---]
[0 0|1 2]
[0 0|3 4]
```

Vectors are rows or columns, whatever you please, and Sage interprets them as appropriate in multiplication contexts.

```
sage: row = vector( (3, -1, 4))
sage: col = vector( QQ, [4, 5] )
sage: row; col
(3, -1, 4)
(4, 5)
```

```
>>> from sage.all import *
>>> row = vector( (Integer(3), -Integer(1), Integer(4)))
>>> col = vector( QQ, [Integer(4), Integer(5)] )
>>> row; col
(3, -1, 4)
(4, 5)
```

```
sage: F = matrix(QQ, 3, 2, range(6)); F
[0 1]
[2 3]
[4 5]
```

```
>>> from sage.all import *
>>> F = matrix(QQ, Integer(3), Integer(2), range(Integer(6))); F
[0 1]
[2 3]
[4 5]
```

```
sage: F*col
(5, 23, 41)
```

```
>>> from sage.all import *
>>> F*col
(5, 23, 41)
```

```
sage: row*F
(14, 20)
```

```
>>> from sage.all import *
>>> row*F
(14, 20)
```

Although our "vectors" (especially over rings other than fields) might be considered as elements of an appropriate free module, they basically behave as vectors for our purposes.

```
sage: ring_vec = vector(SR, [2, 12, -4, 9])
sage: field_vec = vector( QQ, (2, 3, 14) )
sage: ring_vec; field_vec
```

(continues on next page)

(continued from previous page)

```
(2, 12, -4, 9)
(2, 3, 14)
```

```
>>> from sage.all import *
>>> ring_vec = vector(SR, [Integer(2), Integer(12), -Integer(4), Integer(9)])
>>> field_vec = vector( QQ, (Integer(2), Integer(3), Integer(14)) )
>>> ring_vec; field_vec
(2, 12, -4, 9)
(2, 3, 14)
```

```
sage: type( ring_vec )
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category.element_class
↪'>
sage: type( field_vec )
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

```
>>> from sage.all import *
>>> type( ring_vec )
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category.element_class
↪'>
>>> type( field_vec )
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

7.4.2 Left-Handed or Right-handed?

Sage "prefers" rows to columns. For example, the `kernel` method for a matrix A computes the left kernel – the vector space of all vectors v for which $v \cdot A = 0$ – and prints out the vectors as the rows of a matrix.

```
sage: G = matrix(QQ, 2, 3, [[1,2,3],[2,4,6]])
sage: G.kernel()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
```

```
>>> from sage.all import *
>>> G = matrix(QQ, Integer(2), Integer(3), [[Integer(1),Integer(2),Integer(3)],
↪[Integer(2),Integer(4),Integer(6)]])
>>> G.kernel()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
```

```
sage: G.left_kernel()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
```

```
>>> from sage.all import *
>>> G.left_kernel()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
```

The `right_kernel` method computes the space of vectors w so that $A \cdot w = 0$, of course.

7.4.3 Vector Spaces

Since Sage knows the kernel is a vector space, you can compute things that make sense for a vector space.

```
sage: V=G.right_kernel()
sage: V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
```

```
>>> from sage.all import *
>>> V=G.right_kernel()
>>> V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
```

```
sage: V.dimension()
2
```

```
>>> from sage.all import *
>>> V.dimension()
2
```

Here we compute the coordinate vector of $(1, 4, -3)$ relative to V:

```
sage: V.coordinate_vector([1,4,-3])
(1, 4)
```

```
>>> from sage.all import *
>>> V.coordinate_vector([Integer(1),Integer(4),-Integer(3)])
(1, 4)
```

Here we get the basis matrix (note that the basis vectors are the *rows* of the matrix):

```
sage: V.basis_matrix()
[   1    0 -1/3]
[   0    1 -2/3]
```

```
>>> from sage.all import *
>>> V.basis_matrix()
[   1    0 -1/3]
[   0    1 -2/3]
```

Or we can get the basis vectors explicitly as a list of vectors:

```
sage: V.basis()
[
(1, 0, -1/3),
(0, 1, -2/3)
]
```

```
>>> from sage.all import *
>>> V.basis()
[
(1, 0, -1/3),
(0, 1, -2/3)
]
```

Note: Kernels are **vector spaces** and bases are "**echelonized**" (canonicalized).

This is why the `ring` for the matrix is important. Compare the kernels above with the kernel using a matrix which is only defined over the integers.

```
sage: G = matrix(ZZ,2, 3, [[1,2,3],[2,4,6]])
sage: G.kernel()
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[ 2 -1]
```

```
>>> from sage.all import *
>>> G = matrix(ZZ,Integer(2), Integer(3), [[Integer(1),Integer(2),Integer(3)],
↪[Integer(2),Integer(4),Integer(6)]])
>>> G.kernel()
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[ 2 -1]
```

7.4.4 Computations

Here are some more computations with matrices and vectors.

As you might expect, random matrices are random.

```
sage: H = random_matrix(QQ, 5, 5, num_bound = 10, den_bound = 4)
sage: H.det() # random
15416
sage: H.eigenvalues() # random
[-10.08361801792048?, -2.682220984496031?, 4.739405672111427?, -1.320116668180795? -␣
↪10.88676412262347?*I, -1.320116668180795? + 10.88676412262347?*I]
```

```
>>> from sage.all import *
>>> H = random_matrix(QQ, Integer(5), Integer(5), num_bound = Integer(10), den_bound␣
↪= Integer(4))
>>> H.det() # random
15416
>>> H.eigenvalues() # random
[-10.08361801792048?, -2.682220984496031?, 4.739405672111427?, -1.320116668180795? -␣
↪10.88676412262347?*I, -1.320116668180795? + 10.88676412262347?*I]
```

According to the *Numerical analysis quickstart*, the question marks indicate that the actual number is inside the interval found by incrementing and decrementing the last digit of the printed number. So 9.1? is a number between 9.0 and 9.2. Sage knows exactly what number this is (since it's a root of a polynomial), but uses interval notation to print an approximation for ease of use.

The `eigenvectors_right` command prints out a list of `(eigenvalue, [list of eigenvectors], algebraic multiplicity)` tuples for each eigenvalue.

```
sage: H.eigenvectors_right() # random
[(-10.083618017920 48?, [(1, -0.3820692683963385?, -0.4659857618614747?, -0.
↪1264082922197715?, -0.3548156445133095?)], 1), (-2.682220984496031?, [(1, -1.
↪855347152382563?, -0.4203899923232704?, 0.004411201577480876?, -0.5050698736445243?
↪)], 1), (4.739405672111427?, [(1, 0.3284800982819703?, 2.059182569319718?, -1.
↪428547399599918?, 0.5455069936349178?)], 1), (-1.320116668180795? - 10.
↪88676412262347?*I, [(1, 0.710831790589076? + 0.2646474741698805?*I, 0.
↪4504038344112447? + 3.145667601780920?*I, 2.763061217778457? + 0.9994136057023008?
↪*I, 3.092272491890536? - 2.105461094305392?*I)], 1), (-1.320116668180795? + 10.
↪88676412262347?*I, [(1, 0.710831790589076? - 0.2646474741698805?*I, 0.
↪4504038344112447? - 3.145667601780920?*I, 2.763061217778457? - 0.9994136057023008?
↪*I, 3.092272491890536? + 2.105461094305392?*I)], 1)]
```

```
>>> from sage.all import *
>>> H.eigenvectors_right() # random
[(-10.083618017920 48?, [(1, -0.3820692683963385?, -0.4659857618614747?, -0.
↪1264082922197715?, -0.3548156445133095?)], 1), (-2.682220984496031?, [(1, -1.
↪855347152382563?, -0.4203899923232704?, 0.004411201577480876?, -0.5050698736445243?
↪)], 1), (4.739405672111427?, [(1, 0.3284800982819703?, 2.059182569319718?, -1.
↪428547399599918?, 0.5455069936349178?)], 1), (-1.320116668180795? - 10.
↪88676412262347?*I, [(1, 0.710831790589076? + 0.2646474741698805?*I, 0.
↪4504038344112447? + 3.145667601780920?*I, 2.763061217778457? + 0.9994136057023008?
↪*I, 3.092272491890536? - 2.105461094305392?*I)], 1), (-1.320116668180795? + 10.
↪88676412262347?*I, [(1, 0.710831790589076? - 0.2646474741698805?*I, 0.
↪4504038344112447? - 3.145667601780920?*I, 2.763061217778457? - 0.9994136057023008?
↪*I, 3.092272491890536? + 2.105461094305392?*I)], 1)]
```

It may be more convenient to use the `eigenmatrix_right` command, which gives a diagonal matrix of eigenvalues and a column matrix of eigenvectors.

```
sage: D,P=H.eigenmatrix_right()
sage: P*D-H*P
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```

```
>>> from sage.all import *
>>> D,P=H.eigenmatrix_right()
>>> P*D-H*P
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
```

7.4.5 Matrix Solving

We can easily solve linear equations using the backslash, like in Matlab.

```
sage: A = random_matrix(QQ, 3, algorithm='unimodular')
sage: v = vector([2,3,1])
sage: A,v  # random
(
[ 0 -1  1]
[-1 -1 -1]
[ 0  2  2], (2, 3, 1)
)
sage: x=A\v; x  # random
(-7/2, -3/4, 5/4)
sage: A*x  # random
(2, 3, 1)
```

```
>>> from sage.all import *
>>> A = random_matrix(QQ, Integer(3), algorithm='unimodular')
>>> v = vector([Integer(2),Integer(3),Integer(1)])
>>> A,v  # random
(
[ 0 -1  1]
[-1 -1 -1]
[ 0  2  2], (2, 3, 1)
)
>>> x=A * BackslashOperator() * v; x  # random
(-7/2, -3/4, 5/4)
>>> A*x  # random
(2, 3, 1)
```

For *lots* more (concise) information, see the Sage Linear Algebra Quick Reference.

