
SDSU Sage Tutorial Documentation

Release 1.2

Michael O'Sullivan, David Monarres, Matteo Polimeno

Jan 25, 2019

CONTENTS

1	About this tutorial	3
1.1	Introduction	3
1.2	Getting Started	3
1.3	Contributing to the tutorial	11
2	SageMath as a Calculator	13
2.1	Arithmetic and Functions	13
2.2	Solving Equations and Inequalities	19
2.3	Calculus	21
2.4	Statistics	26
2.5	Plotting	27
3	Programming in SageMath	39
3.1	SageMath Objects	39
3.2	Programming Tools	54
3.3	Packages within SageMath	61
3.4	Interactive Demonstrations in the Notebook	66
4	Mathematical Structures	73
4.1	Integers and Modular Arithmetic	73
4.2	Groups	78
4.3	Linear Algebra	88
4.4	Rings	98
4.5	Fields	109
4.6	Coding Theory	114
	Bibliography	123
	Index	125

Contents:

ABOUT THIS TUTORIAL

1.1 Introduction

1.1.1 How to use this tutorial

This tutorial is divided into four parts. This part, *About this tutorial*, discusses some basic properties of SageMath, introduces you to the structure of the tutorial, and explains how to contribute to the project if you so desire.

The second part, *SageMath as a Calculator*, covers topics such as how to do arithmetic, evaluate functions, create simple graphs, solve equations and do basic calculus. We call this section *SageMath as a Calculator* because most of the topics covered are those that are commonly done with a standard graphing calculator. The target audience for this section is any motivated pre-calculus or calculus student.

Programming in SageMath introduces the reader to some more *advanced* topics such as how SageMath handles numbers; how to define and use variables and functions; how to manipulate lists, strings, and sets; and SageMath *universes* and *coercion*.

The final part, *Mathematical Structures*, introduces the reader to topics that one finds in a college-level curriculum: linear algebra, number theory, groups, rings, fields, etc.

Since this tutorial is an introduction to SageMath, we will be using examples to demonstrate ideas and the reader is encouraged to follow along as we progress by entering the commands into their own copy of SageMath. We have included exercises for practice and problems for more extensive exploration of a given topic. The reader is also encouraged to do many of these.

While the tutorial mostly progresses in a linear fashion, we still include at the beginning of each section a list of the most important prerequisite topics. This list follows the text “You should be familiar with.” and by clicking one of those links you will be taken to the relevant portion of the tutorial. We have also included links to further information and other on-line references. These will follow the “**See also:**” text.

Some sections may contain numbered citations such as “¹.” The list of these citations will be at the bottom of a section with at least one citation. These citations will direct the reader to texts which contain more information about the topic being presented.

References:

1.2 Getting Started

1.2.1 About SageMath

SageMath (previously Sage or SAGE) is a free open-source mathematical software system based on the Python pro-

¹ William A. Stein et al. Sage Mathematics Software (Version 8.2), The Sage Development Team, 2018, <http://www.sagemath.org>.

[gramming language](#). Originally created for research into Mathematics, it has been evolving into a powerful tool for Math education. It combines numerous other mathematical software packages with a single interface, using Python. By learning SageMath, you are also learning a lot about Python.

As an open source project, SageMath invites contributions from all of its users. This tutorial is one of many sources of information for learning about how to use SageMath. For more information see the SageMath project's [website](#).

This tutorial assumes that the reader has access to a running copy of SageMath. On most operating systems, installing SageMath usually consists of downloading the proper package from the project's main [website](#), unwrapping it, and executing *sage* from within. For more information on the process of installing sage see SageMath's [Installation Guide](#).

A good alternative is to run SageMath in the cloud using [Cocalc](#). All you need to do is either sign up for a free account or sign in through a Google/Github/Facebook/Twitter account. Once you are signed up, you can start a project using SageMath, and also share it with other users. For more information about Cocalc and its features, visit [Cocalc Tutorial](#).


If you opted for the physical installation and started SageMath, you should know that there are two ways to enter commands: either from the *command line* or by using the web-based *notebook*. The notebook interface is similar in design to the interface of *Matlab*, *Mathematica*, or *Maple* and is a popular choice.

Everything that follows the `sage:` prompt is a command that we encourage the reader to type in on their own. For example, if we wanted to *factor* the integer 1438880 we would give the following example using SageMath's `factor()` command.

```
sage: factor(1438880)
2^5 * 5 * 17 * 23^2
```

The line after the `sage:` contains the output that the user should expect after properly entering the command.

From the command line the interaction would probably look a bit like this:

 SageMath 8.2 Console

```
SageMath version 8.2, Release Date: 2018-05-05
Type "notebook()" for the browser-based notebook interface.
Type "help()" for help.
```

```
Setting permissions of DOT_SAGE directory so only you can read and write i
sage: factor(1438880)
2^5 * 5 * 17 * 23^2
sage: |
```

If the user is using the notebook (most likely) the interaction will look a little like:

The screenshot shows a web browser window with the address bar containing `localhost:8080/home/admin/0/`. The page header includes the logo for "SDGE The Sage Notebook" (Version 8.2) and a navigation menu with links for [admin](#), [Toggle](#), [Home](#), [Published](#), [Log](#), [Settings](#), [Help](#), [Report a Problem](#), and [Sign](#). The main content area is titled "sagemath-tutorial" and shows it was last edited on Aug 7, 2018, at 11:19:55 AM by admin. Below the title is a toolbar with buttons for "File...", "Action...", "Data...", "sage", "Typeset", "Load 3-D Live", and "Use java for 3-D". To the right of the toolbar are buttons for "Print", "Worksheet", "Edit", "Text", "Revisions", "Share", and "Pub". The main workspace contains a code cell with the input `factor(1438880)` and the output $2^5 * 5 * 17 * 23^2$. Below the code cell is an empty input field and an "evaluate" button.

For Cocalc users, it will look like:

If you are in fact using Cocalc, you have probably already noticed the red banner that pops up at the very top of the page that says *Upgrade this project*, every time you create a new project. Just ignore it. It is a bit annoying to the eye, but will do no harm.

1.2.2 Tab Completion

Next we will discuss how to use a couple of important features of the various SageMath interfaces; tab-completion and the built-in help system.

One of the handiest features built into SageMath is the *tab completion* of commands. To use tab completion, just type in the first couple of letters of the command that you would like to use, and then press the tab-key. For instance, suppose that you want to compute $56!$ but don't remember the exact command name to do this. A good guess is that the command will have *factorial* somewhere in its name. To see if that guess is correct, just type the first three letters `fac` and hit the tab-key.

```
sage: fac[TAB]
factor      factorial
sage: factor
```

The output tells you that only two SageMath commands begin with `fac`, `factor()` and `factorial()`. Note that SageMath has already changed the command from `fac` to `factor` because this is the common root for both

commands. Since *factorial* looks like the correct command, we will select this by typing the next letter, *i*, and hitting the tab key again.

```
sage: factorial
```

This time no list is returned because the only command that begins with *factori* is *factorial()*. So to compute $56!$ you just complete the command by adding the argument (56) .

```
sage: factorial(56)
71099858780486345185404564746372494973649797888116845868744704000000000000
```

Another good use of tab-completion is to discover what *methods* an *object* has. Say you have the integer $a = 56$ and were wondering what commands SageMath offers to work with integers like 56. In this case the a is our object and we can find all of the methods associated with integers by typing $a.$ then hitting the tab-key.

```
sage: a = 56
sage: a.[TAB]
a.N                a.kronecker
... A long list of Commands ...
a.divisors         a.parent
a.dump             a.popcount
a.dumps           a.powermod
a.exact_log        a.powermodm_ui
--More--
```

Do not be intimidated by the length of this list. SageMath is a very powerful system and it can do a lot with integers. On the command line, the `--More--` at the bottom of the screen tells you that the list of possible commands is longer than what will fit on a single screen. To scroll through this list a page at a time, just hit any key and SageMath will display the next page.

On the second page you see that `factor()` is an option. To use this method, which *factors* 56 into unique prime factors, you enter `a.factor()`.

```
sage: a.factor()
2^3 * 7
```

Tab-completion can not only reduce the amount of typing needed, but it can be used to *discover* new commands in SageMath.

1.2.3 Help using ?

Once you have identified a command that interests you, the next step is to find out exactly *what* this command does and *how* to use it. SageMath has a built-in help system to help you achieve this very goal.

Let's suppose that you wish to compute the *lowest common multiple* of two integers and are not sure which command does this. A good place to begin the search is by typing `l` at the command prompt and then hitting the tab-key.

```
sage: l[TAB]
laguerre          list_plot3d
lambda           lk
laplace          ll
latex            ln
lattice_polytope lngamma
lazy_attribute   load
lazy_import      load_attach_path
lc              load_session
lcalc           loads
```

```
lcm                local/LIB
ldir               local/bin
...
lisp_console      ls
list              lucas_number1
list_composition  lucas_number2
list_plot         lx
```

Once again you have quite a long list of commands from which to select. Scanning down the list, you see the `lcm()` command listed which seems like what you are trying to compute. To make sure of this enter `lcm?`.

```
sage: lcm?
```

The output of this command is a page that explains both the use and the purpose of the command.

```
Base Class:      <type 'function'>
String Form:    <function lcm at 0x32db6e0>
Namespace:     Interactive
File:          /home/ayeq/sage/local/lib/python2.6/site-packages/sage/rings/arith.py
Definition:    lcm(a, b=None)
Docstring:
    The least common multiple of a and b, or if a is a list and b is
    omitted the least common multiple of all elements of a.

    Note that LCM is an alias for lcm.

INPUT:
* ``a,b`` - two elements of a ring with lcm or
* ``a`` - a list or tuple of elements of a ring with lcm

EXAMPLES:
    sage: lcm(97,100)
    9700
    sage: LCM(97,100)
```

Again, there will be a whole lot of information, usually more than will fit on one screen. On the command line, navigation is easy; the space bar will take you to the next page, and `b`, or the up-arrow key, will move backward in the documentation. To exit the help system hit the `q` key.

When first starting out; the description, the `INPUT`, and the `EXAMPLES` sections are good sections to read. The description gives a short summary describing what the command does, `INPUT` gives you information on what you should provide as *arguments* to the command, and `EXAMPLES` gives concrete examples of the command's usage.

The description in this case is:

```
The least common multiple of a and b, or if a is a list and b is
omitted the least common multiple of all elements of a.
Note that LCM is an alias for lcm.
```

From this description, you can be pretty sure that this is the command that you are looking for. Next examine the `INPUT`:

```
INPUT:
* ``a,b`` - two elements of a ring with lcm or
* ``a`` - a list or tuple of elements of a ring with lcm
```

Here you see that `lcm` can either accept two arguments, for our purposes two integers, or a list of objects. Finally by perusing the `EXAMPLES` you can get a good idea on how this command is actually used in practice.

EXAMPLES:

```
sage: lcm(97,100)
9700
sage: LCM(97,100)
9700
sage: LCM(0,2)
0
sage: LCM(-3,-5)
15
sage: LCM([1,2,3,4,5])
60
sage: v = LCM(range(1,10000)) # *very* fast!
sage: len(str(v))
4349
```

Having a comprehensive help system built into SageMath is one of its best features and the sooner you get comfortable with using it the faster you will be able to use the full power of this CAS.

1.2.4 Source Code, ??

There are probably some readers of this tutorial who like programming and would like to take a look at how a command is built in SageMath. To do this, you simply need to type the command whose source code you are interested in, and then press the `<tab>` key. For instance, going back to our `factor()` command

```
sage: factor??
Source Code (starting at line 2139):

def factor(n, proof=None, int_=False, algorithm='pari', verbose=0, **kwds):
    """comments inserted here"""
    try:
        m = n.factor
    except AttributeError:
        """Maybe n is not a Sage element, try to convert it
        e = py_scalar_to_element(n)

if e is n:
    # Either n was a Sage Element without a factor() method
    # or we cannot it convert it to Sage
    raise TypeError("unable to factor {!r}".format(n))
n = e
m = n.factor

    """code continues"""
```

The output of this command is very long and will not fit in a single page or snapshot (try it yourself). However, we reported some of the lines of the code to show you how the syntax is Pythonic. Once you run the command, you will see a lot of comments (marked by a triple quote `"""`) in the code that will navigate you on how to read it, what it does and to interpret the Python syntax, if you are not familiar with it.

See also:

[SageMath Screencasts](#)

1.3 Contributing to the tutorial

Additions to this tutorial are encouraged as are suggestions for additional topics for inclusion.

When this website was first developed, all of its code was available for download from the original project's [bitbucket](#). However, given the increased and still growing popularity of GitHub over the past few years, we decided to transition the whole repository there. All of its contents can be accessed from [GitHub](#). There you will find a complete copy of the source code for generating this website. To build the site from its source, the reader will need to install the [Sphinx Documentation](#), which is written in the [Python Programming Language](#). We are excited to see any changes that you make so please let us know of any new material that you add. We want this tutorial to be as comprehensive as possible and any assistance toward this goal is welcomed.

The content of this tutorial is written using [reStructured Text](#), which is processed by [Sphinx](#) to produce the HTML and PDF output. Sphinx and reStructured Text are used throughout the official SageMath and Python documentation, so it is useful for contributors to either of these projects.

There are four parts to the tutorial: *About this tutorial* has basic instructions about using and amending the tutorial, and the others have mathematical content. *SageMath as a Calculator* is intended, as the title suggests, to cover straight-forward computations, plotting graphs, and content that one might find in a high school algebra course, introductory statistics or calculus. We intend it to be accessible to an entering college student, or to a bright high school student.

“Programming in SageMath” eases the transition to higher level mathematics by treating topics that relate to the interface between mathematical concepts and computational issues. The first chapter covers universes and coercion (rationals, reals, booleans etc.); variables; and basic structures like lists, sets and strings. The second chapter covers; programming essentials like conditionals and iterative computation; file handling and data handling; etc. The third chapter discusses mathematical software packages within SageMath. Finally, there is a brief discussion of interactive demonstrations with the notebook.

“Mathematical Structures” is written at a more sophisticated level than the earlier material, since the intended audience is college students taking upper division math courses. The emphasis is on learning about specific mathematical structures that have a SageMath class associated to them. We intend each chapter to be independent of the others.

See also:

[reStructured Text Primer](#)

1.3.1 Credits and License

The content and code for this tutorial was written by David Monarres under the supervision of Mike O’Sullivan and was supported by a generous grant from San Diego State University’s President’s Leadership Fund. The tutorial is licensed under the [Creative Commons Attribution-ShareAlike 3.01 License](#). You are free to share and to remix, but attribution should be given to the original funder and creators. You may add your name to the list of contributors below.

Other contributors include:

- Ryan Rosenbaum.
- Matteo Polimeno.

SAGEMATH AS A CALCULATOR

This part of the tutorial examines commands that allow you to use SageMath much like a graphing calculator. The chapter on arithmetic and functions and the chapter on solving equations and inequalities serve as a foundation for the rest of the material. The chapters on plotting, statistics and calculus are independent of each other, although plotting may be useful to read next since plotting graphs is so useful in calculus and in statistics.

2.1 Arithmetic and Functions

2.1.1 Basic Arithmetic

The basic arithmetic operators are $+$, $-$, $*$, and $/$ for addition, subtraction, multiplication and division, while $^$ is used for exponents.

```
sage: 1+1
2
sage: 103-101
2
sage: 7*9
63
sage: 7337/11
667
sage: 11/4
11/4
sage: 2^5
32
```

The $-$ symbol in front of a number indicates that it is negative.

```
sage: -6
-6
sage: -11+9
-2
```

As we would expect, SageMath adheres to the standard order of operations, PEMDAS (parenthesis, exponents, multiplication, division, addition, subtraction).

```
sage: 2*4^2+1
33
sage: (2*4)^2+1
65
sage: 2*4^(2+1)
128
```

```
sage: -3^2
-9
sage: (-3)^2
9
```

When dividing two integers, there is a subtlety: SageMath will return either a fraction or its decimal approximation. Unlike most graphing calculators, SageMath will attempt to be as *precise* as possible and will return the fraction unless told otherwise. One way to tell SageMath that we *want* the decimal approximation is to include a decimal in the expression itself.

```
sage: 11/4.0
2.7500000000000000
sage: 11/4.
2.7500000000000000
sage: 11.0/4
2.7500000000000000
sage: 11/4*1.
2.7500000000000000
```

Exercises:

1. Divide 28 by 2 raised to the 5th power as a rational number, then get its decimal approximation.
2. Compute a decimal approximation of $\sqrt{2}$
3. Use sage to compute $(-9)^{(1/2)}$. Describe the output.

2.1.2 Integer Division and Factoring

You should be familiar with “*Basic Arithmetic*”

Sometimes when we divide, the division operator doesn’t give us all of the information that we want. Often we would like to know not only what the reduced fraction is, or even its decimal approximation, but rather the unique *quotient* and the *remainder* of the division.

To calculate the quotient we use the `//` operator and the `%` operator is used for the remainder.

```
sage: 14 // 4
3
sage: 14 % 4
2
```

If we want both the quotient and the remainder all at once, we use the `divmod()` command

```
sage: divmod(14,4)
(3, 2)
```

Recall that b divides a if 0 is the remainder when we divide the two integers. The integers in SageMath have a built-in command (or ‘method’) which allows us to check whether one integer divides another.

```
sage: 3.divides(15)
True
sage: 5.divides(17)
False
```

A related command is the `divisors()` method. This method returns a list of all positive divisors of the integer specified.

```
sage: 12.divisors()
[1, 2, 3, 4, 6, 12]
sage: 101.divisors()
[1, 101]
```

When the divisors of an integer are only 1 and itself then we say that the number is *prime*. To check if a number is prime in sage, we use its `is_prime()` method.

```
sage: (2^19-1).is_prime()
True
sage: 153.is_prime()
False
```

Notice the parentheses around $2^{19} - 1$ in the first example. They are important to the order of operations in SageMath, and if they are not included then SageMath will compute something very different than we intended. Try evaluating `2^19-1.is_prime()` and notice the result. When in doubt, the judicious use of *parenthesis* is encouraged.

We use the `factor()` method to compute the *prime factorization* of an integer.

```
sage: 62.factor()
2 * 31
sage: 63.factor()
3^2 * 7
```

If we are interested in simply knowing which prime numbers divide an integer, we may use its `prime_divisors()` (or `prime_factors()`) method.

```
sage: 24.prime_divisors()
[2, 3]
sage: 63.prime_factors()
[3, 7]
```

Finally, we have the *greatest common divisor* and *least common multiple* of a pair of integers. A *common divisor* of two integers is any integer which is a divisor of each, whereas a *common multiple* is a number which both integers divide.

The greatest common divisor (gcd), not too surprisingly, is the largest of all of these common divisors. The `gcd()` command is used to calculate this divisor.

```
sage: gcd(14, 63)
7
sage: gcd(15, 19)
1
```

Notice that if two integers share no common divisors, then their gcd will be 1.

The least common multiple is the smallest integer which both integers divide. The `lcm()` command is used to calculate the least common multiple.

```
sage: lcm(4, 5)
20
sage: lcm(14, 21)
42
```

Exercises:

1. Find the quotient and remainder when dividing 98 into 956.

2. Use SageMath to verify that the quotient and remainder computed above are correct.
3. Use SageMath to determine if 3 divides 234878.
4. Compute the list of divisors for each of the integers 134, 491, 422 and 1002.
5. Which of the integers above are *prime*?
6. Calculate $\gcd(a, b)$, $\text{lcm}(a, b)$ and $a \cdot b$ for the pairs of integers (2, 5), (4, 10) and (18, 51). How do the gcd, lcm and the product of the numbers relate?

2.1.3 Standard Functions and Constants

SageMath includes nearly all of the standard functions that one encounters when studying Mathematics. In this section, we shall cover some of the most commonly used functions: the *maximum*, *minimum*, *floor*, *ceiling*, *trigonometric*, *exponential*, and *logarithm* functions. We will also see many of the standard mathematical constants; such as *Euler's constant* (e), π , and *the golden ratio* (ϕ).

The `max()` and `min()` commands return the largest and smallest of a set of numbers.

```
sage: max(1, 5, 8)
8
sage: min(1/2, 1/3)
1/3
```

We may input any number of arguments into the `max` and `min` functions.

In SageMath we use the `abs()` command to compute the *absolute value* of a real number.

```
sage: abs(-10)
10
sage: abs(4)
4
```

The `floor()` command rounds a number down to the nearest integer, while `ceil()` rounds up.

```
sage: floor(2.1)
2
sage: ceil(2.1)
3
```

We need to be very careful while using `floor()` and `ceil()`.

```
sage: floor(1/(2.1-2))
9
```

This is clearly not correct: $\lfloor 1/(2.1 - 2) \rfloor = \lfloor 1/.1 \rfloor = \lfloor 10 \rfloor = 10$. So what happened?

```
sage: 1/(2.1-2)
9.999999999999999
```

Computers store real numbers in *binary*, while we are accustomed to using the decimal representation. The 2.1 in decimal notation is quite simple and short, but when converted to binary it is $10.0001\bar{1} = 10.0001100110011\dots$

Since computers cannot store an infinite number of digits, this gets rounded off somewhere, resulting in the slight error we saw. In SageMath, however, *rational numbers* (fractions) are exact, so we will never see this rounding error.

```
sage: floor(1/(21/10-2))
10
```

Due to this, it is often a good idea to use rational numbers whenever possible instead of decimals, particularly if a high level of precision is required.

The `sqrt()` command calculates the *square root* of a real number. As we have seen earlier with fractions, if we want a decimal approximation we can get this by giving a decimal number as the input.

```
sage: sqrt(3)
sqrt(3)
sage: sqrt(3.0)
1.73205080756888
```

To compute other roots, we use a rational exponent. SageMath can compute any rational power. If either the exponent or the base is a decimal then the output will be a decimal.

```
sage: 3^(1/2)
sqrt(3)
sage: (3.0)^(1/2)
1.73205080756888
sage: 8^(1/2)
2*sqrt(2)
sage: 8^(1/3)
2
```

SageMath also has available all of the standard trigonometric functions: for sine and cosine we use `sin()` and `cos()`.

```
sage: sin(1)
sin(1)
sage: sin(1.0)
0.841470984807897
sage: cos(3/2)
cos(3/2)
sage: cos(3/2.0)
0.0707372016677029
```

Again we see the same behavior that we saw with `sqrt()`, SageMath will give us an exact answer. You might think that since there is no way to simplify `sin(1)`, why bother? Well, some expressions involving sine can indeed be simplified. For example, an important identity from geometry is $\sin(\pi/3) = \sqrt{3}/2$. SageMath has a built-in symbolic π , and understands this identity:

```
sage: pi
pi
sage: sin(pi/3)
1/2*sqrt(3)
```

When we type `pi` in SageMath we are dealing exactly with π , not some numerical approximation. However, we can call for a numerical approximation using the `n()` method:

```
sage: pi.n()
3.14159265358979
sage: sin(pi)
0
sage: sin(pi.n())
1.22464679914735e-16
```

We see that when using the symbolic `pi`, SageMath returns the exact result. However, when we use the approximation we get an approximation back. `e-16` is a shorthand for 10^{-16} and the number `1.22464679914735e-16` should

be zero, but there are errors introduced by the approximation. Here are a few examples of using the symbolic, precise π vs the numerical approximation:

```
sage: sin(pi/6)
1/2
sage: sin(pi.n()/6)
0.5000000000000000
sage: sin(pi/4)
1/2*sqrt(2)
sage: sin(pi.n()/4)
0.707106781186547
```

Continuing on with the theme, there are some lesser known special angles for which the value of sine or cosine can be cleverly simplified.

```
sage: sin(pi/10)
1/4*sqrt(5) - 1/4
sage: cos(pi/5)
1/4*sqrt(5) + 1/4
sage: sin(5*pi/12)
1/12*(sqrt(3) + 3)*sqrt(6)
```

Other trigonometric functions, the inverse trigonometric functions and hyperbolic functions are also available.

```
sage: arctan(1.0)
0.785398163397448
sage: sinh(9.0)
4051.54190208279
```

Similar to `pi` SageMath has a built-in symbolic constant for the number e , the base of the natural logarithm.

```
sage: e
e
sage: e.n()
2.71828182845905
```

While some might be familiar with using `ln(x)` for natural log and `log(x)` to represent logarithm base 10, in SageMath both represent logarithm base e . We may specify a different base as a second argument to the command: to compute $\log_b(x)$ in SageMath we use the command `log(x, b)`.

```
sage: ln(e)
1
sage: log(e)
1
sage: log(e^2)
2
sage: log(10)
log(10)
sage: log(10.0)
2.30258509299405
sage: log(100,10)
2
```

Exponentiation base e can be done using both the `exp()` function and by raising the symbolic constant `e` to a specified power.

```
sage: exp(2)
e^2
```

```
sage: exp(2.0)
7.38905609893065
sage: exp(log(pi))
pi
sage: e^(log(2))
2
```

Exercises:

1. Compute the floor and ceiling of 2.75.
2. Compute the logarithm base e of $1/1000000$, compute the logarithm base 10 of $1/1000000$, then compute the ratio. What should the answer be?
3. Compute the logarithm base 2 of 64.
4. Compare $e^{i\pi}$ with a numerical approximation of it using `pi.n()`.
5. Compute $\sin(\pi/2)$, $\cot(0)$ and $\csc(\pi/16)$.

2.2 Solving Equations and Inequalities

2.2.1 Solving for x

You should be familiar with “*Basic Arithmetic*” and “*Standard Functions and Constants*”

In SageMath, equations and inequalities are defined using the *operators* `==`, `<=`, and `>=` and will return either `True`, `False`, or, if there is a variable, just the equation/inequality.

```
sage: 9 == 9
True
sage: 9 <= 10
True
sage: 3*x - 10 == 5
3*x - 10 == 5
```

To solve an equation or an inequality we use using the, aptly-named, `solve()` command. For the moment, we will only solve for x . The section on variables below explains how to use other variables.

```
sage: solve(3*x - 2 == 5, x)
[x == (7/3)]
sage: solve( 2*x - 5 == 1, x)
[x == 3]
sage: solve( 2*x - 5 >= 17, x)
[[x >= 11]]
sage: solve( 3*x - 2 > 5, x)
[[x > (7/3)]]
```

Equations can have multiple solutions. SageMath returns all solutions found as a list.

```
sage: solve( x^2 + x == 6, x)
[x == -3, x == 2]
sage: solve(2*x^2 - x + 1 == 0, x)
[x == -1/4*I*sqrt(7) + 1/4, x == 1/4*I*sqrt(7) + 1/4]
sage: solve( exp(x) == -1, x)
[x == I*pi]
```

The solution set of certain inequalities consists of the union and intersection of open intervals.

```
sage: solve( x^2 - 6 >= 3, x )
[[x <= -3], [x >= 3]]
sage: solve( x^2 - 6 <= 3, x )
[[x >= -3, x <= 3]]
```

The `solve()` command will attempt to express the solution of an equation without the use of floating point numbers. If this cannot be done, it will return the solution in a symbolic form.

```
sage: solve( sin(x) == x, x )
[x == sin(x)]
sage: solve( exp(x) - x == 0 , x )
[x == e^x]
sage: solve( cos(x) - sin(x) == 0 , x )
[sin(x) == cos(x)]
sage: solve( cos(x) - exp(x) == 0 , x )
[cos(x) == e^x]
```

To find a numeric approximation of the solution we can use the `find_root()` command. Which requires both the expression and a closed interval on which to search for a solution.

```
sage: find_root(sin(x) == x, -pi/2 , pi/2)
0.0
sage: find_root(sin(x) == cos(x), pi, 3*pi/2)
3.9269908169872414
```

This command will only return one solution on the specified interval, if one exists. It will not find the complete solution set over the entire real numbers. To find a complete set of solutions, the reader must use `find_root()` repeatedly over cleverly selected intervals. Sadly, at this point, SageMath cannot do all of the thinking for us. This feature is not planned until SageMath 10.

2.2.2 Declaring Variables

In the previous section we only solved equations in one variable, and we always used x . When a session is started, SageMath creates one symbolic variable, x , and it can be used to solve equations. If you want to use an additional symbolic variable, you have to *declare it* using the `var()` command. The name of a symbolic variable can be a letter, or a combination of letters and numbers:

```
sage: y,z,t = var("y z t")
sage: phi, theta, rho = var("phi theta rho")
sage: x1, x2 = var("x1 x2")
```

Note: Variable names cannot contain spaces, for example “square root” is not a valid variable name, whereas “square_root” is.

Attempting to use a symbolic variable before it has been declared will result in a `NameError`.

```
sage: u
...
NameError: name 'u' is not defined
sage: solve (u^2-1,u)
NameError                                Traceback (most recent call last)
NameError: name 'u' is not defined
```

We can un-declare a symbolic variable, like the variable `phi` defined above, by using the `restore()` command.


```
sage: restore('phi')
sage: phi
...
NameError: name 'phi' is not defined
```

2.2.3 Solving Equations with Several Variables

Small systems of linear equations can be also solved using `solve()`, provided that all the symbolic variables have been declared. The equations must be input as a list, followed by the symbolic variables. The result may be either a unique solution, infinitely many solutions, or no solutions at all.

```
sage: solve([3*x - y == 2, -2*x - y == 1], x, y)
[[x == (1/5), y == (-7/5)]]
sage: solve([2*x + y == -1, -4*x - 2*y == 2], x, y)
[[x == -1/2*r1 - 1/2, y == r1]]
sage: solve([2*x - y == -1, 2*x - y == 2], x, y)
[]
```

In the second equation above, `r1` signifies that there is a free variable which parametrizes the solution set. When there is more than one free variable, SageMath enumerates them `r1, r2, ..., rk`.

```
sage: solve([2*x + 3*y + 5*z == 1, 4*x + 6*y + 10*z == 2, 6*x + 9*y + 15*z == 3], x,
→y, z)
[[x == -5/2*r1 - 3/2*r2 + 1/2, y == r2, z == r1]]
```

`solve()` can be very slow for large systems of equations. For these systems, it is best to use the linear algebra functions as they are quite efficient.

Solving inequalities in several variables can lead to complicated expressions, since the regions they define are complicated. In the example below, SageMath's solution is a list containing the point of intersection of the lines, then two rays, then the region between the two rays.

```
sage: solve([x-y >=2, x+y <=3], x, y)
[[x == (5/2), y == (1/2)], [x == -y + 3, y < (1/2)], [x == y + 2, y < (1/2)], [y + 2
→< x, x < -y + 3, y < (1/2)]]
sage: solve([2*x-y < 4, x+y > 5, x-y < 6], x, y)
[[-y + 5 < x, x < 1/2*y + 2, 2 < y]]
```

Exercises:

1. Find all of the solutions to the equation $x^3 - x = 7x^2 - 7$.
2. Find the complete solution set for the inequality $|t - 7| \geq 3$.
3. Find all x and y that satisfy both $2x + y = 17$ and $x - 3y = -16$.
4. Use `find_root()` to find a solution of the equation $e^x = \cos(x)$ on the interval $[-\pi/2, 0]$.
5. Change the command above so that `find_root()` finds the other solution in the same interval.

