Sage for Power Users

William Stein

February 8, 2012

Contents

1 Introduction to Sage **5**
1.1 Motivation 5
1.2 What is Sage? 6
1.3 "This unique American idea of the entrepreneurial company." 6
1.4 Getting Started 7
1.5 A Tour 8

I Programming Sage **11**

2 Python **12**
2.1 What is Python? 12
2.2 The Sage Preparser 12
2.3 Variables 13
2.4 Control Flow 16
2.5 Functions 18
2.5.1 Further remarks on passing variables to functions 21
2.5.2 Gotcha: Default Arguments 22
2.5.3 Gotcha: Recursion 23
2.5.4 Style 24
2.6 Classes 24
2.6.1 Creating a Number 28
2.7 Data Types: list, tuple, set, str, and file 29
2.7.1 Lists 29
2.7.2 Tuples 30
2.7.3 Strings 31
2.7.4 Sets 33
2.7.5 Files 34
2.8 Exception Handling 35
2.9 Decorators 40
2.10 The Ecosystem 43
2.11 Exercise: Build Python from Source 44

3 Cython **46**
3.1 An Example: Speeding up a Simple Function 46
3.2 Using External C/C++ Code 51
3.2.1 Simple random example 51
3.2.2 Adding rational numbers using MPIR 53
3.3 Important Cython Language Constructions 54

3.3.1 Declaring Cython Variables Using `cdef` . . . 54
3.3.2 Explicit casts . . . 55
3.3.3 Declaring External Data Types and Functions . . . 56
3.3.4 Defining New Cython Functions . . . 58
3.3.5 Defining New Cython Classes . . . 60
3.4 Numpy and Cython . . . 61

4 Resources for Solving Problems Using Sage **72**
4.1 The Sage Library . . . 72
4.2 Question and Answer Sites . . . 73
4.3 Documentation for components of Sage . . . 73
4.4 Live Chat . . . 74

5 Sage Development **75**
5.1 Overview of Sage Development . . . 75
5.1.1 What is a Sage Release? . . . 75
5.1.2 Hurdles . . . 77
5.1.3 Walkthrough . . . 77
5.2 How to modify the Sage library and create a patch . . . 78

6 How Sage Uses Other Math Software Systems **81**

II Using Sage **86**

7 Graphics **87**
7.1 2d Plots . . . 87
7.2 3d Plots . . . 88
7.3 Matplotlib . . . 90

8 Number Theory **96**
8.1 Prime Numbers and the Riemann Hypothesis . . . 96
8.1.1 Primes . . . 96
8.1.2 Factorization . . . 97
8.1.3 Counting Primes . . . 99
8.2 Public-Key Cryptography: Diffie-Hellman . . . 103
8.3 Elliptic Curves and the Birch and Swinnerton-Dyer Conjecture . . . 104
8.3.1 Fields . . . 104
8.3.2 Elliptic Curves . . . 105
8.3.3 Birch and Swinnerton-Dyer . . . 106

9 Statistics **111**
9.1 Using R with Sage . . . 111

10 Abstract Algebra **121**
10.1 Groups, Rings and Fields . . . 121
10.2 Exact Linear Algebra . . . 125
10.2.1 Documentation for Linear Algebra in Sage . . . 125
10.2.2 Underlying Technology . . . 125
10.2.3 Matrices and Vectors . . . 126
10.2.4 Vector Spaces . . . 130

11 Databases **132**
11.1 Saving and Loading Python Objects 132
11.1.1 save and load 132
11.1.2 pickle: Python object serialization 133
11.2 SQLite and SQLAlchemy 140

Preface

This is a book about Sage `http://sagemath.org`, which is a large free open source software project that I started in 2005, whose "mission statement" is to create a viable free open source alternative to the commercial programs Magma, Maple, Mathematica, and Matlab. I have given many talks, tutorials, and workshops on Sage, and this book records what I have personally found to be the most important key ideas that are needed to make effective use of Sage. My intention is that you read the whole book cover-to-cover, and have thus kept the book intentionally short.

I assume that you have some previous computer programming experience, but not necessarily in Python. Though I'll make reference to many mathematical topics when illustrating how to use Sage, do not worry if some are not familiar to you.

This book is licensed under the Creative Commons Attribution 3.0 license[1], so it will always be freely available in many formats.

[1] See `http://creativecommons.org/licenses/by/3.0/`.

Chapter 1

Introduction to Sage

1.1 Motivation

I started the Sage project in early 2005 in order to create a viable free open source mathematical software package that I could use for my research. I was frustrated with not being allowed to easily change or understand the internals of closed source systems[1] and I had a deep concern that my students and colleagues could not easily use the commercially distributed software that I had spent many years developing (and contributing). I started Sage as a new project instead of switching to another system, since the capabilities of any available software for number theory at the time were far behind many key features of commercial systems.[2] Several hundred people have since become involved in Sage development, and the goals have broadened substantially.

Sage uses a mainstream programming language, unlike all popular mathematics software, including Maple, Mathematica, and Matlab, which each use their own special-purpose languages written just for mathematics. One works with Sage using Python, which is one of the world's most popular general purpose scripting languages. By using Python, one can use almost anything ever written in Python directly in Sage. And there is much useful Python code out there that addresses a wide range of application areas.

Instead of writing many of the core libraries from scratch like most math software systems have done in Sage I assembled together the best open source software out there, and built on it[3]. Also, the complete system is easily buildable from source on a range of computers. There are challenges: some of the upstream libraries can be difficult to understand, are written in a range of languages, and have different conventions than Sage. Thus it is important to strongly encouraging good relations with the projects that create many of the components of Sage.

A wide and vibrant community of developers and users have become involved with Sage. Due to the broad interests of this large community of developers, Sage has grown into a project with the following specific goal:

> **Mission Statement:** Provide a viable free open source alternative to Magma, Maple, Mathematica, and Matlab.

[1]For me, this was a powerful niche program called "Magma".

[2]For example, Magma's tools for linear algebra over the rational numbers and finite fields were vastly superior to anything available anywhere else.

[3]Sage includes over 500,000 lines of code that does not come from third party projects.

1.2 What is Sage?

Sage is a free open-source mathematics software system licensed under the GNU Public License (GPL). It combines the power of about 100 open-source packages with a large amount of new code to provide a free open source platform for mathematical computation. Sage has many notable features.

- Sage is free, due mainly to the volunteer effort of hundreds of people and generous funding from the National Science Foundation, private donations, and other organizations such as Google and Microsoft. There are no license codes or copy protection. Sage is also open source, so there are absolutely no secret or proprietary algorithms anywhere in Sage. There is nothing that you are not allowed to see or change.
- Sage uses the mainstream programming language Python. Learning Sage will make you proficient in this popular, widely used, and well supported free programming language, which you will likely also use for other non-mathematics projects. Moreover, Sage features the Cython compiler, which allows one to combine Python, C/C++/Fortran libraries, and native machine types for potentially huge speedups.
- Sage is uniquely able to combine functionality from dozens of other mathematical software programs and programming languages via smart psuedoterminal interfaces. You can combine Lisp, Mathematica, and C code to attack a single problem.
- Sage has both a sophisticated multiuser web-based graphical user interface and a powerful command line interface. Sage can also be made to work with any other Python interactive development environment (IDE).
- Sage may have the widest range of mathematical capabilities of any single mathematical software system available. Sage and its components are developed by an active and enthusiastic worldwide community of people from many areas of mathematics, science, and engineering.
- Modifications to Sage are publicly peer reviewed, and what goes into Sage is decided via community discussions; no matter who you are, if you have a brilliant idea, the energy, and can clearly argue that something should go into Sage, it probably will. Known bugs in Sage, and all discussions about them are available for all to see.

Sage is nothing like Magma, Maple, Mathematica, and Matlab, in which details of their implementations of algorithms is secret, their list of bugs is concealed, how they decided what got included in each release is under wraps, their custom programming language locks you in, and you must fight with license codes, copy protection and intentionally crippled web interfaces.

1.3 "This unique American idea of the entrepreneurial company."

The Mathematica documentation has an argument for why looking at the internals of mathematical software is not necessary.

> "Particularly in more advanced applications of Mathematica, it may sometimes seem worthwhile to try to analyze internal algorithms in order to predict which way of doing a given computation will be the most efficient. And there are indeed occasionally major improvements that you will be able to make in specific computations as a result of such analyses.
>
> But most often the analyses will not be worthwhile. For the internals of Mathematica are quite complicated, and even given a basic description of the algorithm used for a particular purpose, it is usually extremely difficult to reach a reliable conclusion about how the detailed implementation of this algorithm will actually behave in particular circumstances."
>
> — `http://reference.wolfram.com/mathematica/tutorial/WhyYouDoNotUsuallyNeedToKnowAboutInternals.html`

Wolfram, who founded the company that sells Mathematica, admits that the mathematical community hates some of what he has done, arguing that a closed source commercial model is the only approach that can possibly work.

> "There's another thing, quite honestly, that that community has a hard time with. They sort of hate one aspect of what I have done, which is to take intellectual developments and make a company out of them and sell things to people.
>
> My own view of that, which has hardened over the years, is, my god, that's the right thing to do. If you look at what's happened with TeX, for example, which went in the other direction... well, Mathematica could not have been brought to where it is today if it had not been done as a commercial effort. The amount of money that has to be spent to do all the details of development, you just can't support that in any other way than this unique American idea of the entrepreneurial company."
>
> – Stephen Wolfram, 1993, Doctor Dobbs Journal Interview

For the last 20 years, Matlab, Mathematica, and the other commercial systems have dominanted with on the order of a hundred million dollars a year in revenue. If the Sage project succeeds at its goals (still a big if), it will have proved that Wolfram is wrong and radically change the landscape of computational mathematics.

1.4 Getting Started

The easiest way to get started with Sage right *now* is to visit `http://480.sagenb.org` and login using OpenID by clicking one of the buttons at the bottom right. This should work with nearly any operating system and browser combination[4]. Using Sage via the above webpage is fine if you just want to use Sage via the notebook, e.g., for learning Python (Chapter 2) and Cython (Chapter 3).

There are some situations where you will instead want to install Sage on your own computer, or get an account on a command-line server on which Sage is installed:

1. You want to use the Sage command line interface.

2. You want to use the interactive command line profiler and debugger, which haven't been properly ported to the notebook yet (see Chapter **??**).

[4] I recommend that you avoid using Internet Explorer if at all possible.

3. You want to modify Sage and contribute back new code (see Chapter ??).
4. You want to interface nonfree software with Sage (see Chapter 6). It would be illegal for me to allow just anybody to run Maple/Mathematica/etc. code at `http://480.sagenb.org`.
5. You do not have access to the Internet.

Remark 1.4.1. Eliminating all but the last reason above are current goals of the Sage project. A command line interface should be added to the notebook, and it should support the profiler and debugger. It should be possible to edit all files in the source code of Sage, use revision control systems, etc., completely via the web. Even the legal issue involving nonfree software could be resolved by hooking into our University's authentication system, just as you authenticate for off-campus access to library resources.

1.5 A Tour

Sage uses the basic user-interface principle of "question and answer" found in many other mathematical software systems. You enter input written in a well-defined language and, after pressing the return key in the command line interface or pressing shift+return in the notebook interface, Sage evaluates your input and returns the result.

A traditional test that Sage is working is to compute 2+2:

```
sage: 2 + 2
4
```

We factor a whole number.

```
sage: factor(2012)
2^2 * 503
```

Thus $2012 = 2 \times 2 \times 503$. Sage can also factor negative numbers and rational numbers:

```
sage: factor(-2012/2015)
-1 * 2^2 * 5^-1 * 13^-1 * 31^-1 * 503
```

The language that Sage uses is almost the same as the Python programming language. One difference between Sage and Python is that ^ means exponentiation in Sage but exclusive or in Python. Another difference is that integer division results in a rational number in Sage, but is floor division in Python.

```
sage: 2^3
8
sage: 2012/6
1006/3
```

We can also factor symbolic expressions using Sage. To introduce a symbolic variable, use the var command.

```
sage: var('x,y')
(x, y)
sage: F = factor(x^2 - 4*sin(y)^2); F
```

```
(x - 2*sin(y))*(x + 2*sin(y))
```

If you want to put any result in a LaTeX document[5], use the latex command:

```
sage: latex(F)
{\left(x - 2 \, \sin\left(y\right)\right)} {\left(x + 2 \,
 \sin\left(y\right)\right)}
```

which looks like this:

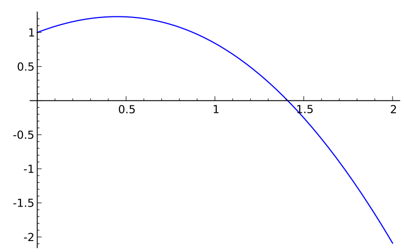
$$(x - 2 \sin(y))(x + 2 \sin(y))$$

Sage knows Calculus:

```
sage: integrate(e^x * sin(x), x)
1/2*(sin(x) - cos(x))*e^x
sage: derivative(1/2*(sin(x) - cos(x))*e^x).expand()
e^x*sin(x)
```

Sage can plot functions:

```
sage: g = plot(sin(x) + (1-x^2), (x, 0, 2)); g
```



To include this plot in a document, save it as a PDF file:

```
sage: g.save('plot1.pdf')
```

We numerically find a root of $\sin(x) + (1 - x^2)$ between 0 and 2, as follows:

```
sage: find_root(sin(x) + (1-x^2), 0, 2)
1.4096240040025754
```

You can use some other programming languages directly from Sage, such as Lisp:

```
sage: s = "(defun factorial(n)"
sage: s += "   (if (= n 1) 1 (* n (factorial (- n 1)))))"
sage: lisp(s)
FACTORIAL
sage: lisp('(factorial 10)')
3628800
```

[5] LaTeX is the dominant tool for producing *professional* quality mathematical papers and books; it is free and open source and you should learn it.

Or use Mathematica (this won't work if you don't have Mathematica):

```
sage: mathematica('Integrate[Sin[x^2],x]') # optional - Mathematica
Sqrt[Pi/2]*FresnelS[Sqrt[2/Pi]*x]
```

Or use Magma, over the web (this should work as long as you have an Internet connection, since it just uses `http://magma.maths.usyd.edu.au/calc/`):

```
sage: magma_free("Factorisation(2012)")
[ <2, 2>, <503, 1> ]
```

Part I

Programming Sage

Chapter 2

Python

Sage uses the Python programming language, which is relatively easy to learn and fun to use. This chapter is a quick Sage-oriented introduction to Python, which you should supplement with a book. Fortunately, the two best books on Python are free: *The Python Tutorial* (see `http://docs.python.org/`) and *Dive Into Python* (see `http://www.diveintopython.net/`).

2.1 What is Python?

Python is a popular free open source language with *no* particular company pushing it. Many big companies such as Google use Python, and support its development. From `http://python.org`:

> "Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs."

- *Work more quickly*: you get stuff done instead of fighting with the language and environment for silly reasons
- *Integrate your systems*: Python is particular useful at creating big systems out of possibly messy collections of software tools.
- *Maintenance costs*: Python code is more likely to be readable and hackable.

Sage is a big integrated system built out of *several million* lines of possibly messy software, code written using Sage tends to be readable and hackable, and people use Sage since it helps them get stuff done immediately.

2.2 The Sage Preparser

When you type commands into Sage, the computer programming language you use is (almost) Python. Each line of code gets automatically run through a preparser before it is sent to the Python interpreter. To see exactly what changes occur, use the `preparse` command:

```
sage: preparse('a = 2.5^3')
"a = RealNumber('2.5')**Integer(3)"
```

As you can see, decimal literals get wrapped using the `RealNumber` command, so when you type 2.5, Python will see `RealNumber('2.5')`. Similarly, integer literals get wrapped using `Integer`. Finally, the caret symbol `^` is replaced by `**`, which is Python's exponentiation operator. One motivation for doing all this is that in Magma, Maple, Mathematica, Matlab and LaTeXthe `^` operator is exponentiation, and making Sage have the same behavior as those systems helps minimize confusion (whereas in Python `^` is "exclusive or"). The preparse does a few other things, but not much more.

If you want to turn off the preparser, type `preparser(False)`:

```
sage: preparser(False)
sage: 2/3 + 2^3
1
```

```
sage: preparser(True)
sage: 2/3 + 2^3
26/3
```

Read more about the preparser at `http://www.sagemath.org/doc/reference/sage/misc/preparser.html`.

2.3 Variables

In Python you create a variable by writing `var = expression`; for example,

```
sage: a = 2
sage: b = 3
sage: a + b
5
```

You can also include several assignment statements on the same line if you separate them with a semicolon:

```
sage: c = 7; d = 15; e = 5
sage: c + d + e
27
```

You do *not* have to end lines with a semicolon. You can also assign the same value to several variables at once:

```
sage: c = d = 10
sage: c + d
20
```

We have only used integers as the expression, but Python supports many other types of objects, such as lists, which we make using square brackets:

```
sage: v = [7, 15, 5]; v
[7, 15, 5]
```

The most important gotcha (and feature) of variables in Python is that variables are a *reference* to a Python object, not a new copy of that object. Thus in the example below, both `v` and `w` "reference" exactly the same Python list:

```
sage: v = [1, 2, 3]
sage: w = v
sage: w[0] = 10
sage: v
[10, 2, 3]
```

Continuing the above example:

```
sage: v[1] = 5
sage: w
[10, 5, 3]
```

If you want a copy of an object, use the `copy` command.

```
sage: v = [1,2,3]
sage: w = v
sage: z = copy(w)
sage: v[0] = 10
sage: print w
[10, 2, 3]
sage: z
[1, 2, 3]
```

The `copy` function only copies the *references* in the list:

```
sage: v = [[1,2], 3, 4]
sage: w = copy(v)
sage: w[1] = 10
sage: w[0][0] = 5
sage: v
[[5, 2], 3, 4]
sage: w
[[5, 2], 10, 4]
```

To recursively make a new copy of everything (as much as possible), use `deepcopy`:

```
sage: v = [[1,2], 3, 4]
sage: w = deepcopy(v)
sage: w[1] = 10; w[0][0] = 5
sage: v
[[1, 2], 3, 4]
sage: w
[[5, 2], 10, 4]
```

You probably won't have to use `deepcopy` often. In over 500,000 lines of code in the core Sage library, deepcopy is used around 177 times:

```
sage -grep deepcopy |wc -l
177
```

The main reason many people are very confused by variables being references in Python is that most other mathematical software (including both Mathematica and Matlab) works differently. For example, in Matlab assignment to a variable creates a new copy. For example, noting that arrays in Matlab are 1-based instead of 0-based,

```
$ matlab
>> v = [1,2,3]
v =
     1     2     3
>> w = v
w =
     1     2     3
>> v(1) = 10
v =
    10     2     3
>> w
w =
     1     2     3
```

And in Mathematica,

```
In[27]:= v = {1,2,3};
In[28]:= w = v;
In[29]:= v[[1]] = 10;
In[30]:= v
Out[30]= {10, 2, 3}
In[31]:= w
Out[31]= {1, 2, 3}
```

But of course in Sage:

```
sage: v = [1,2,3]
sage: w = v
sage: v[0] = 10
sage: v
[10, 2, 3]
sage: w
[10, 2, 3]
```

Remark 2.3.1. Another subtle difference in various computer languages is that exponentiation may associate either left to right or right to left. For example,

```
sage: 3^3^3
7625597484987
```

But in Matlab, we have

```
$ matlab
>> 3^3^3
19683
```

Finally, in Maple we have

```
> 3^3^3
syntax error, ambiguous use of `^`, please use parentheses:
```

Thus watch out: of the two possible design choices about the meaning of `3^3^3`, we quickly find three design decisions made in practice!

Like in Magma, Maple, Matlab, and Mathematica, you do not have to explicitly declare the type of a variable, and it can have several different types in a single snippet of code. This is different to the situation with C, C++ and Java[1]. Use the `type` function to determine the type of a variable, and the `id` function to find out the memory location of the object that a variable references.

```
sage: a = 10
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: id(a)    # random; memory location a points at
4468006416
sage: a = "hello world"
sage: type(a)
<type 'str'>
sage: id(a)    # random; new memory location a now points at
4507478816
```

2.4 Control Flow

The basic control flow statements in Python are `if`, `while` and `for`. The **if** statement lets you choose between alternative code at runtime. Here is an example:

```
a = 2; b = 3
if a > b:
    print(1)
    print("-----")
elif a == b:
    print(2)
else:
    print(3)
```

The Python interpreter evaluates the expression right after `if` and before the colon, and if it evaluates to `True`, then all of the code that is indented before the `elif` or `else` is executed. Otherwise, the expression right after `elif` is evaluated, and if `True`, then the indented code directly below it is evaluated. Otherwise the code under the final `else` is evaluated. The `elif` and `else` are optional, and you can have any number of `elif` blocks.

Unlike nearly every other programming language, there are no explicit begin and end markers around the block of code that will get evaluated when a branch of the if statement is satisfied. Instead the code is indented. There are at least two advantages to Python's choice: (1) you will type and read less, and (2) you will not be fooled by misleading indentation in code like this C code:

[1] Sage is "dynamically typed", whereas C, C++ and Java are "statically typed".

```
if (a>b)
    printf("1");
    printf("-----");
```

Python's `while` statement repeatedly executes the code indented below it until the expression between the `while` and the colon evaluates to False, or until an explicit break statement is executed. Here is an example:

```
i = 5
while i > 0:
    print(i)
    i = i - 1
    if i == 20:
        break
```

When you evaluate this code, you'll see the following output:

```
5
4
3
2
1
```

Each time the indented block of code is executed, the number i is printed out, then the line `i = i - 1` replaces `i` by an integer that is one smaller. Once 0 is reached, the while loop terminates.

If instead, we set `i = 25` at the top, and evaluate the code, we see:

```
25
24
23
22
21
```

This is because the `if` statement evaluates to `True` once i hits 20, and the `break` statement causes the while loop to terminate.

Use the Python `for` loop to iterate over each element in a list or any other "iterable" object. For example,

```
for i in [1, 2, 3, 4, 5]:
    print(i, i*i)
```

will make a table of squares:

```
(1, 1)
(2, 4)
(3, 9)
(4, 16)
(5, 25)
```

You can also use `break` in a `for` loop.