7.5 Sage Quickstart for Multivariable Calculus

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

Because much related material was covered in the calculus tutorial, this quickstart is designed to:

- Give concrete examples of some things not covered there, without huge amounts of commentary, and
- Remind the reader of a few of the things in that tutorial.

7.5.1 Vector Calculus

In Sage, vectors are primarily linear algebra objects, but they are slowly becoming simultaneously analytic continuous functions.

Dot Product, Cross Product

```
sage: v = vector(RR, [1.2, 3.5, 4.6])
sage: w = vector(RR, [1.7,-2.3,5.2])
sage: v*w
17.9100000000000
```

```
>>> from sage.all import *
>>> v = vector(RR, [RealNumber('1.2'), RealNumber('3.5'), RealNumber('4.6')])
>>> w = vector(RR, [RealNumber('1.7'),-RealNumber('2.3'),RealNumber('5.2')])
>>> v*w
17.9100000000000
```

```
sage: v.cross_product(w)
(28.7800000000000, 1.58000000000000, -8.71000000000000)
```

```
>>> from sage.all import *
>>> v.cross_product(w)
(28.7800000000000, 1.58000000000000, -8.71000000000000)
```

Lines and Planes

Intersect $x - 2y - 7z = 6$ with $\frac{x-3}{2} = \frac{y+4}{-3} = \frac{z-1}{1}$. One way to plot the equation of a line in three dimensions is with `parametric_plot3d`.

```
sage: # designed with intersection at t = 2, i.e. (7, -10, 3)
sage: var('t, x, y')
(t, x, y)
sage: line = parametric_plot3d([2*t+3, -3*t-4, t+1], (t, 0, 4),color='red')
sage: plane = plot3d((1/5)*(-12+x-2*y), (x, 4, 10), (y, -13,-7), opacity=0.5)
sage: intersect=point3d([7,-10,3],color='black',size=30)
sage: line+plane+intersect
Graphics3d Object
```

```
>>> from sage.all import *
>>> # designed with intersection at t = 2, i.e. (7, -10, 3)
>>> var('t, x, y')
(t, x, y)
>>> line = parametric_plot3d([Integer(2)*t+Integer(3), -Integer(3)*t-Integer(4),␣
↪t+Integer(1)], (t, Integer(0), Integer(4)),color='red')
>>> plane = plot3d((Integer(1)/Integer(5))*(-Integer(12)+x-Integer(2)*y), (x,␣
↪Integer(4), Integer(10)), (y, -Integer(13),-Integer(7)), opacity=RealNumber('0.5'))
>>> intersect=point3d([Integer(7),-Integer(10),Integer(3)],color='black',
↪size=Integer(30))
>>> line+plane+intersect
Graphics3d Object
```

Vector-Valued Functions

We can make vector-valued functions and do the usual analysis with them.

```
sage: var('t')
t
sage: r=vector((2*t-4, t^2, (1/4)*t^3))
sage: r
(2*t - 4, t^2, 1/4*t^3)
```

```
>>> from sage.all import *
>>> var('t')
t
>>> r=vector((Integer(2)*t-Integer(4), t**Integer(2), (Integer(1)/
↪Integer(4))*t**Integer(3)))
>>> r
(2*t - 4, t^2, 1/4*t^3)
```

```
sage: r(t=5)
(6, 25, 125/4)
```

```
>>> from sage.all import *
>>> r(t=Integer(5))
(6, 25, 125/4)
```

The following makes the derivative also a vector-valued expression.

```
sage: velocity = r.diff(t) # velocity(t) = list(r.diff(t)) also would work
sage: velocity
(2, 2*t, 3/4*t^2)
```

```
>>> from sage.all import *
>>> velocity = r.diff(t) # velocity(t) = list(r.diff(t)) also would work
>>> velocity
(2, 2*t, 3/4*t^2)
```

Currently, this expression does *not* function as a function, so we need to substitute explicitly.

```
sage: velocity(t=1)
(2, 2, 3/4)
```

```
>>> from sage.all import *
>>> velocity(t=Integer(1))
(2, 2, 3/4)
```

```
sage: T=velocity/velocity.norm()
```

```
>>> from sage.all import *
>>> T=velocity/velocity.norm()
```

```
sage: T(t=1).n()
(0.683486126173409, 0.683486126173409, 0.256307297315028)
```

```
>>> from sage.all import *
>>> T(t=Integer(1)).n()
(0.683486126173409, 0.683486126173409, 0.256307297315028)
```

Here we compute the arclength between $t = 0$ and $t = 1$ by integrating the normalized derivative. As pointed out in the *calculus tutorial*, the syntax for `numerical_integral` is slightly nonstandard – we just put in the endpoints, not the variable.

```
sage: arc_length = numerical_integral(velocity.norm(), 0,1)
sage: arc_length
(2.3169847271197814, 2.572369791753217e-14)
```

```
>>> from sage.all import *
>>> arc_length = numerical_integral(velocity.norm(), Integer(0),Integer(1))
>>> arc_length
(2.3169847271197814, 2.572369791753217e-14)
```

We can also plot vector fields, even in three dimensions.

```
sage: x,y,z=var('x y z')
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi),
↪colors=['red','green','blue'])
Graphics3d Object
```

```
>>> from sage.all import *
>>> x,y,z=var('x y z')
>>> plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,Integer(0),pi), (y,Integer(0),
↪pi), (z,Integer(0),pi),colors=['red','green','blue'])
Graphics3d Object
```

If we know a little vector calculus, we can also do line integrals. Here, based on an example by Ben Woodruff of BYU-Idaho, we compute a number of quantities in a physical three-dimensional setting.

Try to read through the entire code. We make an auxiliary function that will calculate $\int(\text{integrand})\,dt$. We'll use our formulas relating various differentials to dt to easily specify the integrands.

```
sage: density(x,y,z)=x^2+z
sage: r=vector([3*cos(t), 3*sin(t),4*t])
sage: tstart=0
sage: tend=2*pi
sage: t = var('t')
sage: t_range=(t,tstart,tend)
sage: def line_integral(integrand):
....:     return RR(numerical_integral((integrand).subs(x=r[0], y=r[1],z=r[2]),␣
↪tstart, tend)[0])
sage: _ = var('x,y,z,t')
sage: r_prime = diff(r,t)
sage: ds=diff(r,t).norm()
sage: s=line_integral(ds)
sage: centroid_x=line_integral(x*ds)/s
sage: centroid_y=line_integral(y*ds)/s
sage: centroid_z=line_integral(z*ds)/s
sage: dm=density(x,y,z)*ds
sage: m=line_integral(dm)
sage: avg_density = m/s
sage: moment_about_yz_plane=line_integral(x*dm)
```

(continues on next page)

(continued from previous page)

```
sage: moment_about_xz_plane=line_integral(y*dm)
sage: moment_about_xy_plane=line_integral(z*dm)
sage: center_mass_x = moment_about_yz_plane/m
sage: center_mass_y = moment_about_xz_plane/m
sage: center_mass_z = moment_about_xy_plane/m
sage: Ix=line_integral((y^2+z^2)*dm)
sage: Iy=line_integral((x^2+z^2)*dm)
sage: Iz=line_integral((x^2+y^2)*dm)
sage: Rx = sqrt(Ix/m)
sage: Ry = sqrt(Iy/m)
sage: Rz = sqrt(Iz/m)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,z"); density = symbolic_expression(x**Integer(2)+z).function(x,y,
↪z)
>>> r=vector([Integer(3)*cos(t), Integer(3)*sin(t),Integer(4)*t])
>>> tstart=Integer(0)
>>> tend=Integer(2)*pi
>>> t = var('t')
>>> t_range=(t,tstart,tend)
>>> def line_integral(integrand):
...     return RR(numerical_integral((integrand).subs(x=r[Integer(0)],␣
↪y=r[Integer(1)],z=r[Integer(2)]), tstart, tend)[Integer(0)])
>>> _ = var('x,y,z,t')
>>> r_prime = diff(r,t)
>>> ds=diff(r,t).norm()
>>> s=line_integral(ds)
>>> centroid_x=line_integral(x*ds)/s
>>> centroid_y=line_integral(y*ds)/s
>>> centroid_z=line_integral(z*ds)/s
>>> dm=density(x,y,z)*ds
>>> m=line_integral(dm)
>>> avg_density = m/s
>>> moment_about_yz_plane=line_integral(x*dm)
>>> moment_about_xz_plane=line_integral(y*dm)
>>> moment_about_xy_plane=line_integral(z*dm)
>>> center_mass_x = moment_about_yz_plane/m
>>> center_mass_y = moment_about_xz_plane/m
>>> center_mass_z = moment_about_xy_plane/m
>>> Ix=line_integral((y**Integer(2)+z**Integer(2))*dm)
>>> Iy=line_integral((x**Integer(2)+z**Integer(2))*dm)
>>> Iz=line_integral((x**Integer(2)+y**Integer(2))*dm)
>>> Rx = sqrt(Ix/m)
>>> Ry = sqrt(Iy/m)
>>> Rz = sqrt(Iz/m)
```