2.3 Calculus

You should be familiar with *Basic Arithmetic*, *Standard Functions and Constants*, and *Declaring Variables*

SageMath has many commands that are useful for the study of differential and integral calculus. We will begin our investigation of these command by defining a few functions that we will use throughout the chapter.

```
sage: f(x) = x*exp(x)
sage: f
x |--> x*e^x
sage: g(x) = (x^2)*cos(2*x)
sage: g
x |--> x^2*cos(2*x)
sage: h(x) = (x^2 + x - 2)/(x-4)
sage: h
x |--> (x^2 + x - 2)/(x-4)
```

SageMath uses `x |-->` to tell you that the expression returned is actually a function and not just a number or string. This means that we can *evaluate* these expressions just like you would expect of any function.

```
sage: f(1)
e
sage: g(2*pi)
4*pi^2
sage: h(-1)
2/5
```

With these functions defined, we will look at how we can use SageMath to compute the *limit* of these functions.

2.3.1 Limits

The limit of $f(x) = xe^x$ as $x \rightarrow 1$ is computed in SageMath by entering the following command into SageMath:

```
sage: limit(f, x=1)
e
```

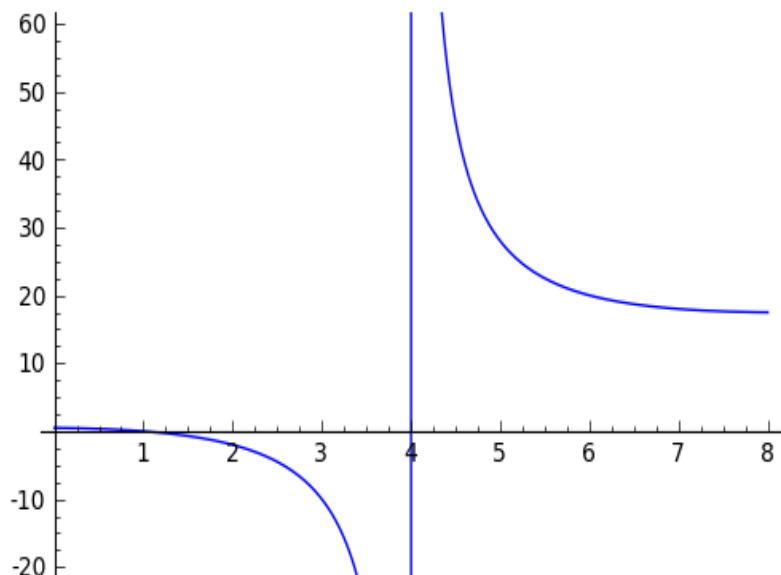
We can do the same with $g(x)$. To evaluate the limit of $g(x) = x^2 \cos(2x)$ as $x \rightarrow 2$ we enter:

```
sage: limit(g, x=2)
4*cos(4)
```

The functions $f(x)$ and $g(x)$ aren't all that exciting as far as limits are concerned since they are both *continuous* for all real numbers. But $h(x)$ has a discontinuity at $x = 4$, so to investigate what is happening near this discontinuity we will look at the limit of $h(x)$ as $x \rightarrow 4$:

```
sage: limit(h, x = 4)
Infinity
```

Now this is an example of why we have to be a little careful when using computer algebra systems. The limit above is not exactly correct. See the graph of $h(x)$ near this discontinuity below.



What we have when $x = 4$ is a *vertical asymptote* with the function tending toward *positive* infinity if x is larger than 4 and *negative* infinity from when x less than 4. We can take these *directional* limits using SageMath to confirm this by supplying the extra *dir* argument.

```
sage: limit(h, x=4, dir="right")
+Infinity
sage: limit(h, x=4, dir="left")
-Infinity
```

2.3.2 Derivatives

The next thing we are going to do is use SageMath to compute the *derivatives* of the functions that we defined earlier. For example, to compute $f'(x)$, $g'(x)$, and $h'(x)$ we will use the `derivative()` command.

```
sage: fp = derivative(f,x)
sage: fp
x |--> x*e^x + e^x
sage: gp = derivative(g, x)
sage: gp
x |--> -2*x^2*sin(2*x) + 2*x*cos(2*x)
sage: hp = derivative(h,x)
sage: hp
x |--> (2*x + 1)/(x - 4) - (x^2 + x - 2)/(x - 4)^2
```

The first argument is the function which you would like to differentiate and the second argument is the variable with which you would like to differentiate with respect to. For example, if I were to supply a different variable, SageMath will hold x constant and take the derivative with respect to that variable.

```
sage: y = var('y')
sage: derivative(f,y)
x |--> 0
sage: derivative(g,y)
x |--> 0
sage: derivative(h,y)
x |--> 0
```

The `derivative()` command returns another function that can be evaluated like any other function.

```
sage: fp(10)
11*e^10
sage: gp(pi/2)
-pi
sage:
sage: hp(10)
1/2
```

With the *derivative function* computed, we can then find the *critical points* using the `solve()` command.

```
sage: solve( fp(x) == 0, x)
[x == -1, e^x == 0]
sage: solve( hp(x) == 0, x)
[x == -3*sqrt(2) + 4, x == 3*sqrt(2) + 4]
sage: solve( gp(x) == 0, x)
[x == 0, x == cos(2*x)/sin(2*x)]
```

Constructing the line *tangent* to our functions at the point $(x, f(x))$ is an important computation which is easily done in SageMath. For example, the following command will compute the line tangent to $f(x)$ at the point $(0, f(0))$.

```
sage: T_f = fp(0)*( x - 0 ) + f(0)
sage: T_f
x
```

The same can be done for $g(x)$ and $h(x)$.

```
sage: T_g = gp(0)*( x - 0 ) + g(0)
sage: T_g
0
sage: T_h = hp(0)*( x - 0 ) + h(0)
sage: T_h
-1/8*x + 1/2
```

2.3.3 Integrals

SageMath has the facility to compute both *definite* and *indefinite* integral for many common functions. We will begin by computing the *indefinite* integral, otherwise known as the *anti-derivative*, for each of the functions that we defined earlier. This will be done by using the `integral()` command which has arguments that are similar to `derivative()`.

```
sage: integral(f,x)
x |--> (x - 1)*e^x
sage: integral(g, x)
x |--> 1/4*(2*x^2 - 1)*sin(2*x) + 1/2*x*cos(2*x)
sage: integral(h, x)
x |--> 1/2*x^2 + 5*x + 18*log(x - 4)
```

The function that is returned is only *one* of the many anti-derivatives that exist for each of these functions. The others differ by a constant. We can verify that we have indeed computed the *anti-derivative* by taking the derivative of our indefinite integrals.

```
sage: derivative(integral(f,x), x)
x |--> (x - 1)*e^x + e^x
sage: f
```

```
x |--> x*e^x
sage: derivative(integral(g,x), x )
x |--> 1/2*(2*x^2 - 1)*cos(2*x) + 1/2*cos(2*x)
sage: derivative(integral(h,x), x )
x |--> x + 18/(x - 4) + 5
```

Wait, none of these look right. But a little algebra, and the use of a trig-identity or two in the case of $1/2*(2*x^2 - 1)*\cos(2*x) + 1/2*\cos(2*x)$, you will see that they are indeed the same.

It should also be noted that there are some functions which are continuous and yet there doesn't exist a *closed form* integral. A common example is e^{-x^2} which forms the basis for the *normal distribution* which is ubiquitous throughout statistics. The antiderivative for er^{-x^2} is commonly called erf, otherwise known as the *error function*.

```
sage: y(x) = exp(-x^2)
sage: integral(y,x)
x |--> 1/2*sqrt(pi)*erf(x)
```

We can also compute the *definite* integral for the functions that we defined earlier. This is done by specifying the *limits of integration* as addition arguments.

```
sage: integral(f, x, 0, 1)
x |--> 1
sage: integral(g, x, 0, 1)
x |--> 1/4*sin(2) + 1/2*cos(2)
sage: integral(h, x, 0, 1)
x |--> 18*log(3) - 18*log(4) + 11/2
```

In each case above, SageMath returns a *function* as its result. Each of these functions is a constant function, which is what we would expect. As it was pointed out earlier, SageMath will return the expression that retains the most precision and will not use decimals unless told to. A quick way to tell SageMath that an approximation is desired is wrap the `integrate()` command with `n()`, the numerical approximation command.

```
sage: n(integral(f, x, 0, 1))
1.0000000000000000
sage: n(integral(g, x, 0, 1))
0.0192509384328492
sage: n(integral(h, x, 0, 1))
0.321722695867944
```

2.3.4 Taylor Series Expansion

Another interesting feature of SageMath is the possibility of computing *Taylor Series* expansions around a point. At first we show how to expand around 0, also called Mclaurin series. Let us give an example with the function $g(x) = \cos(x)$.

```
sage: g = cos(x); g
cos(x)
sage: g_taylor = g.taylor(x, 0, 3)
-1/2*x^2 + 1
```

The first argument in `g.taylor()` is the independent variable of our function, the second argument is the point around which we are expanding, and the third argument is the order of accuracy of the expansion, i.e. where we truncate it. The above case was a pretty simple one, but sometimes you might have to compute a much harder Taylor expansion. For instance, let us try to expand the function $f(x) = \exp(x^2)\sin(x - 5)$ around the point $x = 2$ up to order 3.

```
sage: f = exp(x^3)*sin(x-5); f
e^(x^3)*sin(x - 5)
sage: f_taylor = f.taylor(x,2,3); f_taylor
1/6*(x - 2)^3*(467*cos(3) - 2130*sin(3))*e^8 + 1/2*(x - 2)^2*(24*cos(3) -
↳155*sin(3))*e^8 + (x - 2)*(cos(3) - 12*sin(3))*e^8 - e^8*sin(3)
```

Now the outcome of this computation might be a bit convoluted to visualize, therefore, another interesting feature of SageMath is the possibility of printing the outcome in **Latex** format, which is much nicer to the eye.

```
sage: print; show(f_taylor)
```

$$\frac{1}{6}(x-2)^3(467\cos(3) - 2130\sin(3))e^8 + \frac{1}{2}(x-2)^2(24\cos(3) - 155\sin(3))e^8 + (x-2)(\cos(3) - 12\sin(3))e^8 - e^8\sin(3)$$

Exercises:

1. Use SageMath to compute the following limits:

- $\lim_{x \rightarrow 2} \frac{x^2 + 2x - 8}{x - 2}$
- $\lim_{x \rightarrow (\pi/2)^+} \sec(x)$
- $\lim_{x \rightarrow (\pi/2)^-} \sec(x)$

2. Use SageMath to compute the following *derivatives* with respect to the specified variables:

- $\frac{d}{dx} [x^2 e^{3x} \cos(2x)]$
- $\frac{d}{dt} \left[\frac{t^2 + 1}{t - 2} \right]$ (*remember to define ‘t’*)
- $\frac{d}{dy} [x \cos(x)]$

3. Use SageMath to compute the following integrals:

- $\int \frac{x+1}{x^2+2x+1} dx$
- $\int_{-\pi/4}^{\pi/4} \sec(x) dx$
- $\int x e^{-x^2} dx$

4. Use SageMath to compute the Taylor series expansion of the following functions:

- $\sin(x) * \cos(x)$ around $x = 0$, order 3
- $(x - 2) * \ln(x/2)$ around $x = 1$, order 4
- $\tan(x + 5) - \exp(x^2)$, around $x = \pi$, order 5

2.4 Statistics

You should be familiar with *Basic Arithmetic*

In this section we will discuss the use of some of the basic descriptive statistic functions available for use in SageMath.

To demonstrate their usage we will first generate a pseudo-random list of integers from 0 to 100 to describe. The `random()` function generates a random number from $[0, 1)$, so we will use a trick to generate integers in this specific range. Note, by the nature of random number generation your list of numbers will be different.

```
sage: data = [ int(random()*(100-0) + 0) for i in [ 1 .. 20 ] ]
sage: data
[78, 43, 6, 50, 47, 94, 37, 70, 66, 32, 1, 34, 93,
30, 99, 82, 22, 74, 18, 40]
```

We can compute the mean, median, mode, variance, and standard deviation of this data.

```
sage: mean(data)
254/5
sage: median(data)
45
sage: mode(data)
[32, 1, 66, 99, 82, 37, 6, 40, 74, 43,
34, 78, 47, 50, 30, 22, 18, 70, 93, 94]
sage: variance(data)
83326/95
sage: std(data)
sqrt(83326/95)
```

Note that both the standard deviation and variance are computed in their unbiased forms. If we want to bias these measures then you can use the `bias=True` option.

We can also compute a rolling, or moving, average of the data with the `moving_average()`.

```
sage: moving_average(data, 4)
[177/4, 73/2, 197/4, 57, 62, 267/4, 205/4, 169/4,
133/4, 40, 79/2, 64, 76, 233/4, 277/4, 49, 77/2]
sage: moving_average(data, 10)
[523/10, 223/5, 437/10, 262/5, 252/5, 278/5,
272/5, 529/10, 533/10, 97/2, 493/10]
sage: moving_average(data, 20)
[254/5]
```

Exercises:

1. Use SageMath to generate a list of 20 random integers.
2. The heights of eight students, measured in inches, are 71, 73, 59, 62, 65, 61, 73, 61. Find the *average*, *median* and *mode* of the heights of these students.
3. Using the same data, compute the *standard deviation* and *variance* of the sampled heights.
4. Find the *range* of the heights. (*Hint: use the `max()` and `min()` commands*)

2.5 Plotting

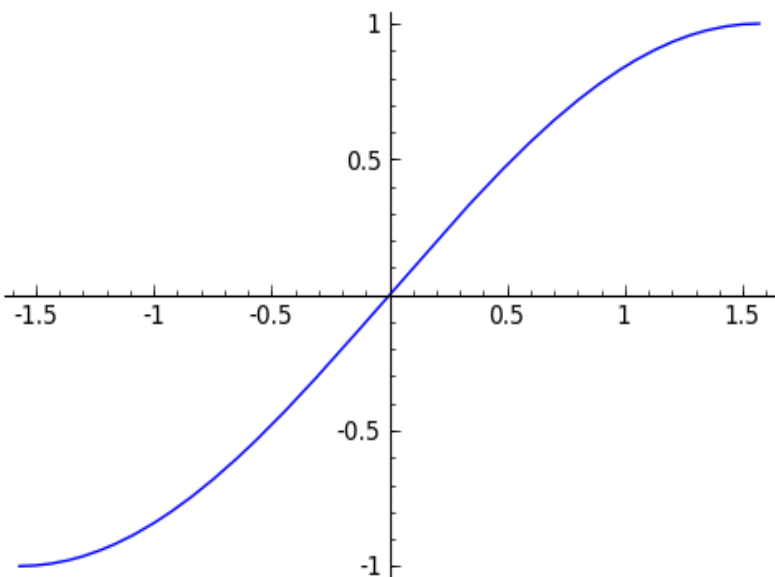
2.5.1 2D Graphics

You should be familiar with *Standard Functions and Constants* and *Solving Equations and Inequalities*

SageMath has many ways for us to visualize the mathematics that we are working with. In this section we will quickly get you up to speed with some of the basic commands used when plotting functions and working with graphics.

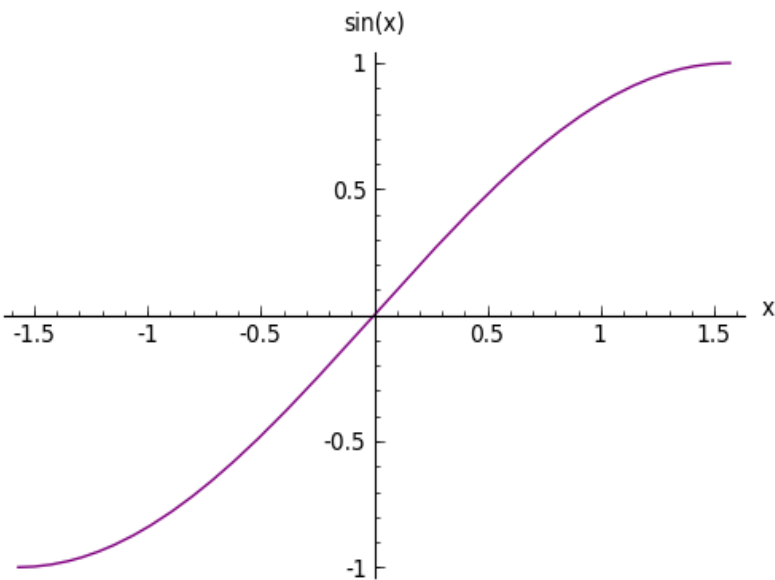
To produce a basic plot of $\sin(x)$ from $x = -\frac{\pi}{2}$ to $x = \frac{\pi}{2}$ we will use the `plot()` command.

```
sage: f(x) = sin(x)
sage: p = plot(f(x), (x, -pi/2, pi/2))
sage: p.show()
```



By default, the plot created will be quite plain. To add axis labels and make our plotted line purple, we can alter the plot attribute by adding the `axes_labels` and `color` options.

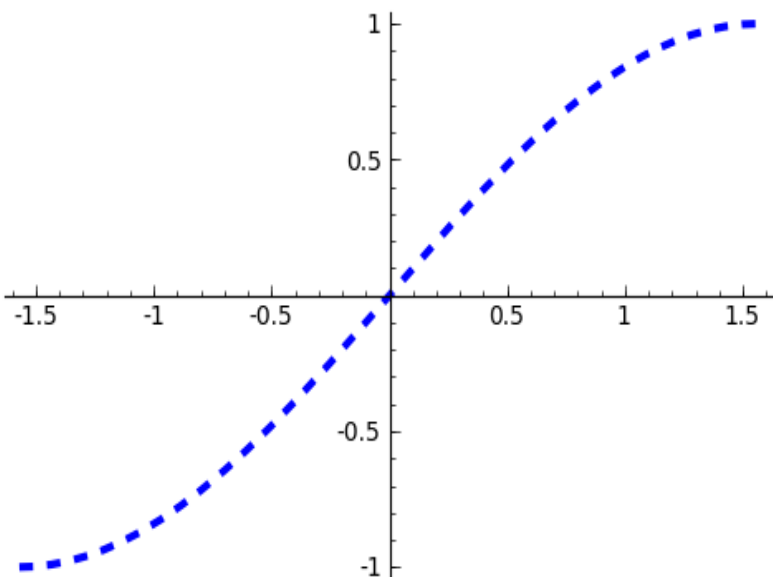
```
sage: p = plot(f(x), (x, -pi/2, pi/2), axes_labels=['x', 'sin(x)'], color='purple')
sage: p.show()
```



The `color` option accepts string color designations ('purple', 'green', 'red', 'black', etc...), an RGB triple such as (.25,.10,1), or an HTML-style hex triple such as #ff00aa.

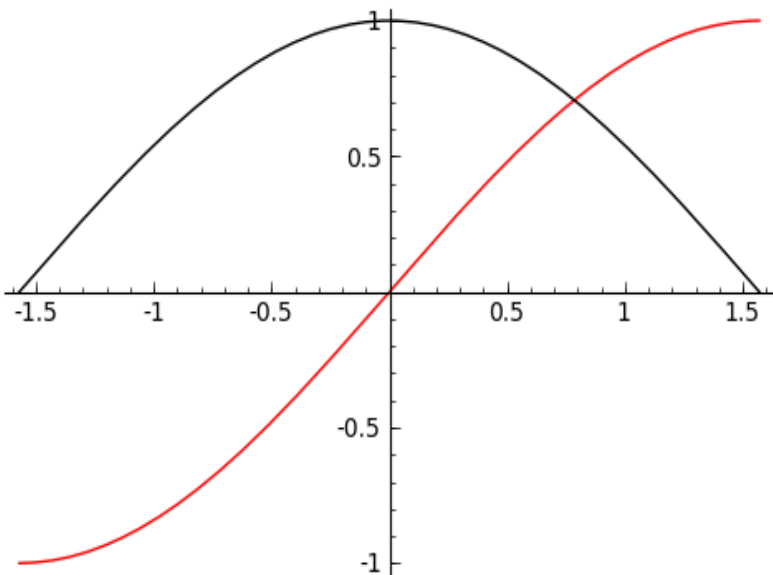
We can change the style of line, whether it is solid, dashed, and its thickness by using the `linestyle` and the `thickness` options.

```
sage: p = plot(f(x), (x, -pi/2, pi/2), linestyle='--', thickness=3)
sage: p.show()
```

We can display the graphs of two functions on the same axes by adding the plots together.

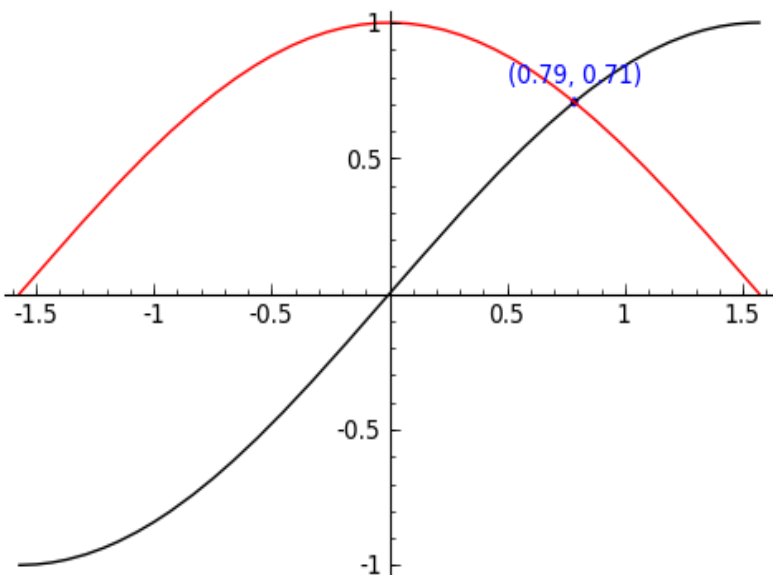
```
sage: f(x) = sin(x)
sage: g(x) = cos(x)
sage: p = plot(f(x), (x, -pi/2, pi/2), color='black')
sage: q = plot(g(x), (x, -pi/2, pi/2), color='red')
sage: r = p + q
sage: r.show()
```



To tie together our plotting commands with some material we have learned earlier, let's use the `find_root()` command to find the point where $\sin(x)$ and $\cos(x)$ intersect. We will then add this point to the graph and label it.

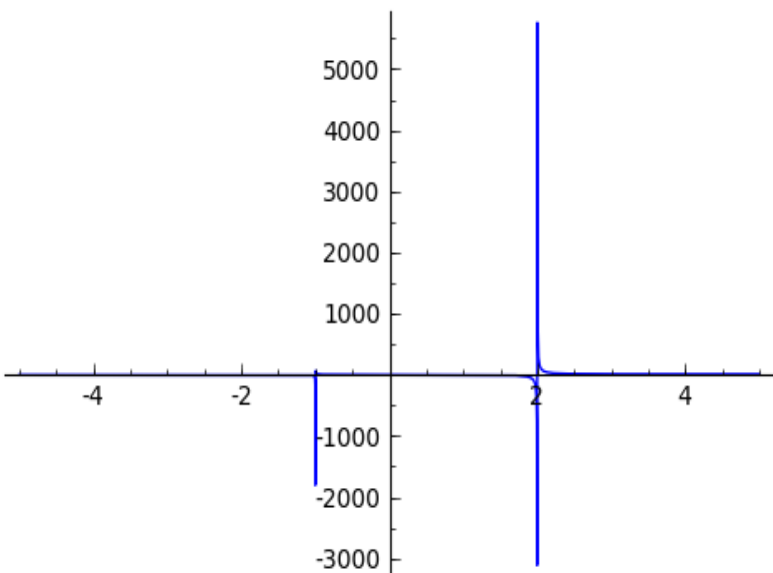
```
sage: find_root( sin(x) == cos(x), -pi/2, pi/2 )
0.78539816339744839
sage: P = point( [(0.78539816339744839, sin(0.78539816339744839))] )
sage: T = text("(0.79,0.71)", (0.78539816339744839, sin(0.78539816339744839) + .10))
```

```
sage: s = P + r + T
sage: s.show()
```



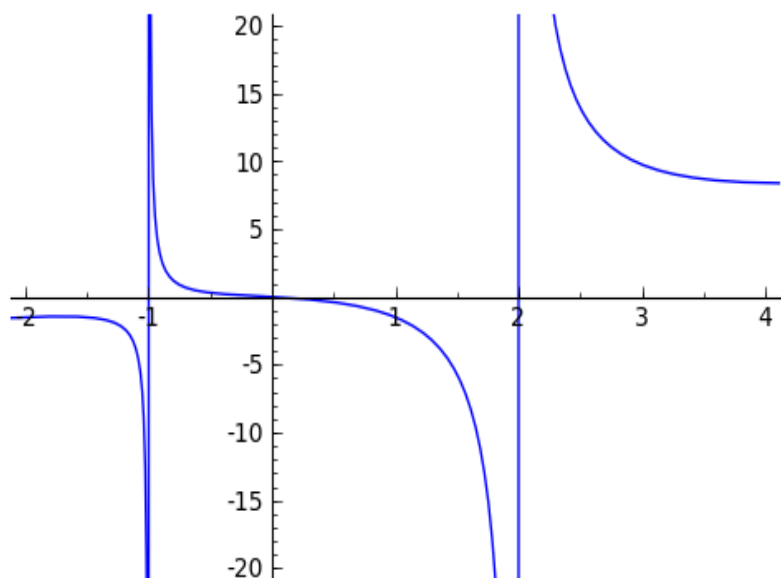
SageMath handles many of the details of producing “nice” looking plots in a way that is transparent to the user. However there are times in which SageMath will produce a plot which isn’t quite what we were expecting.

```
sage: f(x) = (x^3 + x^2 + x)/(x^2 - x - 2)
sage: p = plot(f(x), (x, -5, 5))
sage: p.show()
```



The vertical asymptotes of this rational function cause SageMath to adjust the aspect ratio of the plot to display the rather large y values near $x = -1$ and $x = 2$. This obfuscates most of the features of this function in a way that we may have not intended. To remedy this we can explicitly adjust the vertical and horizontal limits of our plot

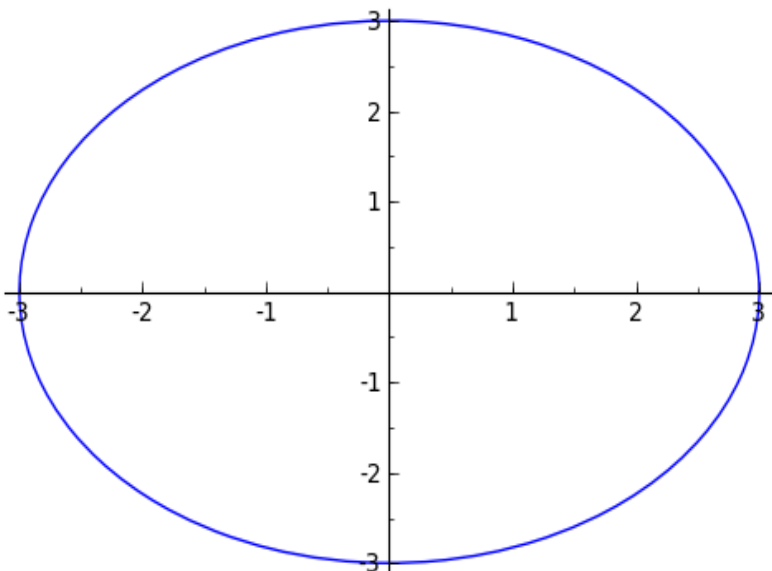
```
sage: p.show(xmin=-2, xmax=4, ymin=-20, ymax=20)
```



This, in the author's opinion, displays the features of this particular function in a much more pleasing fashion.