There are many ways to make lists to iterate over (see Section 2.7.1), for example:

```
sage: range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: range(5,20)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
sage: [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: [n^2 for n in [1..10]]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
sage: [1,3,..,10]
[1, 3, 5, 7, 9]
sage: xrange(10^10,10^10+10^9)   # a "lazy" list
xrange(10000000000, 11000000000)
```

For example,

```
for i in xrange(10^10, 10^10+10^9):
    print(i)
    if i > 10^10 + 5: break
```

results in

```
10000000000
10000000001
10000000002
10000000003
10000000004
10000000005
10000000006
```

2.5 Functions

Use `def` to define a function in Python.

```
def foo(a, bar, w=10):
    if a:
        print bar
    # a block of code that is indented
    print a, bar, w
```

The syntax is similar to the syntax of `if`, `for`, and `while`: a keyword, something, a colon, then an indented block of code that gets executed under certain circumstances. More precisely, define a function put `def`, then the name of the function, then in parenthesis the inputs to the function with possible default values (e.g., `w=10` above makes `w` default to `10` if `w` is not specified). When the function is called, the input variables to the function are set to reference the inputs, and the code in the body of the function is executed.

```
sage: foo(1, 'abc', 5)
abc
1 abc 5
```

```
sage: foo(1, 'xyz')
xyz
1 xyz 10
```

You can explicitly specify the input variables as follows, which can make reading your code later easier:

```
sage: foo(bar='gold', a=False, w=3)
False gold 3
```

Unlike the situation with C/C++/Java, there is **absolutely no way** in the Python language to explicitly declare that the types of the inputs and outputs of a function. You can put constraints on types explicitly using the `isinstance` function, or using decorators (see Section 2.9).

```
def g(a, b):
    if not isinstance(a, Integer):
        raise TypeError
    return a+b
```

Then we have:

```
sage: g(2, 3)
5
sage: g('sage', 'math')
Traceback (click to the left of this block for traceback)
...
TypeError
```

Returning to the function `foo` defined above, it will just work with any inputs for which `+` is defined. For example, `a`, `b`, and `c` could be strings or lists:

```
sage: foo('a', 'b', 'c')
a= a b= b c= c
'abc'
sage: f([1,2], [3,4], [5,6])
a= [1, 2] b= [3, 4] c= [5, 6]
[1, 2, 3, 4, 5, 6]
```

Thus illustrates something in Python called "duck typing". So long as an object quacks like a duck (in our case, something that supports +), then we just treat it like a duck. In this sense, all Python functions are extremely generic.

Any variables that are created in the body of the function are *local to the function*, unless you explicitly use the `global` keyword. For example,

```
c = 1; d = 1
def bar(a, b):
    global d
    c = a;   d = b
    print c, d
```

When we call bar, the global variable d gets changed, but the global c does not change:

```
sage: bar(5, 10)
5 10
sage: print c, d
1 10
```

You can also have functions nested within functions (etc.), where the nested function is completey hidden within the scode of the function that contains it:

```
c = 1; d = 1
def bar(a, b):
    global d    # this is a rare beast.
    c = a;   d = b
    print c, d
    def foo(x, y):
        c = 'fun'; d = 'stuff'
        print c, d
    foo(c,d)
    print c,d
```

Running this, note that the global c is not changed, and locally within foo we have yet another pair of variables also called c and d that have nothing to do with the global c, d or the c, d defined at the top of bar.

```
sage: bar(5,10)
5 10
fun stuff
5 10
sage: c,d
(1, 10)
```

As illustrated above, a Python function can have side effects, and behave differently depending on global variables. Thus Python "functions" are different than the functions $f : X \to Y$ in mathematics. In mathematics, $f(x)$ depends only on x, not on the state of some global variable, the time of day, phase of the moon, etc, but in Python f(x) can depend on more than just x. The following Python function evaluates to x^2 when the number of seconds since the beginning is even, and x^3 when it is odd:

```
def f(x):
    import time
    if int(time.time()) % 2 == 0:
        return x^2
    else:
        return x^3
```

Here we imported a Python module called time using the Python import command. In case you are wondering "what the heck is 'time"', you can type

```
sage: import time
sage: help(time)
```

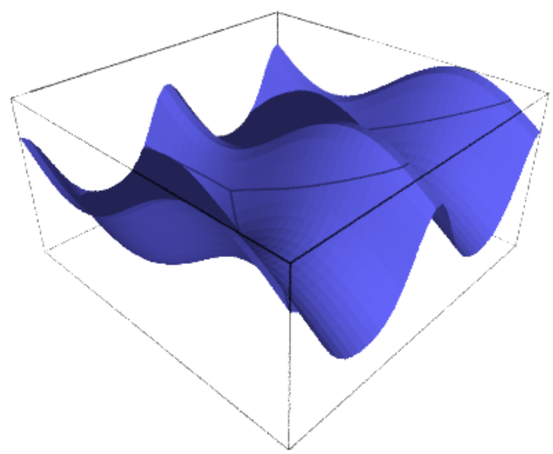
into Sage. In the notebook, you'll get a link to a webpage all about the `time` Python module. Python includes an *enormous* standard library of modules, and you should read all about them at `http://docs.python.org/library/`. I have more than once reimplemented functionality that is already in one of the standard modules because I didn't think to look at the above web page. Want to use Python to parse a webpage? create JSON or XML? use regular expressions? walk a directory tree? compress a file? use a SQLite database? Then consult the Python standard library.

Returning to our function `f` defined above, when we run it, we might get:

```
sage: f(7)
49
sage: f(7)
343
```

Sage (but not Python) also has a notion of Calculus-style functions. For example,

```
sage: f(x,y) = sin(x) + e^cos(y)
sage: f(2,pi)
e^(-1) + sin(2)
sage: f.integrate(x)
(x, y) |--> x*e^cos(y) - cos(x)
sage: plot3d(f, (x,-pi,pi), (y,-2*pi,2*pi), viewer='tachyon')
```



2.5.1 Further remarks on passing variables to functions

We mentioned above that Python uses call by reference semantics. The following example helps clarify this point very explicitly. First we create a list and note where it is stored in memory (at address 69421536 on my computer right now).

```
sage: v = [1,2,3]
sage: id(v)    # random - memory location of v
69421536
```

Next we define a function that prints where in memory its input `w` is stored, and modifies `w`:

```
sage: def foo(w):
...       print "location of w =", id(w)
...       w.append('hello')
...       print "w =", w
```

When we call foo with `v`, note that the variable `w` points to the same memory location as `v`:

```
sage: foo(v)
location of w = 69421536
w = [1, 2, 3, 'hello']
```

Moreover, and it's critical you understand this, the list `v` has now changed!

```
sage: v
[1, 2, 3, 'hello']
```

If we want foo to modify a copy of `v` instead, we have to explicitly use the `copy` function:

```
sage: foo(copy(v))
location of w = 69535936
w = [1, 2, 3, 'hello', 'hello']
```

And this worked fine, as expected:

```
sage: v
[1, 2, 3, 'hello']
```

This illustrates part of the "Zen of Python":

Explicit is better than implicit.

To see the rest of the Zen of Python, type `import this` into Sage.

2.5.2 Gotcha: Default Arguments

Consider the following function `my_append(a,L)` which appends `a` to `L`, but whose second argument is optional, so `my_append(a)` just creates a new list `[a]`:

```
sage: def my_append(a, L=[]):
...       L.append(a)
...       print L
sage: my_append(1)
[1]
sage: my_append(2)    # what?
[1, 2]
sage: my_append(3)    # what? what?
[1, 2, 3]
```

What happened? You might have expected to see output of `[1]`, then `[2]`, then `[3]`. Let's modify the function `my_append` to also print the memory location of `L`.

```
sage: def my_append(a, L=[]):
...       L.append(a)
...       print L, id(L)
sage: my_append(1)          # random memory location
[1] 69438424
sage: my_append(2)          # same random memory location
[1, 2] 69438424
sage: my_append(3)          # same random memory location
[1, 2, 3] 69438424
```

When the function `my_append` is first encountered by the Python interpreter, it evaluates each of the default arguments. When Python sees `L=[]`, it creates a list in memory at location 69438424. Each time you call `my_append` and don't specify the second argument, that same list—at address 69438424—is used, *and modified* in this case.

2.5.3 Gotcha: Recursion

Python supports recursive functions, but they are cripled in that there is by default a fairly small limit on the depth of recursion (you can increase this). This is because Python does not have "tail recursion" like a language such as lisp.

```
def my_factorial(n):
    if n == 1: return n
    assert n > 0
    return n * my_factorial(n-1)
```

This works fine:

```
sage: my_factorial(20)
2432902008176640000
```

But:

```
sage: my_factorial(1000)
Traceback (click to the left of this block for traceback)
...
RuntimeError: maximum recursion depth exceeded in cmp
```

So be careful when writing recursive functions. Often recursive functions will never ever be called with a big depth. However, if you need to write a recursive function that will be called with a big depth, you can simply increase the `recursionlimit` as illustrated below.

```
sage: import sys
sage: sys.getrecursionlimit()
1000
sage: sys.setrecursionlimit(1000000)
sage: a = my_factorial(1000)    # works fine!
```

As an aside, you can in fact write little lisp programs using Sage if you want, since Sage includes an embedded lisp interpreter. For example,

```
sage: lisp.eval('(defun factorial (n) (if (= n 1) 1 (* n (factorial (- n 1)))))')
'FACTORIAL'
sage: lisp('(factorial 10)')
3628800
sage: lisp(10).factorial()
3628800
```

2.5.4 Style

There is a standard coding style that almost everybody uses when writing Python code. Read about it in the Python tutorial:

http://docs.python.org/tutorial/controlflow.html#intermezzo-coding-style

Here is a stylish example:

```
def good_function(a, b = 10):
    """
    This is a good function.

    This function has a docstring and is named using
    lower_case_with_underscores.

    It takes as input integers a and b and outputs something computed
    using them.  (Notice that the above line is <= 79 characters.)
    """
    c = 0

    for i in range(a):
        # add i-th power of b to a and
        # put spaces around operators (comment on line of its own).
        c = b**i + a

    # Another block, and a correctly named class (CamelCase).
    class UselessWrapper(int):
        pass

    return UselessWrapper(c)
```

2.6 Classes

Python classes are typically used to define your own new data type (though they can be used in other ways as well). New classes are easy to define, and support standard object-oriented features such as "multiple inheritance" and "operator overloading".

Here is a trivial example of a class:

```
class CoolThing(object):
    def foo(self, xyz):
        print self, xyz
```

Let’s try it out:

```
sage: z = CoolThing()
sage: z.foo('abc')
<__main__.CoolThing object at 0x...> abc
sage: type(z)
<class '__main__.CoolThing'>
```

The line `class CoolThing(object):` starts declaration of the class `CoolThing`, which derives from the builtin class `object`. Typing `z = CoolThing()` creates a new instance of the class with the variable `z` referencing it. The `foo` method defined above is a function that can only be used with instances, which we call by writing `z.foo('abc')`. Note that the first argument to `def foo(self, xyz)` is `self`, which refers to the particular instance of the class.

Next, we make a more complicated class, which also illustrates how to customize creation of new objects using the `__init__` “dunder method”, define how our objects print themselves using `__repr__`, and define how `+` and `*` implement arithmetic using `__add__` and `__mul__`.

```
class MyRational:
    def __init__(self, n, d):
        self._n = Integer(n); self._d = Integer(d)
    def __repr__(self):
        return '%s/%s'%(self._n, self._d)
    def __add__(self, right):
        return MyRational(self._n*right._d + self._d*right._n,
                          self._d*right._d)
    def __mul__(self, right):
        return MyRational(self._n*right._n, self._d*right._d)
    def reduced_form(self):
        """Return the reduced form of this rational number."""
        a = self._n / self._d
        return MyRational(a.numerator(), a.denominator())
```

Once we define the above class, we have our own new version of “rational numbers”.

```
sage: a = MyRational(2,6); b = MyRational(2, 3)
sage: print a, b
2/6 2/3
sage: a.reduced_form()
1/3
sage: c = a + b; c
18/18
sage: c.reduced_form()
1/1
```

However, notice that subtraction doesn’t work:

```
sage: a - b
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for -: 'instance' and 'instance'
```

This is because we didn't define a `__sub__` method. You can add that method, which looks just like the `__add__` method, except with the `+` replaced by a `-`, and subtraction will work. Alternatively, we can define a derived class that also defines a `__sub__` method, as follows:

```
class MyRational2(MyRational):   # inheritence  (multiple also fully supported)
    def __sub__(self, right):
        return MyRational2(self._n*right._d - self._d*right._n,
                           self._d*right._d)
```

This has absolutely no impact on the original `MyRational` class:

```
sage: MyRational(2,6) - MyRational(2, 3)
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for -: 'MyRational' and 'MyRational'
```

However, instances of `MyRational2` support subtraction, in addition to the multiplication and addition defined above:

```
sage: a = MyRational2(2,6); b = MyRational2(2, 3)
sage: print a, b
2/6 2/3
sage: a + b
18/18
sage: a - b
-6/18
```

Big caveat (!): If you do `a+b`, then the resulting object is an instance of `MyRational`, not of `MyRational2`!

```
sage: type(a-b)
<class '__main__.MyRational2'>
sage: type(a+b)
<class '__main__.MyRational'>
```

This is because the `__add__` method is execute, which explicitly refers to MyRational. You can make the code more robust regarding derivation by using `self.__class__`, as illustrated below:

```
class MyRational3(object):
    def __init__(self, n, d):  # called to initialize object
        self._n = Integer(n); self._d = Integer(d)
    def __add__(self, right):  # called to implement self + right
        return self.__class__(self._n*right._d + self._d*right._n,
                    self._d*right._d)

class MyRational4(MyRational3):
    def __sub__(self, right):  # called to implement self + right
        return self.__class__(self._n*right._d - self._d*right._n,
                   self._d*right._d)
```

Now things work better:

```
sage: a = MyRational4(2,6); b = MyRational4(2, 3)
sage: type(a-b), type(a+b)
(<class '__main__.MyRational4'>, <class '__main__.MyRational4'>)
```

Here is another example that illustrates a default class attribute.

```
class MyClass:
    """
    A simple example class.
    """
    # a Python object attribute; this is basically a default
    # piece of data that is available to each instance of the
    # class, but can be changed in the instance without changing
    # it in the class.  (See example below.)
    i = 12345

    # A function attribute.  Again, this is available to each
    # instance, and can be changed in the instance without
    # changing the class object itself.
    def f(self):
        return 'hello world'
```

First notice that `MyClass` itself is just another Python object (we can have variables reference it, pass it into functions, etc.):

```
sage: MyClass
<class __main__.MyClass at 0x...>
sage: MyClass.i
12345
sage: MyClass.f
<unbound method MyClass.f>
sage: MyClass.__doc__
'A simple example class.'
```

We "call" `MyClass` to create an instance `x` of it:

```
sage: x = MyClass(); x
<__main__.MyClass instance at 0x...>
```

We can then call methods of the instance `x` and get access to its attributes.

```
sage: x.f()
'hello world'
sage: x.i
12345
```

We can also change the attributes and methods of `x`.

```
sage: x.i = 50
sage: def g(): return "goodbye"
sage: x.f = g
sage: x.f()
'goodbye'
```

This does not change the attributes or methods of `MyClass` or new instances of `MyClass`.

```
sage: y = MyClass(); y.i
12345
sage: y.f()
'hello world'
```

We could change those if we wanted to though, as follows:

```
sage: def g(self): return "goodbye"
sage: MyClass.f = g
sage: y = MyClass()
sage: y.f()
'goodbye'
```

As you can see, Python is a *dynamic* language. The above is all happening at runtime. This is different than static languages such as C/C++/Java. It has pros and cons, with the main con being that Python can be slower. We will learn about Cython soon, which is similar to Python but gives you the option of surrending some of the dynamic features of Python in exchange for faster (but less dynamic) static semantics.

2.6.1 Creating a Number

The next example illustrates how to use self and some "dunder" (=double underscore) methods:

```
class Number:
    def __init__(self, x):
        # called when Number is instantiated
        self.x = x
    def __repr__(self):
        # defines how Number prints
        return "The Number %s"%self.x
    def __add__(self, right):
        # defines how "+" works
        return Number(self.x + right.x)
```

Now we create a number `n`, print it, and add it (using +) to another number.

```
sage: n = Number(37)
sage: n
The Number 37
sage: n + Number(15)
The Number 52
```

Try to add subtraction and multiplication to the class `Number` right now. The names of the relevant dunder methods are `__sub__` and `__mul__`.

See `http://docs.python.org/reference/datamodel.html` for long lists of dunder methods.

2.7 Data Types: list, tuple, set, str, and file

2.7.1 Lists

A list in Python is a finite ordered "list" of any Python objects at all. Many useful operations are supported, along with a handy "list comprehension" notation that makes building lists easy.

First we create a list, whose entries are an integer, a string, a data type, and another list with a list in it. Note that `v` has type `list`.

```
sage: v = [3, 'hello', Integer, ['a', [1,2]]]
sage: type(v)
<type 'list'>
sage: v
[3, 'hello', <type 'sage.rings.integer.Integer'>, ['a', [1, 2]]]
```

Lists in Python are 0-based, in that `v[0]` is the first entry in the list. *Remember this!*

```
sage: v[0]
3
sage: v[1]
'hello'
```

You can also index into the list from the other side by using negative numbers:

```
sage: v[-1]
['a', [1, 2]]
sage: v[-2]
<type 'sage.rings.integer.Integer'>
```

You can slice lists. When slicing you specify a start and stop point, and take all the elements between. Keep in mind that it includes the starting point you specify, but excludes the endpoint.

```
sage: v[1:]
['hello', <type 'sage.rings.integer.Integer'>, ['a', [1, 2]]]
sage: v[0:3]
[3, 'hello', <type 'sage.rings.integer.Integer'>]
sage: v[0:3:2]     # just the even-indexed positions
[3, <type 'sage.rings.integer.Integer'>]
```

Use `len` to get the length of a list. New Sage/Python users often get very frustrated trying to figure out how to find the length of a list. Just memorize this right now!

```
sage: len(v)
4
```

You can also sort, append to, delete elements from, extend, etc., lists. See the Python documentation.

```
sage: w = copy(v)
sage: w.sort(); w
[3, ['a', [1, 2]], 'hello', <type 'sage.rings.integer.Integer'>]
```

```
sage: w.extend([1,2,3,4]); w
[3, ['a', [1, 2]], 'hello', <type 'sage.rings.integer.Integer'>,
    1, 2, 3, 4]
```

You can build lists in place using list comprehension, which is a lot like ”set building notation” in mathematics. For example:

```
sage: [n*(n+1)/2 for n in range(1, 10) if n%2 == 1]
[1, 6, 15, 28, 45]
```

The basic structure of a list comprehension is the following (there are more complicated forms):

```
   [ <expression(i)> for i in <iterable> <optional if condition> ]
```

Notice above that `for n in range(1,10)` and `if n%2 == 1` are both valid snippets of Python code. Aside from possible scoping issues, list comprehensions are basically equivalent to combining a `for` loop with an `if` statement in them, where you append to a list. To illustrate this, note that you can literally almost rearrange the code of such a `for` loop into a list comprehension, for example:

```
z = []
for n in range(1, 10):
    if n % 2 == 1:
        z.append(n*(n+1)/2)
```

If you evaluate the above code, then print `z`, you'll see

```
sage: z
[1, 6, 15, 28, 45]
```

If you want to be effective with Sage/Python, you must master lists.

2.7.2 Tuples

Tuples are similar to lists, except you can't change which objects are stored in a tuple. Also, there is no tuple-comprehension; you have to make a list `v`, then change it into a tuple by typing `tuple(v)`. You can however, change the objects themselves if they are mutable.

```
sage: v = (3, 'hello', Integer, ['a', [1,2]]); type(v)
<type 'tuple'>
sage: v[0] = 5  # nope!
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
sage: v[3].append('change a mutable entry'); v
(3, 'hello', <type 'sage.rings.integer.Integer'>,
    ['a', [1, 2], 'change a mutable entry'])
```

BIG FAT WARNING: The following looks like a “tuple comprehension” (if there were such a thing), but it isn't one:

```
sage: w = (n*(n+1)/2 for n in range(1, 10) if n%2 == 1); type(w)
<type 'generator'>
```

Notice that you can't index into `w`:

```
sage: w[0]
Traceback (click to the left of this block for traceback)
...
TypeError: 'generator' object is unsubscriptable
```

You can iterate over `w` though:

```
sage: for n in w: print n,
1 6 15 28 45
```

Here, we get no output since `w` is "used up".

```
sage: for n in w: print n,
```

Anyway, if you want to make a tuple using a list comprehension, be explicit, like so:

```
sage: tuple( n*(n+1)/2 for n in range(1, 10) if n%2 == 1 )
(1, 6, 15, 28, 45)
```

2.7.3 Strings

A string is a finite immutable (unchangeable) sequence of characters. Python supports a wonderful range of string processing functions. To make a string literal:

- Enclose it is in either single or double quotes (just be consistent) – if you use single quotes you can use double quotes in your string without escaping them, and vice versa.
- For a multiline string use three single or double quotes in a row – then you can include newlines directly in your string.
- There are many escape characters for including special characters in strings, e.g., `'\n'` for "newline". If you put the letter `r` right before the quotes you get a raw string, for which a backslash just stays a backslash and you can't escape anything; this is often useful for LaTeX code.

The following examples illustrates some of the above ways of creating strings.

```
sage: s = "this is a string's string using double quotes"; s
"this is a string's string using double quotes"
sage: print s
this is a string's string using double quotes
sage: s = 'this is a string"s using single quotes'; s
'this is a string"s using single quotes'
```

```
s = """this is a
multiline string."""

s = r"""Consider \sin(x) +
\cos(y) and add \pi."""
```

Strings in Python are extremely flexible and easy to manipulate. You can slice them exactly like lists, find substrings, concatenate, etc.

```
sage: s = "This is a string."; s[:10]
'This is a '
sage: s[10:]
'string.'
sage: s[::2]  # get just the even indexed characters
'Ti sasrn.'
sage: s.find('a')
8
sage: s + " Yes, a string."
'This is a string. Yes, a string.'
sage: s.replace('a', 'b')
'This is b string.'
```

The join method is also amazingly useful. If `s` is a string, then `s.join([list of strings])` joins together the list of strings putting `s` between each.

```
sage: ', '.join(['Stein', 'William', 'Arthur'])
'Stein, William, Arthur'
```

Other useful methods are upper and capitalize:

```
sage: s = 'this is lower case'; s.upper()
'THIS IS LOWER CASE'
sage: s.capitalize()
'This is lower case'
```

Finally, the string formating operator % appears constantly in Python code and is extremely useful to know about. Basically, you just put %s's in your string, and these get replaced by the string representations of a tuple of Python objects. Here's how you use it:

```
sage: 'Hi %s. Meet %s.'%('Mom', 2/3)
'Hi Mom. Meet 2/3.'
```

Really what just happened was we created a string and a tuple, and used the mod operator on them, as illustrated below.

```
sage: s = 'Hi %s. Meet %s.'
sage: t = ('Mom', 2/3)
sage: s % t
'Hi Mom. Meet 2/3.'
```

There are many other formating options besides just %s. E.g., %f is useful for numerical computations.

```
sage: '%.2f   %.3f'%(.5,  7/11)
'0.50   0.636'
```

Above, %.2f formats the string with 2 decimal digits after the point, and %.3f with 3 decimal digits.