Finally, we can display everything in a nice *table*. Recall that we use the `r"stuff"` syntax to indicate "raw" strings so that backslashes from LaTeX won't cause trouble.

```
sage: html.table([
....:     [r"Density $\delta(x,y)$", density],
....:     [r"Curve $\vec r(t)$",r],
....:     [r"$t$ range", t_range],
....:     [r"$\vec r'(t)$", r_prime],
....:     [r"$ds$, a little bit of arclength", ds],
....:     [r"$s$ - arclength", s],
```

(continues on next page)

(continued from previous page)

```
....:     [r"Centroid (constant density) $\left(\frac{1}{m}\int x\,ds,\frac{1}{m}\int␣
↪y\,ds, \frac{1}{m}\int z\,ds\right)$", (centroid_x,centroid_y,centroid_z)],
....:     [r"$dm=\delta ds$ - a little bit of mass", dm],
....:     [r"$m=\int \delta ds$ - mass", m],
....:     [r"average density $\frac{1}{m}\int ds$" , avg_density.n()],
....:     [r"$M_{yz}=\int x dm$ - moment about $yz$ plane", moment_about_yz_plane],
....:     [r"$M_{xz}=\int y dm$ - moment about $xz$ plane", moment_about_xz_plane],
....:     [r"$M_{xy}=\int z dm$ - moment about $xy$ plane", moment_about_xy_plane],
....:     [r"Center of mass $\left(\frac1m \int xdm, \frac1m \int ydm, \frac1m \int z␣
↪dm\right)$", (center_mass_x, center_mass_y, center_mass_z)],
....:     [r"$I_x = \int (y^2+z^2) dm$", Ix],[r"$I_y=\int (x^2+z^2) dm$", Iy],[mp(r"
↪$I_z=\int (x^2+y^2)dm$"), Iz],
....:     [r"$R_x=\sqrt{I_x/m}$", Rx],[mp(r"$R_y=\sqrt{I_y/m}"), Ry],[mp(r"$R_z=\sqrt
↪{I_z/m}"),Rz]
....:     ])
```

```
>>> from sage.all import *
>>> html.table([
...     [r"Density $\delta(x,y)$", density],
...     [r"Curve $\vec r(t)$",r],
...     [r"$t$ range", t_range],
...     [r"$\vec r'(t)$", r_prime],
...     [r"$ds$, a little bit of arclength", ds],
...     [r"$s$ - arclength", s],
...     [r"Centroid (constant density) $\left(\frac{1}{m}\int x\,ds,\frac{1}{m}\int y\
↪,ds, \frac{1}{m}\int z\,ds\right)$", (centroid_x,centroid_y,centroid_z)],
...     [r"$dm=\delta ds$ - a little bit of mass", dm],
...     [r"$m=\int \delta ds$ - mass", m],
...     [r"average density $\frac{1}{m}\int ds$" , avg_density.n()],
...     [r"$M_{yz}=\int x dm$ - moment about $yz$ plane", moment_about_yz_plane],
...     [r"$M_{xz}=\int y dm$ - moment about $xz$ plane", moment_about_xz_plane],
...     [r"$M_{xy}=\int z dm$ - moment about $xy$ plane", moment_about_xy_plane],
...     [r"Center of mass $\left(\frac1m \int xdm, \frac1m \int ydm, \frac1m \int z␣
↪dm\right)$", (center_mass_x, center_mass_y, center_mass_z)],
...     [r"$I_x = \int (y^2+z^2) dm$", Ix],[r"$I_y=\int (x^2+z^2) dm$", Iy],[mp(r"$I_
↪z=\int (x^2+y^2)dm$"), Iz],
...     [r"$R_x=\sqrt{I_x/m}$", Rx],[mp(r"$R_y=\sqrt{I_y/m}"), Ry],[mp(r"$R_z=\sqrt{I_
↪z/m}"),Rz]
...     ])
```

7.5.2 Functions of Several Variables

This connects directly to other issues of multivariable functions.

How to view these was mostly addressed in the various plotting tutorials. Here is a reminder of what can be done.

```
sage: # import matplotlib.cm; matplotlib.cm.datad.keys()
sage: # 'Spectral', 'summer', 'blues'
sage: g(x,y)=e^-x*sin(y)
sage: contour_plot(g, (x, -2, 2), (y, -4*pi, 4*pi), cmap = 'Blues', contours=10,␣
↪colorbar=True)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> # import matplotlib.cm; matplotlib.cm.datad.keys()
```

(continues on next page)

(continued from previous page)

```
>>> # 'Spectral', 'summer', 'blues'
>>> __tmp__=var("x,y"); g = symbolic_expression(e**-x*sin(y)).function(x,y)
>>> contour_plot(g, (x, -Integer(2), Integer(2)), (y, -Integer(4)*pi, Integer(4)*pi),␣
↪cmap = 'Blues', contours=Integer(10), colorbar=True)
Graphics object consisting of 1 graphics primitive
```

Partial Differentiation

The following exercise is from Hass, Weir, and Thomas, University Calculus, Exercise 12.7.35. This function has a local minimum at $(4, -2)$.

```
sage: f(x, y) = x^2 + x*y + y^2 - 6*x + 2
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x**Integer(2) + x*y + y**Integer(2) -␣
↪Integer(6)*x + Integer(2)).function(x,y)
```

Quiz: Why did we *not* need to declare the variables in this case?

```
sage: fx(x,y)= f.diff(x)
sage: fy(x,y) = f.diff(y)
sage: fx; fy
(x, y) |--> 2*x + y - 6
(x, y) |--> x + 2*y
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); fx = symbolic_expression(f.diff(x)).function(x,y)
>>> __tmp__=var("x,y"); fy = symbolic_expression(f.diff(y)).function(x,y)
>>> fx; fy
(x, y) |--> 2*x + y - 6
(x, y) |--> x + 2*y
```

```
sage: f.gradient()
(x, y) |--> (2*x + y - 6, x + 2*y)
```

```
>>> from sage.all import *
>>> f.gradient()
(x, y) |--> (2*x + y - 6, x + 2*y)
```

```
sage: solve([fx==0, fy==0], (x, y))
[[x == 4, y == -2]]
```

```
>>> from sage.all import *
>>> solve([fx==Integer(0), fy==Integer(0)], (x, y))
[[x == 4, y == -2]]
```

```
sage: H = f.hessian()
sage: H(x,y)
[2 1]
[1 2]
```

```
>>> from sage.all import *
>>> H = f.hessian()
>>> H(x,y)
[2 1]
[1 2]
```

And of course if the Hessian has positive determinant and f_{xx} is positive, we have a local minimum.

```
sage: html("$f_{xx}=%s$"%H(4,-2)[0,0])
sage: html("$D=%s$"%H(4,-2).det())
```

```
>>> from sage.all import *
>>> html("$f_{xx}=%s$"%H(Integer(4),-Integer(2))[Integer(0),Integer(0)])
>>> html("$D=%s$"%H(Integer(4),-Integer(2)).det())
```

Notice how we were able to use many things we've done up to now to solve this.

- Matrices
- Symbolic functions
- Solving
- Differential calculus
- Special formatting commands
- And, below, plotting!

```
sage: plot3d(f,(x,-5,5),(y,-5,5))+point((4,-2,f(4,-2)),color='red',size=20)
Graphics3d Object
```

```
>>> from sage.all import *
>>> plot3d(f,(x,-Integer(5),Integer(5)),(y,-Integer(5),Integer(5)))+point((Integer(4),
↪-Integer(2),f(Integer(4),-Integer(2))),color='red',size=Integer(20))
Graphics3d Object
```

Multiple Integrals and More

Naturally, there is lots more that one can do.

```
sage: f(x,y)=x^2*y
sage: # integrate in the order dy dx
sage: f(x,y).integrate(y,0,4*x).integrate(x,0,3)
1944/5
sage: # another way to integrate, and in the opposite order too
sage: integrate( integrate(f(x,y), (x, y/4, 3)), (y, 0, 12) )
1944/5
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x**Integer(2)*y).function(x,y)
>>> # integrate in the order dy dx
>>> f(x,y).integrate(y,Integer(0),Integer(4)*x).integrate(x,Integer(0),Integer(3))
1944/5
>>> # another way to integrate, and in the opposite order too
>>> integrate( integrate(f(x,y), (x, y/Integer(4), Integer(3))), (y, Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(12)) )
1944/5
```

```
sage: var('u v')
(u, v)
sage: surface = plot3d(f(x,y), (x, 0, 3.2), (y, 0, 12.3), color = 'blue', opacity=0.3)
sage: domain = parametric_plot3d([3*u, 4*(3*u)*v,0], (u, 0, 1), (v, 0,1), color =
↪'green', opacity = 0.75)
sage: image = parametric_plot3d([3*u, 4*(3*u)*v, f(3*u, 12*u*v)], (u, 0, 1), (v, 0,1),
↪ color = 'green', opacity = 1.00)
sage: surface+domain+image
Graphics3d Object
```

```
>>> from sage.all import *
>>> var('u v')
(u, v)
>>> surface = plot3d(f(x,y), (x, Integer(0), RealNumber('3.2')), (y, Integer(0),␣
↪RealNumber('12.3')), color = 'blue', opacity=RealNumber('0.3'))
>>> domain = parametric_plot3d([Integer(3)*u, Integer(4)*(Integer(3)*u)*v,Integer(0)],
↪ (u, Integer(0), Integer(1)), (v, Integer(0),Integer(1)), color = 'green', opacity␣
↪= RealNumber('0.75'))
>>> image = parametric_plot3d([Integer(3)*u, Integer(4)*(Integer(3)*u)*v,␣
↪f(Integer(3)*u, Integer(12)*u*v)], (u, Integer(0), Integer(1)), (v, Integer(0),
↪Integer(1)), color = 'green', opacity = RealNumber('1.00'))
>>> surface+domain+image
Graphics3d Object
```

Quiz: why did we need to declare variables this time?