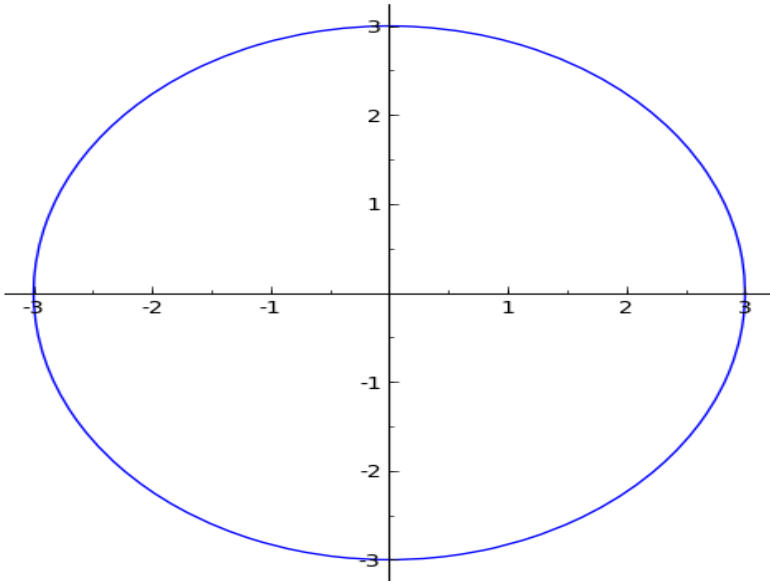
SageMath can handle parametric plots with the `parametric_plot()` command. The following is a simple circle of radius 3.

```
sage: t = var('t')
sage: p = parametric_plot( [3*cos(t), 3*sin(t)], (t, 0, 2*pi) )
sage: p.show()
```



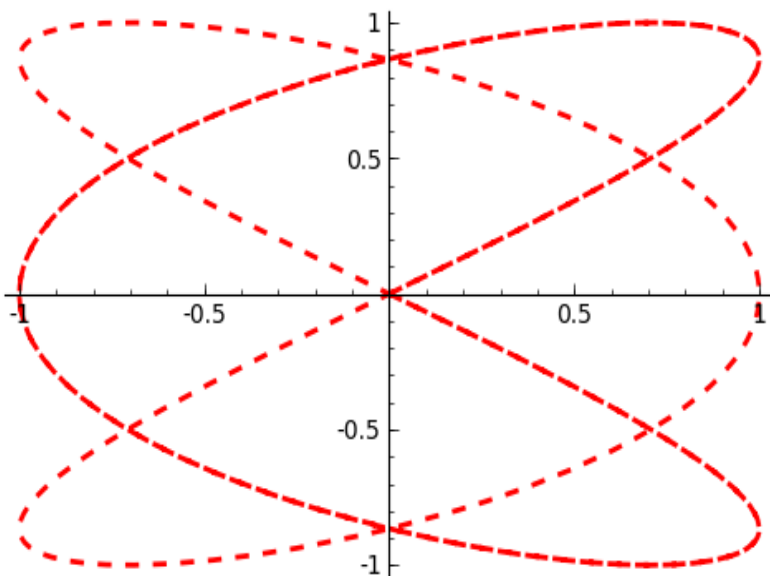
The default choice of aspect ratio makes the plot above decidedly “un-circle like”. We can adjust this by using the `aspect_ratio` option.

```
sage: p.show(aspect_ratio=1)
```



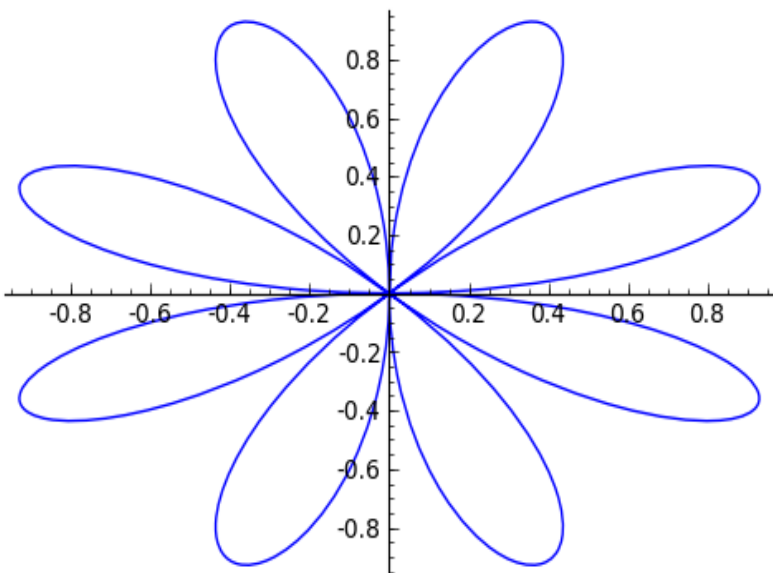
The different plotting commands accept many of the same options as plot. The following generates the Lissajous Curve $L(3, 2)$ with a thick red dashed line.

```
sage: p = parametric_plot( [sin(3*t), sin(2*t)], (t, 0, 3*pi), thickness=2, color='red
↪', linestyle="--")
sage: p.show()
```



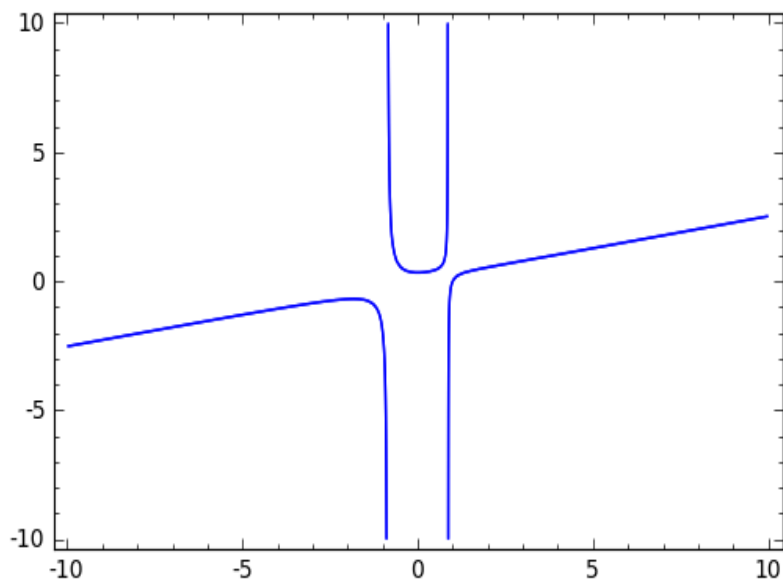
Polar plots can be done using the `polar_plot()` command.

```
sage: theta = var("theta")
sage: r(theta) = sin(4*theta)
sage: p = polar_plot((r(theta)), (theta, 0, 2*pi) )
sage: p.show()
```



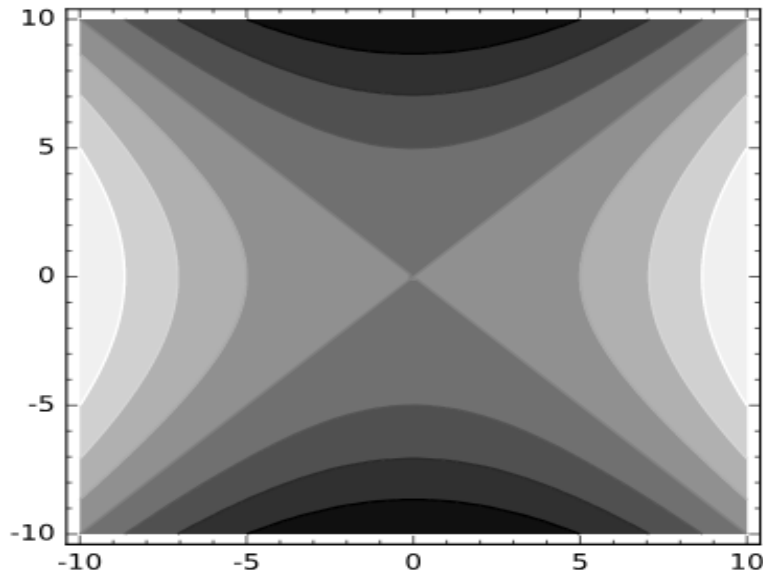
And finally, SageMath can do the plots for functions that are implicitly defined. For example, to display all points (x, y) that satisfy the equation $4x^2y - 3y = x^3 - 1$, we enter the following:

```
sage: implicit_plot(4*x^2*y - 3*y == x^3 - 1, (x,-10,10), (y,-10,10))
```



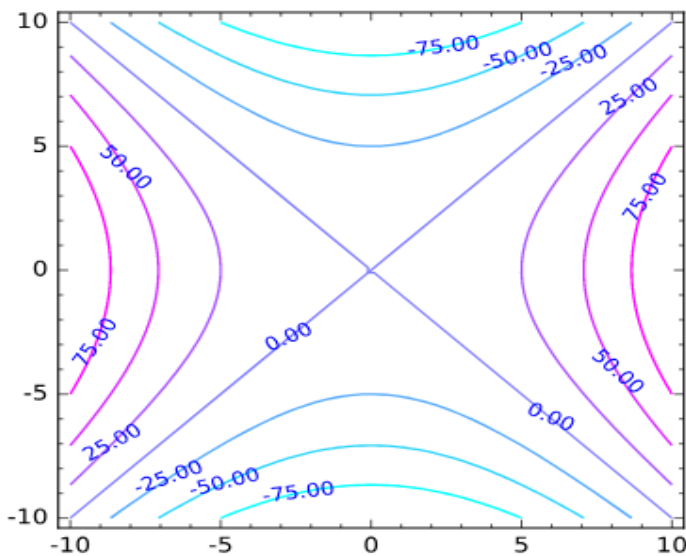
As we transition from 2D-plotting to 3D-plotting, it is worthwhile to briefly mention [contour lines](#). If you are familiar with some basic notions of Multivariable Calculus, you know that a contour line is a curve where a function of two variables holds constant value. Their plots are often useful to gather information about the function itself. Contour lines find use in a variety of fields, from [cartography](#) to [meteorology](#). Here we pick a trivial example for pedagogical purposes: an hyperbola.

```
sage: x,y = var("x,y")
sage: f(x,y) = x^2-y^2
sage: contour = contour_plot(f, (x, -10,10), (y,-10,10))
sage: contour.show()
```



Now, the above picture is not really pleasant to the eye and not very informative either. Thankfully we can customize our contour plot to make it look nicer and clearer.

```
sage: x,y = var("x,y")
sage: f(x,y) = x^2-y^2
sage: contour = contour_plot(f, (x, -10,10), (y,-10,10), cmap='cool', labels=True,
↪fill=False)
sage: contour.show()
```

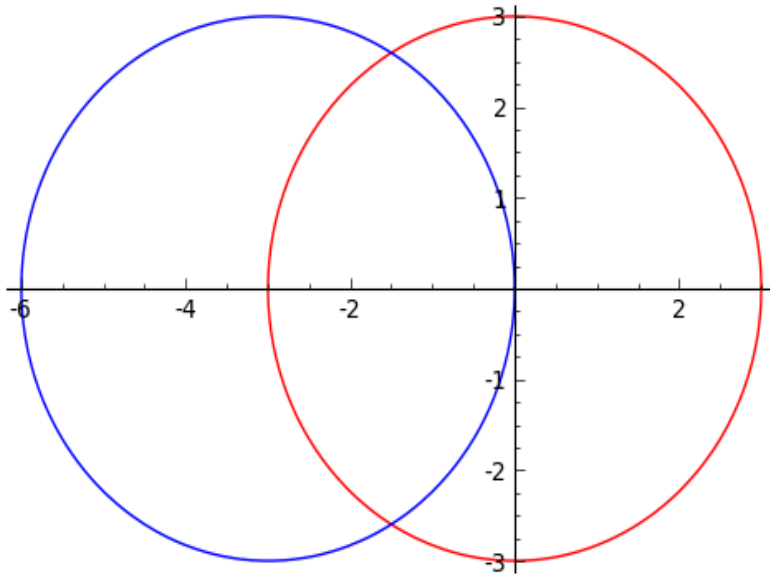


Now, we have printed the values of the function at each contour line and the plot is much clearer. You are free to check for other custom colors [here](#).

****Exercises:****

1. Plot the graph of $y = \sin(\pi x - \pi)$ for $-1 \leq x \leq 1$ using a thick red line.
2. Plot the graph of $\cos(\pi x - \pi)$ on the same interval using a thick blue line.

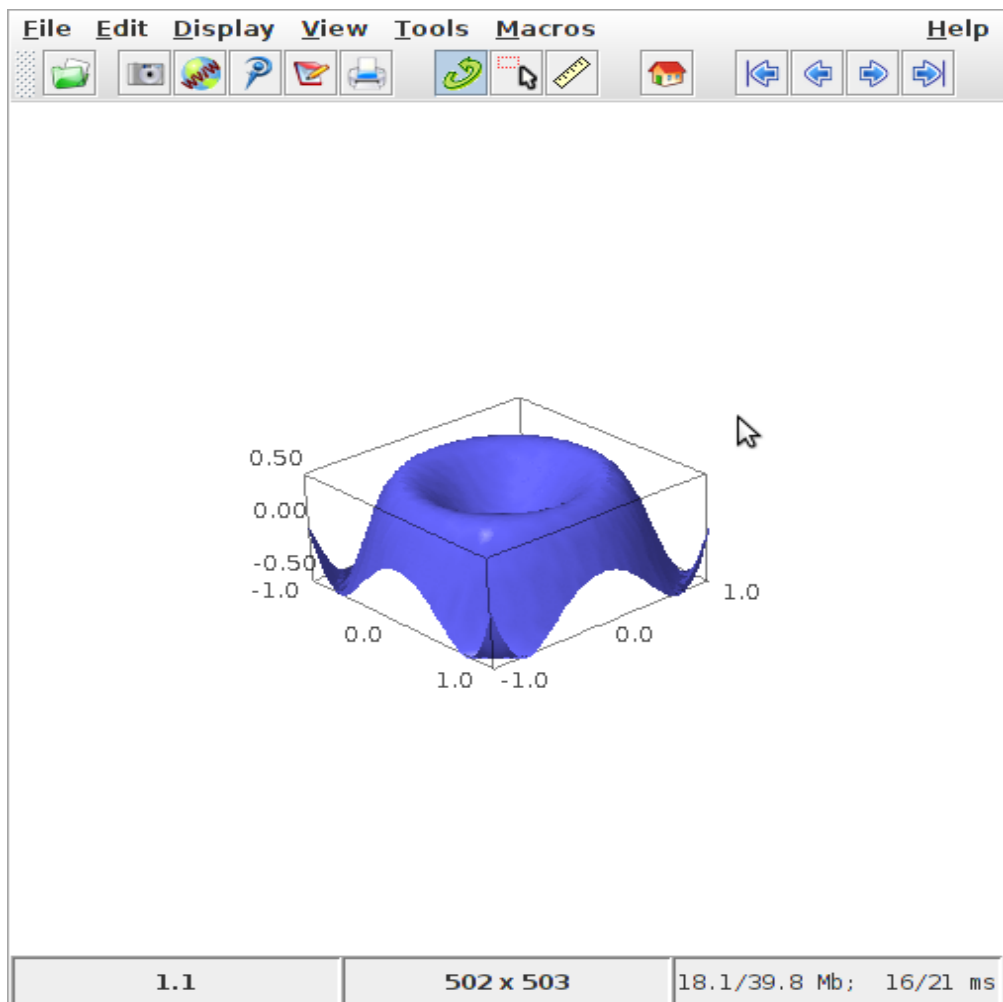
3. Plot the two graphs above on the same set of axes.
4. Plot the graph of $y = 1/x$ for $-1 \leq x \leq 1$ adjusting the range so that only $-10 \leq y \leq 10$.
5. Use the commands in this section to produce the following image:



2.5.2 3D Graphics

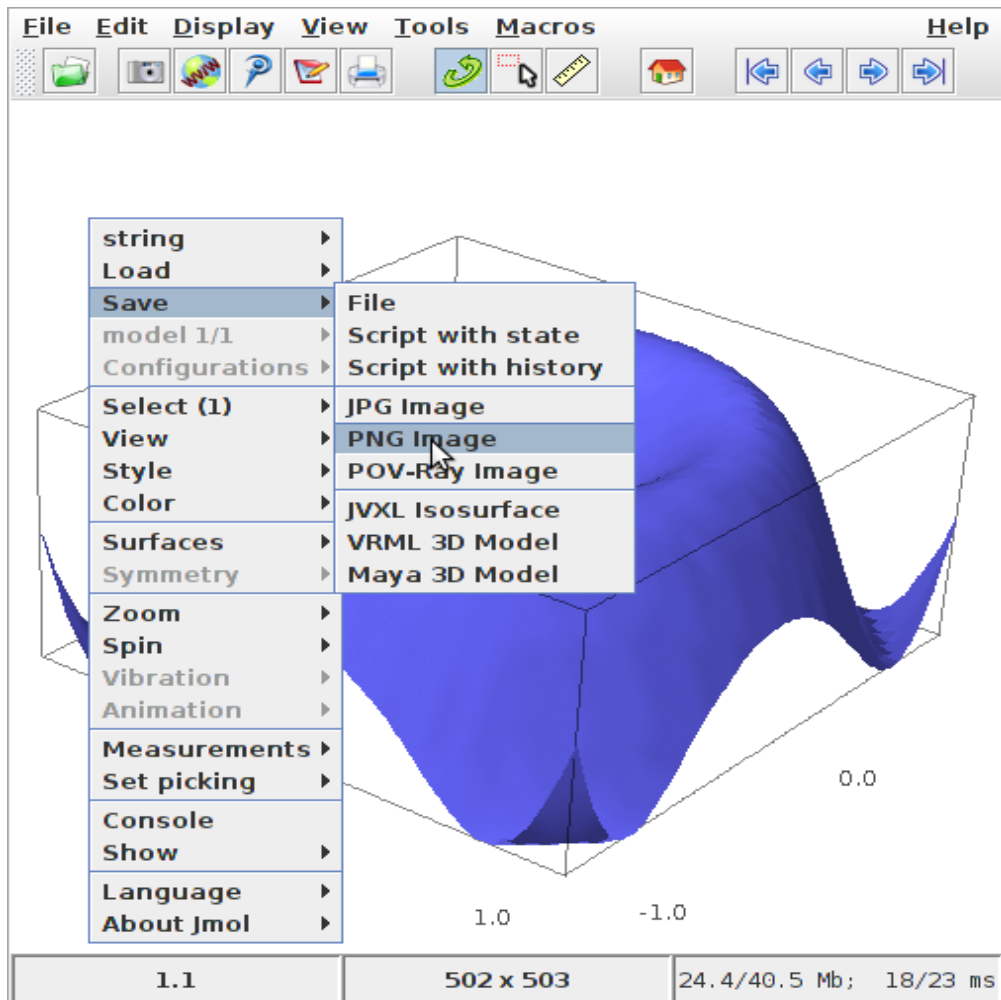
Producing 3D plots can be done using the `plot3d()` command

```
sage: x, y = var("x y")
sage: f(x, y) = x^2 - y^2
sage: p = plot3d(f(x, y), (x, -10, 10), (y, -10, 10))
sage: p.show()
```



SageMath handles 3d plotting a bit differently than what we have seen thus far. It uses a program named jmol to generate interactive plots. So instead of just a static picture we will see either a window like pictured above or, if you are using SageMath's notebook interface, a java applet in your browser's window.

One nice thing about the way that SageMath does this is that you can rotate your plot by just clicking on the surface and dragging it in the direction in which you would like for it to rotate. Zooming in/out can also be done by using your mouse's wheel button (or two-finger vertical swipe on a mac). Once you have rotated and zoomed the plot to your liking, you can save the plot as a file. Do this by right-clicking anywhere in the window/applet and selecting save, then png-image as pictured below



Note: If you are running SageMath on windows or on sagemb.org that your file will be saved either in your VMware virtual machine or on sagemb.org.

PROGRAMMING IN SAGEMATH

This part of the tutorial covers the essential programming tools that you need to use in order to do more advanced mathematics. The first chapter, on SageMath objects, is essential before moving on to study mathematical structures. The second chapter is more specifically about programming: conditionals and iterative loops, creating your own commands and saving your work. It is not necessary for basic computations with mathematical structures, but becomes invaluable for doing in-depth work. The third chapter explains how to interact with some of the main mathematical software packages included in SageMath. Finally there is a brief chapter on interactive use of SageMath.

3.1 SageMath Objects

3.1.1 Universes and Coercion

A key concept in SageMath is the *universe* of an object. The most effective way to gain a familiarity with *universes* and the related concept, *coercion*, is to see a few examples. We begin with the most common universes: the integers, and the rational, real and complex numbers.

In SageMath, `ZZ` indicates the universe where the Integers live, while `QQ`, `RR` and `CC` indicate the universes of the Rationals, Real and Complex numbers, respectively.

```
sage: ZZ
Integer Ring
sage: QQ
Rational Field
sage: RR
Real Field with 53 bits of precision
sage: CC
Complex Field with 53 bits of precision
```

We can check if a given objects *live* in a universe using the `in` operator.

```
sage: 1 in ZZ
True
sage: 1/2 in ZZ
False
sage: 1/2 in QQ
True
sage: sqrt(2) in QQ
False
sage: sqrt(2) in RR
True
```

The letter `I` in SageMath is the square root of `-1` (`i` also works).

```
sage: i^2
-1
sage: i^3
-I
sage: I in RR
False
sage: I in CC
True
```

To directly check which universe a number is in, we use the `parent()` function. SageMath will choose the simplest universe for each number.

```
sage: parent(1)
Integer Ring
sage: parent(1/2)
Rational Field
sage: parent(5.7)
Real Field with 53 bits of precision
sage: parent(pi.n())
Real Field with 53 bits of precision
```

Another important universe is the Symbolic Ring. You might think that $\sqrt{2}$ or π would have parent RR, the real numbers, while I would be in CC. But RR and CC have finite precision, and these numbers satisfy formulas that make them special, for example $\sqrt{2}^2 = 2$ and $\sin(\pi) = 0$. SageMath stores these numbers with special properties in the so-called *Symbolic Ring*, whose variables are aptly-named *symbolic variables*, see “[Variables](#)”.

```
sage: parent(sqrt(2))
Symbolic Ring
sage: parent(I)
Symbolic Ring
sage: parent(pi)
Symbolic Ring
```

We often perform operations with elements from *different* universes as long as there is some sort of natural *conversion* that can be done to both elements so that they live in the *same* universe. For instance, when we compute $1 + 1/2 = 3/2$, we implicitly convert 1 from the Integer universe to the universe of rational numbers, before performing the operation. This conversion is often so natural that we don’t even think about it and, luckily for you, SageMath does many of these conversions without you having to worry about them either.

```
sage: parent(1 + 2)
Integer Ring
sage: parent(1/2 + 2)
Rational Field
sage: parent(1/2 + 2.0)
Real Field with 53 bits of precision
```

SageMath’s treatment of symbolic constants like `pi` is worth-mentioning in its own right. For example, here’s what happens when we mix `pi` with a decimal.

```
sage: exp(1.)*pi
2.71828182845905*pi
sage: parent(exp(1.)*pi)
Symbolic Ring
```

SageMath will always choose the universe which offers the most precision, and the same will be true for other symbolic constants like `e` and `i`, as well for the polynomial indeterminate `x`.

```
sage: parent(2 + i)
Symbolic Ring
sage: parent(2 + x)
Symbolic Ring
sage: parent(2 + 2.0*x)
Symbolic Ring
sage: parent(2*pi + 2.0*e)
Symbolic Ring
```

What if we want to convert a number from a universe to another? Luckily, we can easily accomplish that through a process called *coercion*. We coerce a number into another universe, if it makes sense, by *applying* the parent structure to the object like it was a function. For example:

```
sage: QQ(.5)
1/2
sage: parent(QQ(.5))
Rational Field
sage: RR(sqrt(2))
1.41421356237310
sage: parent(RR(sqrt(2)))
Real Field with 53 bits of precision
```

And in case we try to make *some* nonsensical conversions, SageMath will raise a `TypeError`.

```
sage: QQ(i)
ERROR: An unexpected error occurred while tokenizing input
The following traceback may be corrupted or invalid
The error message is: ('EOF in multi-line statement', (1170, 0))
-----
TypeError                                 Traceback (most recent call last)
... * a lot of noise *
TypeError: Unable to coerce I to a rational
```

Exercises:

1. What *universe* does x live in by default? When you declare a new variable y where does it live?
2. Find the universe of the following expressions:
 - (a) $1 + 1/2$
 - (b) $1 + 1/2.0$
 - (c) $1/2 + i$
 - (d) $e + \pi$
 - (e) $e.n() + \pi$
 - (f) $e.n() + \pi.()$
3. For which of the following does the *coercion* make sense?
 - (a) $RR(1/2)$
 - (b) $QQ(1)$
 - (c) $ZZ(1/2)$
 - (d) $SR(1/2)$ (SR is the *Symbolic Ring*)
 - (e) $CC(x)$

4. If I enter $x=1/2$ into SageMath, what *universe* does x live in?

3.1.2 Booleans

Another important universe is the **Booleans**. The Boolean universe is just known as *bool* in SageMath, and it contains just two elements `True` and `False`.

```
sage: parent(True)
<type 'bool'>
```

There are several *logical* operations on Booleans (i.e. operations like *and*, *or* on `True` and `False`, instead of the operations like $+$, $*$ on numbers). We *negate* a Boolean by using the `not` operator.

```
sage: not True
False
sage: not False
True
```

Suppose we want to combine two Booleans X and Y . To accomplish that, we will use *and/or*.

- (X and Y) is `True` if both X and Y are `True`. If either X or Y is `False`, then (X and Y) is `False`.
- (X or Y) is `True` if either X or Y is `True`.

The following example will show exactly that.

```
sage: True and False
False
sage: True and True
True
sage: True or False
True
sage: False or False
False
```

Above we have a list of *truth statements*. To control their order of evaluation, we can use parentheses.

```
sage: (True or False) and False
False
sage: True or (False and False)
True
```

In the first example (`True or False`) is evaluated to be `True` first, then `True and False` evaluates to be `False`. In the second example, (`False and False`) evaluates to be `False`, but `True or False` is `True`.

Another important operator on Booleans is the **exclusive or** operator, represented by `^^` in SageMath. ($X \text{ ^^ } Y$) is `True` if exactly one between X and Y is `True`, and the other is `False`; otherwise it is `False`.

```
sage: True ^^ True           # xor (exclusive or) operator
False
sage: True ^^ False
True
sage: False ^^ False
False
```

To check if two objects are equal we use the `==` operator. The result is a Boolean:

```
sage: 1 == 1
True
sage: 1 == 0
False
sage: not (True or False) == True and False
True
```

Please notice that we used two equal signs, not one! To check if two things are not equal, we have two options: The `!=` operator and the `<>` operator.

```
sage: 1 != 1
False
sage: 1 != 0
True
sage: 1 <> 0
True
```

If two objects belong to a universe that has an ordering, then we may compare two elements of the universe using `<` and `>` and get a Boolean output. Additionally, we use `>=` for greater-than-or-equal-to and `<=` for less-than-or-equal-to.

```
sage: 1 > 2
False
sage: 2 > 1
True
sage: 4.1 < 5.7
True
sage: 6 < 5
False
sage: 1 >= .99999
True
sage: 1 <= 35
True
```

Exercises:

- Test to see if the following expressions are True, False, or not defined:
 - `not (True or False) == (False and True)`
 - `1 >= 1`
 - `1 + i >= 2 - i`
 - `((3/2) > 1) or (2/3 < 1)`
 - `((3/2) > 1) ^^ (2/3 < 1)`
 - `x > 1/2`
- What is the parent of `x > 1/2`? Why do you think that SageMath treats this expression differently from the rest?
- Use SageMath to find out if e is greater than π ? (*Hint: Remember that both “ e ” and “ π ” are symbolic variables by default.*)

3.1.3 Variables

You should be familiar with “*Declaring Variables*”

The term ‘variable’, can hold different meanings. For instance, in computer programming, a ‘variable’ is a space in memory used to store and retrieve a certain piece of information. In mathematics, a variable such as x is a quantity with indeterminate value: a symbol that we can manipulate with the same rules of arithmetic that are applied to numbers.

In SageMath, both usages are present. We will use the term *variable* for the computer programming variable and *symbolic variable* for the mathematical variable.

SageMath initializes the Symbolic Ring to have one symbolic variable, x . It obeys the arithmetical rules that we expect.

```
sage: 3*x - x
2*x
sage: e*e^x
e^(x + 1)
```

If we need another symbolic variable, we have to declare it, using the `var()` command.

```
sage: e^x*e^y
-----
NameError                                Traceback (most recent call last)

/Users/mosullivan/<ipython console> in <module>()

NameError: name 'y' is not defined
sage: var("y")
y
sage: e^x*e^y
e^(x + y)
sage:
```

Now, let’s look at variables, which are used to store a particular number.

```
sage: m=2^19-1
sage: m
524287
sage: (m+1).factor()
2^19
```

We use an `=` to assign the value on the right to the variable on the left. Having declared a variable, we can reference it by using its name, as seen above.

SageMath allows us to re-assign a different value to a variable.

```
sage: s=12
sage: s
12
sage: s=34
sage: s
34
```

The order of operations in SageMath allows us to reference a variable while assigning it a new value. For instance, we can *increment* the variable `t` by doing the following:

```
sage: t=7
sage: t=t+1
sage: t
8
```

SageMath also offers us a convenient way to assign values to multiple variables at once.


```
sage: a,b=1,2
sage: a
1
sage: b
2
```

Additionally, we can display a sequence of variables using commas.

```
sage: c,d,e=2,3,5
sage: c,d,e
(2, 3, 5)
```

If we are assigning several variables at a time, and for some reason we wish to skip a value on the right-hand side, we may use an underscore on the left hand side. For example,

```
sage: a,_,c=1,2,3
sage: a
1
sage: c
3
sage: _,r = divmod(19,5)
sage: r
4
```

There is also a quick way to initialize two variables with the same value. We do this by just *chaining* together the assignment.

```
sage: a = b = 1
sage: a
1
sage: b
1
```

When you define either a variable or a symbolic variable, it will stay in memory until you quit your session. Sometimes we would like to restore a variable back to its default value. We do this with the `restore()` command.

```
sage: x = 1
sage: a = 2
sage: restore('x')
sage: restore('a')
sage: x
x
sage: a
-----
NameError                                Traceback (most recent call last)
/home/ayeq/sage/local/lib/python2.6/site-packages/sage/all_cmdline.pyc in <module>()
NameError: name 'a' is not defined
```

You can *reset* the entire environment to its defaults by running the `reset()` command.

```
sage: a = 1
sage: b = 2
sage: c = 5
sage: x = 56
sage: reset()
sage: a
-----
NameError                                Traceback (most recent call last)
```

```
/home/ayeq/sage/local/lib/python2.6/site-packages/sage/all_cmdline.pyc in <module> ()
NameError: name 'a' is not defined
sage: x
x
```

And finally if you *really* want the variable obliterated, you can use the sledgehammer of memory management, the `del()` command.

```
sage: a = [2, 3, 4, 5]
sage: del a
sage: a
-----
NameError                                Traceback (most recent call last)
/home/ayeq/sage/local/lib/python2.6/site-packages/sage/all_cmdline.pyc in <module> ()
NameError: name 'a' is not defined
```

Exercises:

1. If you enter the following into SageMath:

```
sage: a = 1
sage: b = a
sage: b = 2
```

What do you expect the value of `a` to be?

2. If you enter the following into SageMath:

```
sage: f = x^2 + x + 1
sage: f
x^2 + x + 1
sage: x = 3
```

What do you expect the value of `f` to be?

3.1.4 Lists

A *list* is an ordered collection of objects. The elements of a list are indexed by the integers, starting with 0. Here is a quick example of how to construct a list and access its elements.

```
sage: [6, 28, 496, 8128]
[6, 28, 496, 8128]
sage: L = [2, 3, 5, 7, 11, 13, 17, 2]
sage: L[0]
2
sage: L[1]
3
sage: L[5]
13
sage: L[6]
17
```

Notice how we access the elements: though 2 is the first element of the list `L`, it is accessed by the index 0.

The `len()` command returns the *length* of a list.

```
sage: len(L)
8
sage: len([2, 3, 5, 7, 11])
5
```

Note that a list of length 5 is indexed from 0 to 4.

Lists can contain numbers from any universe, or even “*Strings*”.

```
sage: M = [ 'apple', 'pear' ]
sage: len(M)
2
parent(M[1])
<type 'str'>
```

We can even have lists of lists!

```
sage: M = [[1, 2], [1, 3], [1, 4]]
sage: M[2]
[1, 4]
sage: len(M)
3
```

To access a particular element within our list of lists we chain their indices. For example, to access the 4 within that list we issue the following command:

```
sage: M[2][1]
4
```

Where we read `M[2][1]` as “Access the element at index 1 within the list with index 2” in `M`. Note that `M[2, 1]` does not work (check it yourself).

Slicing and Indexing

Probably the nicest feature of lists in Python, and thus SageMath, is the *slice* notation. Let’s suppose you have the following list:

```
sage: M = [1, 2, 0, 3, 4, 0, 4, 5]
sage: M
[1, 2, 0, 3, 4, 0, 4, 5]
```

and you would like to access the sub-list `[0, 3, 4]`. Using the slice notation you can do that in the following way:

```
sage: M[2:5]
[0, 3, 4]
```

We use `M[2:5]` since the sub-list that we desire begins with the element with index 2 and ends *before* the element with index 5.

By leaving the last index blank, the slice will extend to the end of the list. Similarly, when the first index is left blank the slice will start at the beginning of the list.

```
sage: M[2:]
[0, 3, 4, 0, 4, 5]
sage: M[:5]
[1, 2, 0, 3, 4]
```

By leaving both indices blank, we get a copy of the entire list.

```
sage: M[:]
[1, 2, 0, 3, 4, 0, 4, 5]
```

Slices also can use negative indices. When a negative number is used the position is measured relative to the end (or beginning) of the list. For example:

```
sage: M[:-2]
[1, 2, 0, 3, 4, 0]
sage: M[-2:]
[4, 5]
```

The first *ends* the slice two elements before the end of the list, while the second *begins* the slice at this same position. And like expected, we can use two negative indices to take slices relative to the last element of a list.

```
sage: M[-4:-2]
[4, 0]
sage: M[-2:-2]
[]
```

You should note that the last *slice* is empty since the beginning of the list is the same position as the end.