2.7.4 Sets

A set consists of unique elements with no ordering. You know what is or isn't in the set and can interate over it. The elements of a set must be immutable, since otherwise there would be no way to guarantee objects stay unique after putting them together in a set. Lists are *not immutable* so can't be put in a set, but strings can be.

```
sage: v = ['this', 'is', 'what', 'this', 'is']; v
['this', 'is', 'what', 'this', 'is']
sage: X = set(v); X
set(['this', 'is', 'what'])
sage: type(X)
<type 'set'>
sage: X[0]  # makes no sense
Traceback (most recent call last):
...
TypeError: 'set' object does not support indexing
sage: 'this' in X
True
sage: 'that' in X
False
sage: for a in X: print a,
this is what
```

Here is how to use the set data structure to obtain the distinct types appearing in a list:

```
sage: v = [1/2, 5/8, 2.5, 5/2, 3.8]
sage: t = [type(a) for a in v]; t
[<type 'sage.rings.rational.Rational'>,
 <type 'sage.rings.rational.Rational'>,
 <type 'sage.rings.real_mpfr.RealLiteral'>,
 <type 'sage.rings.rational.Rational'>,
 <type 'sage.rings.real_mpfr.RealLiteral'>]
sage: list(set(t))
[<type 'sage.rings.real_mpfr.RealLiteral'>,
 <type 'sage.rings.rational.Rational'>]
```

If you create your own class, you can decide whether or not Python should consider it immutable by whether or not you define a `__hash__` dunder method. If defined, then your object is considered immutable, and is allowed in sets. First, notice that sets can't contain lists.

```
sage: v = [[1,2], [1,4]]
```

```
sage: set(v)
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

However, nothing stops us from making a class that derives from list and has a hash method:

```
class NaughtyList(list):
    def __hash__(self):     # a 32 or 64-bit int; equal objects should have the same h
        return hash(str(self))
```

```
sage: v = [NaughtyList([1,2]), NaughtyList([1,4])]; v
[[1, 2], [1, 4]]
sage: X = set(v); X
set([[1, 2], [1, 4]])
```

Do something naughty:

```
sage: v[1][1] = 2
sage: X
set([[1, 2], [1, 2]])
sage: v[0] == v[1]
True
```

The set doesn't know:

```
sage: X
set([[1, 2], [1, 2]])
```

2.7.5 Files

It is straightforward to open, read, write, append to, and close files on disk. For example, below we create a file `foo`, write to it, cose it, open it, then read it.

```
sage: F = open('foo','w')
sage: F
<open file 'foo', mode 'w' at 0x...>
sage: F.write('hello there')
sage: F.close()
sage: print open('foo').read()
hello there
```

In the Sage notebook each input cell is executed in a different directory. Thus if you just create a file in one cell, you can't easily open and read it in another cell. The best workaround is to use the `DATA` variable, which is a string that contains the name of a single directory that all cells have access to, and which you can upload/download files to and from using the Data menu.

```
sage notebook: open(DATA + 'foo','w').write('hi')
sage notebook: print open(DATA + 'foo').read()
```

```
hi
sage notebook: os.system('ls -l %s'%DATA)
total 4
-rw-r--r-- 1 sagenbflask sagenbflask 2 ... ... foo
0
sage notebook: print DATA
/sagenb/flask/sage_notebook.sagenb/home/.../.../data/
```

Another important topic involving files is how to read in interesting files, e.g., png image files, wav audio files, csv files, Excel spreadsheets, etc. There are various ways of loading a huge range of interesting files into Sage, but unfortunately there is still no single simple command that parses them all. This would be a good idea for a student project.

2.8 Exception Handling

Python fully supports exception handling, which allows us to raise and handle error conditions eloquently. The syntax in Python for exception handling is as simple and straightforward as you can possibly imagine.

We would like to write a function `formal_sum` that takes as input two arbitrary objects, and adds them (using +) if *possible*, and if not creates their sum in some formal sense. Our first attempt, of course, does not just magically just work:

```
def formal_sum(a, b):
    return a + b
```

Then:

```
sage: formal_sum(2, 3)    # good
5
sage: formal_sum(5, [1,2,3])    # nope
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+':
           'Integer Ring' and '<type 'list'>'
```

How can we *know* whether or not `a + b` will work? You could try to do something really complicated by attempting to predict (via looking at code?) whether `__add__` will work, but that way insanity lies. Instead, use *exception handling*:

```
class FormalSum(object):
    def __init__(self, a, b):
        self.a = a; self.b = b
    def __repr__(self):
        return '%s + %s'%(self.a, self.b)

def formal_sum(a, b):
    try:
        return a + b
    except TypeError:
        return FormalSum(a,b)
```

The class `FormalSum` block above defines a new Python class whose instances represent the formal sum of the two attributes `a` and `b`, and which print themselves nicely. The function `formal_sum` tries to add `a` and `b`, and if this causes a `TypeError`, it instead creates a `FormalSum` instance.

```
sage: formal_sum(3, 8)
11
sage: formal_sum(5, [1,2,3])
5 + [1, 2, 3]
sage: formal_sum(5, 'five')
5 + five
```

For our next example, instead of catching an error, we create a function `divide` that raises an error if the second input `d` equals `0`.

```
def divide(n, d):
    if d == 0:
        raise ZeroDivisionError, "divide by 0!?!"
    return n/d
```

Try it out:

```
sage: divide(5, 7)
5/7
sage: divide(5, 0)
Traceback (most recent call last):
...
ZeroDivisionError: divide by 0!?!
```

Typically, if you try to divide numbers at the denominator is 0, Sage will raise a `ZeroDivisionError`. Just as above, we can catch this case if we want, and return something else, as illustrated below:

```
def divide2(n, d):
    try:
        return divide(n, d)   # or just put "n/d"
    except ZeroDivisionError:
        return 'infinity'
```

```
sage: divide2(5, 3)
5/3
sage: divide2(5, 0)
'infinity'
```

This web page `http://docs.python.org/lib/module-exceptions.html` lists all the standard builtin exceptions along with what each means. Some common exceptions that often appear in the context of mathematics are: `TypeError, ZeroDivisionError, ArithmeticError, ValueError, RuntimeError, NotImplementedError, OverflowError, IndexError`. We illustrate each of these below:

```
sage: ''.join([1,2])
```

```
Traceback (most recent call last):
...
TypeError: sequence item 0: expected string, sage.rings.integer.Integer found
sage: 1/0
Traceback (most recent call last):
...
ZeroDivisionError: Rational division by zero
sage: factor(0)
Traceback (most recent call last):
...
ArithmeticError: Prime factorization of 0 not defined.
sage: CRT(2, 1, 3, 3)
Traceback (most recent call last):
...
ValueError: No solution to crt problem since gcd(3,3) does not divide 2-1
sage: find_root(SR(1), 0, 5)
Traceback (most recent call last):
...
RuntimeError: no zero in the interval, since constant expression is not 0.
sage: RealField(50)(brun)
Traceback (most recent call last):
...
NotImplementedError: brun is only available up to 41 bits
sage: float(5)^float(902830982304982)
Traceback (most recent call last):
...
OverflowError: (34, 'Numerical result out of range')
sage: v = [1,2,3]
sage: v[10]
Traceback (most recent call last):
...
IndexError: list index out of range
```

The key points to remember about exceptions are:

1. Three keywords: `try`, `except`, `raise`
2. How to catch multiple possible exceptions correctly (there is a gotcha here – see below!).
3. One more keyword: `finally`

There is more to exceptions, but these are the key points. We illustrate the last two below in a contrived example.

```
def divide(n, d):
    try:
        return n/d
    except (ZeroDivisionError, ValueError), msg:
        print msg
        return '%s/%s'%(n,d)
    except TypeError, NotImplementedError:
        # the above line is PURE EVIL(!)
        print "NotImplementedError is now '%s'"%NotImplementedError
        print "What have I done?!"
```

```
    finally:
        print "The finally block is *always* executed."
```

Now try it out:

```
sage: divide(2,3)
The finally block is *always* executed.
2/3
sage: divide(2, 0)
Rational division by zero
The finally block is *always* executed.
'2/0'
sage: divide('hi', 'mom')
NotImplementedError is now 'unsupported operand type(s) for /: 'str' and 'str''
What have I done?!
The finally block is *always* executed.
```

The form of the except statement is:

```
      except [single exception], message
```

```
      except (tuple,of,exceptions), message</p>
```

For example,

```
try:
    import foobar
    1/0
except (ZeroDivisionError, ImportError), msg:
    print "oops --", msg
```

outputs `oops -- No module named foobar` and

```
try:
    1/0
    import foobar
except (ZeroDivisionError, ImportError), msg:
    print "oops --", msg
```

outputs `oops -- Rational division by zero`.

An extremely confusing error, which has cost me hours of frustration, is to write

```
      except exception1, exception2:
```

The result is that if exception1 occurs, then exception2 is set to the error message. Don't make the same mistake.

For example, if we evaluate

```
try:
    1/0
    import foobar
except ZeroDivisionError, ImportError:
```

```
    print "oops --"
```

then evaluate

```
try:
    import foobar
    1/0
except ImportError, ZeroDivisionError:
    print "oops --"
```

we see

```
Traceback (click to the left of this block for traceback)
...
ImportError: No module named foobar
```

Wait, what just happened above? We appear to have totally broken Python!? Actually, we have smashed the `ImportError` variable, making it point at the `ZeroDivisionError` message above!

```
sage: ImportError
ZeroDivisionError('Rational division by zero',)
```

We can fix this for now using the `reset` command, which resets a variable to its default state when Sage started up:

```
sage: reset('ImportError')
sage: ImportError
<type 'exceptions.ImportError'>
```

Another *major mistake* I made once[2] with exceptions is illustrated in the following example code:

```
def divide(n, d):
    if d == 0:
        raise ZeroDivisionError, "error dividing n(=%s) by d(=%s)"%(n,d)
```

It's so friendly and nice having a helpful error message that explains what went wrong in the division, right? (No!):

```
sage: divide(3948,0)
Traceback (click to the left of this block for traceback)
...
ZeroDivisionError: error dividing n(=3948) by d(=0)
```

But if we put a large value of n as input, then several seconds (or minutes!) will be spent just creating the error message. It's ridiculous that divide2 below takes over 3 seconds, given that all the time is spent creating an error message that we just ignore.

```
def divide2(n,d):
    try:
```

[2] Actually, I made it several hundred times in 2005–2006!

```
        divide(n, d)
    except ZeroDivisionError, msg:
        return 'infinity'
```

```
sage: n = 3^(10^7)
sage: time divide2(n, 0)
'infinity'
Time: CPU 3.45 s, Wall: 3.46 s
```

Once the Sage developer David Harvey spent a long time tracking down why certain power series arithmetic in Sage was so slow for his application. It turned out that deep in the code there was a try/except block in which the error message itself took over a minute to construct, and then it was immediately discarded. **Moral:** *be very careful when constructing the error message that you include along with an exception!*

2.9 Decorators

The definition of decorators is remarkably simple, but using them is subtle, powerful, and potentially dangerous. From PEP 318 (see `http://www.python.org/dev/peps/pep-0318`), we have the following new notation in Python (note the first line with the mysterious @ sign):

```
@dec1
def func(arg1, arg2, ...):
    pass
```

This is equivalent to:

```
def func(arg1, arg2, ...):
    pass
func = dec2(dec1(func))
```

That's it!

To motivate the point of decorators, let's make a function called `echo` that takes as input a function `f`, and returns a new function that acts just like `f`, except that it prints all of its inputs. Here we use `*args` and `**kwds`, which is something that we have not discussed before. In Python, use `*args` to refer to all of the positional inputs to a function, and `**kwds` to refer to all of the keyword inputs. When you do this, `args` is a Python tuple containing the positional inputs in order, and `kwds` is a dictionary of the `keyword=value` pairs. You can pass args and kwds on to another function (as illustrated below) by typing `*args` and `**kwds`.

```
def echo(f):
    def g(*args, **kwds):
        print "args =", args
        print "kwds =", kwds
        return f(*args, **kwds)
    return g
```

Now, let's try it out. Define a function:

```
def add_em_up(a, b, c):
    return a + b + c
```

Now use it:

```
sage: add_em_up(1, 2, 3)
6
```

The following works, but it sort of looks funny.

```
sage: add_em_up = echo(add_em_up)
sage: add_em_up(1, 2, 3)
args = (1, 2, 3)
kwds = {}
6
```

Using a decorator right when we define `add_em_up` is much, much cleaner:

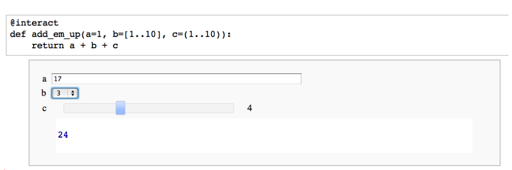
```
@echo
def add_em_up(a, b, c):
    return a + b + c
```

Now we have:

```
sage: add_em_up(1, 2, 3)
args = (1, 2, 3)
kwds = {}
6
```

Here's another example of a very handy decorator (only available in the Sage notebook):

```
@interact
def add_em_up(a=1, b=[1..10], c=(1..10)):
    return a + b + c
```



A hope you can *sense the possibilities....* Here we do type checking:

```
class returns:
    def __init__(self, typ):
        self._typ = typ
    def __call__(self, f):
        return lambda *args, **kwds: self._typ(f(*args, **kwds))
```

```
@returns(float)
def f(n,m):
    """Returns n + m."""
    return n + m
```

Let's try it out:

```
sage: f(2,3)
5.0
sage: type(f(5,6))
<type 'float'>
sage: f('4', '123')
4123.0
```

Here's another example I use all the time. If you put `@parallel(ncpus)` before a function *and* you call the function using a list as input, then the function gets evaluated at each element of the list in parallel, and the results are returned as an iterator. If you call the function without giving a list as input, it just works as usual (not in parallel).

```
@parallel(10)
def f(n):
    sleep(1)   # make function seem slow
    return n*(n+1)/2
```

First, try it not in parallel, which takes a long time.

```
%time
sage: for n in [1..10]: print n, f(n)
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
CPU time: 0.00 s,  Wall time: 10.00 s
```

Now try it in parallel:

```
%time
sage: for X in f([1..10]): print X
(((1,), {}), 1)
(((2,), {}), 3)
(((3,), {}), 6)
(((4,), {}), 10)
(((5,), {}), 15)
(((6,), {}), 21)
(((7,), {}), 28)
(((8,), {}), 36)
(((9,), {}), 45)
```

```
(((10,), {}), 55)
CPU time: 0.19 s,  Wall time: 1.32 s
```

2.10 The Ecosystem

The Sage distributuion itself consists of about 100 open source programs and libraries, which (like Linux) are developed by a loosely knit international group of programmers. Many of these programs are written as Python libraries.

Any software engineer knows that a programming language is much more than just the formal language specification or even a particular implementation. It's also the user community, the general pace of development, and—most importantly—the collections of tools and libraries that are available in that language, especially the free ones. Python excels in available tools, as the following list of many of the Python-based components of Sage attests:

- Pycrypto – fast implementations of many cryptosystems.
- Cython – a Python compiler and tool for efficient use of C/C++ libraries from Python. We will have much more to say about Cython in Chapter 3.
- IPython – interactive interpreter shell
- Jinja2 – HTML and other templating tools; popular for web applications.
- Moinmoin – a standalone wiki, e.g., the one used by `http://wiki.sagemath.org`.
- PIL – Python imaging library (a "programmable Photoshop")
- Pygments – HTML source code highlighting
- SQLalchemy – abstracts interface to most SQL databases and an object:relational mapper.
- Sphinx – ReST documentation system for Python, which is used by many Python projects (including Sage).
- Twisted – a networking framework; everything from web applications to email to ssh servers are implemented in Twisted.
- ZODB – The Zope object-oriented database
- arpack – A sparse numerical linear algebra library.
- CVXopt – A library for solving convex (and other) optimization problems.
- Docutils – related to Python documentation
- easy-install – you can do `easy_install foobar` to install any of the over 13,000 Python packages available at `http://pypi.python.org/`.
- gd – very quickly draw png images with lines, arcs, etc.
- matplotlib – the canonical Python 2d graphics library

- mpmath – arbitrary precision floating point mathematics special functions, numerical integration, matrices, etc.
- NumPy – an n-dimensional array library, which is the fundamental package needed for scientific computing with Python.
- pexpect – control command-line subprocesses
- rpy2 – fast compiled interface to the R statistics program, which is also included in Sage.
- sage – the Sage library; mainly implements mathematical algorithms, especially symbolic ones.
- sagenb – the Sage notebook web application (can be used standalone separate from Sage).
- sagetex – allows you to embed Sage in LaTeX documents
- SciPy – a large library of numerical functions that are useful in mathematics, science, and engineering, including numerical integration, optimization, statistics, differential equations, etc.
- setuptools – package for distributing and working with standalone python packages
- SymPy – a lightweight Python library for symbolic mathematics.

2.11 Exercise: Build Python from Source

If your computer operating system is Linux or OS X (with XCode installed), it is an easy "exercise" to build the Python language from source. This is particularly relevant if you want to understand Python more deeply, since you can change anything you want in the interpreter itself, recompile, and try out the result!

First, go to `http://python.org/download/` and download some version of Python. I am using OS X (with XCode installed) and choose Python 3.2. In a few seconds I have the file `Python-3.2.tar.bz2` in my Downloads folder. Using the Terminal application, I navigate to that folder, extract Python, configure and build it, which takes under 2 minutes (!).

```
deep:~ wstein$ cd Downloads
deep:Downloads wstein$ tar xf Python-3.2.tar.bz2
deep:Downloads wstein$ cd Python-3.2
deep:Python-3.2 wstein$ ./configure; time make -j8
...
real    1m18.284s
user    1m59.552s
sys     0m9.980s
deep:Python-3.2 wstein$
```

And now let's try it out:

```
deep:Python-3.2 wstein$ ./python.exe
Python 3.2 (r32:88445, Mar 30 2011, 10:20:45)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more
information.
>>> 2 + 2
4
```

For fun, let's change something in the core of Python, recompile, and observe our change. On line 288 of `Python-3.2/Objects/listobject.c`, I insert a line that calls the C printf function to print out some graffiti:

```
...
PyObject *
PyList_GetItem(PyObject *op, Py_ssize_t i)
{
    printf("Hi Mom!\n");            /* I added this! */
    if (!PyList_Check(op)) {
        PyErr_BadInternalCall();
...
```

I then type `make` again, wait a few seconds, and try out Python again:

```
deep:Python-3.2 wstein$ ./python.exe
Python 3.2 (r32:88445, Mar 30 2011, 10:25:56)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
Hi Mom!
...
Hi Mom!
>>> v = [1,2,3]
>>> v[0]
1
>>> v['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Hi Mom!
Hi Mom!
Hi Mom!
Hi Mom!
Hi Mom!
TypeError: list indices must be integers, not str
```

Interestingly, the function `PyList_GetItem` appears to not be called when we use an integer to access a list, but it is used when we try to access the list with anything else.

For more information about how the Python source code is laid out, see the README file, especially the section at the end called "Distribution structure".

Chapter 3

Cython

Cython is a compiled variant of the Python language, that can be used to write very, very fast code for Sage, and makes it possible to efficiently use code from existing C/C++ libraries in Sage, or write part of a program in C and the other part in Sage.

Additional references for learning Cython include the Cython Users Guide (see `http://docs.cython.org/src/userguide/`) as you can, and the other Cython documentation (see `http://docs.cython.org/`).

3.1 An Example: Speeding up a Simple Function

Let's start with a first simple example. We write a brute force Python program that computes a double precision (53-bit) floating point approximation to

$$f(n) = \sin(1) + \sin(2) + \sin(3) + \cdots + \sin(n-1) + \sin(n),$$

for n any positive integer. Let's start with the most naive and straightforward implementation of this:

```
def python_sum_symbolic(n):
    return float( sum(sin(n) for n in xrange(1,n+1)) )
```

As a sanity check for each of our implementations, we compute $f(1000)$:

```
sage: python_sum_symbolic(1000)
0.813969634073164
```

Let's benchmark it (these benchmarks were done under OS X on a 2.66GHz Intel Core i7 laptop).

```
sage: timeit('python_sum_symbolic(1000)')
5 loops, best of 3: 193 ms per loop
```

Next, try a bigger input. We use time instead, since that only runs the function once, and this is going to take awhile:

```
sage: time python_sum_symbolic(10^4)
1.6338910217924467
```

```
Time: CPU 15.88 s, Wall: 17.64 s
```

This is really, really, shockingly slow. By the end of this section, you'll see how to compute $f(n)$ over **17 million times** faster!

One reason `python_sum_symbolic` is so bad, is that the `sin` function is a "symbolic function", in that it keeps track of exact values, etc. For example,

```
sage: sum(sin(n) for n in xrange(1, 10+1))
sin(1) + sin(2) + sin(3) + sin(4) + sin(5) + sin(6) +
sin(7) + sin(8) + sin(9) + sin(10)
```

So the `python_sum_symbolic` function is computing a huge formal sum, then converting each summand to a float, and finally adding them up. This is wasteful. Our next implementation using a different sin function, one that is standard with Python, which immediately turns its input into a float (53-bit double precision number), computes `sin`, and returns the result as a float.

```
def python_sum(n):
    from math import sin
    return sum(sin(i) for i in xrange(1, n+1))
```

How does this do?

```
sage: python_sum(1000)  # right answer
0.8139696340731659
sage: timeit('k = python_sum(1000)')
625 loops, best of 3: 185 microseconds per loop
sage: timeit('k = python_sum(10^4)')
125 loops, best of 3: 2.07 ms per loop
```

Thus `python_sum` is over *one thousand times faster* than `python_sum_symbolic`!

```
sage: 193e-3 / 185e-6
1043.24324324324
sage: 15.88 / 2.07e-3
7671.49758454106
```

Perhaps there is some win in not using `sum`?

```
def python_sum2(n):
    from math import sin
    s = float(0)
    for i in range(1, n+1):
        s += sin(i)
    return s
```

Try it out: nope, no win at all (!):

```
sage: timeit('k = python_sum2(10^4)')
125 loops, best of 3: 2.11 ms per loop
```

Next we try using Cython to make our code faster. Note that our "rewritten" program looks identical—the only difference so far is that we told Sage to compile the program using Cython by putting `%cython` at the beginning of the block (if you are using the command line instead of the notebook, put the code without `%cython` in a file `foo.pyx`, then type `load foo.pyx`.)

```
%cython
def cython_sum(n):
    from math import sin
    return sum(sin(i) for i in xrange(1, n+1))
```

If you evaluate the above code in the Sage notebook, you'll see that two linked files appear after the input cell:

1. A file that ends in .c: this is the C program that the above code got turned into. This is compiled and linked automatically into the running copy of Sage.
2. A file that ends in .html: this is an annotated version of the above Cython program; double click on a line to see the corresponding C code.

Is the Cython program any faster?