7.6 Sage Quickstart for Numerical Analysis

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

Sage includes many tools for numerical analysis investigations.

The place to begin is the default decimal number types in Sage.

7.6.1 Basic Analysis

- The `RealField` class using arbitrary precision (implemented with MPFR).
- The default real numbers (`RR`) is `RealField(53)` (i.e., 53 bits of precision).
- But you can make them live at whatever precision you wish.

```
sage: ring=RealField(3)
```

```
>>> from sage.all import *
>>> ring=RealField(Integer(3))
```

To print the actual number (without rounding off the last few imprecise digits to only display correct digits), call the `.str()` method:

```
sage: print(ring('1').nextabove())
1.2
```

```
>>> from sage.all import *
>>> print(ring('1').nextabove())
1.2
```

```
sage: print(ring('1').nextabove().str())
1.2
sage: print(ring('1').nextbelow().str())
0.88
```

```
>>> from sage.all import *
>>> print(ring('1').nextabove().str())
1.2
>>> print(ring('1').nextbelow().str())
0.88
```

Let's change our precision.

```
sage: ring=RealField(20)
sage: print(ring('1').nextabove().str())
1.0000019
sage: print(ring('1').nextbelow().str())
0.99999905
```

```
>>> from sage.all import *
>>> ring=RealField(Integer(20))
>>> print(ring('1').nextabove().str())
1.0000019
>>> print(ring('1').nextbelow().str())
0.99999905
```

You can also specify the rounding mode.

```
sage: ringup=RealField(3,rnd='RNDU')
sage: ringdown=RealField(3,rnd='RNDD')
```

```
>>> from sage.all import *
>>> ringup=RealField(Integer(3),rnd='RNDU')
>>> ringdown=RealField(Integer(3),rnd='RNDD')
```

```
sage: ring(1/9).str()
'0.11111116'
```

```
>>> from sage.all import *
>>> ring(Integer(1)/Integer(9)).str()
'0.11111116'
```

```
sage: ringup(1/9).str()
'0.13'
```

```
>>> from sage.all import *
>>> ringup(Integer(1)/Integer(9)).str()
'0.13'
```

```
sage: ringdown(1/9).str()
'0.10'
```

```
>>> from sage.all import *
>>> ringdown(Integer(1)/Integer(9)).str()
'0.10'
```

```
sage: ring(1/9).str(base=2)
'0.00011100011100011100100'
```

```
>>> from sage.all import *
>>> ring(Integer(1)/Integer(9)).str(base=Integer(2))
'0.00011100011100011100100'
```

Let's see exactly what the fraction is.

```
sage: ring(1/9).exact_rational()
233017/2097152
```

```
>>> from sage.all import *
>>> ring(Integer(1)/Integer(9)).exact_rational()
233017/2097152
```

```
sage: n(233017/2097152)
0.111111164093018
```

```
>>> from sage.all import *
>>> n(Integer(233017)/Integer(2097152))
0.111111164093018
```

7.6.2 Converting to floating point binary (IEEE format)

```
sage: x=13.28125
```

```
>>> from sage.all import *
>>> x=RealNumber('13.28125')
```

Here is `x` as a binary floating point number.

```
sage: x.str(base=2)
'1101.01001000000000000000000000000000000000000000000000'
```

```
>>> from sage.all import *
>>> x.str(base=Integer(2))
'1101.01001000000000000000000000000000000000000000000000'
```

From this binary floating point representation, we can construct `x` again.

```
sage: xbin=2^3 +2^2 + 2^0+2^-2+2^-5; xbin
425/32
```

```
>>> from sage.all import *
>>> xbin=Integer(2)**Integer(3) +Integer(2)**Integer(2) +␣
↪Integer(2)**Integer(0)+Integer(2)**-Integer(2)+Integer(2)**-Integer(5); xbin
425/32
```

```
sage: xbin.n()
13.2812500000000
```

```
>>> from sage.all import *
>>> xbin.n()
13.2812500000000
```

To check, let's ask Sage what `x` is as an exact fraction (no rounding involved).

```
sage: x.exact_rational()
425/32
```

```
>>> from sage.all import *
>>> x.exact_rational()
425/32
```

Now let's convert `x` to the IEEE 754 binary format that is commonly used in computers. For IEEE 754, the first step in getting the binary format is to normalize the number, or express the number as a number between 1 and 2 multiplied by a power of 2. For our `x` above, we multiply by 2^{-3} to get a number between 1 and 2.

```
sage: (x*2^(-3)).str(base=2)
'1.1010100100000000000000000000000000000000000000000000'
```

```
>>> from sage.all import *
>>> (x*Integer(2)**(-Integer(3))).str(base=Integer(2))
'1.1010100100000000000000000000000000000000000000000000'
```

The fraction part of the normalized number is called the *mantissa* (or *significand*).

```
sage: f=(x*2^(-3)).frac() # .frac() gets the fraction part, the part after the decimal
sage: mantissa=f.str(base=2)
sage: mantissa
'0.1010100100000000000000000000000000000000000000000000'
```

```
>>> from sage.all import *
>>> f=(x*Integer(2)**(-Integer(3))).frac() # .frac() gets the fraction part, the part␣
↪after the decimal
>>> mantissa=f.str(base=Integer(2))
>>> mantissa
'0.1010100100000000000000000000000000000000000000000000'
```

Since we multiplied by 2^{-3} to normalize the number, we need to store this information. We add 1023 in order to not have to worry about storing negative numbers. In the below, `c` will be our exponent.

```
sage: c=(3+1023).str(base=2)
sage: c
'10000000010'
```

```
>>> from sage.all import *
>>> c=(Integer(3)+Integer(1023)).str(base=Integer(2))
>>> c
'10000000010'
```

Note that `c` has 11 bits, which is exactly what we want.

```
sage: len(c)
11
```

```
>>> from sage.all import *
>>> len(c)
11
```

Evaluating `mantissa[2:54]` will give the first 52 binary digits after the decimal point of the mantissa. Note that we don't need to store the leading 1 before the decimal point because it will always be there from the way we normalized things. This lets us get 53-bit precision using only 52 bits of storage.

```
sage: len(mantissa[2:54])
52
```

```
>>> from sage.all import *
>>> len(mantissa[Integer(2):Integer(54)])
52
```

Since the original number was positive, our sign bit is zero.

```
sage: sign='0'
```

```
>>> from sage.all import *
>>> sign='0'
```

So here is our 64-bit double-precision floating point number.

```
sage: sign+' '+c+' '+mantissa[2:54] # the [2:] just chops off the '0.', since we just␣
↪need to store the digits after the decimal point
'0 10000000010 1010100100000000000000000000000000000000000000000000'
```

```
>>> from sage.all import *
>>> sign+' '+c+' '+mantissa[Integer(2):Integer(54)] # the [2:] just chops off the '0.
↪', since we just need to store the digits after the decimal point
'0 10000000010 1010100100000000000000000000000000000000000000000000'
```

```
sage: len(sign+c+mantissa[2:54]) # it's 64 bits!
64
```

```
>>> from sage.all import *
>>> len(sign+c+mantissa[Integer(2):Integer(54)]) # it's 64 bits!
64
```

Here we convert back to our original number from the floating point representation that we constructed.

```
sage: ((-1)^(int(sign)) * 2^(int(c,base=2)-1023)*(1+RR(mantissa[:54], base=2)))
13.2812500000000
```

```
>>> from sage.all import *
>>> ((-Integer(1))**(int(sign)) * Integer(2)**(int(c,base=Integer(2))-
↪Integer(1023))*(Integer(1)+RR(mantissa[:Integer(54)], base=Integer(2))))
13.2812500000000
```

```
sage: x
13.2812500000000
```

```
>>> from sage.all import *
>>> x
13.2812500000000
```

So they agree!

Sage uses a cutting-edge numerical library, MPFR, to carry out precise floating point arithmetic using any precision a user specifies. MPFR has a slightly different convention for normalization. In MPFR, we normalize by multiplying by an appropriate power of 2 to make the mantissa an integer, instead of a binary fraction. This allows us to use big integer libraries and sophisticated techniques to carry out calculations at an arbitrary precision.

```
sage: x.sign_mantissa_exponent()
(1, 7476679068876800, -49)
```

```
>>> from sage.all import *
>>> x.sign_mantissa_exponent()
(1, 7476679068876800, -49)
```

```
sage: 7476679068876800*2^(-49)
425/32
```

```
>>> from sage.all import *
>>> Integer(7476679068876800)*Integer(2)**(-Integer(49))
425/32
```

Note that the mantissa here has the same zero/nonzero bits as the mantissa above (before we chopped off the leading 1 above).

```
sage: 7476679068876800.str(base=2)
'11010100100000000000000000000000000000000000000000000'
```

```
>>> from sage.all import *
>>> Integer(7476679068876800).str(base=Integer(2))
'11010100100000000000000000000000000000000000000000000'
```

7.6.3 Interval Arithmetic

Sage also lets you compute using intervals to keep track of error bounds. These basically use the round up and round down features shown above.

```
sage: ring=RealIntervalField(10)
sage: a=ring(1/9)
sage: a
0.112?
```

```
>>> from sage.all import *
>>> ring=RealIntervalField(Integer(10))
>>> a=ring(Integer(1)/Integer(9))
>>> a
0.112?
```

The question mark notation means that the number is contained in the interval found by incrementing and decrementing the last digit of the number. See the documentation for real interval fields for details. In the above case, Sage is saying that 1/9 is somewhere between 0.111 and 0.113. Below, we see that `1/a` is somewhere between 8.9 and 9.1.

```
sage: 1/a
9.0?
```

```
>>> from sage.all import *
>>> Integer(1)/a
9.0?
```

We can get a more precise estimate of the interval if we explicitly print out the interval.

```
sage: print((1/a).str(style='brackets'))
[8.9843 .. 9.0157]
```

```
>>> from sage.all import *
>>> print((Integer(1)/a).str(style='brackets'))
[8.9843 .. 9.0157]
```

7.6.4 Included Software

Scipy (included in Sage) has a lot of numerical algorithms. See the Scipy docs.