If we wish to know the index of an element, we use the `index()` function. It returns the index for the first occurrence of the value given.

```
sage: M = [2, 3, 3, 3, 2, 1, 8, 6, 3]
sage: M.index(2)
0
sage: M.index(3)
1
sage: M.index(14)
...
ValueError: list.index(x): x not in list
```

We can also count the number of times that an element occurs in a list.

```
sage: M.count(3)
4
```

Creating

Since they are used rather frequently, SageMath offers a convenient way to create lists of consecutive integers.

```
sage: [1..7]
[1, 2, 3, 4, 5, 6, 7]
sage: [4..9]
[4, 5, 6, 7, 8, 9]
sage: [2, 4..10]
[2, 4, 6, 8, 10]
```

In the first two examples it is quite clear what is happening. In the last example above, however, it is a bit trickier. If we input `[a, b..c]` for integers a, b and c with $a < b \leq c$, we get back the list `[a, a+d, ..., a+k*d]` where $d = b - a$ and k is the largest integer such that $a + kd \leq c$. If this is a bit overwhelming, hopefully the following examples will clear things up.

```
sage: [1,4..13]
[1, 4, 7, 10, 13]
sage: [1,11..31]
[1, 11, 21, 31]
sage: [1,11..35]
[1, 11, 21, 31]
```

Additionally, we can use this construction method with some of SageMath's symbolic constants such as `pi`.

```
sage: [pi,4*pi..32]
[pi, 4*pi, 7*pi, 10*pi]
```

Modifying lists

Sorting the list `M` can be done using the `sort()` method.

```
sage: M = [2,3,3,3,2,1,8,6,3]
sage: M.sort(); y
[1, 2, 2, 3, 3, 3, 3, 6, 8]
sage: M.index(2)
1
```

The `sort()` method alters the list *in place*, actually changing the ordering of the elements. If we would like to keep the list the same, we should sort a *copy* of the list and not the list itself.

```
sage: M = [2,3,3,3,2,1,8,6,3]
sage: M
[2, 3, 3, 3, 2, 1, 8, 6, 3]
sage: N = M[:]
sage: N.sort()
sage: N
[1, 2, 2, 3, 3, 3, 3, 6, 8]
sage: M
[2, 3, 3, 3, 2, 1, 8, 6, 3]
```

We may alter the elements of a list as follows:

```
sage: L = [1,2,3,4]
sage: L[0]=-1
sage: L
[-1, 2, 3, 4]
```

In “programming vernacular”, data-types that can be changed in place are called *mutable*. Lists are mutable, but some data types in SageMath are not.

To add an element to the end of a list, we use the `append()` method.

```
sage: L = [1,2,3]
sage: L.append(4)
sage: L
[1, 2, 3, 4]
```

Similarly, we may use the `extend()` method to concatenate lists, that is, to *append* a list to the end of another list.

```
sage: L=[1,2]
sage: L.extend([10,11,12])
```

```
sage: L
[1, 2, 10, 11, 12]
```

It is, perhaps, simpler to use the `+` operator to concatenate lists. Since the order of the list is significant, the concatenation `L + M` is not usually the same as `M + L`, though they do contain the same elements.

```
sage: [1,3,5]+[2,4,6]+[100]
[1, 3, 5, 2, 4, 6, 100]
sage: [2,4,6]+[1,3,5]+[100]
[2, 4, 6, 1, 3, 5, 100]
```

If we wish to remove an element from a list, we use the meth:`.remove` method.

```
sage: L = [3,5,11,13,17,19,29,31]
sage: L.remove(11)
sage: L
[3, 5, 13, 17, 19, 29, 31]
```

Note that a list may contain the same element more than once; `remove()` removes only the first instance of the given element.

```
sage: M = [1,2,3,0,3,4,4,0,4,5]
sage: M.remove(3)
sage: M
[1, 2, 0, 3, 4, 4, 0, 4, 5]
sage: M.remove(4)
sage: M
[1, 2, 0, 3, 4, 0, 4, 5]
```

3.1.5 Operations on a List

If your lists contain elements where it makes sense, the `sum()` and `prod()` commands accept a list as argument.

`sum()` returns the sum of its argument:

```
sage: sum([1,2,3])
6
sage: sum([1..100])
5050
```

where `prod()` returns the product.

```
sage: prod([1..4])
24
```

The sum and product commands are defined on lists where the arithmetic make sense and will complain rather loudly when it doesn't.

```
sage: sum([1,2,3,"cat",])
-----
TypeError                                 Traceback (most recent call last)
... (Lengthy error message)
TypeError: unsupported operand parent(s) for '+': 'Integer Ring' and '<type 'str''>
```

Concatenation isn't the only way which we can join together the elements of two lists. One useful tool is the `zip()` command, which joins the elements of two lists by pairing them together in order.

```
sage: zip([1,2,3,4], ['a', 'b', 'c', 'd'] )
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

When the lists aren't of the same length, `zip()` joins the elements up to the items in the shorter list and ignores the rest.

```
sage: zip([1,2,3,4], ['a', 'b', 'c'] )
[(1, 'a'), (2, 'b'), (3, 'c')]
sage: zip([1], ['a', 'b', 'c'] )
[(1, 'a')]
```

Another useful command when dealing with lists is `map()`. This command accepts two arguments, a function `f` and a list `[a0, ..., an-1]` and returns that function applied to each member of that list, `[f(a0), ..., f(an-1)]`

```
sage: map(cos, [0, pi/4, pi/2, 3*pi/4, pi] )
[1, 1/2*sqrt(2), 0, -1/2*sqrt(2), -1]
sage: map(factorial, [1,2,3,4,5])
[1, 2, 6, 24, 120]
sage: sum(map(exp, [1,2,3,4,5]))
e + e^2 + e^3 + e^4 + e^5
```

`map()` is often used in *functional* programming. For more on this style of programming with python see the [Python Documentation](#).

See also:

[An informal introduction to Python: Lists](#)

Exercises:

1. Consider the lists $L = [1, -2, 10, 13]$ and $M = [4, 3, 5, -7]$. Append L onto the end of M . Do the same beginning with M .
2. Consider the list $L = [1, 3, 4, [1, 5, 6], 8, -9]$. At what *index* is the element $[1, 5, 6]$? Remove this element from L .
3. Let $L = [3, 4, 18, 17, 2, 'a']$ and $M = [14, 23, 'b', 'c']$. With SageMath, do the following:
 - (a) Append the elements of the list M to the end of L without changing L .
 - (b) Do the same but this time altering L in place.
 - (c) Insert M as an element at the end of L , altering L in place.
 - (d) Remove the M that you just inserted.
 - (e) Explain the differences between the `extend()` and the `append()` methods.
4. Let $L = [1, 2, 5, 14, 17, 20]$. What are the sub-lists are accessed using the following *slices*.
 - (a) $L[:-1]$
 - (b) $L[-1:]$
 - (c) $L[3:]$
 - (d) $L[0:3]$
 - (e) $L[-4:-1]$
5. Using the same L as the previous problem. Find a slice that will extract the following sub-lists from L : (*Do this in two different ways*)
 - (a) $[5, 14, 17]$.

(b) [1, 2, 5].

(c) [1]

(d) [20]

6. Consider $L = ['a', 9, 10, 17, 'a', 'b', 10]$. Remove all letters from L .

3.1.6 Sets

A *Set* in SageMath is a data type which behaves a lot like a mathematical set and it differs from a list in a few key ways:

- Elements of a Set have no order. So you cannot access elements by an index.
- An element in a Set only appears once.

To see an example of that last point, we will construct a Set by converting a list into a set.

```
sage: y = [2, 3, 3, 3, 2, 1, 8, 6, 3]
sage: A = Set(y)
sage: A
{8, 1, 2, 3, 6}
```

To find the size of a Set we will use the `cardinality()` method.

```
sage: A.cardinality()
5
```

Testing for membership can be done easily by using the `in` operator.

```
sage: 8 in A
True
sage: 10 in A
False
```

All of the usual set operations: `union()`, `intersection()`, `difference()` and `symmetric_difference()` are implemented. For example:

```
sage: B = Set([8, 6, 17, -4, 20, -2])
sage: B
{17, 20, 6, 8, -4, -2}
sage: A.union(B)
{1, 2, 3, 6, 8, 17, 20, -4, -2}
sage: A.intersection(B)
{8, 6}
sage: A.difference(B)
{1, 2, 3}
sage: B.difference(A)
{17, 20, -4, -2}
sage: A.symmetric_difference(B)
{17, 2, 3, 20, 1, -4, -2}
```

Use the `subsets()` method to construct the subsets of a set, or to construct the subsets with a specified number of elements. Notice that the `subsets()` method produces a *list* of subsets.

```
sage: A = Set([1, 2, 3]); A
{1, 2, 3}
sage: powA = A.subsets(); powA
```



```

Subsets of {1, 2, 3}
sage: pairsA = A.subsets(2); pairsA
Subsets of {1, 2, 3} of size 2
sage: powA.list()
[[], {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
sage: pairsA.list()
[{1, 2}, {1, 3}, {2, 3}]

```

Exercises:

1. Consider the sets $A = \{1, -4, 2\}$ and $B = \{3, 2, 1\}$. Compute the following set operations using SageMath:

- (a) $A \cup B$
- (b) $A \cap B$
- (c) $A \setminus B$
- (d) $B \setminus A$
- (e) $(A \setminus B) \cup (B \setminus A)$

See also:

[SageMath Tutorial: Sets](#)

3.1.7 Strings

To construct a string in SageMath we may use single or double quotes.

```

sage: s='I am a string'
sage: s
'I am a string'
sage: print s
I am a string

```

Note the difference between asking for the value of `a` and asking SageMath to `print a`. Like lists, we can access the elements of a string through their indices.

```

sage: a='mathematics'
sage: a[0]
'm'
sage: a[4]
'e'

```

You can find the length of a string using the `len()` command.

```

sage: b='Gauss'
sage: len(b)
5

```

Just like with lists, we can *concatenate* strings just by adding them together.

```

sage: b + " is " + a
'Gauss is mathematics'

```

and we can separate a list by using the `split()` method,

```
sage: s.split()
['I', 'am', 'a', 'string']
```

which divides the string into a list of words. We can divide a list using different characters as *separators*. For example we can get a list from the following *comma separated values*.

```
sage: vals = "18,spam,eggs,28,70,287,cats"
sage: vals.split(',')
['18', 'spam', 'eggs', '28', '70', '287', 'cats']
```

We can use the `map()` and `split()` commands to *convert* a string of integers into something that we can use in sage. This is particularly useful when you must read data from a file.

```
sage: map(Integer, data.split(','))
[17, 18, 20, 19, 18, 20]
```

You should note how the output above differs from what we get when we use only the `split()` method.

```
sage: data.split(',')
['17', '18', '20', '19', '18', '20']
```

The list directly above contains *strings* which represent numbers. We must convert those strings into what we need in order to actually use them.

The opposite of *splitting* up a string into a list is the *joining* of elements of a list. We do this with the `join()` command.

```
sage: L = ['Learning', 'SageMath', 'is', 'easy.']
sage: join(L)
'Learning SageMath is easy.'
```

Just like when I *split* a sting, I can join a list using a different separating value than just a space. I do so by supplying an optional second argument to the `join()` command.

```
sage: join(L, ',')
'Learning,SageMath,is,easy.'
```

Exercises:

1. Consider the string `s = 'This is a string!'`. What is the output of the following commands:
 - (a) `s[:-1] + '.'`
 - (b) `s[0:7] + " not " + s[8:]`
2. Consider the string `s = 'This is a sentence. This is another sentence.'`. Split `s` into a list of two sentences.
3. Consider the list of strings `L = ['This is', 'a', 'string']`. Join the elements of the list to form the string `'This is a string'`.
4. We can use the `map()` and `Integer()` commands to take a string of integers and convert them into *SageMath* integers.

3.2 Programming Tools

SageMath syntax is based on the widely-used language Python, and thereby inherits Python's compact and very readable style. In this chapter we cover the syntax for the essentials of programming in Python. For more complex

issues we provide links to other resources.

3.2.1 Conditionals

You should be familiar with *Solving Equations and Inequalities*, *Boolean operations*, and *Variables*

A *conditional statement* is what we use when we want our code to make *decisions*. For example, suppose we want to divide a number by 2 only *if* it is even. We can do this in SageMath by using an `if` statement.

```
sage: n=44
sage: if n%2 == 0:
....:     print n/2
....:
22
sage: n=37
sage: if n%2 == 0:
....:     print n/2
....:
sage:
```

For $n=44$, the *condition* is met and the `print()` command is executed. Conversely, for $n=37$, nothing will happen since the condition has not been met. Most of what programming is is the skillful application of simple statements like this.

Unlike some other languages, SageMath is picky about indentation, a practice it inherits from Python. Instead of using some kind of punctuation to denote the beginning and ending of a *block* of code, SageMath uses *indentation* (notice though that we do need a *colon* after the condition is written). All of the code to be run under a certain condition must be at the same level of indentation. This might take some time to get used to, but it produces neat, organized code that is often easier to read.

At times, we may wish to check whether our expression satisfies more than one condition. To do so, use the `elif` statement, which is short for else if.

```
sage: m=31
sage: if m%3==0:
....:     print m/3
....: elif m%3==1:
....:     print (m-1)/3
....:
10
```

Notice that we return to the same level of indentation for `elif` as was used for `if`. We may use as many `elif`s as we desire. The tests are evaluated in order and once the first one is met, the associated code is executed and SageMath will leave the entire conditional. For a simple example, consider the following:

```
sage: r=55
sage: if 11.divides(r):
....:     print 11
....: elif r==55:
....:     print 55
....:
11
```

Here both conditions are met, but only the code associated with the first condition is actually executed. Understanding how conditionals are executed is important to controlling the flow of your program.

There is also a subtle shortcut that we used in the previous example. `11.divides(r)` already returns either

True or False, hence we did not need to use an equality here. We could have used the more verbose `11.divides(r)==True` but it is not necessary.

Often we wish to execute some code if none of our conditions above are met. For this we use the `else` operator.

```
sage: n=2*3*5+1
sage: if 2.divides(n):
....:     print 2
....: elif 3.divides(n):
....:     print 3
....: else:
....:     print n
....:
31
```

Since none of the conditions were met, our code *defaulted* to printing the number 31.

3.2.2 While loops

You should be familiar with *Variables* and *Boolean operations*

While loops are one of the most useful techniques in programming. Essentially, a while loop runs a block of code while a condition is still satisfied. Let's see a simple example:

```
sage: i=0
sage: while i < 5:
....:     print i^2
....:     i=i+1
....:
0
1
4
9
16
```

Once the condition `i<5` is False, SageMath exits the loop structure; the variable `i` still exists, though.

3.2.3 For Loops

You should be familiar with *Variables*, *Boolean operations*, and *Lists*

A for loop repeatedly runs a block of code a fixed number of times. In SageMath, for loops iterate over a fixed list.

```
sage: for i in [0..4]:
....:     print i^2
....:
0
1
4
9
16
```

We may iterate over any list, it need not be consecutive integers. Here are a few more (really silly) examples.

```
sage: for str in ["apple", "banana", "coconut", "dates"]:
....:     print str.capitalize()
....:
Apple
Banana
Coconut
Dates
sage: for char in "Leonhard Euler":
....:     print char.swapcase()
....:
l
E
O
N
H
A
R
D

e
U
L
E
R
```

3.2.4 List Comprehensions (Loops in Lists)

You should be familiar with *Lists* and *For Loops*

A particularly useful technique in Python (and SageMath by extension) is the construction of lists using **list comprehensions**. This feature is very similar to the *set builder* notation we often use in mathematics. For example, the set of *even* integers can be written as:

$$\{2 \cdot k \mid k \in \mathbb{Z}\}$$

Where we do not explicitly list the elements of the set but rather give a *rule* which can be used to construct the set. We can do something very similar in python by placing a `for` inside of a list, like in the following example. Here is how we would construct the list of even integers from 0 to 20.

```
sage: [ 2*k for k in [0..10] ]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

This concept may seem a bit intimidating at first, but it is extremely concise way to write some powerful code.

We can use list comprehension to apply a function to each number of a given list, much like we did before with the `map()` command.

```
sage: [pi/4, pi/2..2*pi]
[1/4*pi, 1/2*pi, 3/4*pi, pi, 5/4*pi, 3/2*pi, 7/4*pi, 2*pi]
sage: [ cos(x) for x in [pi/4, pi/2..2*pi]]
[1/2*sqrt(2), 0, -1/2*sqrt(2), -1, -1/2*sqrt(2), 0, 1/2*sqrt(2), 1]
```

We can also use the list comprehension *filter* (or *reduce*) the results by adding a *conditional* to our list comprehension. For example, to construct the list of all natural numbers that are less than 20 which are *relatively prime* to 20 we do the following:

```
sage: [ k for k in [1..19] if gcd(k,20) == 1 ]
[1, 3, 7, 9, 11, 13, 17, 19]
```

Notice that the syntax for the construction is nearly identical to the mathematical way that we would write the same set of numbers:

$$\{k \in \mathbb{N} \mid k < 20 \text{ and } \gcd(k, 20) = 1\}$$

In mathematics we often construct the *Cartesian Product* of two sets:

$$A \times B = \{(a,b) \mid a \in A, b \in B\}$$

We can do something similar by using multiple *for*'s in the list comprehension. For example, to construct the list of all *pairs* of elements in the list constructed earlier we do the following:

```
sage: U = [ k for k in [1..19] if gcd(k,20) == 1 ]
sage: [ (a,b) for a in U for b in U ]
[(1, 1), (1, 3), (1, 7), (1, 9), (1, 11), (1, 13), (1, 17), (1, 19), (3, 1), (3, 3),
↪ (3, 7), (3, 9), (3, 11), (3, 13), (3, 17), (3, 19), (7, 1), (7, 3), (7, 7), (7, 9),
↪ (7, 11), (7, 13), (7, 17), (7, 19), (9, 1), (9, 3), (9, 7), (9, 9), (9, 11), (9,
↪ 13), (9, 17), (9, 19), (11, 1), (11, 3), (11, 7), (11, 9), (11, 11), (11, 13), (11,
↪ 17), (11, 19), (13, 1), (13, 3), (13, 7), (13, 9), (13, 11), (13, 13), (13, 17),
↪ (13, 19), (17, 1), (17, 3), (17, 7), (17, 9), (17, 11), (17, 13), (17, 17), (17,
↪ 19), (19, 1), (19, 3), (19, 7), (19, 9), (19, 11), (19, 13), (19, 17), (19, 19)]
```

It should be noted that you don't have to form *tuples* of the pairs of elements. For instance, you can also find the their product or their sum. Any valid expression involving a and b will be fine.

```
sage: [ a*b for a in U for b in U ]
[1, 3, 7, 9, 11, 13, 17, 19, 3, 9, 21, 27, 33, 39, 51, 57, 7, 21, 49, 63, 77, 91, 119,
↪ 133, 9, 27, 63, 81, 99, 117, 153, 171, 11, 33, 77, 99, 121, 143, 187, 209, 13, 39,
↪ 91, 117, 143, 169, 221, 247, 17, 51, 119, 153, 187, 221, 289, 323, 19, 57, 133, 171,
↪ 209, 247, 323, 361]
sage: [ a + b for a in U for b in U ]
[2, 4, 8, 10, 12, 14, 18, 20, 4, 6, 10, 12, 14, 16, 20, 22, 8, 10, 14, 16, 18, 20, 24,
↪ 26, 10, 12, 16, 18, 20, 22, 26, 28, 12, 14, 18, 20, 22, 24, 28, 30, 14, 16, 20, 22,
↪ 24, 26, 30, 32, 18, 20, 24, 26, 28, 30, 34, 36, 20, 22, 26, 28, 30, 32, 36, 38]
sage: [ gcd(a,b) for a in U for b in U ]
[1, 1, 1, 1, 1, 1, 1, 1, 3, 1, 3, 1, 1, 1, 1, 1, 1, 1, 7, 1, 1, 1, 1, 1, 1, 3, 1, 9,
↪ 1, 1, 1, 1, 1, 1, 1, 1, 11, 1, 1, 1, 1, 1, 1, 1, 1, 1, 13, 1, 1, 1, 1, 1, 1, 1, 1, 17,
↪ 1, 1, 1, 1, 1, 1, 1, 1, 19]
```

Similar constructions work for more than 2 sets; just add more *for* statements.

Since list comprehensions allow for us to put any valid expression, we can add another conditional which affects the output of our list. For example, let take the list of integers which were *relatively prime* to 20 and test if they are prime numbers or not.

```
sage: U
[1, 3, 7, 9, 11, 13, 17, 19]
sage: [ 'prime' if x.is_prime() else 'not prime' for x in U ]
['not prime', 'prime', 'prime', 'not prime', 'prime', 'prime', 'prime', 'prime']
```

See also:

[More on list comprehensions](#)

Exercises:

1. Use a list comprehension to generate lists which have the same members as the following sets:
 - (a) The set of all odd integers greater than -10 and less than 30 .
 - (b) The set of all integers which are divisible by 3 , less than or equal to 100 and greater than -20 .
 - (c) The set of all *prime* numbers less than 100 .
2. Use a list comprehension to compute the $\tan(x)$ for all $x \in \{0, \pi/4, \pi/2, 3\pi/4, \pi\}$

3.2.5 Defining your own commands

Once your computations get complicated enough you may want to hide some of this complexity by creating your own command that can be easily re-used like SageMath's built-in commands. These user-defined commands are commonly called *functions*, though they differ from mathematical functions.

For example, suppose that we wanted to compute the greatest common divisor of 75 and 21 . We can use the *euclidean algorithm* and SageMath to do this. Here is how that would look:

```
sage: def euclid(a,b):
....:     r = a%b
....:     while r != 0:
....:         a=b; b=r
....:         r = a%b
....:     return b
```

a and b are called the *arguments* of the command and the expression following the `return` keyword is called the *return value*. The arguments are in the input of the command whereas the return value is the output.

Those of you who have previous programming experience may notice the absence of end or block *delimiters*, such as `;` or `end`. SageMath, like Python, uses indentation to denote where a block of code begins and ends. This syntax rule forces the programmer to write visually-separated blocks of code, thus making it more readable.

Once the command `euclid` has been defined, the code can easily be re-used with different arguments, just like a built-in command.

```
sage: euclid(75,21)
3
sage: euclid(455,67)
1
sage: euclid(754,99)
1
sage: euclid(756,9)
9
```

User-defined commands may have any number of arguments, including none at all.

```
sage: def g(x,y):
....:     return x*y
....:
sage: g(2,3)
6
sage: g(sqrt(2),sqrt(2))
2
sage: def h():
....:     return 1/2
....:
sage: h()
1/2
```

Defining a return value is also optional, but all commands in SageMath return something. If we do not specify a return value, then SageMath returns the empty object `None`.

```
sage: def lazy(x):
....:     print x^2
....:
sage: lazy(sqrt(3))
3
sage: a = lazy(sqrt(3))
3
sage: a
None
```

What the above is showing is that while the command displays the number 3, the return value is actually **None**. While this is valid code, it is good practice to have your commands actually return the value that you are interested in computing.

By separating the values with commas, your command can have multiple return values.

```
sage: def s(x):
....:     return x^2, x^3
....:
sage: s(1)
(1, 1)
sage: s(2)
(4, 8)
sage: a,b=s(3)
sage: a
9
sage: b
27
```

Defining your own commands in SAGE is easy. However, elegantly encapsulating your code is an art which requires a lot of practice and thought. For a more thorough introduction to functions (commands), [this chapter](#) on Python functions is a good place to start.

3.2.6 External Files and Sessions

In practice, especially when using SageMath for research and projects, it is convenient to being able to load external files. One such instance is when we have a block of code which we wish to run for several different cases. It would be quite tedious to retype all of the code; instead we read it from an external file.

Suppose we have a file in the same directory from which we started SageMath called `pythag.sage` with the following content.

```
# Begin pythag.sage
a=3
b=4
c=sqrt(a^2+b^2)
print c
# End
```

Note that all characters after a # of a SageMath file are ignored when loaded. We may now load the file in SageMath using the `load()` command.

```
sage: load pythag.sage
5
```


After having loaded the file, all of the variables initialized now exist in our SageMath session.

```
sage: a,b,c
(3, 4, 5)
```

SageMath allows us to save a session to pick up where we left off. That is, suppose we have done various calculations and have several variables stored. We may call the `save_session` function to store our session into a file in our working directory (typically `sage_session.sobj`). Following, we may exit SageMath, power off our computer, or what have you. At any later time, we may load the file by opening SageMath from the directory containing the save file and using the `load_session` function.

Here is an example:

```
sage: a=101
sage: b=103
sage: save_session()
sage: exit
Exiting SAGE (CPU time 0m0.06s, Wall time 0m31.27s).
```

Now start SageMath from the same folder as the save file.

```
sage: load_session()
sage: a
101
sage: b
103
```

We may specify the name of a save session, if we so desire.

```
sage: T=1729
sage: save_session('ramanujan')
sage: exit
Exiting SAGE (CPU time 0m0.06s, Wall time 0m16.57s).
```

And again we load our session `ramanujan` with `load_session()`.

```
sage: load_session('ramanujan')
sage: T
1729
```

3.3 Packages within SageMath

There are many open-source software packages available for doing specialized mathematics. One of the objectives of SageMath developers is to create a single clean interface from which these packages may all be accessed. For many computations in advanced mathematics SageMath uses the functionality in one of these packages. A SageMath user can also explicitly call a function from one of the packages. This chapter briefly describes how to do so.

3.3.1 GAP

For this portion of the tutorial we are going to show how to use GAP from within a SageMath session. The commands here follow closely with the [Groups and Homomorphisms](#) section of the GAP tutorial. A reader who is interested in learning more about the capabilities of this system should consult the [Gap Project's main website](#).

You can pass a command to GAP by using `gap()` with the command as a *string*. The following example constructs the *symmetric group* on eight points using GAP.

```
sage: s8 = gap('Group( (1,2), (1,2,3,4,5,6,7,8) )')
sage: s8
Group( [ (1,2), (1,2,3,4,5,6,7,8) ] )
```

s8 has *GAP* as a parent.

```
sage: parent(s8)
Gap
```

The *interface* to the GAP system translates the commands in GAP to *methods* in SageMath. For example, to compute the *Derived Subgroup* of S_8 you use the `DerivedSubgroup()` method.

```
sage: a8 = s8.DerivedSubgroup(); a8
Group( [ (1,2,3), (2,3,4), (2,4)(3,5), (2,6,4), (2,4)(5,7), (2,8,6,4)(3,5) ] )
sage: a8.Size(); a8.IsAbelian(); a8.IsPerfect()
20160
false
true
```

The output of `s8.DerivedSubgroup()` is identical to the output of the GAP command `DerivedSubgroup(s8)` and this is the common convention when the command has one argument. When it requires two, say the group and an additional parameter, the additional parameter is given as an argument to the method. For example, the GAP command `SylowSubgroup(a8,2)` computes the maximal 2-subgroup of A_8 . The following SageMath code does the same, then uses GAP to compute its size.

```
sage: sy12 = a8.SylowSubgroup(2); sy12.Size()
64
```

In the same vein, we can use GAP to compute the *normalizer's* and *centralizers* of these groups.

```
sage: a8.Normalizer(sy12)
Group( [ (1,6)(2,4), (1,6)(5,8), (2,4)(3,7), (2,8)(4,5), (1,7)(2,8)(3,6)(4,5),
(1,8)(2,7)(3,4)(5,6) ] )
sage: a8.Normalizer(sy12) == sy12
True
sage: cent = a8.Centralizer(sy12.Centre());
sage: cent
Group( [ (1,6)(2,4)(3,7)(5,8), (3,5)(7,8), (3,7)(5,8), (2,3)(4,7),
(1,2)(4,6) ] )
sage: cent.Size()
192
```

Gap itself has commands which can manipulate lists of objects. In this example we first compute the *derived series* of `cent` and then compute the size of each of these subgroups using GAP's `List()` command.

```
sage: cent.DerivedSeries(); cent.DerivedSeries().List('Size')
[ Group( [ (1,6)(2,4)(3,7)(5,8), (3,5)(7,8), (3,7)(5,8), (2,3)(4,7),
(1,2)(4,6) ] ),
Group( [ (2,4)(3,7), (1,3)(2,8)(4,5)(6,7), (1,7,4)(2,6,3) ] ),
Group( [ (1,6)(2,4)(3,7)(5,8), (1,6)(3,7),
(1,4)(2,6)(3,5)(7,8), (1,7)(2,5)(3,6)(4,8),
(1,4,6,2)(3,8,7,5) ] ),
Group( [ (1,6)(2,4)(3,7)(5,8) ] ), Group( ) ]
[ 192, 96, 32, 2, 1 ]
```

Since the GAP command constructs a full-fledged SageMath object we can do the same in a more SageMath-y manner by using a list comprehension.

```
sage: [ g.Size() for g in cent.DerivedSeries() ]
[192, 96, 32, 2, 1]
```

To convert a GAP group to a native SageMath one we first extract a list of generators. Then feed that list to the usual group constructor.

```
sage: gens = s8.GeneratorsOfGroup(); gens
[ (1,2), (1,2,3,4,5,6,7,8) ]
sage: SG = PermutationGroup(gens); SG
Permutation Group with generators [(1,2), (1,2,3,4,5,6,7,8)]
sage: parent(SG)
<class 'sage.groups.perm_gps.permgroup.PermutationGroup_generic_with_category'>
```

Going from a SageMath group to a GAP one is even easier.

```
sage: gap(SG)
Group( [ (1,2), (1,2,3,4,5,6,7,8) ] )
sage: parent(gap(SG))
Gap
```

From time to time you will want to just use GAP directly without using the interface. When working from the command line, the `gap_console()` command does just this.

```
sage: gap_console()
GAP4, Version: 4.4.12 of 17-Dec-2008, x86_64-unknown-linux-gnu-gcc
gap>
```

From which we can exit by typing `quit`; at the `gap` prompt.

```
gap> quit;
sage:
```

If the reader is using the notebook then using GAP directly is even easier. It is done by just selecting GAP from a drop down menu (highlighted in yellow in the picture below).

GAP

last edited Aug 14, 2018, 11:20:15 AM by admin

File... Action... Data... **gap** Typeset Load 3-D Live Use java f

```

+ 
s8 := Group( [(1,2), (1,2,3,4,5,6,7,8) ])

Group([ (1,2), (1,2,3,4,5,6,7,8) ])

+ 
NormalSubgroups(s8)

[ Group(()), Alt( [ 1 .. 8 ] ), Group([ (1,2), (1,2,3,4,5,6,7,8) ])

+ 

```

Now the SageMath notebook acts as a web interface to the GAP system.