```
sage: cython_sum(1000)    # it works
0.8139696340731659
sage: timeit('cython_sum(1000)')
625 loops, best of 3: 144 microseconds per loop
sage: timeit('k = cython_sum(10^4)')
625 loops, best of 3: 1.52 ms per loop
```

It's faster, but only about 30% faster. This is very typical of what you should expect by simply putting `%cython` above Python code.

```
sage: 2.07/1.52
1.36184210526316
```

The Cython program is not *that* much faster, because the computer is doing essentially the same thing in both functions. In the case of `python_sum`, the Python interpreter is carrying out a sequence of operations (calling functions in the Python C library), and in the case of `cython_sum`, a C program is running (the compiled Cython module), which is simply calling pretty much the same functions in the Python C library.

To get a further speedup, we declare certain variables to have C data types and use a C version of the sin function.

```
%cython
cdef extern from "math.h":
    double sin(double)

def cython_sum_typed_lib(long n):
    cdef long i
    return sum(sin(i) for i in range(1, n+1))
```

The differences are that we declared `n` to be long and we added a new line `cdef long i`, which declares `i` to also be long. This tells Cython to treat `n` and `i` as being of data type `long`, which is a 32 or 64-bit integer, depending on the computer you're using. This is the same as the long datatype in C. We also call the `sin` function in the math C library. Let's see if this is faster.

```
sage: cython_sum_typed_lib(1000)
0.8139696340731659
sage: timeit('cython_sum_typed_lib(1000)')
625 loops, best of 3: 85.2  s per loop
sage: timeit('cython_sum_typed_lib(10^4)')
625 loops, best of 3: 951  s per loop
sage: 2.07e-3 / 778e-6
2.66066838046272
```

So now our coding is beating the pure Python code by a factor of nearly 3.

In general, you have to be more careful, e.g., `long` integers *overflow*. They may be either 32 or 64-bit depending on the computer you are using. The following example illustrates overflow:

```
%cython
def longmul(long a, long b):
    return a*b
```

Now let's try it:

```
sage: longmul(2^10, 2^20)
1073741824
sage: longmul(2^20, 2^50)   # overflows!
0
sage: 2^20 * 2^50
1180591620717411303424
```

For our next optimization, we use a for loop instead of the sum command:

```
%cython
cdef extern from "math.h":
    double sin(double)

def cython_sum_typed_lib_loop(long n):
    cdef long i
    cdef double s = 0
    for i in range(1, n+1):
        s += sin(i)
    return s
```

In Cython, this is really worth it.

```
sage: cython_sum_typed_lib_loop(1000)
0.8139696340731659
sage: timeit('cython_sum_typed_lib_loop(10^4)')
625 loops, best of 3: 298  s per loop
sage: 2.07e-3 / 298e-6
6.94630872483221
```

We thus obtain a speedup of a factor of about 7 by switching from Python to Cython, and implementing exactly the same algorithm. Way under the hood, the same `sin` function (providing by the math library on the operating system) is being called, but the Cython version of the function avoids a lot of overhead.

Another approach to this particular numerical problem is to use the numpy library, which allows one to evaluate a function on all entries in an array, and sum the entries. This is called "vectorization". Here's what we get in this particular example:

```
def sum_numpy(n):
    import numpy
    return sum(numpy.sin(numpy.array(range(1, n+1))))
```

Let's try it out:

```
sage: sum_numpy(1000)
0.81396963407316592
sage: timeit('sum_numpy(10^4)')
625 loops, best of 3: 1.33 ms per loop
sage: 2.07e-3 / 1.33e-3
1.55639097744361
```

So for this benchmark, our Numpy implementation is slightly better than pure Python, but Cython is still much faster.

Finally, we try a different algorithm, both using Python and Cython. The symbolic capabilities of Sage can be used to find closed form expressions for certain formal sums.

```
sage: var('i, n')
sage: f = sum(sin(i), i, 1, n).full_simplify(); f
1/2*((cos(1) - 1)*sin(n*arctan(sin(1)/cos(1))) +
sin(1)*cos(n*arctan(sin(1)/cos(1))) - sin(1))/(cos(1) - 1)
```

Thus (and I had no idea before trying this),

$$f(n) = \frac{(\cos(1) - 1)\sin\left(n \arctan\left(\frac{\sin(1)}{\cos(1)}\right)\right) + \sin(1)\cos\left(n \arctan\left(\frac{\sin(1)}{\cos(1)}\right)\right) - \sin(1)}{2(\cos(1) - 1)}$$

Using this, we can give an algorithm to compute this sum that is much faster than anything above, and scales much better as well to larger n.

```
def sum_formula(n):
    from math import sin, cos, atan as arctan
    return (1/2*((cos(1) - 1)*sin(n*arctan(sin(1)/cos(1))) +
       sin(1)*cos(n*arctan(sin(1)/cos(1))) - sin(1))/(cos(1) - 1))
```

How does it do?

```
sage: sum_formula(1000)
0.8139696340731664
sage: timeit('sum_formula(10^4)')
625 loops, best of 3: 36.5   s per loop
sage: 2.07e-3 / 36.5e-6
56.7123287671233
```

Finally, we implement the closed formula, but instead in Cython.

```
%cython

cdef extern from "math.h":
    double sin(double)
    double cos(double)
    double atan(double)

def sum_formula_cython(double n):
    return (.5*((cos(1) - 1)*sin(n*atan(sin(1)/cos(1))) +
      sin(1)*cos(n*atan(sin(1)/cos(1))) - sin(1))/(cos(1) - 1))
```

This is 40 times faster than our Python implementation!

```
sage: sum_formula_cython(1000)
0.8139696340731664
sage: timeit('sum_formula_cython(10^4)')
625 loops, best of 3: 906 ns per loop
sage: 36.5e-6 / 906e-9
40.2869757174393
```

And it is over 2000 times faster than `python_sum`.

```
sage: 2.07e-3 / 906e-9
2284.76821192053
```

It is a whopping **17 million times** faster than our first attempt!

```
sagre: 15.88 / 906e-9
1.75275938189845e7
```

3.2 Using External C/C++ Code

Cython is absolutely critical to the design of Sage, and potentially very important to your own work, because it makes it possible to efficiently make use of data types and functions defined in any C/C++ library. Since there is an enormous amount of useful, fast, debugged C/C++ code out there, Cython gives your Sage and Python programs access to vast amounts of useful capabilities. Also, when used correctly, there is no overhead in calling out to the C libraries, unlike the situation with SWIG, ctypes, and many other approaches to writing C library wrappers.

3.2.1 Simple random example

Here's a first simple example. Type `man random` on the command line (or Google it) to find out about the random C library function:

```
RANDOM(3)                BSD Library Functions Manual   RANDOM(3)

NAME
     initstate, random, setstate, srandom, srandomdev -- better
```

```
     random number generator; routines for changing generators

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdlib.h>

     char *
     initstate(unsigned seed, char *state, size_t size);

     long
     random(void);
...
```

Despite `random` being a function defined in the standard C library, we can still call it from Cython, as follows:

```
%cython
cdef extern from "stdlib.h":                    # (1)
    long random()                               # (2)

def random_nums(int n):                         # (3)
    cdef int i                                  # (4)
    v = [random() for i in range(n)]            # (5)
    return v
```

Let's try it out:

```
sage: random_nums(5)
[1315705257, 1147455227, 1571270137, 1106977565, 1805149207]
sage: timeit('v = random_nums(10^5)')
125 loops, best of 3: 5.56 ms per loop
```

It's interesting to see how this compares to pure Python. Here's the same program in Python:

```
%python
import random
k = 2**31-1
def py_random_nums(n):
    return [random.randint(0,k) for i in range(n)]
```

So the speedup is by a factor of nearly 50:

```
sage: py_random_nums(5)
[317567506, 1289482476, 1766134327, 1216261810, 1427493671]
sage: timeit('v = random_nums(10^5)')
5 loops, best of 3: 251 ms per loop
sage: 251/5.56
45.1438848920863
```

Finally we explain the above code line by line. (TODO)

3.2.2 Adding rational numbers using MPIR

We next consider a more mathematical example: arithmetic with arbitrary precision rational numbers. The MPIR C library (which is included with Sage, but can also be downloaded separately for free for any standard operating system from http://mpir.org/) provides highly optimized arithmetic with arbitrary precision integers and rational numbers.[1] We could make use of MPIR by reading the documentation for MPIR and using cdef extern as above. Fortunately, all of the necessary cdef extern declarations needed to use MPIR are already declared in Sage. You can view all the declarations from the notebook by navigating to <url of notebook server>/src/libs/gmp.

Let's use MPIR directly to create two rational numbers and add them together. The code below is complicated and illustrates many issues and techniques, so we will explain it in great depth. Once you understand this, you can deal with many issues that will come up with Cython.

```
%cython
from sage.libs.gmp.all cimport *                        # (1)
def add_rationals(bytes a, bytes b):                    # (2)
    cdef mpq_t x, y, z                                  # (3)
    mpq_init(x); mpq_init(y); mpq_init(z)               # (4)
    mpq_set_str(x, a, 10)    # base 10 string           # (5)
    mpq_set_str(y, b, 10)
    mpq_add(z, x, y)                                    # (6)
    cdef int n = (mpz_sizeinbase (mpq_numref(z), 10)    # (7)
          + mpz_sizeinbase (mpq_denref(z), 10) + 3)
    cdef char* s = <char*>sage_malloc(sizeof(char)*n)   # (8)
    if not s: raise MemoryError                         # (9)
    cdef bytes c = mpq_get_str(s, 10, z)                # (10)
    mpq_clear(x); mpq_clear(y); mpq_clear(z)            # (11)
    sage_free(s)                                        # (12)
    return c
```

Now let's try it out:

```
sage: add_rationals('2/3', '-5/21')
'3/7'
sage: 2/3 - 5/21
3/7
sage: add_rationals('1/29048203984092834823049',
          '-394/29302938402384092834')
'-11444963066794174536188472/851197732045760533660225724673976778930866'
```

Timings suggest we didn't mess up:

```
sage: timeit("add_rationals('2/3', '-5/21')")
625 loops, best of 3: 1.29   s per loop
sage: timeit('2/3 - 5/21')
625 loops, best of 3: 2.16   s per loop
```

Here's a simplistic check that we probably didn't screw up and introduce any memory leaks. (Go up to the code and comment out some frees to see how this changes.)

[1] MPIR and GMP http://gmplib.org/ are basically the same for our discussion; technically they are "forks" of each other, but export essentially the same functions.

```
sage: print get_memory_usage()
sage: timeit("add_rationals('9012038409238411/13',
              '-4/9082309482309487')",number=10^6)
sage: get_memory_usage()
917.5625
1000000 loops, best of 3: 1.72   s per loop
917.5625
```

Finally, we will go line by line through the code and explain exactly what is going on and why. TODO

3.3 Important Cython Language Constructions

In this section we systematically go through *the most important* standard Cython language constructions. We will not talk about using numpy from Cython, dynamic memory allocation, or subtleties of the C language in this section. Instead we cover declaring and using cdef'd variables, explicit type casts, declaring external data types and functions, defining new Cython cdef'd functions, and declaring new Cython cdef'd classes that can have C attributes.

3.3.1 Declaring Cython Variables Using `cdef`

```
cdef type_name variable_name1, variable_name2, ...
```

The single most important statement that Cython adds to Python is

```
cdef type_name
```

This allows you to declare a variable to have a type. The possibilities for the type include:

- C data type: int, float, double, char. Each can be modified by: short, long, signed, unsigned.
- Certain Python types, including: list, dict, str, object (=Python object), etc.
- Name of a known **`cdef class`** (see below). You may have to cimport the class.
- More complicated C/C++ data types: struct, C++ class, typedef, etc., that have been declared using some other method described below.

```
%cython
def C_type_example():
    # ^ = exclusive or -- no preparsering in Cython!
    cdef int n=5/3, x=2^3
    cdef long int m=908230948239489394
    cdef float y=4.5969
    cdef double z=2.13
    cdef char c='c'
    cdef char* s="a C string"
    print n, x, m, y, z, c, s
```

When we run the above function, we get the following. Note the lack of preparsing, and that the char variable `c` is treated as a number.

```
sage: C_type_example()
1 1 908230948239489394 4.59689998627 2.13 99 a C string
```

```
%cython
def type_example2(x, y):
    cdef list v
    cdef dict z
    v = x
    z = y
```

```
sage: type_example2([1,2], {'a':5})
sage: type_example2(17, {'a':5})
Traceback (most recent call last):
...
TypeError: Expected list, got sage.rings.integer.Integer
sage: type_example2([1,2], 17)
Traceback (most recent call last):
...
TypeError: Expected dict, got sage.rings.integer.Integer
```

For the Cython source code of Sage integers, in the Sage library see `rings/integer.pxd` and `rings/integer.pyx`. Also, browse `libs/gmp/` for the definition of functions such as `mpz_set` below.

```
%cython
from sage.rings.integer cimport Integer   # note the cimport!
def unsafe_mutate(Integer n, Integer m):
    mpz_set(n.value, m.value)
```

```
sage: n = 15
sage: print n, id(n)
15 54852752
sage: unsafe_mutate(n, 2011)
sage: print n, id(n)
2011 54852752
```

3.3.2 Explicit casts

```
    <data_type> foo
```

If you need to force the compiler to treat a variable of one data type as another, you have to use an explicit cast. In Java and C/C++ you would use parenthesis around a type name, as follows:

```
int i = 1;
```

```
long j = 3;
i = (int)j;
```

In Cython, you use angle brackets (note: in Cython this particular cast isn't strictly necessary to get the code to compile, but in Java it is):

```
%cython
cdef int i = 1
cdef long j = 3
i = <int> j
print i
```

Here's an example where we convert a Python string to a char* (i.e., a pointer to an array of characters), then change one of the characters, thus mutating an immutable string.

```
%cython
def unsafe_mutate_str(bytes s, n, c):
    cdef char* t = <char*>s
    t[n] = ord(c)
```

Try it out:

```
sage: s = 'This is an immutable string.'
sage: print s, id(s), hash(s)
This is an immutable string. 72268152 -5654925717092887818
sage: unsafe_mutate_str(s, 9, ' ')
sage: unsafe_mutate_str(s, 11, ' ')
sage: unsafe_mutate_str(s, 12, ' ')
print s, id(s), hash(s)
This is a    mutable string. 72268152 -5654925717092887818
sage: hash('This is a    mutable string.')
-7476166060485806082
```

3.3.3 Declaring External Data Types and Functions

In order for Cython to make use of a function or data type defined in external C/C++ library, Cython has to *explicitly* be told what the input and output types are for that function and what the function should be called. Cython will then generate appropriate C/C++ code and conversions based on these assumptions. There are a large number of files in Sage and Cython itself that declare all the functions provided by various standard libraries, but sometimes you want to make use of a function defined elsewhere, e.g., in your own C/C++ library, so you have to declare things yourself. The purpose of the following examples is to illustrate how to do this. It is also extremely useful to look at the Sage library source code for thousands of additional nontrivial working examples.

```
cdef extern from "filename.h":
     declarations ...
```

The following examples illustrates several different possible declarations. We'll describe each line in detail. This first example declares a single type of round function on doubles – it's as straightforward as it gets.

```
%cython
cdef extern from "math.h":
    double round(double)

def f(double n):
    return round(n)
```

Try it out:

```
sage: f(10.53595)
11.0
```

Now suppose we want a version of round that returns a long. By consulting the man page for round, we find that there is a round function declared as follows:

```
    long int lround(double x);
```

We can declare it exactly like the above, or we can use a C "name specifier", which let's us tell Cython we want to call the function `round` in our Cython code, but when Cython generates code it should actually emit `lround`. This is what we do below.

```
%cython
cdef extern from "math.h":
    long int round "lround"(double)

def f(double n):
    return round(n)
```

```
sage: f(10.53595)
11
```

Another case when using C name specifiers is useful if you want to be able to call both a C library version of a function and a builtin Python function with the same name.

```
%cython
cdef extern from "stdlib.h":
    int c_abs "abs"(int i)

def myabs(n):
    print abs(n)
    print c_abs(n)
```

Now use it:

```
sage: myabs(-10)
10
10
```

We can also declare data types and variables using `cdef extern`. To write the code below, I used the `man` command on my computer several times on each referenced function. I knew the relevant functions because I read a book on the C programming language when I was a freshman; learning the basics of the C programming language and standard libraries is a very good idea if you want to be able to make effective use of Cython... or computers in general, since most systems programming is done in C.

Coming up with the declarations below is a little bit of an art form, in that they are not exactly what is given from the man pages, though they are close. Just realize that the declarations you give here do exactly one thing: they inform Cython about what C code it should generate, e.g., it will convert the string `"w"` below to a `char*` before calling the `fopen` function. That's it, that's all the declarations do; they do not have to be perfect. You should evaluate this code in the notebook and click on the .html file that is produced, then look at the corresponding C code, to see what I mean.

```
%cython
cdef extern from "stdio.h":
    # We use void* since we don't care about structure of FILE
    ctypedef void* FILE
    FILE* fopen(char* filename, char* mode)
    int fclose(FILE *stream)
    int fprintf(FILE *stream, char *format, ...)

def f(filename):
    cdef FILE* file
    file = fopen(filename, "w")
    fprintf(file, "Hi Mom!")
    fclose(file)
```

Let's try create create and write to a file using the above code:

```
sage: f('foo.txt')
sage: print open('foo.txt').read()
Hi Mom!
```

It's unlikely you would ever want to access the above functions from Cython, since they are already nicely wrapped by Python itself. Nontheless, if you need total control and speed when doing file access, you have it.

3.3.4 Defining New Cython Functions

In addition to using the `cdef` keyword to define variables as above, we can also define functions. These are like Python functions, but you can declare the input types and the return type explicitly, and calling them is then blazingly fast, as compared to calling regular Python functions. (Remember, most of the point of Cython is speed, speed, speed! The other point of Cython is that you can call C/C++ functions from Cython; that is less relevant if you don't care about speed, because there is something else called ctypes that allows you to do that directly from Python.)

```
cdef return_type function_name(type1 input1, type2 input2...):
    # body of function
```

Here is an example, where we create both a cdef and regular function to add two int's. Note that the return type of the cdef function can itself by a C data type, but the same is not true for the return type of a Python function. We will see below that the cdef function is dramatically faster, since there is very little overhead in calling it.

```
%cython
cdef int add_cython(int a, int b):
    return a + b

def add_python(int a, int b):
    return a + b

def f(int n):
    cdef int i, s=0
    for i in range(n):
        s += add_cython(s, i)
    return s

def g(int n):
    cdef int i, s=0
    for i in range(n):
        s += add_python(s, i)
    return s
```

Let's test it:

```
sage: timeit('f(10^6)')
625 loops, best of 3: 595   s per loop
sage: timeit('g(10^6)')
5 loops, best of 3: 94.6 ms per loop
sage: 94.6/.595
158.991596638655
```

Indeed, we find that the cdef'd function is 159 times faster!

Notice that `add_python` is callable from the interpreter, but `add_cython` isn't:

```
sage: add_python(2,8)
10
sage: add_cython(2,8)
Traceback (most recent call last):
...
NameError: name 'add_cython' is not defined
```

The `cpdef` keyword lets us define a function that is somewhere intermediate between a Python function and a cdef'd function. If we use `cpdef` instead of `cdef` then everything is almost identical, except the `cpdef`'d method can also be called from Python. This is often mainly useful for testing and general usability. The `cpdef` method will be slightly slower though. In this example, it is about 4 times slower.

```
cpdef return_type function_name(type1 input1, type2 input2...):
    # function body
```

Here is the example:

```
%cython

cpdef int add_cython2(int a, int b):
    return a + b

def f2(int n):
    cdef int i, s=0
    for i in range(n):
        s += add_cython2(s, i)
    return s
```

Now test it out:

```
sage: timeit('f2(10^6)')
125 loops, best of 3: 2.63 ms per loop
sage: 2.63/.595
4.42016806722689
sage: add_cython2(2,8)    # the function is available
10
```

3.3.5 Defining New Cython Classes

One of the most powerful features of Cython is that you can define new classes that have C-level attributes and cdef'd methods. The cdef'd attributed and function calls are very, very fast to use.