Mpmath is also included in Sage, and contains a huge amount of numerical stuff. See the mpmath codebase.

The Decimal python module has also been useful for textbook exercises which involved rounding in base 10.

7.6.5 Plotting with precision

Sometimes plotting involves some rather bad rounding errors because plotting calculations are done with machine-precision floating point numbers.

```
sage: f(x)=x^2*(sqrt(x^4+16)-x^2)
sage: plot(f,(x,0,2e4))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_
↪expression(x**Integer(2)*(sqrt(x**Integer(4)+Integer(16))-x**Integer(2))).
↪function(x)
>>> plot(f,(x,Integer(0),RealNumber('2e4')))
Graphics object consisting of 1 graphics primitive
```

We can instead make a function that specifically evaluates all intermediate steps to 100 bits of precision using the `fast_callable` system.

```
sage: R=RealField(100) # 100 bits
sage: g=fast_callable(f, vars=[x], domain=R)
sage: plot(g,(x,0,2e4))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> R=RealField(Integer(100)) # 100 bits
>>> g=fast_callable(f, vars=[x], domain=R)
>>> plot(g,(x,Integer(0),RealNumber('2e4')))
Graphics object consisting of 1 graphics primitive
```

7.7 Sage Quickstart for Number Theory

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

Since Sage began life as a project in algebraic and analytic number theory (and this continues to be a big emphasis), it is no surprise that functionality in this area is extremely comprehensive. And it's also just enjoyable to explore elementary number theory by computer.

7.7.1 Modular Arithmetic

Conveniently, the ring of integers modulo n is always available in Sage, so we can do modular arithmetic very easily. For instance, we can create a number in $\mathbf{Z}/11\mathbf{Z}$. The `type` command tells us that a is not a regular integer.

```
sage: a = mod(2,11); a; type(a); a^10; a^1000000
2
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
1
1
```

```
>>> from sage.all import *
>>> a = mod(Integer(2),Integer(11)); a; type(a); a**Integer(10); a**Integer(1000000)
2
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
1
1
```

Note that we verified Fermat's "Little Theorem" in the previous cell, for the prime $p = 11$ and input $a = 2$.

Recalling the basic programming construct called a *loop* , we can verify this for all a in the integers modulo p. Here, instead of looping over an explicit list like `[0,1,2,3,...8,9,10]`, we loop over a Sage object.

```
sage: for a in Integers(11):
....:     print("{} {}".format(a, a^10))
0 0
1 1
2 1
3 1
4 1
5 1
6 1
```

(continues on next page)

(continued from previous page)

```
7 1
8 1
9 1
10 1
```

```
>>> from sage.all import *
>>> for a in Integers(Integer(11)):
...     print("{} {}".format(a, a**Integer(10)))
0 0
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
10 1
```

Notice that `Integers(11)` gave us an algebraic object which is the ring of integers modulo the prime ideal generated by the element 11.

This works for much bigger numbers too, of course.

```
sage: p=random_prime(10^200,proof=True)
sage: Zp=Integers(p) # Here we give ourselves shorthand for the modular integers
sage: a=Zp(2) # Here we ask for 2 as an element of that ring
sage: p; a; a^(p-1); a^(10^400)
831278801269581161830787498356477572398422896086291267189683307848976468816896107055336280959388241 1
2
1
214280629403801560975383867229133905594454202649654096562327146641561361449201738189364154278449537 58
```

```
>>> from sage.all import *
>>> p=random_prime(Integer(10)**Integer(200),proof=True)
>>> Zp=Integers(p) # Here we give ourselves shorthand for the modular integers
>>> a=Zp(Integer(2)) # Here we ask for 2 as an element of that ring
>>> p; a; a**(p-Integer(1)); a**(Integer(10)**Integer(400))
831278801269581161830787498356477572398422896086291267189683307848976468816896107055336280959388241 1
2
1
214280629403801560975383867229133905594454202649654096562327146641561361449201738189364154278449537 58
```

Whenever you encounter a new object, you should try tab-completion to see what you can do to it. Try it here, and pick one (hopefully one that won't be too long!).

```
sage: Zp.
```

```
>>> from sage.all import *
>>> Zp.
```

Here's one that sounds interesting.

```
sage: Zp.zeta?
```

```
>>> from sage.all import *
>>> Zp.zeta?
```

And we use it to find the fifth roots of unity in this field. We use the *list comprehension* (set-builder) notation from the programming tutorial this time.

```
sage: root_list = Zp.zeta(5,all=True); root_list
[801997703885633241003345486263452400812942732891098664369965252532826865292250889294606853864153831
↪␣
↪698397838955722862975688344850250735578853643480710617154654770618734003597949893674234076839712993
↪␣
↪574074442191990614988012986723235901632385922015745724828366190256768695370076093153868008522043375
↪␣
↪419366418775396766525315677232493679171086389871318795216062437418144020953437245408446072129992970
```

```
>>> from sage.all import *
>>> root_list = Zp.zeta(Integer(5),all=True); root_list
[801997703885633241003345486263452400812942732891098664369965252532826865292250889294606853864153831
↪␣
↪698397838955722862975688344850250735578853643480710617154654770618734003597949893674234076839712993
↪␣
↪574074442191990614988012986723235901632385922015745724828366190256768695370076093153868008522043375
↪␣
↪419366418775396766525315677232493679171086389871318795216062437418144020953437245408446072129992970
```

Are these really fifth roots of unity?

```
sage: [root^5 for root in root_list]
[1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> [root**Integer(5) for root in root_list]
[1, 1, 1, 1]
```

Luckily, it checked out. (If you didn't get any, then your random prime ended up being one for which there *are* no fifth roots of unity – try doing the sequence over again!)

7.7.2 More basic functionality

Similarly to the situation in linear algebra, there is much more we can access at the elementary level, such as

- primitive roots,
- ways to write a number as a sum of squares,
- Legendre symbols,
- modular solving of basic equations,
- etc.

A good way to use Sage in this context is to allow students to experiment with pencil and paper first, then use Sage to see whether patterns they discover hold true before attempting to prove them.

```
sage: p = 13
sage: primitive_root(p); two_squares(p); is_prime(p)
2
```

(continues on next page)

(continued from previous page)

```
(2, 3)
True
```

```
>>> from sage.all import *
>>> p = Integer(13)
>>> primitive_root(p); two_squares(p); is_prime(p)
2
(2, 3)
True
```

This makes it easy to construct elementary cryptographic examples as well. Here is a standard example of a Diffie-Hellman key exchange, for instance. If we didn't do the second line, exponentiation would be impractical.

```
sage: p=random_prime(10^20,10^30) # a random prime between these numbers
sage: q=mod(primitive_root(p),p) # makes the primitive root a number modulo p, not an␣
↪integer
sage: n=randint(1,p) # Alice's random number
sage: m=randint(1,p) # Bob's random number
sage: x=q^n; y=q^m
sage: x; y; x^m; y^n
66786436189350477660
77232558812003408270
45432410008036883324
45432410008036883324
```

```
>>> from sage.all import *
>>> p=random_prime(Integer(10)**Integer(20),Integer(10)**Integer(30)) # a random␣
↪prime between these numbers
>>> q=mod(primitive_root(p),p) # makes the primitive root a number modulo p, not an␣
↪integer
>>> n=randint(Integer(1),p) # Alice's random number
>>> m=randint(Integer(1),p) # Bob's random number
>>> x=q**n; y=q**m
>>> x; y; x**m; y**n
66786436189350477660
77232558812003408270
45432410008036883324
45432410008036883324
```

The final line of the cell first requests Alice and Bob's (possibly) public information, and then verifies that the private keys they get are the same.

It is hard to resist including just one interact. How many theorems do you see here?

```
sage: @interact
sage: def power_table_plot(p=(7,prime_range(50))):
....:     P=matrix_plot(matrix(p-1,[mod(a,p)^b for a in range(1,p) for b in␣
↪srange(p)]),cmap='jet')
....:     show(P)
```

```
>>> from sage.all import *
>>> @interact
>>> def power_table_plot(p=(Integer(7),prime_range(Integer(50)))):
...     P=matrix_plot(matrix(p-Integer(1),[mod(a,p)**b for a in range(Integer(1),p)␣
↪for b in srange(p)]),cmap='jet')
...     show(P)
```

This is a graphic giving the various powers of integers modulo p as colors, not numbers. The columns are the powers, so the first column is the zeroth power (always 1) and the second column gives the colors for the numbers modulo the given prime (first power).