See also:

<http://www.gap-system.org/Manuals/doc/htm/index.htm>

3.3.2 Singular

As with the GAP interface, the SageMath interface to Singular substitutes the language commands with *methods* in SageMath. For example, the following code in Singular:

```

> ring R = 0, (x,y,z), lp;
> R;
// characteristic : 0
// number of vars : 3
//      block   1 : ordering lp
//              : names   x y z
//      block   2 : ordering C

```

Constructs a polynomial ring in three variables; x,y and z over the field of characteristic 0 using the *lexicographic* term ordering. To do the same within SageMath we use the `ring()` method of the `singular` object.

```
sage: R = singular.ring('0', '(x,y,z)', 'lp')
sage: R
// characteristic : 0
// number of vars : 3
//      block 1 : ordering lp
//           : names  x y z
//      block 2 : ordering C
```

Since much of the language that Singular uses is not valid in SageMath the quotations around the arguments are important.

Polynomials are constructed in this ring by using the `poly()` method.

```
sage: p = singular.poly('x^2 * y^2 - 1')
sage: q = singular.poly('x^2 * y^2 - z')
sage: singular.ideal([p,q])
x^2*y^2-1,
x^2*y^2-z
```

To construct the ideal (in R) generated by those polynomials and a Groebner basis it you enter the following.

```
sage: I = singular.ideal([p,q])
sage: I.groebner()
z-1,
x^2*y^2-z
```

Reduction modulo this ideal is accomplished using the `reduce()` method.

```
sage: r = singular.poly('x^3 - x^2 * y^2 - x^2 * z + x')
sage: singular.reduce(p,I)
z-1
sage: singular.reduce(q,I)
0
sage: singular.reduce(r,I)
x^3-x^2*z+x-z
```

and if you would like this reduction done using a Groebner basis, we just combine the methods discussed previously.

```
sage: singular.reduce(q,I.groebner())
0
sage: singular.reduce(p,I.groebner())
0
sage: singular.reduce(r,I.groebner())
x^3-x^2+z-1
```

The quotations are not necessary when passing a Singular object as in the last few examples as there is no ambiguity.

Finally a task that Singular excels at is the factorization of multivariate polynomials. This is done using the `factorize()` method.

```
sage: p.factorize()
[1]:
  _[1]=1
  _[2]=x*y-1
  _[3]=x*y+1
[2]:
  1,1,1
sage: q.factorize()
```

```
[1]:
  _[1]=1
  _[2]=x^2*y^2-z
[2]:
  1,1
sage: r.factorize()
[1]:
  _[1]=-1
  _[2]=x
  _[3]=-x^2+x*y^2+x*z-1
[2]:
  1,1,1
```

See also:

<http://www.singular.uni-kl.de>

3.3.3 Using Python packages in SageMath

3.4 Interactive Demonstrations in the Notebook

In this section we will discuss the creation of interactive “applets” in the SageMath notebook. These are done using the `@interact` decorator and are often called *interacts*. A decorator is just a fancy piece of python which allows for you to create new functions out of old in a quick and concise fashion. You don’t have to fully understand decorators to be able to follow this material but If you are interested you can read a very nice [blog post](#) about decorators by Bruce Eckel of [Thinking in Python](#) Fame.

We will begin with the most simple applet. One that creates a single input box and then displays the results.

Interact Demo -- Sage - Google Chrome

localhost:8000/home/admin/0/

Interact Demo Save Save & quit

last edited on May 09, 2011 09:36 AM by admin

File... Action... Data... sage Typeset Print Worksheet Edit Text Undo Save

```
@interact
def echo_input(a = input_box(default="Hello World", label="Input:", type=str)):
    print "Output: " + a
```

Input: Hello World

Output: Hello World

[evaluate](#)

Notice how changing the text in the input box changes the output. Every time something within the interact changes the “applet” is refreshed and will display those changes. This is the heart of the interactivity.

```
@interact
def echo_input(a = input_box(default="Hello World", label="Input:", type=str)):
    print "Output: " + a
```

Input: Hello World, I am your first interact

Output: Hello World, I am your first interact

[evaluate](#)

Next we will add another control to the applet. This time we will add a *slider*. This control has a handle which the user can slide horizontally, and by sliding change a number in pre-defined increments. For this example, the slider has 0 as its smallest number and 10 as its largest and moves in increments of 1 unit.

The screenshot shows a web browser window titled "Interact Demo -- Sage - Google Chrome". The address bar shows "localhost:8000/home/admin/0/". The page header includes "Sage The Sage Notebook Version 4.6.2" and navigation links: "admin", "Toggle", "Home", "Published", "Log", "Settings", "Help", and "Report a Problem".

The main content area is titled "Interact Demo" and shows the following Python code:

```
@interact
def echo_input(a = input_box(default="Hello World", label="Input:", type=str),
               n = slider(vmin=0, vmax=10, step_size=1, default=5, label="Select a number: ")):
    print "Output: " + a
    print "The number that you have selected is: " + str(n)
```

Below the code is an "evaluate" button. The rendered output shows a user interface with an input box containing "Hello World" and a slider labeled "Select a number:" with a value of 4. The output text is:

```
Output: Hello World
The number that you have selected is: 4
```

Next we will add a selection control. This control allows the user to select one of a finite number of different options. In this case, the user can select any color, as long as that color is red, blue, green, or black.

```
@interact
def echo_input(a = input_box(default="Hello World", label="Input:", type=str),
               n = slider(vmin=0, vmax=10, step_size=1, default=5, label="Select a number: "),
               color = selector(values=["Red", "Blue", "Green", "Black"], label="Select a color:", default="Black")):
    print "Output: " + a
    print "The number that you have selected is: " + str(n)
    print "The color that you have selected is: " + color
```

[evaluate](#)

Input:

Select a number: 5

Select a color:

Output: Hello World
The number that you have selected is: 5
The color that you have selected is: Black

While this initial example shows the use of a couple of common interactive controls, it still does not do anything very interesting. The next example will combine both the use of sliding and selection controls toward creating an applet which plots the trigonometric functions and their standard transformations.

Interact Demo -- Sage - Google Chrome

localhost:8000/home/admin/0/

```
@interact
def trig_graph(f = selector(values = [sin(x),cos(x),tan(x)], label="Select a Trigonometric Function"),
  A = slider(vmin=1/16, vmax=10, step_size=1/16,default=1,label="A = "),
  B = slider(vmin=-10, vmax=10, step_size=1/4, default=0, label="B = "),
  omega = slider(vmin = -pi, vmax= pi, step_size=pi/12,default=1,label="omega = "),
  phi = slider(vmin = -pi, vmax=pi, step_size=pi/12,default=0,label="phi =")):
  show(plot(A*f(omega*x + phi) + B, -2*pi, 2*pi), ymin=-10, ymax=10)
```

Select a Trigonometric Function

A =

B =

omega =

phi =

The example here only scratches the surface of what is possible with SageMath interact. For a growing list of examples of interacts see this page on the [sage wiki](#).

MATHEMATICAL STRUCTURES

The individual chapters in this part of the tutorial are relatively independent of one another. You should be familiar with the chapter `sage_objects` before reading material here. The section *List Comprehensions (Loops in Lists)* is also useful. Eventually, when you are ready for some real experimentation, you will want to read much of the chapter *Programming Tools*. Many sections in this part are incomplete, and we welcome contributions and additions!

4.1 Integers and Modular Arithmetic

4.1.1 Integers Modulo n

You should be familiar with *Universes and Coercion* and *Variables*

In this section we cover how to construct \mathbb{Z}_n , the ring of integers modulo n , and do some basic computations.

To construct \mathbb{Z}_n you use the `Integers` command.

```
sage: Integers(7)
Ring of integers modulo 7
sage: Integers(100)
Ring of integers modulo 100
```

We could do computations modulo an integer by repeatedly using the `%` operator in all of our expressions, but by constructing the ring explicitly we have access to a more natural method for doing arithmetic.

```
sage: R=Integers(13)
sage: a=R(6)
sage: b=R(5)
sage: a + b
11
sage: a*b
4
```

And by explicitly coercing our numbers into the ring \mathbb{Z}_n we can compute some of the mathematical properties of the elements. Like their order, both multiplicative and additive, and whether or not the element is a unit.

```
sage: a.additive_order()
13
sage: a.multiplicative_order()
12
sage: a.is_unit()
True
```

The additive inverse of a is computed using $-a$ and, if a is a unit, the multiplicative inverse is computed using a^{-1} or $1/a$.

```
sage: (-a)
7
sage: (a^(-1))
11
```

These inverses can be checked easily.

```
sage: a + (-a)
0
sage: a*(a^(-1))
1
```

Recall that division in \mathbb{Z}_n is really multiplication by an inverse.

```
sage: R=Integers(24)
sage: R(4)/R(5)
20
sage: R(4)*R(5)^-1
20
sage: R(4/5)
20
```

Not all elements have an inverse, of course. If we try an invalid division, SageMath will complain

```
sage: R(5/4)
...
ZeroDivisionError: Inverse does not exist.
```

We have to be a little bit careful when we are doing this since we are asking SageMath to coerce a rational number into the \mathbb{Z}_{24} . This may cause some unexpected consequences since some reduction is done on rational numbers before the coercion. For an example, consider the following:

```
sage: R(20).is_unit()
False
sage: R(16/20)
20
```

In \mathbb{Z}_{24} , 20 is not a unit, yet at first glance it would seem we divided by it. However, note the order of operations. First sage reduces $16/20$ to $4/5$, and then coerces $4/5$ into \mathbb{Z}_{24} . Since 5 is a unit in \mathbb{Z}_{24} , everything works out ok.

We can also compute some properties of the ring itself.

```
sage: R
Ring of integers modulo 24
sage: R.order()
24
sage: R.is_ring()
True
sage: R.is_integral_domain()
False
sage: R.is_field()
False
```

Since this ring is finite then we can have SageMath list all of its elements.

```
sage: R = Integers(13)
sage: R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

R in this example is a field, since 13 is a prime number. If our ring is not a field then the *units* in \mathbb{Z}_n form a group under multiplication. SageMath can compute a list of generators of the *group of units* using its `unit_gens()` method.

```
sage: R = Integers(12)
sage: R.uni
R.unit_gens          R.unit_group_order
R.unit_group_exponent R.unit_ideal
sage: R.unit_gens()
[7, 5]
```

We can also compute the order of this subgroup.

```
sage: R.unit_group_order()
4
```

Unfortunately, SageMath doesn't seem to have a function which directly returns the units in \mathbb{Z}_n as a group. We can list the elements in a couple of different ways using the information above.

```
sage: (a,b) = R.unit_gens()
sage: a
7
sage: b
5
sage: [ (a^i)*(b^j) for i in range(2) for j in range(2) ]
[1, 5, 7, 11]
```

We can also compute the list of units by using a list comprehension.

```
sage: [ x for x in R if x.is_unit() ]
[1, 5, 7, 11]
```

Exercises:

- Construct the ring of integers modulo 16 and answer the following:
 - Compute the multiplicative orders of 2, 4, 5, 6, 13 and 15?
 - Which of the elements listed above is a unit?
 - What are the generators for the group of units?
 - Compute a list of all of the elements in the group of units.
- Do all of the steps above again, but with the ring of integers modulo 17.
- Use an exhaustive search method to write a function which determines if a is a unit modulo n .
- For $n = 13, 15$ and 21 determine which of 3, 4 and 5 are units in \mathbb{Z}_n . When you find a unit, determine its inverse and compare this to the output of $xgcd(a, n)$. Try to explain this relationship.
- Use SageMath to determine whether the following Rings are fields. For each example, describe the unit group using generators and relations.
 - \mathbb{Z}_{1091}
 - \mathbb{Z}_{1047}
 - \mathbb{Z}_{1037}

(d) \mathbb{Z}_{1087}

4.1.2 Solving Congruences

You should be familiar with *Integers Modulo* and *List Comprehensions (Loops in Lists)*

A linear congruence is an equation of the form $ax = b$ in \mathbb{Z}_n . One way to see if there is a solution to such a problem is an exhaustive search. For example, to determine if there exists a solution to $9x = 6$ we can do the following:

```
sage: R=Integers(21)
sage: a=R(9)
sage: 6 in [ a*x for x in R ]
True
```

Notice that the above tells us only that there exists at least one solution to the equation $9x = 6$ in \mathbb{Z}_{21} . We can construct the list of these solutions by using the following list comprehension.

```
sage: [ x for x in R if R(9)*x == R(6) ]
[3, 10, 17]
```

We can determine when a solution does not exist in a similar fashion.

```
sage: [ x for x in R if R(9)*x == R(2) ]
[]
```

We can also use the `solve_mod()` function to compute the same results.

```
sage: solve_mod( 9*x == 6, 21)
[(3,), (10,), (17,)]
sage: solve_mod( 9*x == 2, 21)
[]
```

`solve_mod()` can handle linear congruences of more than one variable.

```
sage: solve_mod( 9*x + 7*y == 2, 21)
[(15, 14), (15, 8), (15, 2), (15, 17), (15, 11), (15, 5), (15, 20), (1, 14), (1, 8),
↪(1, 2), (1, 17), (1, 11), (1, 5), (1, 20), (8, 14), (8, 8), (8, 2), (8, 17), (8,
↪11), (8, 5), (8, 20)]
```

The solutions are in the form (x, y) , where the variables are listed in the order in which they appear in the equations.

`solve_mod()` can solve systems of linear congruences.

```
sage: solve_mod( [9*x + 2*y == 2, 3*x + 2*y == 11 ], 21)
[(9, 13), (16, 13), (2, 13)]
```

As with the `solve()` command, computations can be slow when working with systems that have a lot of variables and/or equations. For these systems the linear algebra capabilities are recommended.

We can also compute the solutions for non-linear congruences using `solve_mod()`.

```
sage: solve_mod(x^2 + y^2 == 1, 7)
[(0, 1), (0, 6), (1, 0), (2, 2), (2, 5), (5, 2), (5, 5), (6, 0)]
sage: solve_mod([x^2 + y^2 == 1, x^2 - y == 2], 7)
[(2, 2), (5, 2)]
```

Finally, SageMath can compute the simultaneous solution of linear congruences with different moduli under certain circumstances. This is done using the *Chinese Remainder Theorem*, and is implemented in the `crt()` command.

For example, the following computes the smallest nonnegative integer, x that is congruent to $3 \pmod{8}$, $4 \pmod{9}$, and $5 \pmod{25}$.

```
sage: crt([3, 4, 5], [8, 9, 25])
355
```

We can check the validity of this solution using the `mod()` command.

```
sage: mod(355, 8)
3
sage: mod(355, 9)
4
sage: mod(355, 25)
5
```

The set of all integer solutions is those integers congruent to 355 modulo $8 * 9 * 25 = 1800$.

Exercises:

- Find all solutions to the following congruences over \mathbb{Z}_{42} .
 - $41x = 2$
 - $5x = 13$
 - $6x = 0$
 - $6x = 12$
 - $6x = 18$
 - $37x = 21$
- Above you computed the solution sets for the congruences $6x = 0$, $6x = 12$ and $6x = 18$. What are the similarities? What are the differences? Can you use these results to say something in general about the structure of the set $\{6x \mid x \in \mathbb{Z}_{42}\}$?
- Use the `solve_mod()` command find all of the solutions to the following congruences modulo 36.
 - $3x = 21$
 - $7x = 13$
 - $23x = 32$
 - $8x = 14$

4.1.3 Mini-Topic: Euclidean Algorithm

You should be familiar with *Integer Division and Factoring*, *Variables, External Files and Sessions*, and *While loops*

Recall that for $a, b \in \mathbb{Z}$ with $b \neq 0$, there always exists unique $q, r \in \mathbb{Z}$ such that $a = bq + r$ with $0 \leq r < b$. With that in mind, we will use SageMath to calculate the *gcd* of two integers using the *Euclidean Algorithm*. The following code is an implementation of the Euclidean Algorithm in SageMath.

```
# Begin euclid.sage
r=a%b
print (a,b,r)
while r != 0:
    a=b; b=r
    r=a%b
```

```
print (a,b,r)
# End euclid.sage
```

If you create a file `euclid.sage` containing the text above, then the output after loading the file is:

```
sage: a=15; b=4
sage: load euclid.sage
(15, 4, 3) (4, 3, 1) (3, 1, 0)
sage: a=15; b=5
sage: load euclid.sage
(15, 5, 0)
```

In the first case, we see that the gcd was 1, while in the second the gcd was 5.

Exercises:

1. Revise the loop in the `euclid.sage` so that only the gcd and the total number of divisions (i.e. the number of steps through the algorithm) are printed. Compare the speed of this version of the algorithm with the built-in SageMath function `gcd()` by using both functions on large integers.
2. Write your own *Extended Euclidean Algorithm* by revising the loop in `euclid.sage`.

4.2 Groups

There are three major types of groups implemented in sage, `PermutationGroup()`, `MatrixGroup()` and `AbelianGroup()`. We will work with permutation groups first and cover most of the methods that are applied to them. Many of these methods are applicable to arbitrary groups, so the other sections will be somewhat briefer and will focus on methods particular to those structures.

See also:

[Group Theory and SageMath: A Primer by Rob Beezer](#)

4.2.1 Symmetric Groups

The Symmetric Group S_n is the group of all permutations on n elements. First we will construct the symmetric group on $\{1, 2, 3, 4, 5\}$ which is done by using the `SymmetricGroup` command.

```
sage: S5 = SymmetricGroup(5)
S5 Symmetric group of order 5! as a permutation group
```

Once the group has been constructed we can check the number of elements, which is $5!$, and list them all.

```
sage: S5.cardinality()
120
sage: S5.list()
[(), (4,5), (3,4), (3,4,5), (3,5,4), (3,5), (2,3), (2,3)(4,5), (2,3,4), (2,3,4,5), ↵
↵(2,3,5,4), (2,3,5), (2,4,3), (2,4,5,3), (2,4), (2,4,5), (2,4)(3,5), (2,4,3,5), (2,5,
↵4,3), (2,5,3), (2,5,4), (2,5), (2,5,3,4), (2,5)(3,4), (1,2), (1,2)(4,5), (1,2)(3,4),
↵(1,2)(3,4,5), (1,2)(3,5,4), (1,2)(3,5), (1,2,3), (1,2,3)(4,5), (1,2,3,4), (1,2,3,4,
↵5), (1,2,3,5,4), (1,2,3,5), (1,2,4,3), (1,2,4,5,3), (1,2,4), (1,2,4,5), (1,2,4)(3,
↵5), (1,2,4,3,5), (1,2,5,4,3), (1,2,5,3), (1,2,5,4), (1,2,5), (1,2,5,3,4), (1,2,5)(3,
↵4), (1,3,2), (1,3,2)(4,5), (1,3,4,2), (1,3,4,5,2), (1,3,5,4,2), (1,3,5,2), (1,3), ↵
↵(1,3)(4,5), (1,3,4), (1,3,4,5), (1,3,5,4), (1,3,5), (1,3)(2,4), (1,3)(2,4,5), (1,3,
↵2,4), (1,3,2,4,5), (1,3,5,2,4), (1,3,5)(2,4), (1,3)(2,5,4), (1,3)(2,5), (1,3,2,5,4),
↵(1,3,2,5), (1,3,4)(2,5), (1,3,4,2,5), (1,4,3,2), (1,4,5,3,2), (1,4,2), (1,4,5,2), ↵
↵(1,4,2)(3,5), (1,4,3,5,2), (1,4,3), (1,4,5,3), (1,4), (1,4,5), (1,4)(3,5), (1,4,3,
↵5), (1,4,2,3), (1,4,5,2,3), (1,4)(2,3), (1,4,5)(2,3), (1,4)(2,3,5), (1,4,2,3,5), (1,
↵4,2,5,3), (1,4,3)(2,5), (1,4)(2,5,3), (1,4,3,2,5), (1,4)(2,5), (1,4,2,5), (1,5,4,3,
↵2), (1,5,3,2), (1,5,4,2), (1,5,2), (1,5,3,4,2), (1,5,2)(3,4), (1,5,4,3), (1,5,3), ↵
↵(1,5,4), (1,5), (1,5,3,4), (1,5)(3,4), (1,5,4,2,3), (1,5,2,3), (1,5,4)(2,3), (1,
↵5)(2,3), (1,5,2,3,4), (1,5)(2,3,4), (1,5,3)(2,4), (1,5,2,4,3), (1,5,3,2,4), (1,5)(2,
↵4,3), (1,5,2,4), (1,5)(2,4)]
```

As you can see from the list, in SageMath a permutation is written in *cycle notation*. Note that the empty parenthesis `()` is used to represent the identity permutation. We create the identity permutation and a randomly chosen element as follows.

```
sage: id = S5.identity()
()
sage: S5.random_element()
(1,2)(3,4)
sage: r = S5.random_element(), r
(1,3,4)(2,5)
```

As you can see, subsequent calls for a random element give a new element each time. We can also express the element r as a function by listing the images of 1, 2, 3, 4, 5 in order.

```
sage: r.list()
[3,5,4,1,2]
```

We can construct a specific element in S_5 by coercing a permutation, written in *cycle notation*, into S_5 . We must enclose the product of cycles in quotations for SageMath to parse the input correctly.

```
sage: r = S5('(1,3)(2,4)'); r
(1,3)(2,4)
sage: s = S5('(1,4,3,2)'); s
(1,4,3,2)
```

We may also construct an element t using the list of images that it has as a function.

```
sage: t = S5([1,5,4,3,2]); t
(2,5)(3,4)
```

The product of cycles is taken from *left-to-right* and is, of course, not commutative.

```
sage: s*t
(1,4,2,3)
sage: t*s
(1,2,4,3)
sage: id*s
```

Let's compute the order of an element by using the object's `order()` method and check this directly.

```
sage: r.order()
2
sage: r*r
()
sage: s.order()
4
sage: s*s
(1,3)(2,4)
sage: s*s*s*s
()
```

The *exponent* of a group is the least common multiple of the orders of the elements.

```
sage: S5.exponent()
60
```

The `sign()` method is used to compute the sign of a permutation, indicating whether it can be written as the product of an even or an odd number of permutations.

```
sage: S5('(2,3,4)').sign()
1
sage: S5('(4,5)').sign()
-1
```

Each symmetric group S_n is a subgroup of S_{n+1} .

```
sage: S4 = SymmetricGroup(4)
sage: S4.is_subgroup(S5)
True
```

You can construct the subgroup generated by a list of elements by using the `subgroup()` method.

```
sage: H = S5.subgroup([r,s])
sage: H
Subgroup of SymmetricGroup(5) generated by [(1,3)(2,4), (1,4,3,2)]
sage: H.list()
[( ), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
```

We can test to see if the subgroup that we have just created has certain properties by using the appropriate methods. typing `H.is() <tab>` will give a list of several properties to test.

```
sage: H.is_abelian()
True
sage: H.is_cyclic()
True
```

The elements originally used to generate a subgroup are obtained with the `gens()` method. SageMath can't guarantee a minimal generating set, but `gens_small()` makes an attempt.

```
sage: H.gens()
[(1,3)(2,4), (1,4,3,2)]
sage: H.gens_small()
[(1,4,3,2)]
```

A useful tool for examining the structure of a group is the multiplication table, often called the *Cayley Table*. Invoke the group's `cayley_table()` method (also called `multiplication_table()`). The default uses letters to represent the group elements (in the order they appear using `list()`).

```
sage: S3 = SymmetricGroup(3)
sage: S3.cayley_table()
* a b c d e f
+-----+
a| a b c d e f
b| b a d c f e
c| c e a f b d
d| d f b e a c
e| e c f a d b
f| f d e b c a
sage: S3.list()
[( ), (2,3), (1,2), (1,2,3), (1,3,2), (1,3)]
```

We can also use the elements themselves, or give them names. Here we assign name based on the symmetries of a triangle: `u_i()` for reflections through the axis containing vertex `i()` and `r^1, r^2()` for the rotations.

```
sage: S3.cayley_table(names='elements')
*      |      ()      (2,3)      (1,2) (1,2,3) (1,3,2)      (1,3)
-----
()      |      ()      (2,3)      (1,2) (1,2,3) (1,3,2)      (1,3)
(2,3)  |      (2,3)      ()      (1,2,3)  (1,2)  (1,3) (1,3,2)
(1,2)  |      (1,2) (1,3,2)      ()      (1,3)  (2,3) (1,2,3)
(1,2,3)|      (1,2,3) (1,3)  (2,3) (1,3,2)      ()      (1,2)
(1,3,2)|      (1,3,2) (1,2)  (1,3)      () (1,2,3)  (2,3)
(1,3)  |      (1,3) (1,2,3) (1,3,2)  (2,3)  (1,2)      ()

sage: S3.cayley_table(names=['id','u1','u3','r1','r2','u2'])
*  id u1 u3 r1 r2 u2
+-----
id| id u1 u3 r1 r2 u2
u1| u1 id r1 u3 u2 r2
u3| u3 r2 id u2 u1 r1
r1| r1 u2 u1 r2 id u3
r2| r2 u3 u2 id r1 u1
u2| u2 r1 r2 u1 u3 id
```

General Permutation Groups

A permutation group is a subgroup of some symmetric group. We can construct a permutation group directly, without constructing the whole symmetric group, by giving a list of permutations to the `PermutationGroup` command.

```
sage: r = '(1,3)(2,4)(5)'
sage: s = '(1,3,2)'
sage: K = PermutationGroup([r,s])
sage: K
Permutation Group with generators [(1,3,2), (1,3)(2,4)]
sage: K.order()
12
```

Several important permutation groups can also be constructed directly. Here are the simplest.

```
sage: K= KleinFourGroup(); K
The Klein 4 group of order 4, as a permutation group
sage: K.list()
[(), (3,4), (1,2), (1,2)(3,4)]
sage: Q= QuaternionGroup(); Q.list()
[(), (1,2,3,4)(5,6,7,8), (1,3)(2,4)(5,7)(6,8),
(1,4,3,2)(5,8,7,6), (1,5,3,7)(2,8,4,6), (1,6,3,8)(2,5,4,7),
(1,7,3,5)(2,6,4,8), (1,8,3,6)(2,7,4,5)]
sage: [x.order() for x in Q]
[1, 4, 2, 4, 4, 4, 4, 4]
```

There are several families of permutation groups. The `CyclicPermutationGroup` in S_n is generated by the cycle $(1,2,\dots,n)$. The `DihedralGroup` is S_n is the symmetries of a regular n -gon with the vertices enumerated clockwise from 1 to n . It is generated by the rotation $(1,2,\dots,n)$ and a reflection. Use the `gens()` to see which reflection is used. The collection of all even permutations—permutations with positive sign—is a subgroup of S_5 obtained by the command `AlternatingGroup`.

```
sage: C = CyclicPermutationGroup(4); C
Cyclic group of order 4 as a permutation group
sage: C.list()
```

```
[(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
sage: D = DihedralGroup(4); D
Dihedral group of order 8 as a permutation group
sage: D.list()
[(), (2,4), (1,2)(3,4), (1,2,3,4), (1,3), (1,3)(2,4), (1,4,3,2),
(1,4)(2,3)]
sage: D.gens()
[(1,2,3,4), (1,4)(2,3)]
sage: A = AlternatingGroup(4); A
Alternating group of order 4!/2 as a permutation group
sage: A.cardinality()
12
```

Another builtin group is the `DiCyclicGroup` (see [the Group Properties article](#)). Let's check that the A_4 is not isomorphic to the dicyclic group with the same number of elements.

```
sage: B = DiCyclicGroup(3); B
Dicyclic group of order 12 as a permutation group
sage: B.list()
[(), (5,6,7), (5,7,6), (1,2)(3,4), (1,2)(3,4)(5,6,7), (1,2)(3,4)(5,7,6), (1,3,2,4)(6,
↪7), (1,3,2,4)(5,6), (1,3,2,4)(5,7), (1,4,2,3)(6,7), (1,4,2,3)(5,6), (1,4,2,3)(5,7)]
sage: A.is_isomorphic(B)
False
```

With any permutation group we may compute its cardinality, list its elements, compute the order of elements, etc. By using python's *list comprehensions* (see [Lists](#)) we can create a list of elements with certain properties. In this case we can construct the list of all elements of order 2.

```
sage: S5 = SymmetricGroup(5)
sage: T = [s for s in S5 if s.order() == 2]; T
[(4,5), (3,4), (3,5), (2,3), (2,3)(4,5), (2,4), (2,4)(3,5), (2,5), (2,5)(3,4), (1,2),
↪(1,2)(4,5), (1,2)(3,4), (1,2)(3,5), (1,3), (1,3)(4,5), (1,3)(2,4), (1,3)(2,5), (1,
↪4), (1,4)(3,5), (1,4)(2,3), (1,4)(2,5), (1,5), (1,5)(3,4), (1,5)(2,3), (1,5)(2,4)]
```

Next we will construct a permutation group H and list its members. This group H has different elements from `DihedralGroup(5)`, but is isomorphic to it.

```
sage: H = PermutationGroup(['(1,5), (3,4)', '(1,2,5,4,3)']); H
Subgroup of SymmetricGroup(5) generated by [(1,2,5,4,3), (1,5)(3,4)]
sage: H.list()
[(), (2,3)(4,5), (1,2)(3,5), (1,2,5,4,3), (1,3,4,5,2), (1,3)(2,4), (1,4,2,3,5), (1,
↪4)(2,5), (1,5)(3,4), (1,5,3,2,4)]
sage: H.order()
10
sage: D = DihedralGroup(5)
sage: D
Dihedral group of order 10 as a permutation group
sage: D.list()
[(), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5), (1,3)(4,5), (1,3,5,2,4), (1,4)(2,3), (1,4,2,
↪5,3), (1,5,4,3,2), (1,5)(2,4)]
sage: H == D
False
sage: H.is_isomorphic(D)
True
```

As with the symmetric group, we can pass a list of group elements to the method `subgroup()` to create a subgroup of any permutation group.

The list of all subgroups of a permutation group is obtained by the `subgroups()` method. It returns a list whose 0th element is the trivial subgroup.

```
sage: D = DihedralGroup(4)
sage: L = D.subgroups(); L
[Permutation Group with generators [()], Permutation Group with generators [(1,3)(2,
↪4)], Permutation Group with generators [(2,4)], Permutation Group with generators
↪[(1,3)], Permutation Group with generators [(1,2)(3,4)], Permutation Group with
↪generators [(1,4)(2,3)], Permutation Group with generators [(2,4), (1,3)(2,4)],
↪Permutation Group with generators [(1,2,3,4), (1,3)(2,4)], Permutation Group with
↪generators [(1,2)(3,4), (1,3)(2,4)], Permutation Group with generators [(2,4), (1,2,
↪3,4), (1,3)(2,4)]]
```

The join of two subgroups C and K , is the group generated by the union of the two subgroups. We get the union of C and K by “adding” the respective lists. In the example below, we see that the cyclic permutation group generated by $(1, 2, 3, 4, 5)$ and the Klein four group generate the whole symmetric group S_5 . Notice that the Klein four group is a subgroup of S_4 , which itself is a subgroup of S_5 .