```
cdef class ClassName(base_class):
     cdef type_name variable
     # ...
     # Then functions mostly like a Python class, except
     # you can include cdef'd methods with input and output
     # types as in the previous section.
     # ...
     # There are some subtleties with special methods such
     # as __add__ and __hash__; see the Cython documentation.
```

Note that cdef'd classes in Cython can have at most one base class; there is no support for multiple inheritance. This is a basic design decision with Cython, and is very unlikely to ever change. You can of course create non-cdef'd classes in Cython that have multiple inheritance.

Here is an example in which we create a Cython class that wraps a Python string, and provides the ability of changing the entries of the string:

```
%cython
cdef class StringMutator:
    cdef bytes s    # cdef's attribute
    def __init__(self, bytes s):
        self.s = s
    def __setitem__(self, int i, bytes a):
        if i < 0 or i >= len(self.s): raise IndexError
        if len(a) != 1: raise ValueError
```

```
        (<char*> self.s)[i] = (<char*>a)[0]
    def __repr__(self): return self.s
    def __str__(self): return "%s"%self.s
```

```
sage: s = "Hello World"
sage: t = StringMutator(s)
sage: t[4] = 'X'
sage: print s
HellX World
sage: print t
HellX World
```

Notice that setting an entry is fast:

```
sage: timeit("t[4]='X'", number=10^5)
100000 loops, best of 3: 226 ns per loop
```

We did include some bounds checking to avoid crashes:

```
sage: t[100] = 'x'
Traceback (most recent call last):
...
IndexError
```

We can also convert from mutable string back and get a new string:

```
sage: m = str(t); m
'HellX World'
sage: t[0] = 'X'; t
XellX World
sage: m
'HellX World'
```

3.4 Numpy and Cython

```
<h1 style="text-align: center;">Lecture 12: Numpy + Cython = AWESOME</h1>
<p>This lecture is about how to efficiently combine Numpy and Cython to write fast numerical co
<p>We will focus on the problem of computing the <em>standard deviation</em> of a list of
<p><strong>Note</strong>: In statistics it is common to divide by $n-1$ instead of $n$ when com
<p> </p>
<p><strong>Running Example:</strong> Compute the standard deviation of a list of 64-bit floatin

{{{id=3|
set_random_seed(0)
import random
v = [random.random() for _ in range(10^5)]
///
}}}
```

```
{{{id=84|
v[:20]
///
[0.43811732887872634, 0.78344784289564662, 0.7917672531341533, 0.43546157784257289, 0.998796301
}}}

<p>First we write a naive straightforward implementation of computation of the standard deviati

{{{id=9|
def my_std(z):
    mean = sum(z)/len(z)
    return sqrt(sum((x-mean)^2 for x in z)/len(z))
///
}}}

{{{id=10|
time my_std(v)
///
0.28871143425255896
Time: CPU 0.06 s, Wall: 0.06 s
}}}

{{{id=11|
timeit('my_std(v)', number=10)
///
10 loops, best of 3: 64.8 ms per loop
}}}

<p>Next we try the std function in Sage, which was implemented by UW undergrad Andrew Hou as pa

{{{id=1|
time std(v, bias=True)
///
0.28871143425255896
Time: CPU 0.03 s, Wall: 0.03 s
}}}

{{{id=12|
timeit('std(v, bias=True)')
///
25 loops, best of 3: 26.4 ms per loop
}}}

<p>Next we try Numpy, which is much faster than the above.</p>

{{{id=7|
import numpy
v_numpy = numpy.array(v, dtype=numpy.float64)
///
}}}
```

```
{{{id=14|
v_numpy.dtype
///
dtype('float64')
}}}

{{{id=21|
v_numpy.std()
///
0.28871143425255896
}}}

{{{id=6|
timeit('v_numpy.std()')
///
625 loops, best of 3: 1.25 ms per loop
}}}

{{{id=76|
22.5/1.25
///
18.0000000000000
}}}

<p>Sage also has code for working with TimeSeries, which happens to have a method for computing

{{{id=16|
v_stats = stats.TimeSeries(v)
///
}}}

{{{id=85|
v_stats.variance??
///
}}}

{{{id=20|
v_stats.standard_deviation(bias=True)
///
0.28871143425255896
}}}

{{{id=15|
timeit('v_stats.standard_deviation(bias=True)')
///
625 loops, best of 3: 240 s per loop
}}}

<p>The TimeSeries code is nearly optimal.  A TimeSeries is represented by a contiguous arr
```

```
{{{id=17|
1.25/.236
///
5.29661016949153
}}}

<p><strong>Goal: </strong>Write a function that computes the standard deviation of a numpy arra
<p>First approach: Use numpy "vectorized operations".  This doesn't help at all (and is al

{{{id=86|
def std_numpy1_oneline(v):
    return math.sqrt(((v - v.mean())**2).mean())
///
}}}

{{{id=23|
def std_numpy1(v):
    m = v.mean()  # mean of entries
    w = v - m     # subtracts m from each entry: "broadcasting"
    w2 = w**2     # squares each entry componentwise.
    return math.sqrt(w2.mean())
///
}}}

{{{id=88|
get_memory_usage()
///
864.90625
}}}

{{{id=89|
w = v_numpy**2
///
}}}

{{{id=90|
get_memory_usage()
///
865.671875
}}}

{{{id=19|
std_numpy1(v_numpy)
///
0.28871143425255896
}}}

{{{id=87|
std_numpy1_oneline(v_numpy)
```

```
///
0.28871143425255896
}}}

{{{id=18|
timeit('std_numpy1(v_numpy)')
///
625 loops, best of 3: 1.25 ms per loop
}}}

<p>Let's see how the time gets spent between each step.  It turns out to be about equally

{{{id=34|
m = v_numpy.mean()
timeit('v_numpy.mean()')
///
625 loops, best of 3: 140 s per loop
}}}

{{{id=37|
w = v_numpy - m
timeit('v_numpy - m')
///
625 loops, best of 3: 241 s per loop
}}}

{{{id=38|
w2 = w**2
timeit('w**2')
///
625 loops, best of 3: 157 s per loop
}}}

{{{id=36|
m2 = w2.mean()
timeit('math.sqrt(w2.mean())')
///
625 loops, best of 3: 143 s per loop
}}}

{{{id=91|
sqrt(2)
///
sqrt(2)
}}}

{{{id=92|
math.sqrt(2)
///
1.4142135623730951
```

```
}}}

{{{id=93|
a = float(2)
timeit('sqrt(a)', number=10^5)
///
100000 loops, best of 3: 589 ns per loop
}}}

{{{id=94|
timeit('math.sqrt(a)', number=10^5)
///
100000 loops, best of 3: 216 ns per loop
}}}

{{{id=39|

///
}}}

<p>Next try Cython with no special type declarations.  Not surprisingly, this does not hel

{{{id=28|
%cython

import math
def std_numpy2(v):
    m = v.mean()  # mean of entries
    w = v - m     # subtracts m from each entry: "broadcasting"
    w2 = w**2     # squares each entry componentwise.
    return math.sqrt(w2.mean())
///
}}}

{{{id=25|
std_numpy2(v_numpy)
///
0.28871143425255896
}}}

{{{id=24|
timeit('std_numpy2(v_numpy)')
///
625 loops, best of 3: 1.3 ms per loop
}}}

<p>Next try Cython with special support for Numpy.  This gets powerful... as we will see.<

{{{id=30|
%cython
```

```
from numpy cimport ndarray
import math
def std_numpy3(ndarray v not None):
    m = v.mean()  # mean of entries
    w = v - m     # subtracts m from each entry: "broadcasting"
    w2 = w**2     # squares each entry componentwise.
    return math.sqrt(w2.mean())
///
}}}

{{{id=96|
std_numpy3(None)
///
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "_sage_input_68.py", line 10, in <module>
    exec compile(u'open("___code___.py","w").write("# -*- coding: utf-8 -*-\\n" + _support_.pre
  File "", line 1, in <module>

  File "/tmp/tmpXdvvgn/___code___.py", line 2, in <module>
    exec compile(u'std_numpy3(None)' + '\n', '', 'single')
  File "", line 1, in <module>

  File "_sagenb_flask_sage_notebook_sagenb_home_openidSfmMv1OuVE_44_code_sage70_spyx_0.pyx", li
    def std_numpy3(ndarray v not None):
TypeError: Argument 'v' has incorrect type (expected numpy.ndarray, got NoneType)
}}}

{{{id=33|
std_numpy3(v_numpy)
///
0.28871143425255896
}}}

{{{id=42|
timeit('std_numpy3(v_numpy)')
///
625 loops, best of 3: 1.7 ms per loop
}}}

<p>Look at Cython + Numpy documentation (by Googling "cython numpy"), and we learn that if we d

{{{id=46|
%cython

cimport numpy as alex
import math

def std_numpy4a(alex.ndarray[alex.float64_t, ndim=1] v not None):
```

```
    cdef Py_ssize_t i
    cdef Py_ssize_t n = v.shape[0]   # how many entries

    # Compute the mean
    cdef double m # = 0
    for i in range(n):
        m += v[i]
    m /= n
    # just doing the mean for now...
    return m
///
}}}

{{{id=45|
std_numpy4a(v_numpy)
///
0.49896857465357608
}}}

<p>Timing looks good...</p>

{{{id=44|
timeit('std_numpy4a(v_numpy)')
///
625 loops, best of 3: 376 s per loop
}}}

{{{id=79|

///
}}}

<p>Let's finish it the function and see how it compares.</p>

{{{id=56|
%cython

cimport numpy as np
cdef extern:
    double sqrt(double)

def std_numpy4b(np.ndarray[np.float64_t, ndim=1] v):
    cdef Py_ssize_t i
    cdef Py_ssize_t n = v.shape[0]   # how many entries

    # Compute the mean
    cdef double m = 0
    for i in range(n):
        m += v[i]
    m /= n
```

```
    # Compute variance
    cdef double s = 0
    for i in range(n):
        s += (v[i] - m)**2

    return sqrt(s/n)
///
}}}

{{{id=55|
std_numpy4b(v_numpy)
///
0.28871143425255896
}}}

{{{id=63|
timeit('std_numpy4b(v_numpy)')
///
625 loops, best of 3: 274 s per loop
}}}

{{{id=54|
timeit('v_stats.standard_deviation(bias=True)')
///
625 loops, best of 3: 238 s per loop
}}}

{{{id=58|
timeit('v_numpy.std()')
///
625 loops, best of 3: 1.27 ms per loop
}}}

<p>Very nice!!</p>

{{{id=60|

///
}}}

<p>Finally, we try again, after disabling bounds checking.   This is even better; almost a

{{{id=50|
%cython

cimport numpy as np
cdef extern:
    double sqrt(double)
```

```
# turn of bounds-checking for entire function
cimport cython
@cython.boundscheck(False)
def std_numpy5a(np.ndarray[np.float64_t, ndim=1] v):
    cdef Py_ssize_t i
    cdef Py_ssize_t n = v.shape[0]   # how many entries
    # Compute the mean
    cdef double m = 0
    for i in range(n):
        m += v[i]
    m /= n
    # Compute variance
    cdef double s = 0
    for i in range(n):
        s += (v[i] - m)**2
    return sqrt(s/n)
///
}}}

{{{id=49|
timeit('std_numpy5a(v_numpy)')
///
625 loops, best of 3: 227 s per loop
}}}

{{{id=43|
timeit('v_stats.standard_deviation(bias=True)')
///
625 loops, best of 3: 240 s per loop
}}}

<h1><span style="color: #800000;">Yeah, we did it!!  </span> </h1>
<p>For smaller input, interestingly we get a massive win over Numpy.   If you were, e.g.,

{{{id=65|
a = numpy.array([1,2,3,4], dtype=float); a
///
array([ 1.,  2.,  3.,  4.])
}}}

{{{id=67|
timeit('std_numpy5a(a)')
///
625 loops, best of 3: 483 ns per loop
}}}

{{{id=68|
timeit('a.std()')
///
625 loops, best of 3: 24.4 s per loop
```

```
}}}

{{{id=69|
b = stats.TimeSeries(a)
timeit('b.standard_deviation(bias=True)')
///
625 loops, best of 3: 534 ns per loop
}}}
```

Chapter 4

Resources for Solving Problems Using Sage

4.1 The Sage Library

You can do a Google search on all of the Sage documentation, web pages and discussion groups all in one go by visiting the webpage http://sagemath.org/search.html and typing in your search, then waiting as the page dynamically updates.

Of course you can find links to the standard Sage documentation, including the tutorial, constructions guide, FAQ, developer's guide, and reference manual at http://sagemath.org/help.html. There are also links to videos and many other helpful materials there.

There are numerous quick reference cards at http://wiki.sagemath.org/quickref which list numerous Sage commands on a single page in specific areas such as Calculus and Linear Algebra. Much work went into creating these cards, and they are an excellent resource to print out.

If you want to search the documentation of the functions defined in the Sage library, use the search_doc command. This just does a straight search through all the docstrings of the functions in the HTML version of the Sage documentation, without any prebuilt index. It is written in Python and uses regularly expressions on the source code to extract docstrings out and find your search terms. The search_doc command works on both the command line and in the notebook. On the command line it displays one line from each HTML document, so is tedious to actually use. In the notebook, it displays the relevant html documents, which links to each. If you click on a link, you'll go to an interactive version of the relevant section of the documentation, where you can search that page for relevant text. Watch out, since stupidly you then have to use the back button to get back to your worksheet – it would be better if the html output of search_doc used <a target="_new" href="...">; until this is changed, you may want to right click and select "open in new tab".

```
sage: search_doc('eigenvalue')
...
```

The HTML documentation for Sage is far from complete; there is lots of code in the Sage library that isn't documented at all in the HTML documentation of Sage, for whatever reason. You can easily search through all of this code by typing search_src(...)

on either the command line or in the notebook.

```
sage: search_src('eigenvalue')
...
```

On the command line you'll get a list of each line in each file that contains the given search term. In the notebook, you will get a list of all files in the Sage library that contain the search term, along with links to the files. The same caveats regarding clicking on the links applies as with `search_doc` (see above). When you click on a file, it will look funny for a moment (a bug, in my opinion), then suddenly refresh and display as a very nicely formated and syntax highlighted page. You should then search this page for your term, in order to see it in context. At the top of the page there is also a link called "browse directory", which lets you browse to any file in the Sage library and similarly view it.

To search the definitions of function, use `search_def`. This works just like `search_src` but restricts the search to the definition lines of functions.

```
sage: search_def('other_graph')
...
```

4.2 Question and Answer Sites

The Sage project hosts their own question and answer site devoted to Sage at `http://ask.sagemath.org`. You can instantly sign in using OpenID and ask a question, or answer one. Specific answerable questions are best. You can also easily search all the questions, and see if anybody has asked a similar question before (and what the answers were). The answers are ranked based on user voting, and the questions are sorted by tags. People are motivated to give good answers, since they get "karma points" for posting useful answers.

One of the first big question/answer sites is `http://stackoverflow.com/`, which has a huge number of questions and answers about all things related to coding. One of the top most popular tags is "python", with over 50000 question. There are also a few dozen questions taged "sage" (some about the Sage math software, and some about the unrelated Sage accounting software). If you run into Python programming questions, this can be an excellent site on which to search for answers or ask questions.

4.3 Documentation for components of Sage

There are many components of Sage that offer vast amounts of functionality, and have excellent documentation, but you'll find almost nothing about them in any of the standard Sage documentation. For example, for numerical computing numpy, scipy, and cvxopt are all included in Sage, and often many, many capabilities that are well documented in their respected documentation. Thus it is quite useful to know that you can do the following:

```
sage: import scipy.special
sage: scipy.special.<call some function>
```

There is a list of all packages included in every recent copy of Sage at `http://sagemath.org/packages/standard/`.

Usually the best way to find the documentation for one of these optional components of Sage is to use Google. Search for the component by name and possibly throw in a word like "math" or if it the component is a Python library the word "python". For example, there is a component of Sage called `mpmath`, and you can find its website by doing a google search for... `mpmath`. Once there, it is easy to find the documentation, and you should quickly be able to start using mpmath's functionality from within Sage.

If you have your own Sage install, you can also install nearly any Python library you want into it. However, this is not an easy option if you're using a public Sage notebook server that somebody else administers.

4.4 Live Chat

There is a live IRC chatroom where you can ask for help anytime, and maybe get some feedback. All you have to do is point your webbrowser to `http://sagemath.org/help-irc.html`, fill in the form, and you're chatting. Type `/who` to see a list of people logged into the forum.

Chapter 5

Sage Development

5.1 Overview of Sage Development

Motivating Problem: *Suppose you want to modify or improve Sage in some way, and want your changes to be included in a future release of Sage. How do you do this?*

5.1.1 What is a Sage Release?

New versions of Sage are released about once every month or two, so it is possible for your contribution to get into Sage and start being used by people relatively quickly. A new release of Sage consists of both an updated version of the code in the Sage library, and updated versions of some of the roughly 90 third-party packages that Sage includes. Before each release, this code (over 6 million lines) is all built from source on dozens of hardware/OS combinations, and hundreds of thousands of tests are run to increase the chances Sage will actually work correctly when you use it.

The way in which Sage is distributed—as both a core library and its dependencies—is somewhat unusual in the world of open source software, though it is similar to how other mathematical software of comparable size and scope to Sage is released (Magma, Mathematica, Matlab, Maple, Enthought's Python Distribution, etc.) Mathematical software is highly interrelated and extremely sensitive to even the slightest changes anywhere in the system, and because Sage has such a large test suite, we actually notice these issues. It is a constant and difficult battle just to keep the components of Sage working together as new and hopefully improved releases of each component appear.

Each new Sage release has an associated changelog, which lists all of the changes that were made to Sage in that release, along with everybody who contributed to the release. You can find a list of these at `http://sagemath.org/mirror/src/changelogs/`

The changelog for Sage-4.6.2 looks like this:

```
Sage 4.6.2 was released on 28 February 2011. It is available at
           http://www.sagemath.org/download.html
  * About Sage (http://www.sagemath.org)
...
The following 100 people contributed to this release. Of those, 25
made their first contribution to Sage:

  * Alain Filbois [first contribution]
  * Alain Giorgetti [first contribution]
```

```
  * Alexandre Blondin Mass
  * Alexey U. Gudchenko [first contribution]
  * Alex Ghitza
  * Aly Deines
  ...
  * William Stein
  * Wolfgang Steiner [first contribution]
  * Yann Laigle-Chapuy
  * Yann Ponty [first contribution]

* Release manager: Jeroen Demeyer.

* Doctesting coverage:

  * Overall weighted coverage score:  84.8%  (84.4% for 4.6.1)
  * Total number of functions:        27200  (26816 for 4.6.1)

* We closed 221 tickets in this release. For details, see

  http://sage.math.washington.edu/home/release/sage-4.6.2/tickets.html

Closed tickets:

 #116: notebook doctest -- should be able to doctest a worksheet,
       so we can distribute worksheets with SAGE [Reviewed by
       Willem Jan Palenstijn]
#5389: Creating a  updated GAP workspace with -tp is racy
       [Reviewed by Willem Jan Palenstijn]
#8216: Make David Perkinson's sandpile 2.2 module an experimental
       (at least) package [Reviewed by David Kirkby]
#9641: Race condition with sage -tp [Reviewed by Willem Jan
       Palenstijn]
#9809: Graph.num_edges() gives wrong answer [Reviewed by Minh
       Van Nguyen]
...
#10816: Volker Braun: Subscheme creation does not work from the
       notebook [Reviewed by Jeroen Demeyer]
#10842: Jeroen Demeyer: Increase timeouts in sage/tests/cmdline.py
       [Reviewed by Volker Braun]
```

Notice that 100 (!) different people contributed improvements and bug fixes to Sage-4.6.2, which was a release that took just over a month to appear. Of these, 25 were first-time contributors.

There were 221 "trac tickets" closed in this release. Each ticket description is listed after a number. Visit `http://sage.math.washington.edu/home/release/sage-4.6.2/tickets.html` for an easy-to-navigate list of these tickets, which links to `http://trac.sagemath.org`, or search for `#number` in the search box in the upper right.

For example, consider ticket #10336. This ticket is an "enhancement", not a bug fix, that includes a code submission by `novoselt`, a.k.a. Andrew Novoseltsov, who is a Russian graduate student studying algebraic geometry in Canada (who used to be a Univ. of Washington graduate student). When you view the ticket you'll see how long ago the patch was posted, that little improvements were made, and that `vbraun`, a.k.a. Volker Braun (a Physicist in Ireland) gives the work a positive review. Moreover, two months after the first code was submitted, it was merged into sage-4.6.2.alpha2 by `jdemeyer`, who is a Belgium number theorist.

Notice that before a final Sage release is made there are a sequence of alpha releases, e.g., sage-4.6.1.alpha1, sage-4.6.1.alpha2, and also release candidates. It is important to emphasize that these are all completely public releases, which anybody can try out, and the source code for all of them, including in progress releases, is available at `http://sage.math.washington.edu/home/release/`. There are other open source projects, even components of Sage[1]), that keep their alpha releases secret or semisecret; we believe this is a counterproductive approach to the creation of open source software, and that it is best to keep every step of the development process open.

5.1.2 Hurdles

There are several hurdles to getting your code into Sage:

- You have to use the *command line*. It is currently simply not possible to use only the notebook for Sage development... yet!
- You have to know *basic UNIX* commands: ls, cp, cd, mv, etc.
- You have to have some understanding of *the Sage Python library* and our coding conventions and requirements.
- You have to submit *patches* to the trac webpage, which requires using the Mercurial distributed revision control system. Thus you must become familiar with the basic use of a distributed revision control system. This is good for you anyways.
- All patches go through a *peer review* process, just like a formally published paper. Somebody has to referee your work, signing off on it publicly, before your work can go into Sage. Beyond testing whether the code works and is stylish, this process also includes asking whether it makes sense to include your code in Sage at all; we usually do not want third-rate code.

Fortunately, the process is well documented (see `http://sagemath.org/doc/developer/`), there are thousands of examples of tickets along with the complete review process at `http://trac.sagemath.org`, and there are numerous Sage Days workshops that help people get up to speed. Around five hundred people have successfully got code into Sage, and you can too if you are serious.

5.1.3 Walkthrough

We will do a careful slow step-by-step live demo that illustrates some of Sage development.

```
my_laptop ssh math480@sage.math.washington.edu

math480@sage:~$ cd scratch
math480@sage:~/scratch$ ls
sage-4.6.2-sage.math.washington.edu-x86_64-Linux.tar.gz
math480@sage:~/scratch$ mkdir wstein
math480@sage:~/scratch$ cd wstein/
math480@sage:~/scratch/wstein$ ls
```

[1]For example, GAP `http://www.gap-system.org/`.

```
math480@sage:~/scratch/wstein$ tar xf ../sage-4.6.2-sage.math.washington.edu-x86_64-Linu
[[Wait about 1 minute.]]
math480@sage:~/scratch/wstein$ mv sage-4.6.2-sage.math.washington.edu-x86_64-Linux sage
math480@sage:~/scratch/wstein$ cd sage/
math480@sage:~/scratch/wstein/sage$ ls
COPYING.txt  devel     ipython  Makefile    sage                  spkg
data         examples  local    README.txt  sage-README-osx.txt   VERSION.txt
math480@sage:~/scratch/wstein/sage$ here  # sets up path
math480@sage:~/scratch/wstein/sage$ sage
----------------------------------------------------------------------
| Sage Version 4.6.2, Release Date: 2011-02-25
|
| Type notebook() for the GUI, and license() for information.
|
----------------------------------------------------------------------
The Sage install tree may have moved
...
Done resetting paths
sage: 2 + 3
5
```

Now make some change (using vim, emacs, pico, etc.), do "sage -br" to make change take effect. Then make a patch and export it.

5.2 How to modify the Sage library and create a patch

1. Use an ssh client to connect to `math480@sage.math.washington.edu`. I told you the password during class. On Windows, putty is a good client. On OS X or Linux, just open a terminal and type: `ssh math480@sage.math.washington.edu`

2. The minimum commands you'll need to know to use the command prompt for this assignment are. So look these over:

   - `pwd` = print working directory = "where am I"?
   - `ls` = list files in working directory
   - `cd ..` = move up one directory
   - `cd directory_name` = move into `directory_name`
   - `tar xf /home/math480/scratch/sage-4.8-sage.math.washington.edu-x86_64-Linux.tar.gz` = extract your own copy of Sage into the current directory
   - `mv dir1 dir2` = rename dir1 to dir2 (or file1 to file2)
   - `mkdir directory_name` = make a directory
   - `pico filename` = edit the given file
   - `hg commit` = commit your changes to your local repository
   - `hg export tip > my.patch` = export your changes to the file my.patch
   - `hg rollback` = use this only if you want to undo the commit.
   - `/home/math480/scratch/your_name/sage/sage -br` = build and run modified sage

- You can browse your files at
  `http://sage.math.washington.edu/home/math480/scratch/`

Now we'll proceed step-by-step to use the above commands to make your patch.

3. Make your own directory:

```
cd scratch
mkdir your_name
cd your_name
```

4. Extract sage into your directory:

```
tar xf /home/math480/scratch/sage-4.8-sage.math.washington.edu-x86_64-Linux.tar.gz
```

This will extract over 50,000 files and takes between 1 minute and 15 minutes – be patient.

5. Rename your sage install:

```
mv sage-4.8-sage.math.washington.edu-x86_64-Linux sage
```

6. Find some file(s) to edit:

```
cd sage/devel/sage/sage/
ls
cd some_subdirectory
pico some_file
```

7. Test out your changes:

```
/home/math480/scratch/your_name/sage/sage -br
```

The first time you do this it will take about a minute. Afterwards it should only take a second or two.

8. Commit your changes (enter a 1-line description into the editor that pops up):

```
hg commit
```

9. Oops, change your mind?!