One more very useful object is the prime counting function $\pi(x)$. This comes with its own custom plotting.

```
sage: prime_pi(100); plot(prime_pi,1,100)
25
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> prime_pi(Integer(100)); plot(prime_pi,Integer(1),Integer(100))
25
Graphics object consisting of 1 graphics primitive
```

A very nice aspect of Sage is combining several aspects of mathematics together. It can be very eye-opening to students to see analytic aspects of number theory early on. (Note that we have to reassign x to a variable, since above it was a cryptographic key!)

```
sage: var('x')
x
sage: plot(prime_pi,2,10^6,thickness=2)+plot(Li,2,10^6,color='red')+plot(x/ln(x),2,10^
↪6,color='green')
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> var('x')
x
>>> plot(prime_pi,Integer(2),Integer(10)**Integer(6),thickness=Integer(2))+plot(Li,
↪Integer(2),Integer(10)**Integer(6),color='red')+plot(x/ln(x),Integer(2),
↪Integer(10)**Integer(6),color='green')
Graphics object consisting of 3 graphics primitives
```

7.7.3 Advanced Number Theory

For those who are interested, more advanced number-theoretic objects are easy to come by; we end with a brief sampler of these.

In the first example, K is the field extension $\mathbf{Q}(\sqrt{-14})$, where the symbol a plays the role of $\sqrt{-14}$; we discover several basic facts about K in the next several cells.

```
sage: K.<a> = NumberField(x^2+14); K
Number Field in a with defining polynomial x^2 + 14
```

```
>>> from sage.all import *
>>> K = NumberField(x**Integer(2)+Integer(14), names=('a',)); (a,) = K._first_
↪ngens(1); K
Number Field in a with defining polynomial x^2 + 14
```

```
sage: K.discriminant(); K.class_group().order(); K.class_group().is_cyclic()
-56
4
True
```

```
>>> from sage.all import *
>>> K.discriminant(); K.class_group().order(); K.class_group().is_cyclic()
-56
4
True
```

Various zeta functions are also available; here is a complex plot of the Riemann zeta.

```
sage: complex_plot(zeta, (-30,30), (-30,30))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> complex_plot(zeta, (-Integer(30),Integer(30)), (-Integer(30),Integer(30)))
Graphics object consisting of 1 graphics primitive
```

7.7.4 Cryptography

Sage supports various more advanced cryptographic procedures as well as some basic pedagogical ones natively. This example is adapted from the documentation.

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES(); sdes
Simplified DES block cipher with 10-bit keys
```

```
>>> from sage.all import *
>>> from sage.crypto.block_cipher.sdes import SimplifiedDES
>>> sdes = SimplifiedDES(); sdes
Simplified DES block cipher with 10-bit keys
```

```
sage: bin = BinaryStrings()
sage: P = [0,1,0,0,1,1,0,1] # our message
sage: K = sdes.random_key() # generate a random key
sage: C = sdes.encrypt(P, K) # encrypt our message
sage: plaintxt = sdes.decrypt(C, K) # decrypt it
sage: plaintxt # print it
[0, 1, 0, 0, 1, 1, 0, 1]
```

```
>>> from sage.all import *
>>> bin = BinaryStrings()
>>> P = [Integer(0),Integer(1),Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(1)] # our message
>>> K = sdes.random_key() # generate a random key
>>> C = sdes.encrypt(P, K) # encrypt our message
>>> plaintxt = sdes.decrypt(C, K) # decrypt it
>>> plaintxt # print it
[0, 1, 0, 0, 1, 1, 0, 1]
```

See also the cryptography example in the *discrete math quickstart*.

7.8 Sage Quickstart for Statistics

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071). It is licensed under the Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA).

7.8.1 Basic Descriptive Statistics

Basic statistical functions are best to get from numpy and scipy.stats.

NumPy provides, for example, functions to compute the arithmetic mean and the standard deviation:

```
sage: import numpy as np
sage: np.mean([1, 2, 3, 5])
2.75

sage: np.std([1, 2, 2, 4, 5, 6, 8])  # rel to 1e-13
2.32992949004287
```

```
>>> from sage.all import *
>>> import numpy as np
>>> np.mean([Integer(1), Integer(2), Integer(3), Integer(5)])
2.75

>>> np.std([Integer(1), Integer(2), Integer(2), Integer(4), Integer(5), Integer(6),␣
↪Integer(8)])  # rel to 1e-13
2.32992949004287
```

SciPy offers many more functions, for example a function to compute the harmonic mean:

```
sage: from scipy.stats import hmean
sage: hmean([1, 2, 2, 4, 5, 6, 8])  # rel to 1e-13
2.5531914893617023
```

```
>>> from sage.all import *
>>> from scipy.stats import hmean
>>> hmean([Integer(1), Integer(2), Integer(2), Integer(4), Integer(5), Integer(6),␣
↪Integer(8)])  # rel to 1e-13
2.5531914893617023
```

We do not recommend to use Python's built in `statistics` module with Sage. It has a known incompatibility with number types defined in Sage, see Issue #28234.

7.8.2 Distributions

Let's generate a random sample from a given type of continuous distribution using native Python random generators.

- We use the simplest method of generating random elements from a log normal distribution with (normal) mean 2 and $\sigma = 3$.
- Notice that there is really no way around making some kind of loop.

```
sage: my_data = [lognormvariate(2, 3) for i in range(10)]
sage: my_data # random
[13.189347821530054, 151.28229284782799, 0.071974845847761343, 202.62181449742425, 1.
↪9677158880100207, 71.959830176932542, 21.054742855786007, 3.9235315623286406, 4129.
↪9880239483346, 16.41063858663054]
```

```
>>> from sage.all import *
>>> my_data = [lognormvariate(Integer(2), Integer(3)) for i in range(Integer(10))]
>>> my_data # random
[13.189347821530054, 151.28229284782799, 0.071974845847761343, 202.62181449742425, 1.
↪9677158880100207, 71.959830176932542, 21.054742855786007, 3.9235315623286406, 4129.
↪9880239483346, 16.41063858663054]
```

We can check whether the mean of the log of the data is close to 2.

```
sage: np.mean([log(item) for item in my_data]) # random
3.0769518857697618
```

```
>>> from sage.all import *
>>> np.mean([log(item) for item in my_data]) # random
3.0769518857697618
```

Here is an example using the Gnu scientific library under the hood.

- Let `dist` be the variable assigned to a continuous Gaussian/normal distribution with standard deviation of 3.
- Then we use the `.get_random_element()` method ten times, adding 2 each time so that the mean is equal to 2.

```
sage: dist=RealDistribution('gaussian',3)
sage: my_data=[dist.get_random_element()+2 for _ in range(10)]
sage: my_data # random
[3.18196848067, -2.70878671264, 0.445500746768, 0.579932075555, -1.32546445128, 0.
↪985799587162, 4.96649083229, -1.78785287243, -3.05866866979, 5.90786474822]
```

```
>>> from sage.all import *
>>> dist=RealDistribution('gaussian',Integer(3))
>>> my_data=[dist.get_random_element()+Integer(2) for _ in range(Integer(10))]
>>> my_data # random
[3.18196848067, -2.70878671264, 0.445500746768, 0.579932075555, -1.32546445128, 0.
↪985799587162, 4.96649083229, -1.78785287243, -3.05866866979, 5.90786474822]
```

For now, it's a little annoying to get histograms of such things directly. Here, we get a larger sampling of this distribution and plot a histogram with 10 bins.

```
sage: my_data2 = [dist.get_random_element()+2 for _ in range(1000)]
sage: T = stats.TimeSeries(my_data)
sage: T.plot_histogram(normalize=False,bins=10)
Graphics object consisting of 10 graphics primitives
```

```
>>> from sage.all import *
>>> my_data2 = [dist.get_random_element()+Integer(2) for _ in range(Integer(1000))]
>>> T = stats.TimeSeries(my_data)
>>> T.plot_histogram(normalize=False,bins=Integer(10))
Graphics object consisting of 10 graphics primitives
```

To access discrete distributions, we again use Scipy.

- We have to `import` the module `scipy.stats`.
- We use `binom_dist` to denote the binomial distribution with 20 trials and 5% expected failure rate.
- The `.pmf(x)` method gives the probability of x failures, which we then plot in a bar chart for x from 0 to 20. (Don't forget that `range(21)` means all integers from *zero to twenty*.)

```
sage: import scipy.stats
sage: binom_dist = scipy.stats.binom(20,.05)
sage: bar_chart([binom_dist.pmf(x) for x in range(21)])
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> import scipy.stats
>>> binom_dist = scipy.stats.binom(Integer(20),RealNumber('.05'))
>>> bar_chart([binom_dist.pmf(x) for x in range(Integer(21))])
Graphics object consisting of 1 graphics primitive
```

The `bar_chart` function performs some of the duties of histograms.

Scipy's statistics can do other things too. Here, we find the median (as the fiftieth percentile) of an earlier data set.

```
sage: scipy.stats.scoreatpercentile(my_data, 50) # random
0.51271641116183286
```

```
>>> from sage.all import *
>>> scipy.stats.scoreatpercentile(my_data, Integer(50)) # random
0.51271641116183286
```

The key thing to remember here is to look at the documentation!

7.8.3 Using R from within Sage

The R project is a leading software package for statistical analysis with widespread use in industry and academics.

There are several ways to access R; they are based on the `rpy2` package.

- One of the easiest is to just put `r()` around things you want to make into statistical objects, and then …
- Use R commands via `r.method()` to pass them on to Sage for further processing.

The following example of the Kruskal-Wallis test comes directly from the examples in `r.kruskal_test?` in the notebook.