```
sage: K = KleinFourGroup(); K.list()
[(), (3,4), (1,2), (1,2)(3,4)]
sage: C = CyclicPermutationGroup(5)
sage: CjK = PermutationGroup(C.list()+K.list())
Permutation Group with generators [(), (3,4), (1,2), (1,2)(3,4), (1,2,3,4,5), (1,3,5,
↪2,4), (1,4,2,5,3), (1,5,4,3,2)]
sage: CjK.gens_small(); CjK.cardinality()
[(1,2)(3,5,4), (1,4,5,3)]
120
sage: CjK == SymmetricGroup(5)
True
```

The centralizer of an element a (the subgroup of elements that commute with a) and the center of a group are constructed in the way you’d expect.

```
sage: D.center()
Subgroup of (Dihedral group of order 8 as a permutation group) generated by [(1,3)(2,
↪4)]
sage: D.centralizer(D('(1,3)(2,4)'))
Subgroup of (Dihedral group of order 8 as a permutation group) generated by [(1,2,3,
↪4), (1,4)(2,3)]
```

Quotients of Permutation Groups

In this section we explore normal subgroups and the quotient of a group by a normal subgroup. First we consider cosets and conjugation.

The alternating group A_4 has a subgroup isomorphic to the Klein four group that is normal.

```
sage: A4 = AlternatingGroup(4)
sage: g1 = A4('(1,4)(3,2)'); g2 = A4('(2,4)(1,3)')
sage: H = A4.subgroup([g1, g2]);
sage: H.is_normal(A4); H.is_isomorphic(KleinFourGroup())
True
True
```

Let’s compare the right and left cosets of H in A_4 .

```
sage: Hr = A4.cosets(H, side = 'right')
sage: Hl = A4.cosets(H, side = 'left')
sage: Hr; Hl
[[(), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3)], [(2,3,4), (1,3,2), (1,4,3), (1,2,4)], [(2,
↪4,3), (1,4,2), (1,2,3), (1,3,4)]]
[[(), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3)], [(2,3,4), (1,2,4), (1,3,2), (1,4,3)], [(2,
↪4,3), (1,2,3), (1,3,4), (1,4,2)]]
sage: Hr == Hl
False
```

We can see they are equal, but sage is comparing each coset as lists, and notes that the elements of the last two cosets are not listed in the same order. To rectify this, use `sorted()` to remind sage to order each coset. We are fortunate with this example that the cosets themselves are listed in the same order. Otherwise we would have to apply `sorted()` to the two lists of cosets.

```
sage: Hr_sorted = [sorted(S) for S in Hr]
sage: Hl_sorted = [sorted(S) for S in Hl]
sage: Hr_sorted == Hl_sorted
True
```

The conjugate by a of an element g is the element $a^{-1}ga$. The set of all conjugates of g as a varies is the conjugacy class of g . Below, we create a 3-cycle and compute its conjugacy class in S_4 and then in A_4 . This shows that two elements may be conjugate in S_4 but not in A_4 .

```
sage: S4 = SymmetricGroup(4)
sage: A4 = AlternatingGroup(4)
sage: g = S4('(1,3,4)')
sage: Set([a^(-1)*g*a for a in A4])
{(1,3,4), (1,4,2), (1,2,3), (2,4,3)}
sage: Set([a^(-1)*g*a for a in S4])
{(1,2,3), (1,3,4), (2,3,4), (2,4,3), (1,4,3), (1,2,4), (1,3,2), (1,4,2)}
```

The method `conjugacy_class_representatives()` chooses one element from each conjugacy class. Notice that there are two classes for 3-cycles in A_4 , but only one in S_4 .

```
sage: S4.conjugacy_classes_representatives()
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4)]
sage: A4.conjugacy_classes_representatives()
[(), (1,2)(3,4), (1,2,3), (1,2,4)]
```

The conjugate by a of a subgroup H is the group $a^{-1}Ha$ (recall that multiplication is left-to right). The group encompassing a and H need not be specified; sage just considers them inside the symmetric group containing all the integers that appear.

```
sage: H = CyclicPermutationGroup(4)
sage: K = H.conjugate(PermutationGroupElement('(3,5)')); K
Permutation Group with generators [(1,2,5,4)]
```

The normalizer of H in S_4 is the subgroup of elements of $a \in S_4$ such that $a^{-1}Ha = H$.

```
sage: S4.normalizer(H)
Permutation Group with generators [(2,4), (1,2,3,4), (1,3)(2,4)]
sage: H1 = H.conjugate(PermutationGroupElement('(2,4)')); H1
Permutation Group with generators [(1,4,3,2)]
sage: H1 == H
True
```


SageMath can compute all normal subgroups of a group G . Let's verify that S_4 has 2 non-trivial normal subgroups, the alternating group, and a group isomorphic to the Klein four group (but not equal to sage's standard Klein four group).

```
sage: S4 = SymmetricGroup(4)
sage: S4norms = S4.normal_subgroups(); S4norms
[Permutation Group with generators [], Permutation Group with generators [(1,3)(2,
↪4), (1,4)(2,3)], Permutation Group with generators [(2,4,3), (1,3)(2,4), (1,4)(2,
↪3)], Permutation Group with generators [(1,2), (1,2,3,4)]]
sage: K = S4norms[1]; K==KleinFourGroup()
False
sage: K.is_isomorphic(KleinFourGroup())
True
sage: A = S4norms[2]; A == AlternatingGroup(4)
True
```

We may now compute the quotient of G by the normal subgroups K and A in the previous example. As expected G/A is isomorphic to S_2 . Since G has 24 elements and K has 4 elements, the quotient has 6 elements. We can check that it is isomorphic to S_3 .

```
sage: G.quotient(A)
Permutation Group with generators [(1,2)]
sage: H = G.quotient(K); H
Permutation Group with generators [(1,2)(3,6)(4,5), (1,3,5)(2,4,6)]
sage: H.is_isomorphic(SymmetricGroup(3))
True
```

SageMath can also compute the normalizer of a subgroup H of G , which is the largest subgroup of G containing H in which H is normal. Here we compute the normalizer of the cyclic permutation group H created above inside of S_4 . We get the dihedral group D_4 . If we had used a different 4-cycle the resulting group may have been isomorphic to D_4 but not equal to it.

```
sage: G.normalizer(H).cardinality()
8
sage: HK.normalizer(H) == DihedralGroup(4)
True
```

For some groups the list of all subgroups may be large. To better understand the subgroups of G we may compute one group from each conjugacy class. The following computations show that there are 30 subgroups of S_4 but only 11 up to conjugacy. Every other subgroup is not only isomorphic to one of the 11, given by `conjugacy_classes_subgroups()`, but is also isomorphic via conjugation by some element of G .

```
sage: G
Symmetric group of order 4! as a permutation group
sage: G.subgroups()
[Permutation Group with generators [], Permutation Group with generators [(1,2)(3,
↪4)], Permutation Group with generators [(1,3)(2,4)], Permutation Group with
↪generators [(1,4)(2,3)], Permutation Group with generators [(3,4)], Permutation
↪Group with generators [(2,3)], Permutation Group with generators [(2,4)],
↪Permutation Group with generators [(1,2)], Permutation Group with generators [(1,
↪3)], Permutation Group with generators [(1,4)], Permutation Group with generators
↪[(2,4,3)], Permutation Group with generators [(1,2,3)], Permutation Group with
↪generators [(1,4,2)], Permutation Group with generators [(1,3,4)], Permutation
↪Group with generators [(1,4)(2,3), (1,3)(2,4)], Permutation Group with generators
↪[(1,2)(3,4), (3,4)], Permutation Group with generators [(1,4)(2,3), (2,3)],
↪Permutation Group with generators [(1,3)(2,4), (2,4)], Permutation Group with
↪generators [(1,2)(3,4), (1,3,2,4)], Permutation Group with generators [(1,3)(2,4),
↪(1,4,3,2)], Permutation Group with generators [(1,4)(2,3), (1,2,4,3)], Permutation
↪Group with generators [(3,4), (2,4,3)], Permutation Group with generators [(3,4),
↪(1,3,4)], Permutation Group with generators [(1,2), (1,2,3)], Permutation Group
↪with generators [(1,2), (1,4,2)], Permutation Group with generators [(1,3)(2,4), (1,3)
↪(4,2,3), (1,2)], Permutation Group with generators [(1,2)(3,4), (1,3)(2,4), (1,4)],
↪Permutation Group with generators [(1,4)(2,3), (1,2)(3,4), (1,3)], Permutation
↪Group with generators [(1,3)(2,4), (1,4)(2,3), (2,4,3)], Permutation Group with
↪generators [(1,3)(2,4), (1,4)(2,3), (2,4,3), (1,2)]]
```

```
sage: len(G.subgroups())
30
sage: G.conjugacy_classes_subgroups()
[Permutation Group with generators [()], Permutation Group with generators [(1,3)(2,
↪4)], Permutation Group with generators [(3,4)], Permutation Group with generators
↪[(2,4,3)], Permutation Group with generators [(1,4)(2,3), (1,3)(2,4)], Permutation
↪Group with generators [(1,2)(3,4), (3,4)], Permutation Group with generators [(1,
↪2)(3,4), (1,3,2,4)], Permutation Group with generators [(3,4), (2,4,3)],
↪Permutation Group with generators [(1,3)(2,4), (1,4)(2,3), (1,2)], Permutation
↪Group with generators [(1,3)(2,4), (1,4)(2,3), (2,4,3)], Permutation Group with
↪generators [(1,3)(2,4), (1,4)(2,3), (2,4,3), (1,2)]]
sage: len(G.conjugacy_classes_subgroups())
11
```

Exercises:

1. Find two subgroups of A_4 that are conjugate in S_4 but are not conjugate in A_4 .

Permutation Group Homomorphisms

To construct a homomorphism between two permutation groups we use the `PermutationGroupMorphism()` command. For an example let us use the two isomorphic groups that we constructed earlier.

```
sage: G = SymmetricGroup(5)
sage: r = G('(1,2,5,4,3)')
sage: s = G('(1,5),(3,4)')
sage: H = G.subgroup([r,s])
sage: H
Subgroup of SymmetricGroup(5) generated by [(1,2,5,4,3), (1,5)(3,4)]
sage: D = DihedralGroup(5)
sage: D
Dihedral group of order 10 as a permutation group
```

A homomorphism between these is constructed by listing an association between the *generators* of one group to the generators of the other. To see these we will use the `gens()` method provided by our groups

```
sage: H.gens()
[(1,2,5,4,3), (1,5)(3,4)]
sage: D.gens()
[(1,2,3,4,5), (1,5)(2,4)]
```

We construct the homomorphism $\phi : H \rightarrow D$ that sends $(1, 2, 5, 4, 3) \rightarrow (1, 2, 3, 4, 5)$ and $(1, 5)(3, 4) \rightarrow (1, 5)(2, 4)$ as follows:

```
sage: phi = PermutationGroupMorphism(H,D,H.gens(), D.gens())
sage: phi
Homomorphism : Permutation Group with generators [(1,2,5,4,3), (1,5)(3,4)] -->
↪Dihedral group of order 10 as a permutation group
```

We can apply this homomorphism as we would any function, by calling it.

```
sage: phi( '(2,3)(4,5) ' )
(1,3)(4,5)
sage: phi( '(1,5,3,2,4) ' )
(1,3,5,2,4)
sage: phi( '(1,5) ' )
-----
```

```
AttributeError                                Traceback (most recent call last)
...
AttributeError: 'str' object has no attribute '_gap_init_'
```

Note that we get an `AttributeError` because the permutation $(1, 5)$ is not in the domain of `phi()`.

The homomorphism also comes equipped with a few useful methods, the most useful is the `kernel()` method, which yields the kernel of the homomorphism. Since this homomorphism is an injection, the kernel is just the trivial group.

```
sage: phi.kernel()
Permutation Group with generators [()]
```

The *direct product* of two `PermutationGroups` produces another `PermutationGroup`, but in a larger symmetric group. The output is a list of length five consisting of the direct product followed by four homomorphisms. The first two homomorphisms are the natural ones from each factor into the product. The second two homomorphisms are the natural projections from the product on to each factor.

```
sage: C4 = CyclicPermutationGroup(4)
sage: C3 = CyclicPermutationGroup(3)
sage: C4xC3 = C4.direct_product(C3); C4xC3
(Permutation Group with generators [(5,6,7), (1,2,3,4)], Permutation group morphism:
From: Cyclic group of order 4 as a permutation group
To: Permutation Group with generators [(5,6,7), (1,2,3,4)]
Defn: Embedding( Group( [ (1,2,3,4), (5,6,7) ] ), 1 ), Permutation group morphism:
From: Cyclic group of order 3 as a permutation group
To: Permutation Group with generators [(5,6,7), (1,2,3,4)]
Defn: Embedding( Group( [ (1,2,3,4), (5,6,7) ] ), 2 ), Permutation group morphism:
From: Permutation Group with generators [(5,6,7), (1,2,3,4)]
To: Cyclic group of order 4 as a permutation group
Defn: Projection( Group( [ (1,2,3,4), (5,6,7) ] ), 1 ), Permutation group morphism:
From: Permutation Group with generators [(5,6,7), (1,2,3,4)]
To: Cyclic group of order 3 as a permutation group
Defn: Projection( Group( [ (1,2,3,4), (5,6,7) ] ), 2 ))
```

If we just want the direct product group, we must select the 0th element of the direct product.

```
sage: C4xC3[0]
Permutation Group with generators [(1,2,3,4), (5,6,7)]
```

Exercises:

1. There is a homomorphism from the dicyclic group of index n to the dihedral group of index n . Construct it and find the kernel.

4.2.2 Matrix Groups

Please contribute!

4.2.3 Abelian Groups

Please contribute!

4.3 Linear Algebra

4.3.1 Vectors and Matrices

To create a vector, use the `vector()` command with a list of entries. Scalar multiples and the dot product are straightforward to compute. As with lists, vectors are indexed starting from 0.

```
sage: v= vector([1,2,3,4])
sage: v[0]
1
sage: v[4]
ERROR: An unexpected error occurred while tokenizing input
```

Arithmetic on vectors is what one would expect. SageMath will produce an error message if you add two vectors of different lengths.

```
sage: 7*v
(7, 14, 21, 28)
sage: v + vector([2,1,4,5])
(3, 3, 7, 9)
sage: v*v
sage: v + vector([2,1,4])
-----
TypeError                                 Traceback (most recent call last)
/Users/mosullivan/Work/SageMath/Tutorial/sdsu-sage-tutorial/<ipython console> in
↳<module>()

/Applications/sage/local/lib/python2.6/site-packages/sage/structure/element.so in
↳sage.structure.element.ModuleElement.__add__ (sage/structure/element.c:7627) ()

/Applications/sage/local/lib/python2.6/site-packages/sage/structure/coerce.so in sage.
↳structure.coerce.CoercionModel_cache_maps.bin_op (sage/structure/coerce.c:6995) ()

TypeError: unsupported operand parent(s) for '+': 'Ambient free module of rank 4 over
↳the principal ideal domain Integer Ring' and 'Ambient free module of rank 3 over
↳the principal ideal domain Integer Ring'
```

We use the `matrix()` command to construct a matrix with a list of the *rows* of the matrix as the argument.

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

We can also construct a matrix by specifying all of the coordinates in a single matrix while specifying the dimensions of the matrix. The following command creates a matrix with 4 rows and 2 columns.

```
sage: matrix(4,2, [1,2,3,4,5,6,7,8])
[1 2]
[3 4]
[5 6]
[7 8]
```

If the matrix that we want to construct is square we can omit the number of columns from the argument.

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

By default, SageMath constructs the matrix over the smallest universe which contains the coordinates.

```
sage: parent(matrix(2, [1, 2, 3, 4]))
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: parent(matrix(2, [1, 2/1, 3, 4]))
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: parent(matrix(2, [x, x^2, x-1, x^3]))
Full MatrixSpace of 2 by 2 dense matrices over Symbolic Ring
```

We can specify the universe for the coordinates of a matrix or vector by giving it as an optional argument.

```
sage: matrix(QQ, 2, [1.1, 1.2, 1.3, 1.4])
[11/10 6/5]
[13/10 7/5]
```

There are shortcuts in SageMath to construct some of the more commonly used matrices. To construct the identity matrix we use the `identity_matrix()` function.

```
sage: identity_matrix(3)
[1 0 0]
[0 1 0]
[0 0 1]
```

To construct the zero matrix we may use `zero_matrix()` or the regular matrix function with no list input.

```
sage: zero_matrix(2, 2)
[0 0]
[0 0]
sage: matrix(2)
[0 0]
[0 0]
sage: matrix(2, 3)
[0 0 0]
[0 0 0]
```

Note that if we use `zero_matrix()` we must input two integers.

Exercises:

1. Use SageMath to construct the vector $v = (4, 10, 17, 28, 2)$
2. Construct the following matrix over the rational numbers in SageMath.

$$\begin{pmatrix} 5 & 3 & 2 \\ 4 & 7 & 10 \\ 2 & 11 & 1 \end{pmatrix}$$

3. Construct a 10x10 identity matrix.
4. Construct a 20x10 zero matrix.

4.3.2 Matrix Arithmetic

You should be familiar with *Vectors and Matrices*.

We may use $+$, $-$, $*$ and $^$ for matrix addition, subtraction, multiplication and exponents.

```
sage: A=matrix(2, [1,1,0,1])
sage: B=matrix(2, [1,0,1,1])
sage: A+B
[2 1]
[1 2]
sage: A*B
[2 1]
[1 1]
sage: B*A
[1 1]
[1 2]
sage: A-B
[ 0 1]
[-1 0]
sage: A^3
[1 3]
[0 1]
```

We can compute the *inverse* of a matrix by raising it to the -1 -th power.

```
sage: A^-1
[ 1 -1]
[ 0  1]
```

If the matrix is not invertible SageMath will complain about a `ZeroDivisionError`.

```
sage: A = matrix([[4,2],[8,4]])
sage: A^-1
-----
ZeroDivisionError                                Traceback (most recent call last)
... (Long error message)
ZeroDivisionError: input matrix must be nonsingular
```

When multiplying vectors and matrices; vectors can be considered both as rows or as columns, so you can multiply a 3-vector by a $3 \times n$ matrix on the right, or by a $n \times 3$ matrix on the left.

```
sage: x = vector([12,3,3])
sage: x
(12, 3, 3)
sage: A
[1 2 3]
[4 5 6]
sage: A*x
(27, 81)
sage: B = transpose(A)
sage: B
[1 4]
[2 5]
[3 6]
sage: x*B
(27, 81)
```

We use the `det()` method to calculate the *determinant* of a square matrix.

```
sage: A = matrix([[[-1/2,0,-1],[0,-2,2],[1,0,-1/2]]]); A
[-1/2  0  -1]
```

```
[ 0 -2 2]
[ 1 0 -1/2]
sage: A.det()
-5/2
```

We use the `trace()` method to compute the **trace** of a square (or any) matrix.

```
sage: A = matrix([[ -1/2, 0, -1], [0, -2, 2], [1, 0, -1/2]]); A.trace()
-3
```

This was a trivial case that could have been easily done by hand, but there will be circumstances when knowing the `trace()` method will turn out to be useful.

To check if a matrix is invertible we use the `is_invertible()` method.

```
sage: A=matrix(2, [1,1,0,1])
sage: A.is_invertible()
True
sage: A.det()
1
```

The invertibility of a matrix depends on the ring or field it is defined over. For example:

```
sage: B=matrix(2, [1,2,3,4])
sage: B.is_invertible()
False
```

In this example, SageMath assumes that the matrix `B` is defined over the integers and not the rationals, where it does not have an inverse. But if we define `B` as a matrix over the rationals, we obtain different results.

```
sage: B = matrix(QQ, 2, [1,2,3,4])
sage: B
[1 2]
[3 4]
sage: B.is_invertible()
True
```

If we ask SageMath to compute the inverse of a matrix over the integers it will automatically coerce `B` into a matrix over the rationals if necessary.

```
sage: B = matrix(2, [1,2,3,4])
sage: parent(B)
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: B^-1
[ -2 1]
[ 3/2 -1/2]
sage: parent(B^-1)
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

Exercises:

1. Consider the matrices:

$$A = \begin{pmatrix} 1 & 3 \\ 7 & 8 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 4 & 8 \\ 9 & 15 \end{pmatrix}$$

Compute the following :

$$a) : \text{math} : 'A + B' : \text{math} : 'AB' : \text{math} : 'B^{-1}d' : \text{math} : 'B^{-1}AB'$$

2. Which of the following matrices is invertible over \mathbb{Z} ? What about \mathbb{Q} ?

$$A = \begin{pmatrix} 2 & 8 \\ 4 & 16 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 7 \\ 13 & 24 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 4 \\ 2 & 7 \end{pmatrix} \quad D = \begin{pmatrix} 4 & 6 \\ 8 & -2 \end{pmatrix}$$

4.3.3 Matrix Manipulation

You should be familiar with *Vectors and Matrices* and *Matrix Arithmetic*.

In this section we will cover some of the commands that we can use to *manipulate* matrices. Let's begin by defining a matrix over the rational numbers.

```
sage: M = matrix(QQ, [[1,2,3],[4,5,6],[7,8,9]]); M
[1 2 3]
[4 5 6]
[7 8 9]
```

To get a list of row and column vectors, we use the `rows()` and `columns()` methods.

```
sage: M.rows()
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
sage: M.columns()
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

The following examples show how to get a particular row or column vector. Remember that SageMath follows Python's convention that all of the indexes begin with zero.

```
sage: M.row(0)
(1, 2, 3)
sage: M.row(2)
(7, 8, 9)
sage: M.column(1)
(2, 5, 8)
sage: M.column(2)
(3, 6, 9)
```

You can even get a list of the diagonal entries, by calling the `diagonal()` method.

```
sage: M.diagonal()
[1, 5, 9]
```

SageMath also allows us to construct new matrices from the row and/or column vectors.

```
sage: M.matrix_from_columns([0,2])
[1 3]
[4 6]
[7 9]
sage: M.matrix_from_rows([0,2])
[1 2 3]
[7 8 9]
sage: M.matrix_from_rows_and_columns([0,2],[0,2])
[1 3]
[7 9]
```

It should be noted that the `matrix_from_rows_and_columns()` returns the *intersection* of the rows and columns specified. In the above example we are selecting the matrix that consists of the four 'corners' of our 3×3 matrix.

Next we will discuss some of the elementary row operations. To multiply a row or column by a number we use the `rescale_row()` or `rescale_column()` methods. Note that these commands change the matrix itself.

```
sage: M.rescale_row(1, -1/4); M
[ 1  2  3]
[ -1 -5/4 -3/2]
[ 7  8  9]
sage: M.rescale_col(2, -1/3); M
[ 1  2 -1]
[ -1 -5/4 1/2]
[ 7  8 -3]
sage: M.rescale_row(1, -4); M
[ 1  2 -1]
[ 4  5 -2]
[ 7  8 -3]
```

We can add a multiple of a row or column to another row or column by using the `add_multiple_of_row()` method. The first command takes -4 times the row 0 and adds it to row 1.

```
sage: M.add_multiple_of_row(1, 0, -4); M
[ 1  2 -1]
[ 0 -3  2]
[ 7  8 -3]
sage: M.add_multiple_of_row(2, 0, -7); M
[ 1  2 -1]
[ 0 -3  2]
[ 0 -6  4]
```

The same can be done with the column vectors, which are also zero indexed.

```
sage: M.add_multiple_of_column(1, 0, -2); M
[ 1  0 -1]
[ 0 -3  2]
[ 0 -6  4]
sage: M.add_multiple_of_column(2, 0, 1); M
[ 1  0  0]
[ 0 -3  2]
[ 0 -6  4]
```

If we don't like the ordering of our rows or columns we can swap them in place.

```
sage: M.swap_rows(1, 0); M
[ 0 -3  2]
[ 1  0  0]
[ 0 -6  4]
sage: M.swap_columns(0, 2); M
[ 2 -3  0]
[ 0  0  1]
[ 4 -6  0]
```

If we want to change a row or column of M then we use the `set_column()` or `set_row()` methods.

```
sage: M.set_column(0, [1, 2, 3]); M
[ 1 -3  0]
[ 2  0  1]
[ 3 -6  0]
sage: M.set_row(0, [1, 2, 5]); M
[ 1  2  5]
```

```
[ 2  0  1]
[ 3 -6  0]
```

And finally if we want to change a whole “block” of a matrix, we use the `set_block()` method with the coordinates of where we want the upper left corner of the block to begin.

```
sage: B = matrix(QQ, [ [1,0 ], [0,1]]); B
[1 0]
[0 1]
sage: M.set_block(1,1,B); M
[1 2 5]
[2 1 0]
[3 0 1]
```

Of course, if all we want is the *echelon form* of the matrix we can use either the `echelon_form()` or `echelonize()` methods. The difference between the two is the former returns a copy of the matrix in echelon form without changing the original matrix and the latter alters the matrix itself.

```
sage: M.echelon_form()
[1 0 0]
[0 1 0]
[0 0 1]

sage: M.echelonize(); M
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
```

Next we use the *augmented* matrix and the echelon form to solve a 3×4 system of the form $Mx = b$. First we define the matrix M and the vector b

```
sage: M = matrix(QQ, [[2,4,6,2,4], [1,2,3,1,1], [2,4,8,0,0], [3,6,7,5,9]]); M
[2 4 6 2 4]
[1 2 3 1 1]
[2 4 8 0 0]
[3 6 7 5 9]
sage: b = vector(QQ, [56, 23, 34, 101])
```

Then we construct the augmented matrix ($M | b$), store it in the variable M_aug and compute its echelon form.

```
sage: M_aug = M.augment(b); M_aug
[ 2  4  6  2  4 56]
[ 1  2  3  1  1 23]
[ 2  4  8  0  0 34]
[ 3  6  7  5  9 101]
sage: M_aug.echelon_form()
[ 1  2  0  4  0 21]
[ 0  0  1 -1  0 -1]
[ 0  0  0  0  1  5]
[ 0  0  0  0  0  0]
```

This tells us that we have a one dimensional solution space that consists of vectors of the form $v = c(-2, 1, 0, 0, 0) + (21, 0, 1, 0, 5)$.

```
sage: M*vector([21,0,-1,0,5])
(56, 23, 34, 101)
sage: M*vector([-2,1,0,0,0])
```

```
(0, 0, 0, 0)
```

If all we need is a *single* solution to this system, we can use the `solve_right()` method.

```
sage: M.solve_right(b)
(21, 0, -1, 0, 5)
```

Exercises:

1. Consider the matrix.

$$A = \begin{pmatrix} 4 & 17 & 23 \\ 1/32 & 2 & 17 \\ 16 & -23 & 27 \end{pmatrix}$$

Use only the elementary row operations discussed to put A into *echelon* form.

2. Using the commands discussed in this section, transform the matrix on the left into the matrix on the right.

- 1.

$$\begin{pmatrix} -7 & -1 & 1 & 4 & 0 \\ -8 & -2 & 4 & 2 & 6 \\ 1 & 1 & -3 & 3 & 0 \\ 0 & 8 & 13 & -2 & 0 \\ 1 & 4 & 0 & -1 & 4 \end{pmatrix} \quad \begin{pmatrix} -7 & -8 & 1 & 0 & 1 \\ -1 & -2 & 1 & 8 & 4 \\ 1 & 4 & -3 & 13 & 0 \\ 4 & 2 & 3 & -2 & -1 \\ 0 & 6 & 0 & 0 & 4 \end{pmatrix}$$

- 2.

$$\begin{pmatrix} -1 & -2 & 1 & -13 \\ -3 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ -2 & -1 & -9 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 & 100 \\ 0 & 1 & 0 & 12 \\ 0 & 0 & 1 & 111 \\ 0 & 0 & 0 & 202 \end{pmatrix}$$

- 3.

$$\begin{pmatrix} 0 & -1 & 1 \\ -2 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & -1 & 1 & -4 \\ -2 & 1 & -1 & -1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

4.3.4 Vector and Matrix Spaces

It is sometimes useful to create the space of all matrices of particular dimension, for which we use the `MatrixSpace()` function. We must specify the field (or indeed any ring) where the entries live.

```
sage: MatrixSpace(QQ, 2, 3)
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

If we input a ring R and an integer n we get the matrix ring of $n \times n$ matrices over R . Coercion can be used to construct the zero matrix, the identity matrix, or a matrix with specified entries as shown.

```
sage: Mat = MatrixSpace(ZZ, 2); Mat
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: Mat(1)
[1 0]
```

```
[0 1]
sage: Mat(0)
[0 0]
[0 0]
sage: Mat([1,2,3,4])
[1 2]
[3 4]
```

We may compute various spaces associated to a matrix.

```
sage: Mat = MatrixSpace(QQ, 3, 4)
sage: A = Mat([[1,2,3,4], [1,3,4,4], [2,5,7,8]])
sage: A
[1 2 3 4]
[1 3 4 4]
[2 5 7 8]
sage: A.rank()
2
sage: A.right_kernel()
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 0 -1/4]
[ 0 1 -1 1/4]
sage: A.left_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 1 -1]
sage: A.row_space()
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[1 0 1 4]
[0 1 1 0]
```

Exercises:

1. For the following 5x3 matrix:

$$\begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & -3 \\ 1 & 1 & 1 \\ 0 & -6 & -20 \\ 0 & 0 & 0 \end{pmatrix}$$

Use SageMath to compute the bases for the following spaces:

- (a) The right and left kernel.
- (b) The row space.
- (c) The column space.