```
hg rollback
```

Now make more changes and go to step 7. Make sure to do `hg commit` before going to step 10.

10. Export your changes

```
hg export tip > my.patch
```

11. Find the file `my.patch` by browsing to
    `http://sage.math.washington.edu/home/math480/scratch/`

12. Download `my.patch` and attach it to your email to me.

Chapter 6

How Sage Uses Other Math Software Systems

```
<p>The goal of this lecture is to give you a deeper understanding of some of the fundamental an
<p>I built Sage partly from other complete mathematical software systems because I wanted to fi
<h3 style="text-align: center;">"Building the car instead of reinventing the wheel."</h3>
<p>Some of the major components included in Sage are:</p>
<ul>
<li>PARI/GP - number theory</li>
<li>GAP - group theory</li>
<li>Singular - commutative algebra</li>
<li>Maxima - symbolic calculus</li>
<li>R - statistics</li>
</ul>
<p>Each of the above is a full standalone project with its own custom programming language, his
<p>I also wanted to make it easy to call the following systems from Sage for the purposes of be
<ul>
<li>Magma</li>
<li>Maple</li>
<li>Mathematica</li>
<li>Mupad</li>
<li>Matlab</li>
<li>Axiom, Octave, REDUCE, Macaulay2, Scilab, Kash, Lisp</li>
</ul>
<p> </p>
<p><strong>The Big Problem:</strong> How can we make use of the above systems from Python?</p>
<p>This question is difficult partly because there are so many answers, each with pros and cons
<p> </p>
<p> </p>
sage: number_of_partitions(10)
42
sage: list(Partitions(10))
[[10], [9, 1], [8, 2], [8, 1, 1], [7, 3], [7, 2, 1], [7, 1, 1, 1], [6, 4], [6, 3, 1], [6, 2, 2]
sage: time number_of_partitions(10^7)
92027175502604546685596278166825605430729405281023979395328576351741298526232350197882291654710
Time: CPU 0.32 s, Wall: 0.32 s
```

```
<p><strong>Problem 1:</strong> Availability of a specific known version of a third party softwa
<p>Even if we solve the big problem above, a "vendor" will often just release a new version of
<p><strong>Solution: </strong>For the free open systems that (1) we really need, and (2) we can
<p>For the non-free systems or the free systems that are hard to build, the problem just doesn’
<p> </p>
<p> </p>
<p><strong>Problem 2</strong>: Make a specific version of some mathematics software (call it M)
<p>Here are some of the many potential approaches to this problem:</p>
<ol>
<li><strong>Naive Subprocess. </strong>Start up M, tell it to read in a file, and save the resu
<li><strong>Create network protocols.</strong>  Define an openmath/XML based protocol for
<li><strong>Pseudo-tty’s (ptty) = pexpect. </strong>  Create a simulated command line prom
<li><strong>C/C++ library interfaces.</strong>  Create a C/C++ library interface and link
</ol>
<p>As of now, people have written fairly polished versions of both (3) and (4) for all of: PARI
<p>The rest of this worksheet is about how to use (3) above: the pexpect based interfaces. &nbs
<p>Here are the basic points, which we’ll follow with several examples illustrating them.  
<ul>
<li><strong>x = m.eval(s): </strong>sets x equal to the string obtained by typing the string s
<li><strong>x = m(s): </strong>creates a new Python object that "wraps" the result of evaluatin
</ul>
<p>And that is pretty much it.  </p>
<p><strong>WARNING:</strong> There is latency.  <strong>Any</strong> time you call any fun
sage: %lisp
sage: (* 5 7)
35
sage: lisp.eval('(* 5 7)')
'35'
sage: %maxima
sage: a:5
sage: b:7
sage: a*b
5
7
35
<h2>Examples</h2>
<p>Another note: the very first time you do m.eval(...) it may take surprisingly long, since an

<p>We use Maxima to illustrate evaluation of a simple string:</p>
sage: s = maxima.eval("2 + 3")
sage: type(s)
<type 'str'>
sage: s
'5'
sage: maxima.eval("""
...      a : 2;
...      b : 3;
...      c : a +b;
...
sage: """)
```

```
sage: maxima.eval('c')
'5'
sage: timeit('maxima.eval("2+2")')
625 loops, best of 3: 1.2 ms per loop
sage: a = maxima('2')
sage: timeit('a + a')
625 loops, best of 3: 1.37 ms per loop
sage: timeit('2+2')
625 loops, best of 3: 331 ns per loop
<p>There is now a separate Maxima subprocess running.  Each process has an id number assoc
sage: maxima.pid()   # the "pin id" of the subprocess
9259
<p>Next will illustrate creating a Python object that wraps an expression in Maxima.</p>
sage: s = maxima('sin(x^3) * tan(y)')
sage: type(s)
<class 'sage.interfaces.maxima.MaximaElement'>
sage: float(1.31*10^(-3) / (330*10^(-9)))
3969.69696969697
<p>The name of the object in the corresponding Maxima session:</p>
sage: s.name()
'sage2656'
<p>The object prints nicely:</p>
sage: s
sin(x^3)*tan(y)
<p>Latex output happens to be supported:</p>
sage: show(s)
<html><div class="math">\newcommand{\Bold}[1]{\mathbf{#1}}\sin x^3\,\tan y</div></html>
sage: maxima.eval('sage2656 + 1')
'sin(x^3)*tan(y)+1'
<p>You can call functions on objects in a Pythonic way.</p>
sage: s.integrate('y')
sin(x^3)*log(sec(y))
<p>Or use maxima.function(...)</p>
sage: maxima.integrate(s, 'y')
sin(x^3)*log(sec(y))
<p>The result is another Python object (which wraps another object defined in Maxima).  We
sage: z = s.integrate('y')
sage: type(z)
<class 'sage.interfaces.maxima.MaximaElement'>
sage: z
sin(x^3)*log(sec(y))
sage: z.name()
'sage2662'
sage: z.diff('y')
sin(x^3)*tan(y)
sage: z + z
2*sin(x^3)*log(sec(y))
<p><strong>Conclusion:</strong> If you understand the above, you are in extremely good shape. &
sage: z.jksadhflksd()
jksadhflksd(sin(x^3)*log(sec(y)))
```

```
sage: z_sage = z.sage(); z_sage
log(sec(y))*sin(x^3)
sage: type(z_sage)
<type 'sage.symbolic.expression.Expression'>
sage: maxima(z_sage)
sin(x^3)*log(sec(y))
<p>It is possible in some systems to seriously mess things up and get things "out of sync". &nb
<p>Here is an example with each of the five big systems included in Sage:</p>
sage: maxima('2+3')  # maxima
5
sage: gp('2+3')  # pari/gp
5
sage: singular('2+3')
5
sage: gap('2+3')
5
sage: r('2 + 3')
[1] 5
sage: z_sage._maxima_init_()
'(log(sec(y)))*(sin((x)^(3)))'
<p>You can follow standard R tutorials and have the computations (except graphics at present) t
sage: x = r('c(1,3,2,10,5)');  y = r('1:5')
sage: print x
sage: print y
[1]  1  3  2 10  5
[1] 1 2 3 4 5
sage: x + y
[1]  2  5  5 14 10
sage: x/y
[1] 1.0000000 1.5000000 0.6666667 2.5000000 1.0000000
sage: x.length()
[1] 5
sage: x > 3
[1] FALSE FALSE FALSE  TRUE  TRUE
sage: x[x > 3]
[1] 10  5
<p>There is also an interface to Octave, which is very similar to Matlab (but free).</p>
sage: A = octave('rand(3)'); A
 0.401446 0.286955 0.396858
 0.606625 0.371021 0.515619
 0.96863 0.683554 0.837288
sage: A*A
 0.719642 0.492938 0.639562
 0.968042 0.664185 0.863772
 1.61454 1.1039 1.43791
sage: A.rref()
 1 0 0
 0 1 0
 0 0 1
<p><strong>Bonus:</strong> There is even a pexpect interface to Sage itself.   (Trivia: th
```

```
sage: sage0('2 + 3')
5
sage: A = sage0('matrix(QQ, 3, [1..9])'); A
[1 2 3]
[4 5 6]
[7 8 9]
sage: type(A)
<class 'sage.interfaces.sage0.SageElement'>
sage: A.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
<p>Let's get crazy: a pexpect interface inside a pexpect interface.  And of course, this c
sage: sage0.eval('sage0 = Sage()')
sage: z = sage0('sage0("3+5")')
sage: type(z)
<class 'sage.interfaces.sage0.SageElement'>
sage: z
8
sage: sage0.type(z)
<class 'sage.interfaces.sage0.SageElement'>
```

Part II

Using Sage

Chapter 7

Graphics

7.1 2d Plots

```
<p>Sage has plotting support that covers:</p>
<ul>
<li>most 2d plotting that Mathematica has (with a similar interface)</li>
<li>3d plotting, somewhat like Mathematica</li>
<li>most 2d plotting that Matlab has (with a similar interface)</li>
</ul>
<p>Sage uses the Python library Matplotlib (<a href="http://matplotlib.sourceforge.net/" target
<p>In this worksheet, we'll explain how to use the "mathematica-style" 2d plotting capabilities
<h1>Drawing Lines</h1>
<p>First, we'll discuss a simple but very powerful plotting command in Sage called "line". &nbs
sage: L = line([(-2,-2), (3,8), (5, 5)])
sage: print L
Graphics object consisting of 1 graphics primitive
<p>To <em><strong>see</strong></em> the actual plot of L, just put L by itself on a line or typ
sage: L
<html><font color='black'><img src='cell://sage0.png'></font></html>
sage: show(L)
<html><font color='black'><img src='cell://sage0.png'></font></html>
sage: L.show()
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p>Incidentally, there are many, many options that you can pass to the show command.   The
<ul>
<li>frame=True:   Make it so the x-y axis are replaced by a frame, which is much better wh
<li>gridlines=True: Adds a background grid, which makes it easier to understand the plot in som
<li>figsize=[w,h]:  Allows you to adjust the size of the output.  Think of w and h as
</ul>
<p>You can combine these options.  For example:</p>
sage: L.show(frame=True, gridlines=True, figsize=[8,2])
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p> In the notebook you can just click and download the default plots displayed above, sin
sage: L.save('image.pdf')
sage: L.save('image.eps')
sage: L.save('image.svg')
```

```
<p>Lines (and all other graphics objects) have numerous properties that you can adjust, which y
<ul>
<li>color=...: where for the color you can give a string, e.g., 'red'; or an html color, e.g.,
<li>thickness=4:  the thickness of the line</li>
<li>linestyle='--':   the style of the line: '--', '-.', '-', ':'</li>
</ul>
sage: line([(-2,-2), (3,8), (5, 5)], color='purple', thickness=3, linestyle='--')
<html><font color='black'><img src='cell://sage0.png'></font></html>
sage: line([(-2,-2), (3,8), (5, 5)], color='#042a99', thickness=1.5, linestyle=':')
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p>Let's have some fun:</p>
sage: line([(random(), random()) for _ in range(100)], color='purple')
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p>Arithmetic: a key unusual idea in Sage graphics is that you combine together different graph
sage: L1 = line([(0,0), (1,1), (2,0)], color='green', thickness=7)
sage: L2 = line([(1,0), (2,5), (3,0)], color='purple', thickness=10, alpha=.7)  # alpha = trans
sage: L1 + L2
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p>There are numerous other important plotting commands in Sage, including point, circle, polyg
sage: G = point((1,1), pointsize=200) + circle((1,1), .5)
sage: # zorder makes sure that triangle is on top
sage: G += polygon([(0,0), (1,.6), (2,0)], color='purple', zorder=5)
sage: G += arrow((1,1), (2,1.2), color='green')
sage: # You can use TeX formulas:
sage: G += text(r"$\sqrt{\sin(\pi x^2)}$",(1.8,1.35),color='black',fontsize=20)
sage: G.show(aspect_ratio = 1)
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p>There are also a function just called "plot" that makes a plot of a wide range of Sage objec
sage: plot(x*sin(1/x), (x, -1, 5), color='green', thickness=2)
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p>matrix_plot is another similar plotting function, which allows you to visualize a matrix.</p
sage: A = random_matrix(RDF,100);
sage: matrix_plot(A)
<html><font color='black'><img src='cell://sage0.png'></font></html>
sage: matrix_plot(A^2)
<html><font color='black'><img src='cell://sage0.png'></font></html>
<p>Finally, there is a graphics_array function that lets you assemble several independent plots
sage: graphics_array([[matrix_plot(A), matrix_plot(A^2)], [plot(sin), plot(cos,color='red')]])
<p>Bonus -- you can animate graphics.  Given any list of graphics objects, the animate com
sage: v = [plot(sin(a*x), (x,0,10)) for a in [0,0.2,..,pi]]
sage: z = animate(v, xmin=0,xmax=10,ymin=-1,ymax=1)
sage: z.show(delay=10)
```

7.2 3d Plots

```
<h1>Sage 3d Graphics</h1>
<p>In Sage, just as with 2d graphics, you make 3d graphics by creating various primitives and c
<p>There are many 3d graphics primitives in Sage.   For example, you can draw platonic sol
<p>All 3d graphics objects have <strong>translate</strong> and <strong>rotate</strong> (and <st
<p>Also, you can set the color and opacity of any 3d object when you create it, as an optional
```

```
<p>Finally, you can display a 3d scene G using either  jmol (java) via <strong>G.show()</s
<p>There are also some rudimentary 3d plotting capabilities in matplotlib.  I had once ann
<p><strong>History:</strong> William Stein included Tachyon in Sage, then Tom Boothby, Josh Kan
<p><strong>Note: </strong>The 3d plotting in Sage is mainly oriented toward mathematical  visua
<p><strong>Shortcoming:</strong> The biggest shortcomings are that (1) realtime interaction wit
<p>The rest of this worksheet illustrates with examples how to create 3d images using Sage.</p>
<h2>Platonic Solids</h2>
<p><strong>Problem</strong>: Draw all of the platonic solids next to each other in different co
sage: G = tetrahedron(color='red')
sage: G += cube((2,0,0), color='green')
sage: G += octahedron((4,0,0), color='purple')
sage: G += dodecahedron((6,0,0), color='orange')
sage: G += icosahedron((8,0,0), color='brown')
sage: G.show(frame=False, aspect_ratio=1, zoom=1.3)
sage: G.show(viewer='tachyon', frame=False, aspect_ratio=1, zoom=1.5)
sage: G.show(viewer='canvas3d', frame=False, aspect_ratio=1, zoom=1.5)
<h2>Points and Spheres</h2>
<p><strong>Problem: </strong>Plot 40 semi-transparent random spheres.   Similarly, pl
sage: G = sum( sphere((random(), random(), random()), color=hue(random()),
...         size=.1*random(), opacity=.5) for _ in range(40))
...
sage: G.show(spin=True, frame=False)
sage: G = sum( point3d((random(), random(), random()), color=hue(random())) for _ in range(1000
sage: G.show(spin=True, frame=False)
<h2>1d Curves Through Space</h2>
<p>Draw a 3d random walk.</p>
sage: v = [(0,0,0)]
sage: for i in range(300):
...       v.append([a+random()-.5 for a in v[-1]])
...
sage: line3d(v, color='black')
sage: line3d(v, color='red', thickness=3)
<h2>3D Text</h2>
<p><strong>Problem:</strong> Draw some text in 3d.</p>
sage: G = sum([text3d('%.1f'%n, (cos(n),sin(n),n), color=hue(n/8)) for n in [0,0.3,..,12]])
sage: G.show(spin=True)
<h2>Plotting Functions</h2>
<p><strong>Problem</strong>: Plot a function $z=f(x,y)$.</p>
sage: var('x,y')
sage: B=1.5
sage: plot3d( sin(pi*(x^2+y^2))/2,(x,-B,B),(y,-B,B), plot_points=100, color='gree' )
<p><strong>Problem: </strong>Plot an implicit 3d surface defined by an equation $f(x,y,z)=0$.</
sage: T = RDF(golden_ratio)
sage: p(x,y,z) = 2 - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z) + cos(y - T*z) + cos(z - T*x)
sage: r = 4.77
sage: implicit_plot3d(p, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=40)
sage: implicit_plot3d(p==1, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=40, color='green')
<h2>Models</h2>
<p><strong>Problem:</strong> Plot Yoda.</p>
<p><strong>Solution:</strong> use a standard mesh one finds online as follows, which describes
```

```
sage: # Yoda! (over 50,000 triangles)
sage: from scipy import io
sage: x = io.loadmat(DATA + 'yodapose.mat')
sage: from sage.plot.plot3d.index_face_set import IndexFaceSet
sage: V = x['V']; F3=x['F3']-1; F4=x['F4']-1
sage: Y = IndexFaceSet(F3,V,color=Color('#444444')) + IndexFaceSet(F4,V,color=Color('#007700'))
sage: Y = Y.rotateX(-1)
sage: Y.show(aspect_ratio=1, frame=False, zoom=1.2)
```

7.3 Matplotlib

```
<p>Though Sage provides its own functions (e.g,. plot, line, point, text, circle, etc.) for dra
<p>Also, if you're drawing an image that involves a huge amount of data points, directly using
<h2>Important Caveat</h2>
<p>There are<strong> two absolutely critical</strong> things to remember when using matplotlib
<ol>
<li>Instead of <strong>plt.show()</strong> use <strong>plt.savefig('a.png').  Memorize thi
<li>You might have to put your input in a <strong>%python</strong> cell or turn off the prepars
</ol>
<p>With these two hints, you should be able to to try out the examples at <a href="http://matpl
<p>In fact, try it now [in class, go to the above website, scroll, and let students choose an e
<ol>
<li>Click on the thumbnail image.</li>
<li>Click source code in the upper left</li>
<li>Paste the code into a notebook cell.</li>
<li>Put <strong>%python</strong> as the first line of the cell.</li>
<li>Change any .show() to .savefig('a.png')</li>
</ol>
<p>Note: There are some images in the gallery that require some external data file (e.g, the br
<p>For example, if students choose <a href="http://matplotlib.sourceforge.net/examples/api/arti
sage: %hide
sage: %python
sage: """
sage: Show examples of matplotlib artists
sage: http://matplotlib.sourceforge.net/api/artist_api.html
sage: Several examples of standard matplotlib graphics primitives (artists)
sage: are drawn using matplotlib API. Full list of artists and the
sage: documentation is available at
sage: http://matplotlib.sourceforge.net/api/artist_api.html
sage: Copyright (c) 2010, Bartosz Telenczuk
sage: License: This work is licensed under the BSD. A copy should be
sage: included with this source code, and is also available at
sage: http://www.opensource.org/licenses/bsd-license.php
sage: """
sage: import numpy as np
sage: import matplotlib.pyplot as plt
sage: import matplotlib
sage: from matplotlib.collections import PatchCollection
sage: import matplotlib.path as mpath
sage: import matplotlib.patches as mpatches
```

```
sage: import matplotlib.lines as mlines
sage: font = "sans-serif"
sage: fig = plt.figure(figsize=(5,5))
sage: ax = plt.axes([0,0,1,1])
sage: # create 3x3 grid to plot the artists
sage: pos = np.mgrid[0.2:0.8:3j, 0.2:0.8:3j].reshape(2, -1)
sage: patches = []
sage: # add a circle
sage: art = mpatches.Circle(pos[:,0], 0.1,ec="none")
sage: patches.append(art)
sage: plt.text(pos[0,0], pos[1,0]-0.15, "Circle", ha="center",
...           family=font, size=14)
...
sage: # add a rectangle
sage: art = mpatches.Rectangle(pos[:,1] - np.array([0.025, 0.05]), 0.05, 0.1,
...           ec="none")
...
sage: patches.append(art)
sage: plt.text(pos[0,1], pos[1,1]-0.15, "Rectangle", ha="center",
...           family=font, size=14)
...
sage: # add a wedge
sage: wedge = mpatches.Wedge(pos[:,2], 0.1, 30, 270, ec="none")
sage: patches.append(wedge)
sage: plt.text(pos[0,2], pos[1,2]-0.15, "Wedge", ha="center",
...           family=font, size=14)
...
sage: # add a Polygon
sage: polygon = mpatches.RegularPolygon(pos[:,3], 5, 0.1)
sage: patches.append(polygon)
sage: plt.text(pos[0,3], pos[1,3]-0.15, "Polygon", ha="center",
...           family=font, size=14)
...
sage: #add an ellipse
sage: ellipse = mpatches.Ellipse(pos[:,4], 0.2, 0.1)
sage: patches.append(ellipse)
sage: plt.text(pos[0,4], pos[1,4]-0.15, "Ellipse", ha="center",
...           family=font, size=14)
...
sage: #add an arrow
sage: arrow = mpatches.Arrow(pos[0,5]-0.05, pos[1,5]-0.05, 0.1, 0.1, width=0.1)
sage: patches.append(arrow)
sage: plt.text(pos[0,5], pos[1,5]-0.15, "Arrow", ha="center",
...           family=font, size=14)
...
sage: # add a path patch
sage: Path = mpath.Path
sage: verts = np.array([
...        (0.158, -0.257),
...        (0.035, -0.11),
```

```
...        (-0.175, 0.20),
...        (0.0375, 0.20),
...        (0.085, 0.115),
...        (0.22, 0.32),
...        (0.3, 0.005),
...        (0.20, -0.05),
...        (0.158, -0.257),
...       ])
...
sage: verts = verts-verts.mean(0)
sage: codes = [Path.MOVETO,
...             Path.CURVE4, Path.CURVE4, Path.CURVE4, Path.LINETO,
...             Path.CURVE4, Path.CURVE4, Path.CURVE4, Path.CLOSEPOLY]
...
sage: path = mpath.Path(verts/2.5+pos[:,6], codes)
sage: patch = mpatches.PathPatch(path)
sage: patches.append(patch)
sage: plt.text(pos[0,6], pos[1,6]-0.15, "PathPatch", ha="center",
...            family=font, size=14)
...
sage: # add a fancy box
sage: fancybox = mpatches.FancyBboxPatch(
...            pos[:,7]-np.array([0.025, 0.05]), 0.05, 0.1,
...            boxstyle=mpatches.BoxStyle("Round", pad=0.02))
...
sage: patches.append(fancybox)
sage: plt.text(pos[0,7], pos[1,7]-0.15, "FancyBoxPatch", ha="center",
...            family=font, size=14)
...
sage: # add a line
sage: x,y = np.array([[-0.06, 0.0, 0.1], [0.05,-0.05, 0.05]])
sage: line = mlines.Line2D(x+pos[0,8], y+pos[1,8], lw=5.,
...            alpha=0.4)
...
sage: plt.text(pos[0,8], pos[1,8]-0.15, "Line2D", ha="center",
...            family=font, size=14)
...
sage: colors = 100*np.random.rand(len(patches))
sage: collection = PatchCollection(patches, cmap=matplotlib.cm.jet, alpha=0.4)
sage: collection.set_array(np.array(colors))
sage: ax.add_collection(collection)
sage: ax.add_line(line)
sage: ax.set_xticks([])
sage: ax.set_yticks([])
sage: plt.savefig('a.png')
<h2>Pyplot</h2>
<p>Matplotlib has an interface that works much like Matlab.  This will be very helpful if
<p>Below we replicate several examples from this tutorial in Sage, and you should read this tut
sage: import matplotlib.pyplot as plt
sage: plt.clf()
```