```
sage: # optional - rpy2
sage: x = r([2.9, 3.0, 2.5, 2.6, 3.2])       # normal subjects
sage: y = r([3.8, 2.7, 4.0, 2.4])            # with obstructive airway disease
sage: z = r([2.8, 3.4, 3.7, 2.2, 2.0])       # with asbestosis
sage: a = r([x,y,z])                         # make a long R vector of all the data
sage: b = r.factor(5*[1] + 4*[2] + 5*[3])    # create something for R to tell
....:                                        #   which subjects are which
sage: a; b                                   # show them
 [1] 2.9 3.0 2.5 2.6 3.2 3.8 2.7 4.0 2.4 2.8 3.4 3.7 2.2 2.0
 [1] 1 1 1 1 1 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

```
>>> from sage.all import *
>>> # optional - rpy2
>>> x = r([RealNumber('2.9'), RealNumber('3.0'), RealNumber('2.5'), RealNumber('2.6'),
↪ RealNumber('3.2')])      # normal subjects
>>> y = r([RealNumber('3.8'), RealNumber('2.7'), RealNumber('4.0'), RealNumber('2.4
↪')])           # with obstructive airway disease
>>> z = r([RealNumber('2.8'), RealNumber('3.4'), RealNumber('3.7'), RealNumber('2.2'),
↪ RealNumber('2.0')])      # with asbestosis
>>> a = r([x,y,z])                          # make a long R vector of all the data
>>> b = r.factor(Integer(5)*[Integer(1)] + Integer(4)*[Integer(2)] +␣
↪Integer(5)*[Integer(3)])   # create something for R to tell
...                                         #   which subjects are which
>>> a; b                                    # show them
 [1] 2.9 3.0 2.5 2.6 3.2 3.8 2.7 4.0 2.4 2.8 3.4 3.7 2.2 2.0
 [1] 1 1 1 1 1 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

```
sage: r.kruskal_test(a,b)             # do the KW test!                      #␣
↪optional - rpy2
    Kruskal-Wallis rank sum test

data:  sage17 and sage33
Kruskal-Wallis chi-squared = 0.7714, df = 2, p-value = 0.68
```

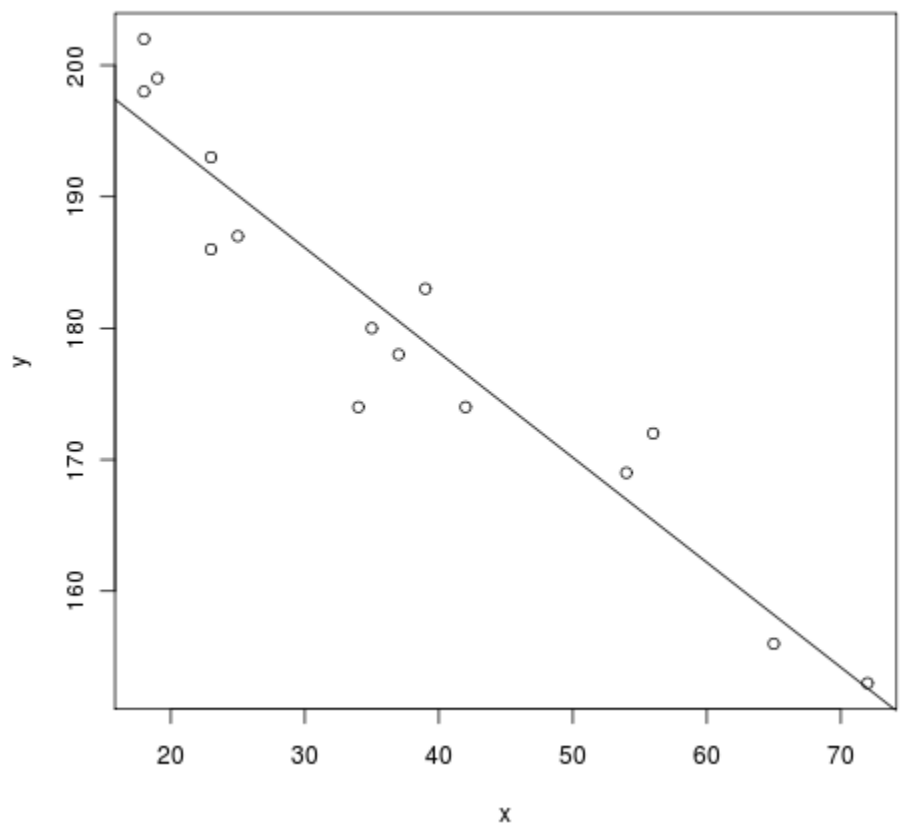
```
>>> from sage.all import *
>>> r.kruskal_test(a,b)             # do the KW test!                      #␣
↪optional - rpy2
    Kruskal-Wallis rank sum test

data:  sage17 and sage33
Kruskal-Wallis chi-squared = 0.7714, df = 2, p-value = 0.68
```

Looks like we can't reject the null hypothesis here.

The best way to use R seriously is to simply ask each individual cell to evaluate completely in R, using a so-called "percent directive". Here is a sample linear regression from John Verzani's simpleR text. Notice that R also uses the # symbol to indicate comments.

```
sage: %r                                                                      #␣
↪optional - rpy2
....: x = c(18,23,25,35,65,54,34,56,72,19,23,42,18,39,37) # ages of individuals
....: y = c(202,186,187,180,156,169,174,172,153,199,193,174,198,183,178) # maximum␣
↪heart rate of each one
....: png() # turn on plotting
....: plot(x,y) # make a plot
....: lm(y ~ x) # do the linear regression
....: abline(lm(y ~ x)) # plot the regression line
....: dev.off()     # turn off the device so it plots
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
   210.0485      -0.7977

null device
          1
```

```
>>> from sage.all import *
>>> %r                                                                    #␣
↪optional - rpy2
... x = c(Integer(18),Integer(23),Integer(25),Integer(35),Integer(65),Integer(54),
↪Integer(34),Integer(56),Integer(72),Integer(19),Integer(23),Integer(42),Integer(18),
↪Integer(39),Integer(37)) # ages of individuals
... y = c(Integer(202),Integer(186),Integer(187),Integer(180),Integer(156),
↪Integer(169),Integer(174),Integer(172),Integer(153),Integer(199),Integer(193),
↪Integer(174),Integer(198),Integer(183),Integer(178)) # maximum heart rate of each␣
↪one
... png() # turn on plotting
... plot(x,y) # make a plot
... lm(y ~ x) # do the linear regression
... abline(lm(y ~ x)) # plot the regression line
... dev.off()     # turn off the device so it plots
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
   210.0485      -0.7977

null device
          1
```



To get a whole worksheet to evaluate in R (and be able to ignore the `%`), you could also drop down the `r` option in the menu close to the top which currently has `sage` in it.

7.9 Sage Interact Quickstart

This Sage quickstart tutorial was developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071).

Invaluable resources are the Sage wiki http://wiki.sagemath.org/interact (type "sage interact" into Google), UTMOST Sage Cell Repository (a collection of contributed interacts).

7.9.1 Start with just one command

How would one create an interactive cell? First, let's focus on a new thing to do! Perhaps we just want a graph plotter that has some options.

So let's start by getting the commands for what you want the output to look like. Here we just want a simple plot.

```
sage: plot(x^2,(x,-3,3))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(x**Integer(2),(x,-Integer(3),Integer(3)))
Graphics object consisting of 1 graphics primitive
```

Then abstract out the parts you want to change. We'll be letting the user change the function, so let's make that a variable `f`.

```
sage: f=x^3
sage: plot(f,(x,-3,3))
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> f=x**Integer(3)
>>> plot(f,(x,-Integer(3),Integer(3)))
Graphics object consisting of 1 graphics primitive
```

This was important because it allowed you to step back and think about what you would really be doing.

Now for the technical part. We make this a `def` function - see the *programming tutorial*.

```
sage: def myplot(f=x^2):
....:     show(plot(f,(x,-3,3)))
```

```
>>> from sage.all import *
>>> def myplot(f=x**Integer(2)):
...     show(plot(f,(x,-Integer(3),Integer(3))))
```

Let's test the `def` function `myplot` by just calling it.

```
sage: myplot()
```

```
>>> from sage.all import *
>>> myplot()
```

If we call it with a different value for `f`, we should get a different plot.

```
sage: myplot(x^3)
```

```
>>> from sage.all import *
>>> myplot(x**Integer(3))
```

So far, we've only defined a new function, so this was review. To make a "control" to allow the user to interactively enter the function, we just preface the function with `@interact`.

```
sage: @interact
sage: def myplot(f=x^2):
....:     show(plot(f,(x,-3,3)))
```

```
>>> from sage.all import *
>>> @interact
>>> def myplot(f=x**Integer(2)):
...     show(plot(f,(x,-Integer(3),Integer(3))))
```

Note: Technically what `@interact` does is wrap the function, so the above is equivalent to:

```
def myplot(..): ...
myplot=interact(myplot)
```

Note that we can still call our function, even when we've used `@interact`. This is often useful in debugging it.