4.3.5 Mini-Topic: The Jordan Canonical Form

For every linear transformation $T : \mathbb{R}^n \rightarrow \mathbb{R}^n$ there is a basis of \mathbb{R}^n such that the matrix $[m]_{\mathcal{B}}$ is in an *almost* diagonal form. This unique matrix is called the *Jordan Canonical Form* of T . For more information on this please refer to this [article](#) on Wikipedia. To demonstrate some common tools that we use in SageMath we will compute this basis for the linear transformation

$$T(x, y, z, t) = (2x + y, 2y + 1, 3z, y - z + 3t).$$

We will begin by defining T in SageMath.

```
sage: T(x,y,z,t) = (2*x+y, 2*y+1, 3*z, y - z + 3*t)
```

Now, let's use the standard ordered basis of \mathbb{R}^3 to find the matrix form of T .

```
sage: T(1,0,0,0), T(0,1,0,0), T(0,0,1,0), T(0,0,0,1)
((2, 1, 0, 0), (1, 3, 0, 1), (0, 1, 3, -1), (0, 1, 0, 3))
```

Note that since SageMath uses rows to construct a matrix we must use the `transpose()` function to get the matrix we expect.

```
sage: M = transpose(matrix([[2,1,0,0],[0,2,1,0], [0,0,3,0],[0,1,-1,3]])); M
[ 2  1  0  0]
[ 0  2  1  0]
[ 0  0  3  0]
[ 0  1 -1  3]
```

Once we have the matrix we will compute its *characteristic polynomial* and then factor it in order to find its eigenvalues.

```
sage: f = M.characteristic_polynomial(); f
x^4 - 10*x^3 + 37*x^2 - 60*x + 36
sage: f.factor()
(x - 3)^2 * (x - 2)^2
```

Alternatively, you can compute the eigenvalues directly.

```
sage: eval_M = M.eigenvalues(); eval_M;
[3, 3, 2, 2]
```

Above we have two eigenvalues $\lambda_1 = 3$ and $\lambda_2 = 2$ and both are of algebraic multiplicity 2. Now we need to look at the associated *eigenvectors*. To do so we will use the `eigenvectors_right()` method.

```
sage: evec_M = M.eigenvectors_right(); evec_M
[(3, [
(1, 1, 1, 0),
(0, 0, 0, 1)
]), 2), (2, [
(1, 0, 0, 0)
]), 2]]
sage: evec_M[1][1][0]
(1, 0, 0, 0)
```

What is returned is a `list()` of ordered triples. Each triple consists of an eigenvalue followed by a list with a basis for the associated eigenspace followed by the dimension of the associated eigenspace. Note that the eigenvalue 2 has algebraic multiplicity of 2 but geometric multiplicity only 1. This means that we will have to compute a *generalized eigenvector* for this eigenvalue. We will do this by solving the system $(M - 2I)v = x$, where x is the eigenvector $(1, 0, 0, 0)$. We will use the `echelon_form()` of the augmented matrix to solve the system.

```
sage: (M - 2*identity_matrix(4)).augment(evec_M[1][1][0])
[ 0  1  0  0  1]
[ 0  0  1  0  0]
[ 0  0  1  0  0]
[ 0  1 -1  1  0]
sage: __.echelon_form()
[ 0  1  0  0  1]
[ 0  0  1  0  0]
[ 0  0  0  1 -1]
```

```
[ 0 0 0 0 0]
sage: gv = vector([1,1,0,-1]); gv
(1, 1, 0, -1)
```

With the generalized eigenvector gv , we now have the right number of linearly independent vectors to form a basis for our *Jordan Form* matrix. We will next form the *change of basis matrix* that consists of these vectors as columns.

```
sage: S = transpose( matrix( [[1,1,1,0],[0,0,0,1],[1,0,0,0],gv])); S
[ 1 0 1 1]
[ 1 0 0 1]
[ 1 0 0 0]
[ 0 1 0 -1]
```

Now we will compute the matrix representation of T with respect to this basis.

```
sage: S.inverse()*M*S
[3 0 0 0]
[0 3 0 0]
[0 0 2 1]
[0 0 0 2]
```

And there it is, the *Jordan Canonical Form* of the linear transformation T . Of course we could have just used SageMath's built in `jordan_form()` method to compute this directly.

```
sage: M.jordan_form()
[3|0|0 0]
[---+---]
[0|3|0 0]
[---+---]
[0|0|2 1]
[0|0|0 2]
```

But that wouldn't be any fun!

Exercises:

1. Compute a jordan basis for the following matrix using the steps outlined in this section.

$$\begin{pmatrix} 1 & 2 & 0 & 2 \\ 0 & 2 & 0 & 0 \\ -1 & 2 & -\frac{1}{2} & -2 \\ 0 & 2 & 0 & 2 \end{pmatrix}$$

4.4 Rings

4.4.1 Polynomial Rings

Constructing polynomial rings in SageMath is fairly straightforward. We just specify the name of the “indeterminate” variable as well as the coefficient ring.

```
sage: R.<x>=PolynomialRing(ZZ)
sage: R
Univariate Polynomial Ring in x over Integer Ring
```

Once the polynomial ring has been defined we can construct a polynomial without any special coercions.

```
sage: p = 2*x^2 + (1/2)*x + (3/5)
sage: parent(p)
Univariate Polynomial Ring in x over Rational Field
```

Though x is the most common choice for a variable, we could have chosen any letter for the indeterminate.

```
sage: R.<Y>=PolynomialRing(QQ)
sage: R
Univariate Polynomial Ring in Y over Rational Field
```

Polynomials with rational coefficients in Y are now valid objects in SageMath.

```
sage: q = Y^4 + (1/2)*Y^3 + (1/3)*Y + (1/4)
sage: q
Y^4 + 1/2*Y^3 + 1/3*Y + 1/4
sage: parent(q)
Univariate Polynomial Ring in Y over Rational Field
```

We can define polynomial rings over any ring or field.

```
sage: Z7=Integers(7)
sage: R.<x>=PolynomialRing(Z7); R
Univariate Polynomial Ring in x over Ring of integers modulo 7
```

When entering a polynomial into SageMath the coefficients are automatically coerced into the ring or field specified.

```
sage: p = 18*x^2 + 7*x + 16; p
4*x^2 + 2
sage: parent(p)
Univariate Polynomial Ring in x over Ring of integers modulo 7
```

Of course this coercion has to be well defined.

```
sage: q = x^4 + (1/2)*x^3 + (1/3)*x^2 + (1/4)*x + (1/5)
-----
TypeError                                 Traceback (most recent call last) ...
TypeError: unsupported operand parent(s) for '*': 'Rational Field' and 'Univariate_
↳Polynomial Ring in x over Ring of integers modulo 7'
```

When prudent, SageMath will extend the universe of definition to fit the polynomial entered. For example, if we ask for a rational coefficient in a polynomial ring over \mathbb{Z} , SageMath will naturally coerce this polynomial into a ring over \mathbb{Q}

```
sage: S.<y>=PolynomialRing(ZZ)
sage: 1/2*y
1/2*y
sage: parent(1/2*y)
Univariate Polynomial Ring in y over Rational Field
```

It should be noted that the ring S hasn't been changed at all. Nor is $(1/2)*y$ in the universe ``S. This can be easily verified.

```
sage: S
Univariate Polynomial Ring in y over Integer Ring
sage: (1/2)*y in S
False
```

Once constructed, the basic arithmetic with polynomials is straightforward.

```
sage: R.<x>=PolynomialRing(QQ)
sage: f=x+1
sage: g=x^2+x-1
sage: h=1/2*x+3/4
sage: f+g
x^2 + 2*x
sage: g-h
x^2 + 1/2*x - 7/4
sage: f*g
x^3 + 2*x^2 - 1
sage: h^3
1/8*x^3 + 9/16*x^2 + 27/32*x + 27/64
```

We can also divide elements of the polynomial ring, but this changes the parent.

```
sage: f/g
(x + 1)/(x^2 + x - 1)
sage: parent(f/g)
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

A fundamental attribute of a polynomial is its degree. We use the `degree()` method to calculate this.

```
sage: R.<x>=PolynomialRing(QQ)
sage: (x^3+3).degree()
3
sage: R(0).degree()
-1
```

Notice that by convention SageMath sets the degree of 0 to be -1.

The polynomial ring over a field has a division algorithm. As with the integers, we may use the `//` operator to determine the *quotient* and the `%` operator to determine the *remainder* of a division.

```
sage: R.<x>=PolynomialRing(Integers(7))
sage: f=x^6+x^2+1
sage: g=x^3+x+1
sage: f // g
x^3 + 6*x + 6
sage: f % g
2*x^2 + 2*x + 2
```

Additionally, if the coefficients of the polynomial are in \mathbb{Z} or \mathbb{Q} , we may use the `divmod()` command to compute both at the same time.

```
sage: S.<y>=PolynomialRing(QQ)
sage: a=(y+1)*(y^2+1)
sage: b=(y+1)*(y+5)
sage: a // b
y - 5
sage: a % b
26*y + 26
sage: divmod(a,b)
(y - 5, 26*y + 26)
```

For a field F , the polynomial ring $F[x]$ has a division algorithm, so we have a unique greatest common divisor (gcd) of polynomials. This can be computed using the `gcd()` command.


```
sage: R.<x> = PolynomialRing(QQ)
sage: p = x^4 + 2*x^3 + 2*x^2 + 2*x + 1
sage: q = x^4 - 1
sage: gcd(p,q)
x^3 + x^2 + x + 1
```

The greatest common divisor of two integers can be represented as a linear combination of the two integers, and the extended Euclidean algorithm is used to determine one such linear combination. Similarly, the greatest common divisor of polynomials $a(x)$ and $b(x)$ may be written in the form $a(x)f(x) + b(x)g(x)$ for some polynomials $f(x)$ and $g(x)$. We may use the `xgcd()` function to compute the multipliers $f(x)$ and $g(x)$.

```
sage: R.<x>=PolynomialRing(ZZ)
sage: a=x^4-1
sage: b=(x+1)*x
sage: xgcd(a,b)
(x + 1, -1, x^2 - x + 1)
sage: d,u,v=xgcd(a,b)
sage: a*u+b*v
x + 1
```

To check whether a polynomial is irreducible, we use its `is_irreducible()` method.

```
sage: R.<x>=PolynomialRing(Integers(5))
sage: (x^3+x+1).is_irreducible()
True
sage: (x^3+1).is_irreducible()
False
```

This method is only suitable for polynomial rings that are defined over a field, as polynomials defined more generally may not possess a unique factorization.

To compute the *factorization* of a polynomial, where defined, we use the `factor()` command.

```
sage: R.<x>=PolynomialRing(Integers(5))
sage: factor(x^3+x+1)
x^3 + x + 1
sage: factor(x^3+1)
(x + 1) * (x^2 + 4*x + 1)
```

In the example above, we see a confirmation that $x^3 + x + 1$ is irreducible in $\mathbb{Z}_5[x]$ whereas $x^3 + 1$ may be factored, hence is reducible.

We can also consider polynomials in $R[x]$ as functions from R to R by *evaluation*, that is by substituting the indeterminate variable with a member of the coefficient ring. Evaluation of polynomials in SageMath works as expected, by *calling* the polynomial like a function.

```
sage: R.<x>=PolynomialRing(Integers(3))
sage: f=2*x+1
sage: f(0)
1
sage: f(1)
0
sage: f(2)
2
```

Calculating the *roots*, or *zeros*, of a polynomial can be done by using the `roots()` method.

```
sage: ((x-1)^2*(x-2)*x^3).roots()
[(2, 1), (1, 2), (0, 3)]
```

SageMath returns a list of pairs (r, m) where r is the root and m is its multiplicity. Of course, a polynomial need not have any roots and in this case the *empty list* is returned.

```
sage: (x^2+1).roots()
[]
```

Multivariate Polynomial Rings

Defining a polynomial ring with more than one variable can be done easily by supplying an extra argument to `PolynomialRing()` which specifies the number of variables desired.

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: p = -1/2*x - y*z - y + 8*z^2; p
-y*z + 8*z^2 - 1/2*x - y
```

Unlike with univariate polynomials, there is not a single way that we can order the terms of a polynomial. So to specify things like the *degree* and the *leading term* of a polynomial we must first fix a rule for deciding when one term is larger than another. If no argument is specified, SageMath defaults to the *graded reverse lexicographic* ordering, sometimes referred to as *grevlex*, to make these decisions. To read more about *Monomial Orderings*, see this [page](#) on Wikipedia.

To access a list of the monomials with nonzero coefficients in p , you use the `monomials()` method.

```
sage: p.monomials()
[y*z, z^2, x, y]
```

These monomials are listed in descending order using the term ordering specified when the ring was constructed.

To access a list of *coefficients* we use the `coefficients()` method.

```
sage: p.coefficients()
[-1, 8, -1/2, -1]
```

The list of coefficients is provided in the same order as the monomial listing computed earlier. This means that we can create a list of *terms* of our polynomial by `zip()`-ing up the two lists.

```
sage: [ a*b for a,b in zip(p.coefficients(),p.monomials()) ]
[-y*z, 8*z^2, -1/2*x, -y]
```

Often you want to compute information pertaining to the *largest*, or *leading*, term. We can compute the *lead coefficient*, *leading monomial*, and the *lead term* as follows:

```
sage: p.lc()
-1
sage:
sage: p.lm()
y*z
sage: p.lt()
-y*z
```

We can also compute the polynomial's *total degree* using the `total_degree()` method.

```
sage: p.total_degree()
2
```

The exponents of each variable in each term, once again specified in descending order, is computed using the `exponents()` method.

```
sage: p.exponents()
[(0, 1, 1), (0, 0, 2), (1, 0, 0), (0, 1, 0)]
```

and the exponent of the lead term is computed by chaining together two of the methods just presented.

```
sage: p.lm().exponents()
[(0, 1, 1)]
```

To change the term ordering we must reconstruct both the ring itself and all of the polynomials with which we were working. The following code constructs a multivariate polynomial ring in x, y , and z using the *lexicographic* monomial ordering.

```
sage: R.<x,y,z> = PolynomialRing(QQ,3,order='lex')
sage: p = -1/2*x - y*z - y + 8*z^2; p
-1/2*x - y*z - y + 8*z^2
```

Once the term order changes, all of the methods discussed earlier, even how SageMath displays the polynomial, take this into account.

```
sage: p.lm()
x
sage: p.lc()
-1/2
sage: p.lt()
-1/2*x
sage: p.monomials()
[x, y*z, y, z^2]
```

Note that the order in which the indeterminates are listed affects the monomial ordering. In the example above we have the lexicographic ordering, with $x > y > z$. We may redefine the ring to use the lexicographic order $z > y > x$.

```
sage: R.<z,y,x> = PolynomialRing(QQ,3,order='lex')
sage: p = -1/2*x - y*z - y + 8*z^2
sage: p
8*z^2 - z*y - y - 1/2*x
sage: p.lm()
z^2
sage: p.lc()
8
sage: p.lt()
8*z^2
```

Note again how all of the methods automatically take the new ordering into account.

Finally we can *reduce* a polynomial modulo a list of polynomials using the `mod()` method.

```
sage: r = -x^2 + 1/58*x*y - y + 1/2*z^2
sage: r.mod([p,q])
-238657765/29696*y^2 + 83193/14848*y*z^2 + 68345/14848*y - 1/1024*z^4 + 255/512*z^2 -
↪ 1/1024
```

Exercises:

1. Use SageMath to find out which of the following polynomials with rational coefficients are irreducible.

(a) $3y^4 - \frac{1}{2}y^2 - \frac{1}{2}y - \frac{1}{2}$

- (b) $2y^4 - y^2 - y$
- (c) $\frac{1}{5}y^5 - \frac{1}{3}y^4 + y^3 - \frac{17}{2}y^2 - 21y$
- (d) $y^3 + \frac{1}{4}y^2 - 6y + \frac{1}{8}$
- (e) $3y^7 + y^6 + \frac{9}{2}y^4 - y^3 + y^2 - \frac{1}{2}y$
2. Factor all of the polynomials over $\mathbb{Z}[x]$.
- (a) $-x^{10} + 4x^9 - x^8 + x^7 - x^6 + 2x^3 + x^2 - 1$
- (b) $x^5 + 2x^4 + x^3 + 3x^2 - 3$
- (c) $x^4 + x^3 - x^2 - x$
- (d) $2x^8 - 5x^7 - 3x^6 + 15x^5 - 3x^4 - 15x^3 + 7x^2 + 5x - 3$
- (e) $6x^6 - x^5 - 8x^4 - x^3 + 3x^2 + x$
3. Compute all of the *roots* and of the following polynomials defined over \mathbb{Z}_7 . Compare this list to their factorizations.
- (a) $2x^7 + 3x^6 + 6x^5 + 4x^4 + x^3 + 5x^2 + 2x + 5$
- (b) $3x^3 + x^2 + 2x + 1$
- (c) $3x^8 + 5x^7 + 5x^5 + x^3 + 2x^2 + 6x$
- (d) $x^5 + 2x^4 + x^3 + 2x^2 + 2x + 1$
- (e) $2x^{10} + 2x^8 + 5x^6 + x^5 + 3x^4 + 5x^3 + 2x^2 + 6x + 5$

4.4.2 Ideals and Quotients

In this section we will construct and do common computations with ideals and quotient rings.

Ideals

Once a ring is constructed and a list of generating elements have been selected, the ideal generated by this list is constructed by using the `*` operator.

```
sage: R.<x> = PolynomialRing(QQ)
sage: I = [2*x^2 + 8*x - 10, 10*x - 10]*R; I
Principal ideal (x - 1) of Univariate Polynomial Ring in x over Rational Field
sage: J = [x^2 + 1, x^3 + x]*R; J
Principal ideal (x^2 + 1) of Univariate Polynomial Ring in x over Rational Field
sage: K = [x^2 + 1, x - 2]*R; K
Principal ideal (1) of Univariate Polynomial Ring in x over Rational Field
```

SageMath automatically reduces the set of generators. This can be seen by using the `gens()` method which returns the list of the ideal's generating elements.

```
sage: I.gens()
(x - 1,)
sage: J.gens()
(x^2 + 1,)
sage: K.gens()
(1,)
```

Ideal membership can be determined by using the `in` conditional.

```
sage: R(x-1) in I
True
sage: R(x) in I
False
sage: R(2) in J
False
sage: R(2) in K
True
```

You can determine some properties of the ideal by using the corresponding `is_` methods. For example, to determine whether the ideals are *prime*, *principal*, or *idempotent* we enter the following:

```
sage: J.is_prime()
True
sage: K.is_prime()
False
sage: I.is_idempotent()
False
sage: K.is_principal()
True
```

Ideals in Multivariate Polynomial Rings

To construct an ideal within a multivariate polynomial ring, we must first construct the Polynomial ring with a term ordering and a collection of polynomials that will generate the ideal.

```
sage: R.<x,y,z> = PolynomialRing(QQ,3,order='lex')
sage: p = -1/2*x - y*z - y + 8*z^2
sage: q = -32*x + 2869*y - z^2 - 1
```

The ideal is constructed in the same manner as before.

```
sage: I = [p,q]*R
sage: I
Ideal (-1/2*x - y*z - y + 8*z^2, -32*x + 2869*y - z^2 - 1) of Multivariate Polynomial_
↪Ring in x, y, z over Rational Field
```

When the ring is a multivariate polynomial, we can compute a special list of generators for `I`, called a *groebner_basis*.

```
sage: I.groebner_basis()
[x - 2869/32*y + 1/32*z^2 + 1/32, y*z + 2933/64*y - 513/64*z^2 - 1/64]
```

There are different algorithms for computing a Groebner basis. We can change the algorithm by supplying an optional argument to the `groebner_basis()` command. The following commands compute a Groebner basis using the Buchberger algorithm while showing the intermediate results. Very useful for teaching!

```
sage: set_verbosity(3)
sage: I.groebner_basis('toy:buchberger')
(-32*x + 2869*y - z^2 - 1, -1/2*x - y*z - y + 8*z^2) => -2*y*z - 2933/32*y + 513/32*z^2
↪ + 1/32
G: set([-2*y*z - 2933/32*y + 513/32*z^2 + 1/32, -1/2*x - y*z - y + 8*z^2, -32*x +
↪ 2869*y - z^2 - 1])
(-1/2*x - y*z - y + 8*z^2, -32*x + 2869*y - z^2 - 1) => 0
G: set([-2*y*z - 2933/32*y + 513/32*z^2 + 1/32, -1/2*x - y*z - y + 8*z^2, -32*x +
↪ 2869*y - z^2 - 1])
(-1/2*x - y*z - y + 8*z^2, -2*y*z - 2933/32*y + 513/32*z^2 + 1/32) => 0
```

```
G: set([-2*y*z - 2933/32*y + 513/32*z^2 + 1/32, -1/2*x - y*z - y + 8*z^2, -32*x +
↪2869*y - z^2 - 1])
(-32*x + 2869*y - z^2 - 1, -2*y*z - 2933/32*y + 513/32*z^2 + 1/32) => 0
G: set([-2*y*z - 2933/32*y + 513/32*z^2 + 1/32, -1/2*x - y*z - y + 8*z^2, -32*x +
↪2869*y - z^2 - 1])
3 reductions to zero.
[x + 2*y*z + 2*y - 16*z^2, x - 2869/32*y + 1/32*z^2 + 1/32, y*z + 2933/64*y - 513/
↪64*z^2 - 1/64]
```

We can compute the various *elimination ideals* by using the `elimination_ideal()` method.

```
sage: I.elimination_ideal([x])
Ideal (64*y*z + 2933*y - 513*z^2 - 1) of Multivariate Polynomial Ring in x, y, z over
↪Rational Field
sage: I.elimination_ideal([x,y])
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Rational Field
sage: I.elimination_ideal([x,z])
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Rational Field
sage: I.elimination_ideal([x])
Ideal (64*y*z + 2933*y - 513*z^2 - 1) of Multivariate Polynomial Ring in x, y, z over
↪Rational Field
sage: I.elimination_ideal([y])
Ideal (64*x*z + 2933*x + 2*z^3 - 45902*z^2 + 2*z + 2) of Multivariate Polynomial Ring
↪in x, y, z over Rational Field
sage: I.elimination_ideal([z])
Ideal (263169*x^2 + 128*x*y^2 - 47095452*x*y + 16416*x - 11476*y^3 + 2106993608*y^2 -
↪1468864*y + 256) of Multivariate Polynomial Ring in x, y, z over Rational Field
sage: I.elimination_ideal([x,y])
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

4.4.3 Quotient Rings

To construct the *quotient ring* of a ring with an ideal we use the `quotient()` method.

```
sage: R = ZZ
sage: I = R*[5]
sage: I
Principal ideal (5) of Integer Ring
sage: Q = R.quotient(I)
sage: Q
Ring of integers modulo 5
```

To perform arithmetic in the quotient ring, we must first *coerce* elements into this universe. For more on why we do this see [Universes and Coercion](#).

```
sage: Q(10)
0
sage: Q(12)
2
sage: Q(10) + Q(12)
2
sage: Q(10 + 12)
2
```

When working with quotients of polynomial rings it is a good idea to give the indeterminate a new name.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: parent(x)
Univariate Polynomial Ring in x over Integer Ring
sage: I = R.ideal(x^2 + 1)
sage: Q.<a> = R.quotient(I)
sage: parent(a)
Univariate Quotient Polynomial Ring in a over Integer Ring with modulus x^2 + 1
sage: a^2
-1
sage: x^2
x^2
```

Then we can do arithmetic in this quotient ring without having to explicitly coerce all of our elements.

```
sage: 15*a^2 + 20*a + 1
20*a - 14
sage: (15 + a)*(14 - a)
-a + 211
```

4.4.4 Properties of Rings

You can check some of the properties of the rings which have been constructed. For example, to check whether a ring is an *integral domain* or a *field* we use the `is_integral_domain()` or `is_field()` methods.

```
sage: QQ.is_field()
True
sage: ZZ.is_integral_domain()
True
sage: ZZ.is_field()
False
sage: R=Integers(15)
sage: R.is_integral_domain()
False
sage: S=Integers(17)
sage: S.is_field()
True
```

These properties are often determined instantaneously since they are built into the definitions of the rings and not calculated on the fly.

For a complete listing of properties that are built into a ring, you can use SageMath's built in *tab-completion*. For example, to see all of the properties which can be determined for the rational numbers we type `QQ.is` then the tab key. What we get is a list of all of the properties that we can compute.

```
sage: QQ.is[TAB]
QQ.is_absolute           QQ.is_finite           QQ.is_ring
QQ.is_atomic_repr       QQ.is_integral_domain  QQ.is_subring
QQ.is_commutative       QQ.is_integrally_closed QQ.is_zero
QQ.is_exact              QQ.is_noetherian
QQ.is_field              QQ.is_prime_field
```

The *characteristic* of the ring can be computed using the ring's `characteristic()` method.

```
sage: QQ.characteristic()
0
sage: R=Integers(43)
```

```
sage: R.characteristic()
43
sage: F.<a> = FiniteField(9)
sage: F.characteristic()
3
sage: ZZ.characteristic()
0
```

4.4.5 Mini-Topic: Multivariate Polynomial Division Algorithm

In this section we will use SageMath to construct a *division* algorithm for multivariate polynomials. Specifically, for a given polynomial f (the dividend) and a sequence of polynomials f_1, f_2, \dots, f_k (the divisors) we want to compute a sequence of quotients a_1, a_2, \dots, a_k and a remainder polynomial r so that

$$f = \sum_{i=1}^{i=k} a_i \cdot f_i + r$$

where no terms of r are divisible by any of the leading terms of f_i .

The first thing that we will do is to construct the base field for the polynomial ring and determine how many variables we want for the polynomial ring. In this case, let's define a two variable polynomial ring over the finite field \mathbb{F}_2 .

```
sage: K = GF(2)
sage: n = 2
```

Next we will construct the polynomial ring.

```
sage: P.<x,y> = PolynomialRing(F,2,order="lex")
```

Since we are working with more than one variable we must tell SageMath how to order the terms, in this case we selected a *lexicographic* ordering. The default term ordering is *degree reverse lexicographic*, where the *total degree* is used first to determine the order of the monomials, then a *reverse lexicographic* order is used to break ties. Other options for monomial orderings are *deglex* (degree lexicographic) or you can define a *block* ordering by using the `TermOrder()` command. You can read more on monomial orderings on-line on [Wikipedia](#) and on [MathWorld](#), or the book [\[Cox2007\]](#).

Now we will begin our division algorithm. The first thing we will do is define a function which determines whether two monomial *divide* each other.

```
def does_divide(m1,m2):
    for c in (vector(ZZ, m1.degrees()) - vector(ZZ,m2.degrees())):
        if c < 0:
            return False
    return True
```

Then we will define a sequence of polynomials which we will use to reduce our *dividend*.

```
sage: F = [x^2 + x, y^2 + y]
```

Next we will define the polynomial which will be reduced.

```
sage: f = x^3* y^2
```

Now we will define the list of quotients and the remainder and initialize them to 0.


```
sage: A = [P(0) for i in range(0, len(F)) ]
sage: r = P(0)
```

Now because we alter f through the algorithm we will create a copy of it so that we can keep the value of f for later to verify the algorithm.

```
sage: p = f
```

Now we are ready to define the main loop of our algorithm.

```
while p != P(0):
    i = 0
    div_occurred = False
    while (i < len(F) and div_occurred == False):
        print A, p, r
        if does_divide(p.lm(), F[i]):
            q = P(p.lm()/F[i].lm())
            A[i] = A[i] + q
            p = p - q*F[i]
            div_occurred = True
        else:
            i = i + 1
    if div_occurred == False:
        r = r + p.lm()
        p = p - p.lm()

print A, p, r
```

4.5 Fields

4.5.1 Number Fields

We create a number field by specifying an irreducible polynomial and a name for the root of that polynomial. We may use the indeterminate x , which is already defined in sage. We can also create a polynomial ring over the rationals and use the indeterminate for that polynomial ring.

```
sage: P.<t> = PolynomialRing(QQ)
sage: K.<a> = NumberField(t^3-2)
sage: K
Number Field in a with defining polynomial t^3 - 2
sage: K.polynomial()
t^3 - 2
```

A “random element” may be constructed producing an element with degree at most 2 (one less than the degree of the defining polynomial). The options `num_bound()` or `den_bound()` may be used to bound the numerator or denominator.

```
sage: K.random_element()
-5/14*a^2 + a - 3
sage: K.random_element()
-2*a
sage: K.random_element(num_bound= 2)
-a^2 + 1
```

Every irrational element will have a minimal polynomial of degree 3.

```
sage: a.minpoly()
x^3 - 2
sage: (a^2-3*a).minpoly()
x^3 + 18*x + 50
```

We can test isomorphism of fields.

```
sage: K.<a>= NumberField(t^3-2)
sage: L.<b> = NumberField(t^3-6*t-6)
sage: K.is_isomorphic(L)
True
```

The number of real embeddings and the number of pairs of complex embeddings are given by the signature of the field. The embeddings into the real field, $\mathbb{R}()$, or complex field $\mathbb{C}()$ may also be constructed.

```
sage: K.signature()
(1, 1)
sage: K.real_embeddings()
[
Ring morphism:
  From: Number Field in a with defining polynomial t^3 - 2
  To:   Real Field with 53 bits of precision
  Defn: a |--> 1.25992104989487
]
sage: K.complex_embeddings()
[
Ring morphism:
  From: Number Field in a with defining polynomial t^3 - 2
  To:   Complex Field with 53 bits of precision
  Defn: a |--> -0.629960524947437 - 1.09112363597172*I,
Ring morphism:
  From: Number Field in a with defining polynomial t^3 - 2
  To:   Complex Field with 53 bits of precision
  Defn: a |--> -0.629960524947437 + 1.09112363597172*I,
Ring morphism:
  From: Number Field in a with defining polynomial t^3 - 2
  To:   Complex Field with 53 bits of precision
  Defn: a |--> 1.25992104989487
]
sage: phi1, phi2, phi3 = K.complex_embeddings()
sage: phi1(a)
-0.629960524947437 - 1.09112363597172*I
sage: phi2(a)
-0.629960524947437 + 1.09112363597172*I
sage: phi3(a^2+3*a+5)
10.3671642016528
```

The `Galois_group()` method computes the Galois group of the Galois closure, not of the field itself. When the Galois group is not cyclic, as in the second example, you need to name one of the generators. The generators may also be accessed as shown below.

```
sage: G = L.galois_group()
sage: G.gens()
[(1, 2, 3)]
sage: H.<g>= K.galois_group()
sage: H.gens()
```

```
[ (1,2) (3,4) (5,6), (1,4,6) (2,5,3) ]
sage: H.0
(1,2) (3,4) (5,6)
sage: H.1
(1,4,6) (2,5,3)
```

The Galois closure of K .

```
sage: L.<b> = K.galois_closure()
sage: L
Number Field in b with defining polynomial t^6 + 40*t^3 + 1372
```

Field Extensions

Now let's construct field extensions, which may be done in a few different ways. The methods `absolute_()` refer to the prime field \mathbb{Q} , while the methods `relative_()` refer to a field extension as constructed, which may be relative to some intermediate field.

```
sage: P.<t> = PolynomialRing(QQ)
sage: K.<a> = NumberField(t^3-2)
sage: L.<b> = NumberField(t^3-a)
sage: L.relative_degree(); L.relative_polynomial()
3
t^3 - a
sage: L.base_field()
Number Field in a with defining polynomial t^3 - 2
sage: L.absolute_degree(); L.absolute_polynomial()
9
x^9 - 2
sage: L.gens()
(b, a)
```

We may also create the compositum of several fields defined by a list of polynomials over the rationals. We must specify a root for each polynomial. SageMath creates a sequence of 3 fields in the following example, starting at the far right in the list.

```
sage: M.<a,b,c> = NumberField([t^3-2, t^2-3, t^3-5])
sage: M
Number Field in a with defining polynomial t^3 - 2 over its base field
sage: M.relative_degree()
3
sage: M.absolute_degree()
18
sage: d = M.absolute_generator(); d
a - b + c
sage: d.minpoly()
x^3 + (3*b - 3*c)*x^2 + (-6*c*b + 3*c^2 + 9)*x + (3*c^2 + 3)*b - 9*c - 7
sage: d.absolute_minpoly()
x^18 - 27*x^16 - 42*x^15 + 324*x^14 + 378*x^13 - 2073*x^12 + 1134*x^11 - 6588*x^10 -
↪ 23870*x^9 + 88695*x^8 + 79002*x^7 - 147369*x^6 - 1454922*x^5 + 431190*x^4 +
↪ 164892*x^3 + 2486700*x^2 - 1271592*x + 579268
```

The next example computes the Galois closure of $K()$ and asks for the roots of unity. The generator for $L()$ is something that sage computes, so it may have a complicated minimum polynomial, as we see. We know that $L()$ contains cube roots of unity, so let's verify it.

```

sage: K.<a> = NumberField(t^3-2)
sage: L.<b> = K.galois_closure()
sage: b.minpoly()
x^6 + 40*x^3 + 1372
sage: units= L.roots_of_unity(); units
[1/36*b^3 + 19/18, 1/36*b^3 + 1/18, -1, -1/36*b^3 - 19/18, -1/36*b^3 - 1/18, 1]
sage: len(units)
6
sage: [u^3 for u in units]
[-1, 1, -1, 1, -1, 1]

```

Special Number Fields

There are two classes of number fields with special properties that you can construct directly. For a *quadratic field extension* simply specify a square free integer.

```

sage: F.<a> = QuadraticField(17)
sage: a^2
17
sage: (7*a-3).minpoly()
x^2 + 6*x - 824

```

A *cyclotomic field* is created by indentifying its primitive root of unity.