```
sage: plt.plot([1,2,3,4])
sage: plt.ylabel('some numbers')
sage: plt.savefig('a.png', dpi=70)
sage: plt.clf()
sage: plt.plot([1,2,3,4], [1,4,9,16])
sage: plt.savefig('a.png', dpi=70)
sage: plt.clf()
sage: # 'ro' = red circles, like in MATLAB;  'bx' = blue crosses.
sage: plt.plot([1,2,3,4], [1,4,9,16], 'ro',  [5,5.5], [2,2], 'bx')
sage: plt.axis([0, 6, 0, 20])
sage: plt.savefig('a.png', dpi=70)
<h3>Use Numpy instead of Python lists:</h3>
sage: import numpy as np
sage: plt.clf()
sage: # evenly sampled time at 200ms intervals
sage: t = np.arange(0., 5., 0.2)
sage: # red dashes, blue squares and green triangles
sage: plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
sage: plt.savefig('a.png', dpi=70)
<p>Multiple figures and axis all at once:</p>
sage: import numpy as np
sage: import matplotlib.pyplot as plt
sage: def f(t):
...       return np.exp(-t) * np.cos(2*np.pi*t)
...
sage: t1 = np.arange(0.0, 5.0, 0.1)
sage: t2 = np.arange(0.0, 5.0, 0.02)
sage: plt.clf()
sage: plt.figure(1)
sage: plt.subplot(121)
sage: plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
sage: plt.subplot(122)
sage: plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
sage: plt.savefig('a.png')
<p>An example involving text</p>
sage: import numpy as np
sage: import matplotlib.pyplot as plt
sage: plt.clf()
sage: mu, sigma = 100, 15
sage: x = mu + sigma * np.random.randn(10000)
sage: # the histogram of the data
sage: n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)
sage: plt.xlabel('Smarts')
sage: plt.ylabel('Probability')  # bug -- gets chopped out below :-(
sage: plt.title('Histogram of IQ')
sage: plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
sage: plt.axis([40, 160, 0, 0.03])
sage: plt.grid(True)
sage: plt.savefig('a.png', dpi=70)
<p>Incidentally, you can of course combine matplotlib graphics with @interact</p>
```

```
sage: import numpy as np
sage: import matplotlib.pyplot as plt
sage: plt.clf()
sage: mu, sigma = 100, 15
sage: x = mu + sigma * np.random.randn(10000)
sage: @interact
sage: def f(bins=(5..150)):
...       plt.clf()
...       n, bins, patches = plt.hist(x, bins, normed=1, facecolor='g', alpha=0.75)
...       plt.xlabel('Smarts', fontsize=18, color='red')
...       plt.ylabel('Probability')     # bug -- gets chopped out below
...       plt.title('Histogram of IQ')
...       plt.text(60, .025, r'$\mu=100,\ \sigma=15$')  # latex!
...       plt.axis([40, 160, 0, 0.03])
...       plt.grid(True)
...       plt.savefig('a.png', dpi=70)
<p>Annotation Example</p>
sage: import numpy as np
sage: import matplotlib.pyplot as plt
sage: plt.clf()
sage: ax = plt.subplot(111)
sage: t = np.arange(0.0, 5.0, 0.01)
sage: s = np.cos(2*np.pi*t)
sage: line, = plt.plot(t, s, lw=2)
sage: plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
...               arrowprops=dict(facecolor='black', shrink=0.07))
...
sage: plt.ylim(-2,2)
sage: plt.savefig('a.png')
<p>There are tons of other examples of pyplot at the matplotlib website here: <a href="http://m
<p>For example we have the following economics example:</p>
sage: """
sage: make a scatter plot with varying color and size arguments
sage: """
sage: import matplotlib
sage: import numpy as np
sage: import matplotlib.pyplot as plt
sage: import matplotlib.mlab as mlab
sage: import matplotlib.cbook as cbook
sage: # load a numpy record array from yahoo csv data with fields date,
sage: # open, close, volume, adj_close from the mpl-data/example directory.
sage: # The record array stores python datetime.date as an object array in
sage: # the date column
sage: datafile = cbook.get_sample_data('goog.npy')
sage: r = np.load(datafile).view(np.recarray)
sage: r = r[-250:] # get the most recent 250 trading days
sage: delta1 = np.diff(r.adj_close)/r.adj_close[:-1]
sage: # size in points ^2
sage: volume = (15*r.volume[:-2]/r.volume[0])**2
sage: close = 0.003*r.close[:-2]/0.003*r.open[:-2]
```

```
sage: fig = plt.figure()
sage: ax = fig.add_subplot(111)
sage: ax.scatter(delta1[:-1], delta1[1:], c=close, s=volume, alpha=0.75)
sage: #ticks = arange(-0.06, 0.061, 0.02)
sage: #xticks(ticks)
sage: #yticks(ticks)
sage: ax.set_xlabel(r'$\Delta_i$', fontsize=20)
sage: ax.set_ylabel(r'$\Delta_{i+1}$', fontsize=20)
sage: ax.set_title('Volume and percent change')
sage: ax.grid(True)
sage: plt.savefig('a.png')
<h2>There is more to matplotlib than just pyplot...</h2>
<p>There is more to matplotlib than just a Matlab like interface.    Matplotlib has i
<h2>A 3d Example</h2>
<p>This is basically this example: <a href="http://matplotlib.sourceforge.net/examples/mplot3d/
sage: %python
sage: from mpl_toolkits.mplot3d import Axes3D
sage: from matplotlib import cm
sage: from matplotlib.ticker import LinearLocator, FixedLocator, FormatStrFormatter
sage: import matplotlib.pyplot as plt
sage: import numpy as np
sage: fig = plt.figure()
sage: ax = fig.gca(projection='3d')
sage: X = np.arange(-7, 7, 0.25)
sage: Y = np.arange(-7, 7, 0.25)
sage: X, Y = np.meshgrid(X, Y)
sage: R = np.sqrt(X**2 + Y**2)
sage: Z = np.sin(R)
sage: surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.jet,
...           linewidth=0, antialiased=False)
...
sage: ax.set_zlim3d(-1.01, 1.01)
sage: ax.w_zaxis.set_major_locator(LinearLocator(10))
sage: ax.w_zaxis.set_major_formatter(FormatStrFormatter('%.03f'))
sage: fig.colorbar(surf, shrink=0.5, aspect=5)
sage: plt.savefig('a.png')
```

Chapter 8

Number Theory

8.1 Prime Numbers and the Riemann Hypothesis

8.1.1 Primes

An integer $p \geq 2$ is *prime* if its only divisors are 1 and p. For example, the first few primes are

$$2, 3, 5, 7, 11, 13, 17, 19, \ldots.$$

You can find primes in Sage using the `prime_range` command:

```
sage: prime_range(10)
[2, 3, 5, 7]
sage: prime_range(7, 23)
[7, 11, 13, 17, 19]
sage: range(7, 23)
[7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
sage: prime_range(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
 61, 67, 71, 73, 79, 83, 89, 97]
```

Note that `prime_range` works like the range command, in that it doesn't include its upper endpoint. There is also an iterator over the prime numbers called `primes`:

```
sage: P = primes(10^100); P
<generator object primes at 0x5a49280>
sage: for p in P:
...       print p
...       if p > 10: break
2
3
5
7
11
```

Though memory efficient, the `primes` iterator can be much slower than `prime_range` since it uses a different algorithm and caching strategy.

```
sage: v = list(primes(10^6))       # this will take much longer
```

```
sage: v = prime_range(10^6)         # than this takes
```

Mankind has been fascinated with prime number for thousands of years.

Theorem 8.1.1 (Euclid)**.** *There are infinitely many prime numbers.*

Proof. This is easier to prove than you might guess. We will describe an algorithm that takes as input a finite list $p_1, \ldots, p_k$ of primes, and outputs a prime not in this list. The existence of this algorithm implies that there *must* be infinitely many primes. The algorithm works as follows. First, let $n = p_1 p_2 \cdots p_k + 1$. It is easy to see by induction that every integer ≥ 2 is divisible by some prime; in particular, n is divisible by some prime q (for concreteness, take q to be the smallest prime divisor of n). But n is not divisible by any p_i, since if you divide n by p_i the remainder is 1. Thus $q \neq p_i$ for any i, so q is the new prime output by our algorithm. $\square$

The number $p = 2^{43112609} - 1$ is a prime number with 12978189 digits.

```
sage: p = 2^43112609 - 1
sage: k = p.digits(10)  # long time: about 20 seconds
sage: len(k)
12978189
```

As of May 2011, it is the largest *explicitly* known prime number. The people (the GIMPS project) who found the prime p above won a \$100,000 prize from the Electronic Frontier Foundation (EFF) for finding this prime (the first known prime with more than 10 million digits), and the EFF offers \$150,000 to anybody who can explicitly exhibit a prime with at least 100 million digits.

In Sage we can test whether or not a number is prime using the `is_prime` function. There is also a function `is_pseudoprime`, which is potentially much, much, much faster, but in theory could claim a number to be composite even though it is prime (there are no known examples of this, but it surely happens).

```
sage: is_prime(2011)
True
sage: is_prime(2009)
False
sage: is_pseudoprime(2009)
False
```

The commands `next_prime` and `next_probable_prime` find the next prime (or pseudoprime) after a number.

```
sage: n = next_probable_prime(10^300)
sage: is_pseudoprime(n)   # takes about 0.01 seconds
True
sage: is_prime(n)   # long time -- about 10 seconds!
True
```

8.1.2 Factorization

Theorem 8.1.2 (Euclid)**.** *"The Fundamental Theorem of Arithmetic" Every positive integer factors uniquely as a product of primes* $p_1^{e_1} \cdots p_r^{e_r}$*.*

This is much harder to prove than you might at first guess, since there are other rings, which are very similar to the ring of integers, but where this statement fails. For example, consider the ring $R = \mathbb{Z}[\sqrt{-5}]$. Here we have

$$6 = 2 \cdot 3 = (1 + \sqrt{-5})(1 - \sqrt{-5}),$$

which exhibits two different factorizations of 6 into elements that cannot be factored further. For a proof of Theorem 8.1.2, see Chapter 1 of the book *Elementary Number Theory: Primes, Congruences, and Secrets*, which is freely available at `http://wstein.org/ent/`.

It is not known whether or not there is a fast ("polynomial time") algorithm to factor integers, though many people suspect that there is no such algorithm on a classical computer. (There is a quantum algorithm to factor quickly using quantum computers; unfortunately, it may not be possible to build a sufficiently powerful quantum computer.)

In Sage, use the command `factor` to factor an integer.

```
sage: factor(2012)
2^2 * 503
sage: factor(10^50 + 4)
2^2 * 13 * 89 * 216076058772687986171132238547968885047536732 93
```

The output of the factor command looks like a factorization, but you can work with it as if it were a list of prime-exponent pairs (p, e).

```
sage: v = factor(10^50 + 4); v
2^2 * 13 * 89 * 21607605877268798617113223854796888504753673293
sage: v[0]
(2, 2)
sage: v[1]
(13, 1)
sage: len(v)
4
sage: list(v)
[(2, 2), (13, 1), (89, 1),
 (21607605877268798617113223854796888504753673293, 1)]
```

The `factor` command also has a `verbose=` option, which if set to 4 or 8 produces a huge amount of output about the factoring algorithms that are being used. (I do not know a good place to read about the format of the verbose= output, except for reading the source code of PARI, which implements the underlying factorization algorithm in Sage, at present.)

```
sage: factor(10^50 + 4, verbose=8)
OddPwrs: is 227694421180235176194517066805197 3
        ...a 3rd, 5th, or 7th power?
        modulo: resid. (remaining possibilities)
           211:   88   (3rd 1, 5th 1, 7th 0)
           209:   34   (3rd 0, 5th 1, 7th 0)
            61:   30   (3rd 0, 5th 0, 7th 0)
OddPwrs: examining 2276944211802351761945170668051973
  ***   Warning: IFAC: untested integer declared prime.
        2276944211802351761945170668051973
Starting APRCL: Choosing t = 840
```

```
Solving the triangular system
Solving the triangular system

Jacobi sums and tables computed
Step4: q-values (# = 14): 421 281 211 71 61 43 41 31 29 13 11 7 5 3
Step5: testing conditions lp
Step6: testing potential divisors
Individual Fermat powerings:
  2  :   7
  3  :   7
  4  :   8
  5  :   8
  7  :   6
  8  :   2
Number of Fermat powerings = 38
Maximal number of nondeterministic steps = 0
2^2 * 13 * 89 * 216076058772687986171132238547968885047536732993
```

8.1.3 Counting Primes

Trying to understand how prime numbers are distributed is a problem that has intrigued mathematicians for hundreds of years. To make this question precise, we introduce the function $\pi(x)$, which counts the number of primes up to x:

$$\pi(x) = \#\{p : p \leq x \text{ is prime}\}.$$

For example,

$$\pi(10) = \#\{2, 3, 5, 7\} = 4.$$

Use `prime_pi` to compute with $\pi(x)$ in Sage.

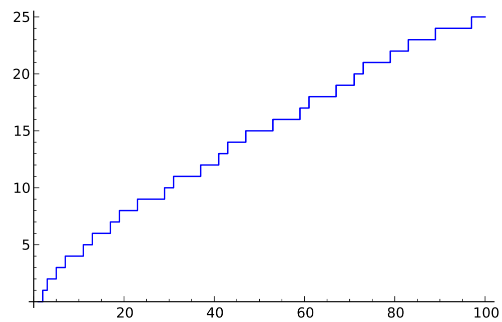
```
sage: prime_pi(10)
4
sage: prime_pi(10.7)
4
sage: prime_pi(100)
25
sage: prime_pi(1000)
168
```

You can count more primes than you might at first suspect:

```
sage: prime_pi(10^11)  # takes about 1 second
4118054813
```

The plot of $\pi(x)$ looks like a staircase:

```
sage: plot(prime_pi, 1, 100)
```



Based on heuristic evidence and numerical data, people conjectured (in the 1700s) that the bumpy staircase $\pi(x)$ behaves somewhat like the nice smooth function $x/\log(x)$. The Prime Number Theorem makes this precise; it is one of the deepest and most important theorems we have about $\pi(x)$, and its proof is quite difficult.

Theorem 8.1.3 (Prime Number Theorem)**.** *We have $\pi(x) \sim x/\log(x)$, which means that*

$$\lim_{x\to\infty} \frac{x/\log(x)}{\pi(x)} = 1.$$

This theorem means that if you want to use $x/\log(x)$ to estimate say 10 digits of $\pi(x)$, then there is definitely some B such that for all $x \geq B$, we have that $\pi(x)$ and $x/\log(x)$ have at least the same first 10 digits. However, the theorem itself makes no explicit claim about what B is; maybe it is 10^{30}, or maybe it is 10^{1000}.

There is conjecturally a vastly better smooth function that estimates $\pi(x)$, which is the special function $\mathrm{Li}(x)$:

$$\mathrm{Li}(x) = \int_2^x \frac{t}{\log(t)}.$$

The following conjecture is widely believed, but so far nobody has a clue how to prove it.

Conjecture 8.1.4 (Riemann Hypothesis)**.** *For all $x > 2.01$, we have*

$$|\pi(x) - \mathrm{Li}(x)| \leq \sqrt{x} \cdot \log(x).$$

In other words, if we estimate $\pi(x)$ using $\mathrm{Li}(x)$, then about half of the digits will be right. Moreover, there is no limit here; this is a statement about all $x > 2.01$, which is really amazing.

Some consider this conjecture to be the most important unsolved problem in mathematics. For example, it was selected as one of the Clay Mathematics Institute million dollar prize problems: `http://www.claymath.org/millennium/Riemann_Hypothesis/`

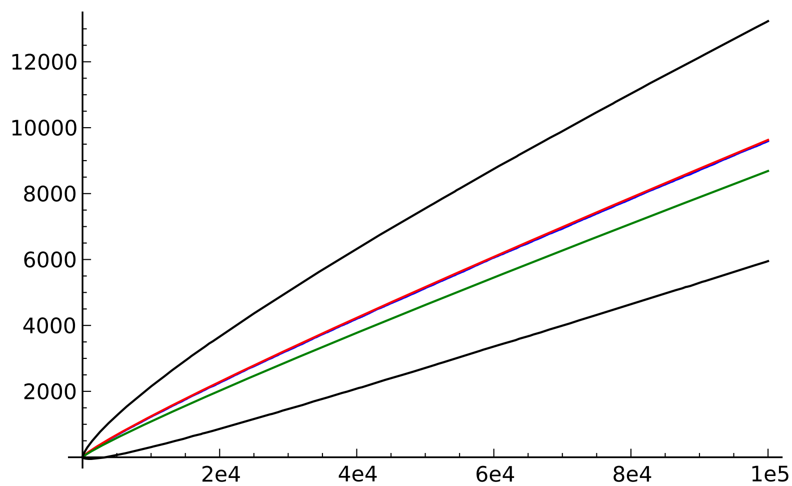
We illustrate the above conjecture using Sage.

```
sage: def rh(x):
...       pp = prime_pi(x)
...       print 'pi(x)          = %10.1f'%pp
...       print 'Li(x)          = %10.1f'%Li(x)
...       print 'x/log(x)       = %10.1f'%(x/math.log(x))
...       print 'sqrt(x)*log(x)= %10.1f'%(math.sqrt(x)*math.log(x))
...       print '|pi(x)-Li(x)|  = %10.1f'%abs(pp - Li(x))
...       print '|pi(x)-x/l(x)|= %10.1f'%abs(pp - x/math.log(x))
```

```
...
sage: rh(10^9)
pi(x)          = 50847534.0
Li(x)          = 50849233.9
x/log(x)       = 48254942.4
sqrt(x)*log(x)=   655327.2
|pi(x)-Li(x)| =     1699.9
|pi(x)-x/l(x)|=  2592591.6
```

The following plot illustrates Conjecture 8.1.4. In the plot $\pi(x)$ and $\mathrm{Li}(x)$ are visibly *on top of each other*!

```
sage: x = var('x')
sage: B = 10^5
sage: G = (plot(prime_pi, 2, B) + plot(Li, 2, B, color='red')
...              + plot(x/log(x), 2, B, color='green'))
sage: G += plot(lambda x: prime_pi(x) - math.sqrt(x)*math.log(x),
...              2, B, color='black')
sage: G += plot(lambda x: prime_pi(x) + math.sqrt(x)*math.log(x),
...              2, B, color='black')
sage: G
```



Conjecture 8.1.4 is typically stated in terms of a complex analytic function called the *Riemann Zeta function*.

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \prod_{\text{primes } p} \frac{1}{1 - p^{-s}}.$$

The function $\zeta(s)$ has a (uniquely determined) analytic continuation to $\mathbb{C} \setminus \{1\}$, and a simple pole at $s = 1$. In Sage, you can evaluate it anywhere using the command `zeta`:

```
sage: zeta(2)
1/6*pi^2
sage: zeta(3+I)
zeta(I + 3)
```

Use the N command or coercion to CC (the complex field) to give a numerical answer.

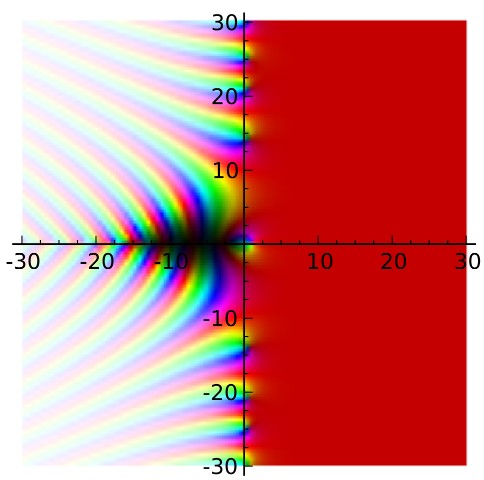
```
sage: CC(zeta(3+I))
1.10721440843141 - 0.148290867178175*I
sage: zeta(CC(3+I))
1.10721440843141 - 0.148290867178175*I
sage: N(zeta(3+I))
1.10721440843141 - 0.148290867178175*I
sage: N(zeta(3+I), 100)
1.1072144084314091956251002058 - 0.14829086717817534849076412567*I
```

An equivalent version of Conjecture 8.1.4 is the following statement about where the function $\zeta(s)$ takes the value 0.

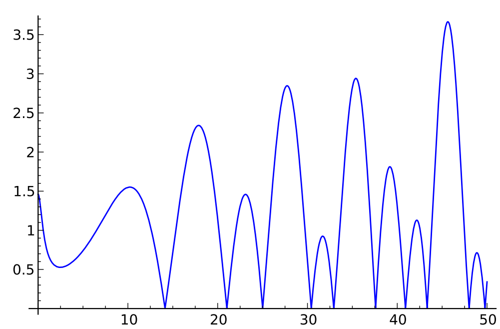
Conjecture 8.1.5. *The zeros of* $\zeta(s)$ *with* $\mathrm{Re}(s) \geq 0$ *all satisfy* $\mathrm{Re}(s) = 1/2$.

We can draw several plots of $\zeta(s)$, some of which illustrate the zeros of $\zeta(s)$.

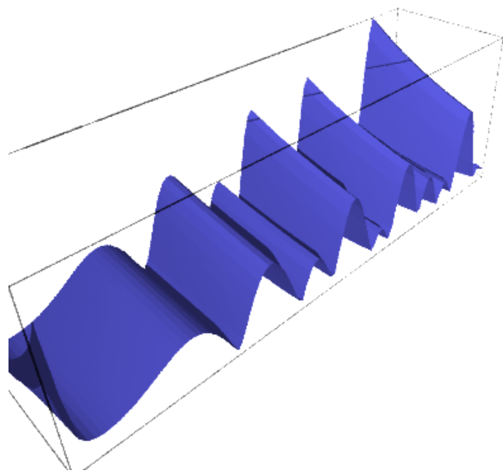
```
sage: complex_plot(zeta, (-30,30), (-30,30))
```




```
sage: plot(lambda y: abs(zeta(1/2+I*y)), (0,50))
```



```
sage: plot3d(lambda x, y: abs(zeta(x+I*y)), (.2,.7), (0,50),
...              plot_points=100)
```



8.2 Public-Key Cryptography: Diffie-Hellman

(for this section, there is a lot more in my handwritten notes...)