```
sage: myplot(x^4)
```

```
>>> from sage.all import *
>>> myplot(x**Integer(4))
```

7.9.2 Adding Complexity

We can go ahead and replace other parts of the expression with variables. Note that _ is the function name now. That is a just convention for throw-away names that we don't care about.

```
sage: @interact
sage: def _(f=x^2, a=-3, b=3):
....:     show(plot(f,(x,a,b)))
```

```
>>> from sage.all import *
>>> @interact
>>> def _(f=x**Integer(2), a=-Integer(3), b=Integer(3)):
...     show(plot(f,(x,a,b)))
```

If we pass `('label', default_value)` in for a control, then the control gets the label when printed. Here, we've put in some text for all three of them. Remember that the text must be in quotes! Otherwise Sage will think that you are referring (for example) to some variable called "lower", which it will think you forgot to define.

```
sage: @interact
sage: def _(f=('$f$', x^2), a=('lower', -3), b=('upper', 3)):
....:     show(plot(f,(x,a,b)))
```

```
>>> from sage.all import *
>>> @interact
>>> def _(f=('$f$', x**Integer(2)), a=('lower', -Integer(3)), b=('upper',␣
↪Integer(3))):
...     show(plot(f,(x,a,b)))
```

We can specify the type of control explicitly, along with options. See *below* for more detail on the possibilities.

```
sage: @interact
sage: def _(f=input_box(x^2, width=20, label="$f$")):
....:     show(plot(f,(x,-3,3)))
```

```
>>> from sage.all import *
>>> @interact
>>> def _(f=input_box(x**Integer(2), width=Integer(20), label="$f$")):
...     show(plot(f,(x,-Integer(3),Integer(3))))
```

Here we demonstrate a bunch of options. Notice the new controls:

- `range_slider`, which passes in *two* values, `zoom[0]` and `zoom[1]`
- `True`/`False` gets converted to checkboxes for the end user

```
sage: @interact
sage: def _(f=input_box(x^2,width=20),
....: color=color_selector(widget='colorpicker', label=""),
....: axes=True,
....: fill=True,
....: zoom=range_slider(-3,3,default=(-3,3))):
....:     show(plot(f,(x,zoom[0], zoom[1]), color=color, axes=axes,fill=fill))
```

```
>>> from sage.all import *
>>> @interact
>>> def _(f=input_box(x**Integer(2),width=Integer(20)),
... color=color_selector(widget='colorpicker', label=""),
... axes=True,
... fill=True,
... zoom=range_slider(-Integer(3),Integer(3),default=(-Integer(3),Integer(3)))):
...     show(plot(f,(x,zoom[Integer(0)], zoom[Integer(1)]), color=color, axes=axes,
↪fill=fill))
```

There is also one button type to *disable automatic updates*.

The previous interact was a bit ugly, because all of the controls were stacked on top of each other. We can control the layout of the widget controls in a grid (at the top, bottom, left, or right) using the `layout` parameter.

```
sage: @interact(layout=dict(top=[['f', 'color']],
....: left=[['axes'],['fill']],
....: bottom=[['zoom']]))
sage: def _(f=input_box(x^2,width=20),
....: color=color_selector(widget='colorpicker', label=""),
....: axes=True,
....: fill=True,
....: zoom=range_slider(-3,3, default=(-3,3))):
....:     show(plot(f,(x,zoom[0], zoom[1]), color=color, axes=axes,fill=fill))
```

```
>>> from sage.all import *
>>> @interact(layout=dict(top=[['f', 'color']],
... left=[['axes'],['fill']],
... bottom=[['zoom']]))
>>> def _(f=input_box(x**Integer(2),width=Integer(20)),
... color=color_selector(widget='colorpicker', label=""),
... axes=True,
... fill=True,
... zoom=range_slider(-Integer(3),Integer(3), default=(-Integer(3),Integer(3)))):
...     show(plot(f,(x,zoom[Integer(0)], zoom[Integer(1)]), color=color, axes=axes,
↪fill=fill))
```

7.9.3 Control Types

There are many potential types of widgets one might want to use for interactive control. Sage has all of the following:

- boxes
- sliders
- range sliders
- checkboxes
- selectors (dropdown lists or buttons)
- grid of boxes
- color selectors
- plain text

We illustrate some more of these below.

```
sage: @interact
sage: def _(frame=checkbox(True, label='Use frame')):
....:     show(plot(sin(x), (x,-5,5)), frame=frame)
```

```
>>> from sage.all import *
>>> @interact
>>> def _(frame=checkbox(True, label='Use frame')):
...     show(plot(sin(x), (x,-Integer(5),Integer(5))), frame=frame)
```

```
sage: var('x,y')
sage: colormaps=sage.plot.colors.colormaps.keys()
sage: @interact
sage: def _(cmap=selector(colormaps)):
....:     contour_plot(x^2-y^2,(x,-2,2),(y,-2,2),cmap=cmap).show()
```

```
>>> from sage.all import *
>>> var('x,y')
>>> colormaps=sage.plot.colors.colormaps.keys()
>>> @interact
>>> def _(cmap=selector(colormaps)):
...     contour_plot(x**Integer(2)-y**Integer(2),(x,-Integer(2),Integer(2)),(y,-
↪Integer(2),Integer(2)),cmap=cmap).show()
```

```
sage: var('x,y')
sage: colormaps=sage.plot.colors.colormaps.keys()
sage: @interact
sage: def _(cmap=selector(['RdBu', 'jet', 'gray','gray_r'],buttons=True),
sage: type=['density','contour']):
....:     if type=='contour':
....:         contour_plot(x^2-y^2,(x,-2,2),(y,-2,2),cmap=cmap, aspect_ratio=1).show()
....:     else:
....:         density_plot(x^2-y^2,(x,-2,2),(y,-2,2),cmap=cmap, frame=True,axes=False,
↪aspect_ratio=1).show()
```

```
>>> from sage.all import *
>>> var('x,y')
>>> colormaps=sage.plot.colors.colormaps.keys()
>>> @interact
>>> def _(cmap=selector(['RdBu', 'jet', 'gray','gray_r'],buttons=True),
>>> type=['density','contour']):
...     if type=='contour':
...         contour_plot(x**Integer(2)-y**Integer(2),(x,-Integer(2),Integer(2)),(y,-
↪Integer(2),Integer(2)),cmap=cmap, aspect_ratio=Integer(1)).show()
...     else:
...         density_plot(x**Integer(2)-y**Integer(2),(x,-Integer(2),Integer(2)),(y,-
↪Integer(2),Integer(2)),cmap=cmap, frame=True,axes=False,aspect_ratio=Integer(1)).
↪show()
```

By default, ranges are sliders that divide the range into 50 steps.

```
sage: @interact
sage: def _(n=(1,20)):
....:     print(factorial(n))
```

```
>>> from sage.all import *
>>> @interact
>>> def _(n=(Integer(1),Integer(20))):
...     print(factorial(n))
```

You can set the step size to get, for example, just integer values.

```
sage: @interact
sage: def _(n=slider(1,20, step_size=1)):
....:     print(factorial(n))
```

```
>>> from sage.all import *
>>> @interact
>>> def _(n=slider(Integer(1),Integer(20), step_size=Integer(1))):
...     print(factorial(n))
```

Or you can explicitly specify the slider values.

```
sage: @interact
sage: def _(n=slider([1..20])):
....:     print(factorial(n))
```

```
>>> from sage.all import *
>>> @interact
```

(continues on next page)

(continued from previous page)

```
>>> def _(n=slider((ellipsis_range(Integer(1),Ellipsis,Integer(20))))):
...     print(factorial(n))
```

And the slider values don't even have to be numbers!

```
sage: @interact
sage: def _(fun=('function', slider([sin,cos,tan,sec,csc,cot]))):
....:     print(fun(4.39293))
```

```
>>> from sage.all import *
>>> @interact
>>> def _(fun=('function', slider([sin,cos,tan,sec,csc,cot]))):
...     print(fun(RealNumber('4.39293')))
```

Matrices are automatically converted to a grid of input boxes.

```
sage: @interact
sage: def _(m=('matrix', identity_matrix(2))):
....:     print(m.eigenvalues())
```

```
>>> from sage.all import *
>>> @interact
>>> def _(m=('matrix', identity_matrix(Integer(2)))):
...     print(m.eigenvalues())
```

Here's how to get vectors from a grid of boxes.

```
sage: @interact
sage: def _(v=('vector', input_grid(1, 3, default=[[1,2,3]], to_value=lambda x:␣
↪vector(flatten(x))))):
....:     print(v.norm())
```

```
>>> from sage.all import *
>>> @interact
>>> def _(v=('vector', input_grid(Integer(1), Integer(3), default=[[Integer(1),
↪Integer(2),Integer(3)]], to_value=lambda x: vector(flatten(x))))):
...     print(v.norm())
```

7.9.4 The option not to update

As a final problem, what happens when the controls get so complicated that it would counterproductive to see the interact update for each of the changes one wants to make? Think changing the endpoints and order of integration for a triple integral, for instance, or the example below where a whole matrix might be changed.

In this situation, where we don't want any updates until we specifically say so, we can use the `auto_update=False` option. This will create a button to enable the user to update as soon as he or she is ready.

```
sage: @interact
sage: def _(m=('matrix', identity_matrix(2)), auto_update=False):
....:     print(m.eigenvalues())
```

```
>>> from sage.all import *
>>> @interact
```

(continues on next page)

(continued from previous page)

```
>>> def _(m=('matrix', identity_matrix(Integer(2))), auto_update=False):
...     print(m.eigenvalues())
```