CyclotomicField()

QuadraticField()

4.5.2 Finite Fields

In a prior section we constructed rings of integers modulo n . We know that when n is a prime number the *ring* \mathbb{Z}_n is actually a *field*. SageMath will allow us to construct this same object as either a ring or a field.

```

sage: R = Integers(7)
sage: F7 = GF(7)
sage: R, F7
(Ring of integers modulo 7, Finite Field of size 7)

```

To take advantage of the extra structure it is best to use the command `GF()` (or equivalently, `FiniteField()`) to construct this object. As with modular rings we have to coerce integers into the field in order to do arithmetic in the field.

```

sage: F7(4 + 3)
0
sage: F7(2*3)
6
sage: F7(3*7)
0
sage: F7(3/2)
5

```

We can use SageMath to construct any *finite field*. Recall that a finite field is always of order $n = p^k$ where p is a prime number. To construct the field of order $25 = 5^2$ we input the following command.

```
sage: F25.<a> = GF(25)
```

Recall that the finite field of order 5^2 can be thought of as an *extension* of \mathbb{Z}_5 using a root of a polynomial of degree 2. The `a` that you specified is a root of this polynomial. There are different polynomials that can be used to construct this extension and SageMath chooses one for you. You can see the polynomial chosen by using the, aptly named, `polynomial()` method.

```
sage: p = F25.polynomial();
sage: p
a^2 + 4*a + 2
```

We can verify that `a` satisfies this polynomial.

```
sage: a^2 + 4*a + 2
0
```

It should be noted that `a` already lives in the field and no special coercion is necessary to do arithmetic using `a`.

```
sage: parent(a)
Finite Field in a of size 5^2
sage: a^2
a + 3
sage: a*(a^2 + 1)
3
```

But if we are using only integers we must coerce the arithmetic into the field.

```
sage: 3+4
7
sage: parent(3+4)
Integer Ring
sage: F25(3 + 4)
2
sage: parent(F25(3+4))
Finite Field in a of size 5^2
```

Sometimes we would like to specify the polynomial used to construct our extension. To do so we just need to add the *modulus* option to our field constructor.

```
sage: F25.<a> = GF(25, modulus=x^2 + x + 1)
sage: a^2 + a + 1
0
sage: a^2
4*a + 4
```

Remember that the modulus must be a polynomial which is *irreducible* over $\mathbb{F}_5[x]$. Many times we would like for the modulus to not just be irreducible, but to be *primitive*. Next we will construct all of the primitive polynomials of degree 2. The following example uses *Polynomial Rings* and *List Comprehensions (Loops in Lists)*. First thing that we will do is construct a list of all monic polynomials over $\text{GF}(5)$

```
sage: F5 = GF(5)
sage: P.<x> = PolynomialRing(F, 'x')
sage: AP = [ a0 + a1*x + a2*x^2 for (a0,a1) in F^3]
sage: AP
[x^2, x^2 + 1, x^2 + 2, x^2 + 3, x^2 + 4, x^2 + x, x^2 + x + 1, x^2 + x + 2, x^2 + x
↪ + 3, x^2 + x + 4, x^2 + 2*x, x^2 + 2*x + 1, x^2 + 2*x + 2, x^2 + 2*x + 3, x^2 + 2*x
↪ + 4, x^2 + 3*x, x^2 + 3*x + 1, x^2 + 3*x + 2, x^2 + 3*x + 3, x^2 + 3*x + 4, x^2 +
↪ 4*x, x^2 + 4*x + 1, x^2 + 4*x + 2, x^2 + 4*x + 3, x^2 + 4*x + 4]
```

Next we will *filter* out the primitive polynomials out of this list.

```
sage: PR = [ p for p in AP if p.is_primitive() ]
sage: PR
[x^2 + x + 2, x^2 + 2*x + 3, x^2 + 3*x + 3, x^2 + 4*x + 2]
```

If we wanted all of the *irreducible* polynomials we would only change the last command slightly.

```
sage: IR = [ p for p in AP if p.is_irreducible() ]
sage: IR
[x^2 + 2, x^2 + 3, x^2 + x + 1, x^2 + x + 2, x^2 + 2*x + 3, x^2 + 2*x + 4, x^2 + 3*x +
↪ 3, x^2 + 3*x + 4, x^2 + 4*x + 1, x^2 + 4*x + 2]
```

It should be noted that the above code will only work if the polynomials are over *finite* rings or fields.

Exercises:

1. Compute the list of all *primitive polynomials* of degree 3 over $GF(5)$.
2. Compute the number of *primitive elements* in $GF(125)$.
3. Explain the relationship between the number of primitive polynomials and the number of primitive elements in the previous exercises.

4.5.3 Function Fields

4.6 Coding Theory

4.6.1 Linear Codes

A *linear code* is just a finite-dimensional vector space commonly defined over a finite field. To construct a linear code in SageMath we first define a finite field and a matrix over this field whose range will define this vector space.

```
sage: F = GF(2)
sage: G = matrix(F, [(0,1,0,1,0), (0,1,1,1,0), (0,0,1,0,1), (0,1,0,0,1)]); G
[0 1 0 1 0]
[0 1 1 1 0]
[0 0 1 0 1]
[0 1 0 0 1]
```

The code itself is constructed by the `LinearCode()` command.

```
sage: C = LinearCode(G); C
Linear code of length 5, dimension 4 over Finite Field of size 2
```

While the *length* and *dimension* of the code are displayed in the object's *description*, you can also obtain these properties at anytime using the code's `length()` and `dimension()` methods.

```
sage: C.length()
5
sage: C.dimension()
4
```

Given two code words, we can compute their *Hamming Weight* and *Distance* both by using the `hamming_weight()` function.

```
sage: w1 = vector(F, (0,1,0,1,0)); w1
(0, 1, 0, 1, 0)
sage: hamming_weight(w1)
2
sage: w2 = vector(F, (0,1,1,0,1)); w2
(0, 1, 1, 0, 1)
sage: hamming_weight(w2)
3
sage: hamming_weight(w1 - w2)
3
```

The *minimum distance* of C can be computed by using the `minimum_distance()` method.

```
sage: C.minimum_distance()
1
```

SageMath can also compute the *distribution* of weights for the code.

```
sage: C.weight_distribution()
[1, 4, 6, 4, 1, 0]
```

Where the value listed at index i of the list, starting with zero and ending with the length of the code, is the number of codewords with that weight.

Related to the weight distribution is the *weight enumerator* polynomial, which you compute using the code's `weight_enumerator()` method.

```
sage: C.weight_enumerator()
x^5 + 4*x^4*y + 6*x^3*y^2 + 4*x^2*y^3 + x*y^4
```

The *generating* and *check* matrices are computed using the `gen_mat()` and `check_mat()` methods.

```
sage: C.gen_mat()
[0 1 0 1 0]
[0 1 1 1 0]
[0 0 1 0 1]
[0 1 0 0 1]
sage: C.check_mat()
[1 0 0 0 0]
```

The *systematic* form of the generating matrix is computed using `gen_mat_systematic()`.

```
sage: C.gen_mat_systematic()
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

SageMath can both *extend* and *puncture* our code. The *extended code* is computed as follows:

```
sage: Cx = C.extended_code(); Cx
Linear code of length 6, dimension 4 over Finite Field of size 2
sage: Cx.gen_mat()
[0 1 0 1 0 0]
[0 1 1 1 0 1]
[0 0 1 0 1 0]
[0 1 0 0 1 0]
sage: Cx.check_mat()
```

```
[1 0 0 0 0 0]
[0 1 1 1 1 1]
```

The *punctured* code is computed by supplying the code's `punctured()` method a list of coordinates in which to delete. The following commands construct the code that results when the 1st and 3rd coordinate from every code word in `C` are deleted. Note that unlike vectors, lists and matrices the 1st column is indexed by 1 and not 0 when puncturing a code.

```
sage: Cp = C.punctured([1,3]); Cp
Linear code of length 3, dimension 2 over Finite Field of size 2
sage: Cp.gen_mat()
[0 1 0]
[0 0 1]
sage: Cp.check_mat()
[1 0 0]
```

SageMath can also compute the *dual* of `C`.

```
sage: Cd = C.dual_code(); Cd
Linear code of length 5, dimension 1 over Finite Field of size 2
sage: Cd.gen_mat()
[1 0 0 0 0]
sage: Cd.check_mat()
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

And finally SageMath can *decode* a received vector. The following simulates a communications channel; We begin with a code word, introduce an error and then correct this error by *decoding* the received message.

```
sage: wrd = vector(F, (0,0,0,0,1))
sage: err = vector(F, (0,0,1,0,0))
sage: msg = wrd + err; msg
(0, 0, 1, 0, 1)
sage: C.decode(msg)
(0, 0, 0, 0, 1)
sage: C.decode(msg) == wrd
True
```

It should be noted that since the above code has a minimum distance of only 1 that decoding will not always produce the code word that you may have expected.

These are only some of the commands that SageMath offers for computing and working with linear codes. There is much more information on the following web sites:

See also:

1. http://www.sagemath.org/doc/constructions/linear_codes.html
2. http://www.sagemath.org/doc/reference/sage/coding/linear_code.html

4.6.2 Cyclic Codes

To construct a cyclic code of length 3 over \mathbb{F}_2 we first need a *generating polynomial*, which can be any *irreducible* factor of $x^3 - 1$. A list of irreducible factors is computed using the `factor()` command.


```
sage: P.<x> = PolynomialRing(GF(2), 'x')
sage: factor(x^3 - 1)
(x + 1) * (x^2 + x + 1)
```

The output above tells you that there are 2 choices for non-trivial generating polynomials. The following commands will construct the code generated by $g(x) = x + 1$.

```
sage: g = x + 1
sage: C = CyclicCode(3, g)
sage: C.list()
[(0, 0, 0), (1, 0, 1), (0, 1, 1), (1, 1, 0)]
```

Cyclic codes are a special type of linear code. So the commands that you worked with in the prior section all work in the same way. For example, the generating matrix is computed, in the usual and systematic forms, as follows:

```
sage: G = C.gen_mat(); G
[1 1 0]
[0 1 1]
sage: Gs = C.gen_mat_systematic(); Gs
[1 0 1]
[0 1 1]
```

Just to verify that this is the generating matrix, and to practice working with matrices and vectors, we will see if the image of G spans the code.

```
sage: vector(GF(2), [0, 0]) * G
(0, 0, 0)
sage: vector(GF(2), [1, 0]) * G
(1, 1, 0)
sage: vector(GF(2), [1, 1]) * G
(1, 0, 1)
sage: vector(GF(2), [0, 1]) * G
(0, 1, 1)
```

SageMath can also compute a *parity check* matrix of C using the code's `check_mat()` method.

```
sage: H = C.check_mat()
[1 1 1]
```

Verifying that H is a *check matrix* for C is straightforward.

```
sage: H*vector(GF(2), [0, 1, 1])
(0)
sage: H*vector(GF(2), [1, 0, 1])
(0)
sage: H*vector(GF(2), [1, 0, 0])
(1)
```

You can also compute the *dual code* and its generating and parity check matrices.

```
sage: Cp = C.dual_code()
sage: Cp.gen_mat()
[1 1 1]
sage: Cp.check_mat()
[1 0 1]
[0 1 1]
```

4.6.3 Mini-Topic: Factoring $x^n - 1$

The smallest field containing \mathbb{F}_q and containing the roots of $x^n - 1$ is $GF(q^t)$ where t is the order of q in $\mathbb{Z} \bmod n$.

The factors of $x^n - 1$ over \mathbb{F}_q must all have degree dividing t .

Let us begin by first defining n and q and constructing the ambient rings.

```
sage: n = 19
sage: q = 2
sage: F = GF(2)
sage: P.<x> = PolynomialRing(F, 'x')
```

Remembering that since we are constructing a finite field that q has to either be prime or a prime power. Now let us compute all of the irreducible factors of $x^n - 1$ over \mathbb{F}_q .

```
sage: A = factor(x^n-1); A
```

Now to verify the facts about the degrees of the factors computed that was stated earlier. Compare the list above with the order of 2 in \mathbb{Z}_n .

```
sage: Integers(19)(2).multiplicative_order()
```

Remembering that since \mathbb{Z}_n is a ring, we have to specify which type of *order* we want to compute, either *additive* or *multiplicative*.

Now let us repeat what we just did, but this time letting $q = 2^2$. Changing q alone will not change the base field nor the polynomial ring. So we will have to re-construct everything using our new parameter.

```
sage: q = 4
sage: F.<a> = GF(4, 'a')
sage: P.<x> = PolynomialRing(F, 'x')
```

Now let us factor $x^n - 1$ again. This time over a non-prime field.

```
sage: A = factor(x^n-1); A
(x + 1) * (x^9 + a*x^8 + a*x^6 + a*x^5 + (a + 1)*x^4 + (a + 1)*x^3 + (a + 1)*x + 1) *
↪ (x^9 + (a + 1)*x^8 + (a + 1)*x^6 + (a + 1)*x^5 + a*x^4 + a*x^3 + a*x + 1)
```

Exercises:

1. Try repeating the above for $F = \mathbb{F}_8$.
2. Compute the order of 2, 4, 8 mod 19. What are your observations?
3. Try other values of n and other fields.

4.6.4 Mini-Topic: Idempotent Polynomials

We'll find the idempotent which is 1 modulo the i th factor of $x^n - 1$. Continuing with \mathbb{F}_4 .

```
sage: F.<a> = GF(4, 'a')
sage: P.<x> = PolynomialRing(F, 'x')
```

Then we will create the quotient ring.

```
sage: R.<y> = P.quotient(x^19 - 1)
sage: A = factor(x^19 - 1); A
(x + 1) * (x^9 + a*x^8 + a*x^6 + a*x^5 + (a + 1)*x^4 + (a + 1)*x^3 + (a + 1)*x + 1) *
↪ (x^9 + (a + 1)*x^8 + (a + 1)*x^6 + (a + 1)*x^5 + a*x^4 + a*x^3 + a*x + 1)
```

Since the `factor()` command returns a list of polynomial factors and their multiplicities, which we do not need, we will strip those out.

```
sage: A = [p[0] for p in A]
```

Now we will just select one of these factors. The reader should also try different factors for themselves.

```
sage: p0 = A[2]
```

Now we take the product of all of the other factors.

```
sage: ap = prod( [p for p in A if p != a])
x^10 + (a + 1)*x^9 + a*x^8 + a*x^7 + x^5 + (a + 1)*x^3 + (a + 1)*x^2 + a*x + 1
```

Then compute the `xgcd()` of p_0 and ap .

```
sage: d, s, t = xgcd(p0, ap)
```

You should recall that $d = s \cdot p_0 + t \cdot ap$ is the extended gcd. You should check that $s \cdot p_0 \equiv 1 \pmod p$ for all $p \neq p_0$ and $s \cdot p_0 \equiv 0 \pmod{p_0}$

```
sage: s*p0 % A[1]
1
sage: s*p0 % A[2]
0
sage: s*p0 % A[0]
1
```

Now check that $t \cdot ap \equiv 0 \pmod p$ for $p \neq p_0$ and $t \cdot ap \equiv 1 \pmod{p_0}$.

```
sage: t*ap % A[0]
0
sage: t*ap % A[1]
0
sage: t*ap % A[2]
1
```

Now we will check that the polynomial that we computed is an idempotent in $F[x] / \langle x^n - 1 \rangle$.

```
sage: f = R(bp*ap)
sage: f^2 == f
True
```

Check the generating polynomial.

```
sage: gcd(b*p0, x^19-1)
x^9 + (a + 1)*x^8 + (a + 1)*x^6 + (a + 1)*x^5 + a*x^4 + a*x^3 + a*x + 1
sage: p0
x^9 + (a + 1)*x^8 + (a + 1)*x^6 + (a + 1)*x^5 + a*x^4 + a*x^3 + a*x + 1
```

Exercises:

1. Find the idempotent element of $F[x] / \langle x^n - 1 \rangle$ For $q = 4$ and $n = 3, 5, 11$ and 17 .

For the reciprocal polynomials of idempotents, see Theorem 5 [MacWilliams1977] p. 219

4.6.5 Other Codes

Hamming Codes

A Hamming Code is a simple linear code which has the capability to detect up to 2 contiguous errors and correct for any single error.

We will begin by constructing a binary Hamming code with 3 parity checks.

```
sage: F = GF(2)
sage: C = HammingCode(3,F); C
Linear code of length 7, dimension 4 over Finite Field of size 2
```

Hamming codes always have a length, $|\mathbb{F}|^r - 1$ where r is the number of parity checks and \mathbb{F} is the finite-field over which the code is defined. This is because the columns of its *parity check* matrix consists of all non-zero elements of \mathbb{F}^r .

```
sage: C.check_mat()
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
```

A Ternary Hamming Code is constructed by supplying a non-binary finite field as the base field. Here we will construct the ternary Hamming code over $GF(2^3)$ also with 3 parity checks.

```
sage: C = HammingCode(3, F); C
Linear code of length 73, dimension 70 over Finite Field in a of size 2^3
```

See also:

http://en.wikipedia.org/wiki/Hamming_code

BCH Codes

BCH codes, or Bose-Chaudhuri-Hocquenghem codes, are a special class of the cyclic codes with 3 required parameters, n, δ, F and one optional one b . Here n is the length of the code words, δ is a parameter called the *designed distance* and F is a finite field of order q^n where $\gcd(n, q) = 1$.

If b is not provided then a default value of zero is used. For example, you construct construct a BCH code of length $n = 13$ with $\delta = 5$ over $F = GF(9)$.

```
sage: F.<a> = GF(3^2, 'a')
sage: C = BCHCode(13, 5, F)
sage: C
Linear code of length 13, dimension 6 over Finite Field in a of size 3^2
```

We can compute the code's minimum distance using its `minimum_distance()` method.

```
sage: C.minimum_distance()
6
```

Since BCH codes are also linear, you can use SageMath to compute the code's generating and check matrices.

```
sage: C.gen_mat()
[2 2 1 2 0 0 1 1 0 0 0 0 0]
[0 2 2 1 2 0 0 1 1 0 0 0 0]
[0 0 2 2 1 2 0 0 1 1 0 0 0]
```

```

[0 0 0 2 2 1 2 0 0 1 1 0 0]
[0 0 0 0 2 2 1 2 0 0 1 1 0]
[0 0 0 0 0 2 2 1 2 0 0 1 1]
sage: C.check_mat()
[1 0 0 0 0 0 0 1 2 1 2 2 2]
[0 1 0 0 0 0 0 1 0 0 0 1 1]
[0 0 1 0 0 0 0 2 2 2 1 1 2]
[0 0 0 1 0 0 0 1 1 0 1 0 0]
[0 0 0 0 1 0 0 0 1 1 0 1 0]
[0 0 0 0 0 1 0 0 0 1 1 0 1]
[0 0 0 0 0 0 1 2 1 2 2 2 1]

```

We can also compute its *dual* code.

```

sage: Cp = C.dual_code(); Cp
Linear code of length 13, dimension 7 over Finite Field in a of size 3^2
sage: Cp.gen_mat()
[1 0 0 0 0 0 0 1 2 1 2 2 2]
[0 1 0 0 0 0 0 1 0 0 0 1 1]
[0 0 1 0 0 0 0 2 2 2 1 1 2]
[0 0 0 1 0 0 0 1 1 0 1 0 0]
[0 0 0 0 1 0 0 0 1 1 0 1 0]
[0 0 0 0 0 1 0 0 0 1 1 0 1]
[0 0 0 0 0 0 1 2 1 2 2 2 1]
sage: Cp.check_mat()
[1 0 0 0 0 0 2 2 1 2 0 0 1]
[0 1 0 0 0 0 1 0 1 2 2 0 2]
[0 0 1 0 0 0 2 0 1 0 2 2 1]
[0 0 0 1 0 0 1 0 2 2 0 2 1]
[0 0 0 0 1 0 1 2 2 0 2 0 1]
[0 0 0 0 0 1 1 2 1 0 0 2 2]

```

See also:

http://en.wikipedia.org/wiki/BCH_code

BIBLIOGRAPHY

[Cox2007] Cox, David and Little, John and O'Shea, Donald, *Ideals, varieties, and algorithms*. Springer 2007

[MacWilliams1977] MacWilliams, F. J. and Sloane, N. J. A., *The theory of error-correcting codes*. North-Holland Publishing Co. 1977

Symbols

=, 43

==, 19, 42

?, 8

>=, 19, 43

<, 43

<=, 19, 43

<>, 43

A

About SageMath, 3

About this tutorial, 1

abs, 16

absolute value, 16

add_multiple_of_column, 93

add_multiple_of_row, 93

addition, 13

additive_order, 73

AlternatingGroup, 80, 81

and, 42

Anti-derivative, 24

append, 49

arctan, 18

arithmetic, 13

matrix, 89

aspect_ratio, 31

augment, 94, 97

axes_labels, 28

B

Basic Arithmetic, 13

Basic Arithmetic and Functions, 13

BCH Codes, 120

BCHCode, 120

bitbucket, 11

bool, 42

Booleans, 42

C

Calculus Commands, 21

cardinality, 52

Cartesian product, 87

cayley_table, 80

CC, 39

ceil, 16

ceiling, 16

center, 82

characteristic, 107

characteristic_polynomial, 97

check_mat, 115

Chinese Remainder Theorem, 76

cocalc, 4

Coding Theory, 114

check matrix, 115

generating matrix, 115

hamming distance, 114

hamming weight, 114

minimum distance, 115

coefficients, 102

Coercion, 40

explicit, 41

implicit, 40

column, 92

columns, 92

command line, 4

concatenation of strings, 53

conditionals, 55

Contributing to the tutorial, 10

contribution, 11

cos, 17

count, 48

countour_plot, 33

crt, 76

Cyclic Codes, 116

CyclicCode, 117

CyclicPermutationGroup, 81, 82

D

decimal approximation, 17

Declaring Variables, 20

decode, 116

def, 59

defining mathematical functions, 22

degree, 100

del, 46
 derivative, 23
 Derivatives, 23
 partial, 23
 det, 90
 diagonal, 92
 DiCyclicGroup, 81
 dimension, 114
 direct product, 87
 divides, 14
 division, 13, 14
 Division and factoring, 14
 divisors, 14, 15
 divmod, 14, 100
 dual_code, 116

E

e, 18
 echelon_form, 94
 echelonize, 94
 eigenvalues, 97
 eigenvectors_right, 97
 elif, 55
 elimination_ideal, 106
 else, 56
 equations, 19
 euclidean algorithm, 77
 exp, 18
 exponential function, 18
 exponentiation, 13
 exponents, 103
 extend, 49
 extended_code, 115
 external files, 60
 external programs, 61
 gap, 61
 singular, 64

F

factor, 15, 97, 101, 118
 factoring, 15
 False, 19, 42
 Field Extensions
 generating polynomial, 113
 modulus, 113
 find_root, 20
 Finding critical points, 24
 floor, 16
 for, 56
 for statement, 56
 fractional powers, 17
 functions, 59
 arguments, 59
 definition, 59

multiple arguments, 59
 return values, 59

G

gcd, 15, 77, 100, 119
 gen_mat, 115
 gen_mat_systematic, 115
 gens, 80, 104
 gens_small, 80
 Getting Started, 3
 GF, 108, 112
 Graphics, 27
 adding two plots, 29
 aspect_ratio, 31
 color, 28
 greatest common divisor, 15
 groebner_basis, 105
 groups, 78
 alternating, 80
 Cayley table, 80
 cyclic, 82
 DihedralGroup, 81
 dihedral, 81
 generators, 80
 homomorphisms, 86
 kernel of homomorphism, 87
 Klein 4, 82
 order, 79
 permutation, 81
 subgroup, 80

H

Hamming Codes, 120
 hamming_weight, 114
 HammingCode, 120
 help, 8
 command line, 8
 DESCRIPTION, 9
 EXAMPLES, 9
 INPUT, 9
 How to use this tutorial, 3
 hyperbolic trigonometric functions, 18

I

ideal, 106
 Ideals, 104
 construction, 105
 gens, 104
 membership, 104
 Idempotent Polynomials, 118
 identity_matrix, 89, 97
 if statement, 55
 if-else statement, 55
 in, 39, 52

Indeterminants, 106
 index, 48
 inequalities, 19
 Integers, 73, 99
 integers, 77
 integral, 24
 Integrals, 24

- definite, 24
- indefinite, 24

 interact, 66
 interactive applets, 66
 intersection, 52
 inverse, 74, 98
 inverse trigonometric functions, 18
 is_abelian, 80
 is_cyclic, 80
 is_field, 74, 107
 is_idempotent, 105
 is_integral_domain, 74, 107
 is_invertible, 91
 is_irreducible, 101, 114
 is_prime, 15, 105
 is_primitive, 114
 is_principal, 105
 is_ring, 74
 is_subgroup, 80
 is_unit, 73, 75

J

joining strings, 54
 Jordan Canonical Form, 96
 jordan_form, 98

K

kernel, 87

L

lc, 102
 lcm, 15
 least common multiple, 15
 left_kernel, 96
 len, 46, 53
 length, 114
 lexicographic monomial ordering, 103
 limit, 22
 Limits, 22

- directional, 23

 Line tangent to a curve, 24
 linear algebra, 87
 Linear Codes, 114

- dimension, 114
- length, 114

 linear congruences, 76
 LinearCode, 114

linestyle, 28
 list, 46, 74

- append, 49
- comprehensions, 57
- concatenation, 50
- count, 48
- definition, 46
- extend, 49
- index, 48
- length, 46
- remove, 50
- slice, 47
- sort, 49
- zip, 50

lists

map, 51

lm, 102

ln, 18

load, 60

load_session, 61

loading a file, 60

log, 18

logarithms, 18

loops, 56

for, 56

while, 56

lt, 102

M

map, 51, 54

mathematical structures, 73

matrix, 88, 92

column, 92

determinant, 90

diagonal, 92

invertibility, 91

manipulation, 92

row, 92

rows, 92

trace, 91

matrix arithmetic, 89

matrix_from_columns, 92

matrix_from_rows, 92

matrix_from_rows_and_columns, 92

MatrixSpace, 95

max, 16

maximum, 16

mean, 27

median, 27

min, 16

Mini-Topic

Idempotent Polynomials, 118

minimum, 16

minimum_distance, 115

- mod, 103
- mode, 27
- modular arithmetic, 73
- modulus, 14
- Monomial Orderings, 102
- monomials, 102
- moving_average, 27
- multiplication, 13
- multiplicative_order, 73, 118
- Multivariate Polynomial Division Algorithm, 108
- Multivariate Polynomial Rings, 102
 - monomials, 102

N

- n, 17
- natural base, 18
- not, 42
- notebook, 4
- nth root, 17
- number fields, 109

O

- Objects, 39
- or, 42
- order, 73, 74, 79
 - additive, 73
 - multiplicative, 73
- Order of operations, 13

P

- parametric_plot, 31
- parent, 40, 98, 99, 106
- PermutationGroup, 81
- PermutationGroupMorphism, 86
- plot, 27
 - parametric, 31
- plot3d, 35
- Plotting, 27
 - 2D, 27
 - 3D, 35
 - aspect_ratio, 31
 - contour, 33
 - polar, 32
- polar_plot, 32
- polynomial, 113
- Polynomial Arithmetic, 100
- Polynomial Rings, 98
 - gcd, 100
 - xgcd, 101
- PolynomialRing, 98–100
- prime, 15
- prime_divisors, 15
- prod, 50, 119
- Programming in SageMath, 37

- Properties of Rings, 107
- punctured, 116

Q

- QQ, 39
- quotient, 106, 118
- Quotient Rings, 106
- quotient, 14

R

- random, 26
- rank, 96
- Reduction modulo an ideal, 103
- remainder, 14
- remove, 50
- rescale_col, 92
- rescale_row, 92
- reset, 45
- restore, 20, 45
- reStructured Text, 11
- return, 59
- right_kernel, 96
- Rings, 98
 - characteristic, 107
- rings, 73
 - integers modulo n, 73
 - list, 74
 - order, 73
 - size, 74
 - unit group, 75
 - units, 73
- roots, 101
- row, 92
- row_space, 96
- rows, 92
- RR, 39

S

- SageMath as a Calculator, 11
- save_session, 61
- sessions, 60
- Set, 52
 - cardinality, 52
 - difference, 52
 - intersection, 52
 - subsets, 52
 - symmetric difference, 52
 - union, 52
- set difference, 52
- set_block, 94
- set_column, 93
- set_row, 93
- show, 27
- sign, 80

sin, 17
 sinh, 18
 slices, 47
 solve, 19
 solve_mod, 76
 solve_right, 95
 solving equations, 19
 Solving equations and inequalities, 19
 Solving equations with several variables, 21
 Solving for x, 19
 sort, 49
 source code
 ?, 10
 split, 53, 54
 sqrt, 17
 square root, 17
 standard deviation, 27
 Standard functions and constants, 16
 Statistics, 26
 std, 27
 string, 53
 strings
 concatenation, 53
 join, 54
 len, 53
 split, 53
 subgroup, 82
 subgroups, 80
 subsets, 52
 subtraction, 13
 sum, 50
 swap_columns, 93
 swap_rows, 93
 symbolic variables, 20
 symmetric difference, 52
 SymmetricGroup, 78

T

tab completion, 7
 Tab-completion, 107
 tan, 17
 Taylor, 25
 terms, 102
 thickness, 28
 total_degree, 102
 trace, 91
 transpose, 90, 97, 98
 trigonometric functions, 17
 True, 19, 42
 tutorial source, 11

U

union, 52
 unit group, 75

unit_gens, 75
 unit_group_order, 75
 Universes, 39
 Universes and coercion, 39

V

var, 20, 44
 Variables, 43
 assignment, 44
 symbolic, 44
 variables
 deleting, 46
 reset, 45
 restore, 45
 variance, 27
 vector, 88, 90
 Vector and Matrix Spaces, 95

W

weight_distribution, 115
 weight_enumerator, 115
 while statement, 56

X

xgcd, 101, 119

Z

zero_matrix, 89
 zip, 50
 ZZ, 39