```
<p>Naive modular exponentiation is not good.</p>
sage: 7^11
1977326743
sage: (7^11) % 13
2
<p>But Sage implements a vastly better algorithm. </p>
sage: a = Mod(18, 11); a
7
sage: type(a)
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: parent(a)
Ring of integers modulo 11
sage: 18 % 11
7
sage: parent(18 % 11)
Integer Ring
sage: type(18 % 11)
<type 'sage.rings.integer.Integer'>
sage: a^139299208340283408230482348032984023948
9
<p>Here is a bigger example.</p>
sage: p = next_prime(10^100); p
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
sage: g = Mod(2, p)
sage: a = ZZ.random_element(p); a
```

```
38994629840785861384457662111217990522007745403201488128250840383333872299659576835783489303389
sage: g^a
79473887545115165760984304429323579667762572891406147323458363740845141926839332940954748013603
sage: timeit('g^a')
625 loops, best of 3: 71.1 s per loop
<p>We illustrate the Diffie-Hellman key exchange.</p>
sage: @interact
sage: def _(bits=(5..1024), g=2, seed=(0..100)):
...       t = cputime()
...       set_random_seed(seed)
...       p = next_prime(2^(bits-1))
...       print "<html>"
...       print "p = %s"%p
...       a = ZZ.random_element(p)
...       b = ZZ.random_element(p)
...       print "a = %s"%a
...       print "b = %s"%b
...       g = Mod(g, p)
...       print "g^a (mod p) = %s"%(g^a)
...       print "g^b (mod p) = %s"%(g^b)
...       print "secret = %s = %s"%((g^a)^b, (g^b)^a)
...       print "total time = %s seconds"%cputime(t)
...       print "</html>"
sage: time next_probable_prime  (2^(1024-1))
89884656743115795386465259539451236680898848947115328636715040578866337902750481566354238661203
Time: CPU 0.21 s, Wall: 0.21 s
<p>References:</p>
<ol>
<li>For math -- see <a href="http://wstein.org/ent " target="_blank">http://wstein.org/ent 
<li>More on cryptography using Sage -- see the book by David Kohel that is <a href="http://sage
<li>There is a library called <a href="http://www.dlitz.net/software/pycrypto/" target="_blank"
</ol>
```

8.3 Elliptic Curves and the Birch and Swinnerton-Dyer Conjecture

8.3.1 Fields

A *field* is a set of objects equipped with rules for multiplication and addition that satisfy certain axioms (for example, every nonzero element has an inverse). Standard examples of fields include the field $\mathbb{C}$ of all complex numbers and the field $\mathbb{Q}$ of all rational numbers. Also, for every prime number p we have the *finite field*

$$\mathbb{F}_p = \{0, 1, 2, \ldots, p-2, p-1\}$$

of numbers modulo p. In the field $\mathbb{F}_p$, arithmetic is defined by multiply or adding two numbers, then taking the remainder modulo p.

8.3.2 Elliptic Curves

Definition 8.3.1 (Elliptic Curve)**.** An *elliptic curve* over a field K is a curve defined by an equation $y^2 = x^3 + ax + b$ with $a, b \in K$ such that the cubic $x^3 + ax + b$ has distinct roots; equivalently, the discriminant $-4a^3 - 27b^2$ of the cubic is nonzero.

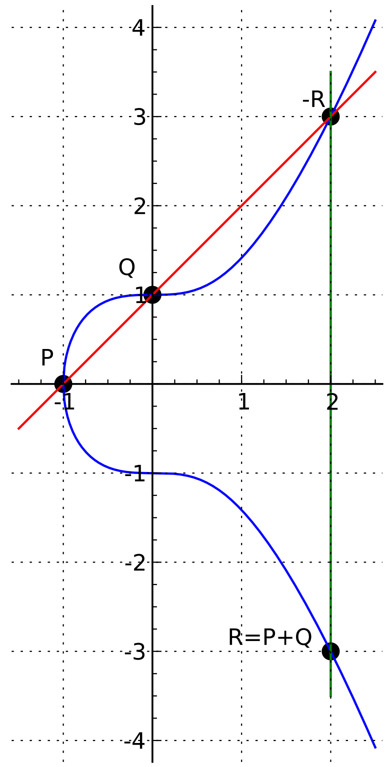
Suppose now that E is an elliptic curve over a field K. Then the set of K-rational points on E is

$$E(K) = \{(X, Y) \in K \times K : Y^2 = X^3 + aX + b\} \cup \{\mathcal{O}\}.$$

The extra point $\mathcal{O}$ should be thought of as being "at infinity" and is included since we view E as a curve in the "projective plane".

There is a natural way to define a way off *adding together* two elements $P, Q \in E(K)$ to get another element $R = P+Q \in E(K)$, thus generating possibly new points. This is the "chord and tangent" procedure; the following diagram illustrates using it to compute $R = P + Q = (0, 1) + (-1, 0) = (2, -3)$.

```
sage: E = EllipticCurve([0,1])
sage: P = E([0,1]); Q = E([-1,0]); R = P+Q; mR = -R
sage: G = E.plot(-1.5,2.5, plot_points=300)
sage: v = [(0,1), (-1,0), (2,-3), (2,3)]
sage: G += points(v, pointsize=50, color='black')
sage: G += line([(-1.5,-.5), (2.5,3.5)], color='red')
sage: G += text("P", (-1.2,.3), color='black')
sage: G += text("Q", (-.3,1.3), color='black')
sage: G += text("-R", (1.8,3.2), color='black')
sage: G += text("R=P+Q", (1.3,-2.85), color='black')
sage: G += line([(2,3.5), (2,-3.5)], color='green')
sage: G.show(gridlines=True, aspect_ratio=1)
```



When $K = \mathbb{Q}$ is the rational numbers, there is an amazing theorem about $E(K)$.

Theorem 8.3.2 (Mordell)**.** *Let E be any elliptic curve over $\mathbb{Q}$. Then there are finitely many points $P_1, \ldots, P_k$ in $E(\mathbb{Q})$ such that* every *point in $E(\mathbb{Q})$ is of the form $n_1P_1 + \cdots + n_kP_k$ for some integers $n_1, \ldots, n_k \in \mathbb{Z}$.*

Mordell's theorem means that $E(\mathbb{Q})$ is a finitely generated abelian group, so $E(\mathbb{Q})$ is isomorphic to $\mathbb{Z}^r \oplus T$, for some finite group T, and some nonnegative integer r. The number $r = \mathrm{rank}(E)$ is called the *rank* of E and is a fundamental and mysterious invariant of E.

Open Problem 8.3.3. Give an algorithm that takes as input an elliptic curve E over $\mathbb{Q}$ and outputs the rank of E.

Problem 8.3.3 goes back over 1000 years, making it perhaps the oldest *"interesting"* problem in all of mathematics, where the problem is interesting because of its connections to many ideas in modern number theory, and the numerous partial results that mathematicians have obtained. In particular, around 1000 years ago the Arabs asked for an algorithm to decide whether or not an integer n is the area of a rational right triangle, i.e., a right triangle all three of whose side lengths are rational numbers. The connection with Problem 8.3.3 arises because n is the area of a rational right triangle if and only if the rank of the elliptic curve $y^2 = x^3 + n^2x$ is positive.

8.3.3 Birch and Swinnerton-Dyer

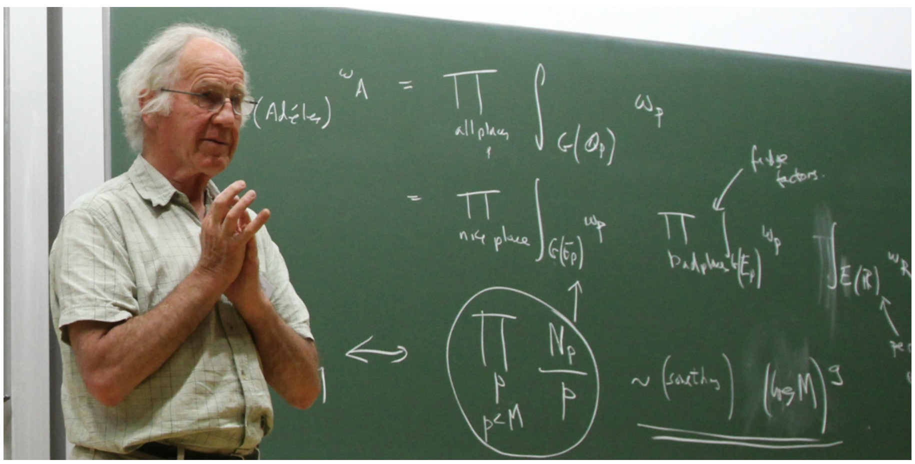
In the 1960s two British mathematicians, Bryan Birch and Sir Peter Swinnerton-Dyer (BSD), had an amazing idea related to Problem 8.3.3. After a huge amount of work and difficult hard won 1960s computer use, they obtained precise data relating two quantities for many elliptic curves. Let $N_p = \#E(\mathbb{F}_p)$, where $\#E(\mathbb{F}_p)$ is the number of points on the elliptic curve obtained by reducing the equation that defines E modulo p, when this makes sense.

$$\mathrm{rank}(E) \longleftrightarrow f_E(M),$$

where

$$f_E(M) = \prod_{\text{good primes } p<M} \frac{N_p}{p}.$$

This function is something that is dramatically simpler to contemplate computing than $\mathrm{rank}(E)$. You simply reduce the equation that defines E modulo p, and count all the solutions modulo p to the reduced equation. It is easy to come up with a (slow) algorithm to do that for any given p.



Birch explaining the conjecture in Cambrige on May 4, 2011

We will compute $f_E(M)$ in Sage using the following:

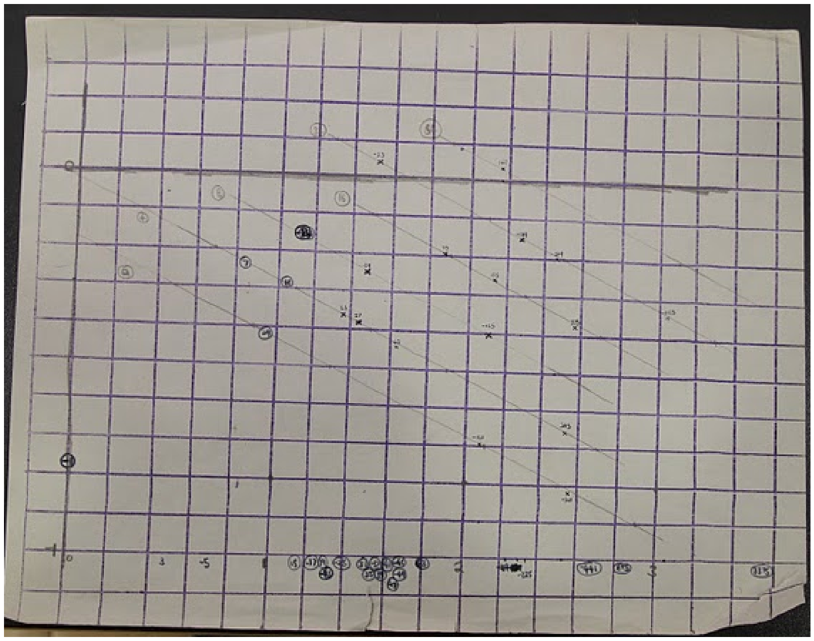
```
def f(E,M):
    N = E.conductor()
    return prod(E.Np(p)/float(p) for p in primes(M) if N%p)
```

BSD considered mainly curves of the form $y^2 = x^3 + b$, with b an integer. For example, we have the following table for various values of b:

```
for b in [1,2,-11,-6,316]:
    E = EllipticCurve([0,b])
    v = (b, E.rank(), f(E,10^3), f(E,10^4), f(E,10^6))
    print '%4s%4s%10.3f%10.3f%10.3f'%v


   1   0     1.895     2.060     1.849
   2   1     6.804     8.735    11.693
 -11   2    36.523    49.215   143.102
  -6   0     0.461     0.551     1.013
 316   3   100.158   261.144   879.231
```

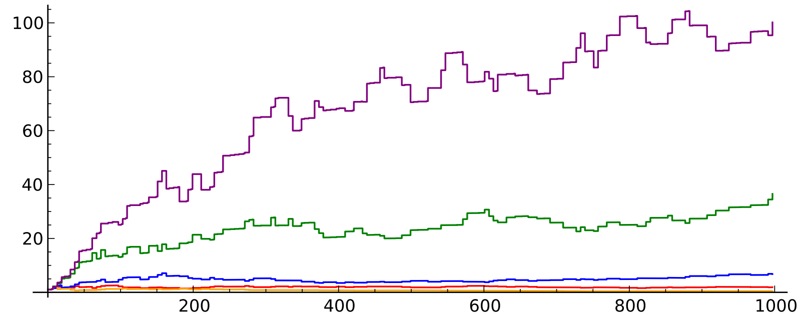
Here is a photo I snapped of the very piece of paper on which BSD came up with the basic idea of matching the ranks up with the behavior of $f_E(M)$ (do not ask me to explain it):



What they found was that by eyeballing the plots of $f_E(M)$, they were able in practice to predict the rank. Incidentally, we can plot $f_E(M)$ in Sage using the following code:

```
def f_plot(E, M, **kwds):
    N = E.conductor()
    v = [(0,1)]
    pr = 1
    for p in primes(M):
        if N%p:
            pr *= E.Np(p)/float(p)
            v.append((p, v[-1][1]))
            v.append((p, pr))
    return line(v, **kwds)

B = 10^5
show(f_plot(EllipticCurve([0,1]), B, color='red') +
    f_plot(EllipticCurve([0,2]), B, color='blue') +
    f_plot(EllipticCurve([0,-11]), B, color='green') +
    f_plot(EllipticCurve([0,-6]), B, color='orange') +
    f_plot(EllipticCurve([0,316]), B, color='purple')
)
```



I hope you will agree that looking at the above pictures suggests the rank... but certainly doesn't feel rock solid and precise. Fortunately, there is another approach to the same problem that involves an object much like the Riemann Zeta function, which appeared above in Section 8.3.

Fix an elliptic curve E. For each prime number p, set $a_p = p + 1 - N_p$. Let

$$L^*(E,s) = \prod_p \frac{1}{1 - a_p p^{-s} + p^{1-2s}}. \tag{8.3.1}$$

Remark 8.3.4. There is also a way to define factors for all primes p, and one obtains a function that we denote $L(E,s)$. For the conjecture we make below, it makes no difference whether we use $L^*(E,s)$ or $L(E,s)$.

A big theorem proved in 2000, but proved in many special cases already in 1960s is:

Theorem 8.3.5. *The function $L(E,s)$ has a unique analytic continuation to the whole complex plane.*

In other words, despite the right hand side of (8.3.1) possibly not converging, there is a natural and meaningful "nice" way of making sense of $L(E,s)$ for any complex number s. The reason $L(E,s)$ is so relevant to the function $f_E(M)$ that we considered above is that *formally*[1]

$$L(E,1) \text{``}=\text{''} \prod_p \frac{1}{1 - a_p p^{-1} + p^{1-2}} = \prod_p \frac{p}{p - a_p + 1} = \prod_p \frac{p}{N_p} \text{``} = \frac{1}{f_E(\infty)}\text{''}.$$

Motivated by the above formal observation and their other data, BSD made the following conjecture:

Conjecture 8.3.6 (Birch and Swinnerton-Dyer)**.** *Let E be an elliptic curve over $\mathbb{Q}$. Then*

$$\mathrm{ord}_{s=1} L(E,s) = \mathrm{rank}(E).$$

This is a Clay Million Dollar prize problem: `http://www.claymath.org/millennium/Birch_and_Swinnerton-Dyer_Conjecture/`.

The work of many, many people over several decades has resulted in the following theorem:

Theorem 8.3.7. *If $\mathrm{ord}_{s=1} L(E,s) \le 1$, then Conjecture 8.3.6 holds for E.*

Sage is good at computing with $L(E,s)$. For example,

[1]And in fact this equality is probably true only true up to a factor of $\sqrt{2}$...

```
sage: E = EllipticCurve([0,-6])
sage: L = E.lseries().dokchitser()
sage: L(2)
0.970573503589685
sage: L(1)
1.80166139420421
sage: L(1+I)
1.37330247586099 + 0.672104565160637*I
sage: L.taylor_series(1, 5)
1.80166139420421 - 4.34358857895219*z + 10.6996108328594*z^2
   - 16.6581015345210*z^3 + 17.7188237405279*z^4 + O(z^5)
```

Here is an example of rank 2:

```
sage: E = EllipticCurve([0,-11])
sage: L = E.lseries().dokchitser()
sage: L.taylor_series(1, 5)
2.66270802215019e-23 + (-6.18778237886993e-23)*z
     + 5.92327478382316*z^2 - 13.7649096437350*z^3
     + 17.0105571907034*z^4 + O(z^5)
```

Finally, here is an example of rank 3:

```
sage: E = EllipticCurve([0,316])
sage: E.rank()
sage: L = E.lseries().dokchitser()
sage: L.taylor_series(1, 5)
(8.21208956591497e-23)*z + (-3.64556152695356e-22)*z^2
   + 25.3581351256025*z^3 - 112.571399845523*z^4 + O(z^5)
sage: E.analytic_rank() # order of vanishing of L
3
```

Though we can numerically evaluate $L(E, s)$ at any point to any number of digits, we do not have a way in general to provably compute $\mathrm{ord}_{s=1} L(E, s)$. For example, we may suspect that $\mathrm{ord}_{s=1} L(E, s) = 4$ since *numerically* to 10,000 digits (say) we find that $L^{(k)}(E, 1) = 0.00000...$ for $k = 0, 1, 2, 3$, but this is not a proof.

Open Problem 8.3.8. Verify with proof Conjecture 8.3.6 for one single elliptic curve of rank 4, e.g., for the curve $y^2 = x^3 - 102627x + 12560670$.

Chapter 9

Statistics

9.1 Using R with Sage

TODO:

```
<p>See <a href="http://rpy.sourceforge.net/rpy2/doc-2.0/html/introduction.html" target="_blank"
sage: %auto
sage: import rpy2.robjects as robjects
sage: R = robjects.r
<p>We get pi from the R namespace.</p>
sage: v = R['pi']; v
<RVector - Python:0x433e320 / R:0x4fd90c8>
<p>Note that we have to explicitly use print to see a nice representation:</p>
sage: print v
[1] 3.141593
<p>There is a pexpect interface to r called "r" by default when you start Sage.  This tuto
sage: r
R Interpreter
sage: r('2 + 3')   # the pexpect interface
[1] 5
sage: import rpy2.robjects as robjects
sage: R = robjects.r
sage: print R('2 + 3')  # the rpy2 cython interface (note the import!)
[1] 5
sage: R("""
sage: a = 5
sage: b = 7
sage: c = a + b""")
sage: print R("c")
[1] 12
sage: timeit("r('2+3')")
625 loops, best of 3: 1.44 ms per loop
sage: timeit("R('2+3')")
625 loops, best of 3: 650 s per loop
sage: timeit("pari('2+3')")
625 loops, best of 3: 5.72 s per loop
<p>(frankly, I'm shocked at how slow the rpy2 interface actually is...!)</p>
```

```
<p>This is how to get started with rpy2:</p>

<p>Beware the preparser:</p>
sage: v = R['pi']; v
<RVector - Python:0x433ec20 / R:0x4fd90c8>
sage: print v
[1] 3.141593
sage: repr(v)
'<RVector - Python:0x433dcf8 / R:0x48e3178>'
sage: str(v)
'[1] 3.141593'
sage: w =  v + int(1); print w
[1] 3.141593 1.000000
sage: w[0]
3.1415926535897931
sage: v + 3
Traceback (most recent call last):
...
ValueError: Nothing can be done for the type <type 'sage.rings.integer.Integer'> at the moment.
<p>And note again that v is a vector not a number.</p>
sage: print v + int(3)
[1] 3.141593 3.000000
sage: print v[0] + int(3)
6.14159265359
<p>WARNING:  Python indexing starts at 0 and R indexing starts at 1.</p>
sage: print R('c(5,2,-3)[1]')
[1] 5
sage: timeit('R("f <- function(r) { 2 * pi * r }")')
625 loops, best of 3: 460 s per loop
<p>Define a function in R:</p>
sage: R("f <- function(r) { 2 * pi * r }")
<RFunction - Python:0x433f440 / R:0x529ec00>
<p>Now call the function:</p>
sage: print R("f(3)")
[1] 18.84956
<p>The function is now defined in the global R namespace:</p>
sage: r_f = R['f']
sage: print r_f(int(3))
[1] 18.84956
sage: timeit('r_f(int(3))')
625 loops, best of 3: 41.4 s per loop
sage: print R("f")
function(r) { 2 * pi * r }
<p>Most R objects have a string representation that can be directly parsed by R, which can be h
sage: letters = R['letters']
sage: print letters.r_repr()
c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l",
"m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y",
"z")
<p>Here is an example of how we might use this:</p>
```

```
sage: rcode = 'paste(%s, collapse="-")' %(letters.r_repr())
sage: print R(rcode)
[1] "a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
sage: timeit('robjects.IntVector(range(10))')
625 loops, best of 3: 9.65 s per loop
sage: time w = robjects.IntVector(range(10^6))
Time: CPU 0.74 s, Wall: 0.74 s
sage: time print R['mean'](w)
[1] 499999.5
Time: CPU 0.16 s, Wall: 0.17 s
sage: time print R['sd'](w)
[1] 288675.3
Time: CPU 0.18 s, Wall: 0.18 s
sage: time w = r(range(10^3))
Time: CPU 1.12 s, Wall: 2.56 s
sage: time z = pari(range(10^6))
Traceback (most recent call last):
...
KeyboardInterrupt: evaluating PARI string
__SAGE__
<h2>Vectors</h2>
<p>Vectors are an important basic data structure in R:</p>
sage: print robjects.StrVector(['abc', 'def'])
[1] "abc" "def"
sage: print robjects.IntVector([1, 2, 3])
[1] 1 2 3
sage: print robjects.FloatVector([1.1, 2.2, 3.3])
[1] 1.1 2.2 3.3
<p>You can also create R matrices, which are R vectors with a dim attribute:</p>
sage: v = robjects.FloatVector([1.1, 2.2, 3.3, 4.4, 5.5, 6.6])
sage: m = R['matrix'](v, nrow = int(2))
sage: print m
     [,1] [,2] [,3]
[1,]  1.1  3.3  5.5
[2,]  2.2  4.4  6.6
<h2>R functions</h2>
<p>The above illustrates how to call an R function.  You get it from the R namespace, then
sage: v = robjects.IntVector([1..10])
sage: print R['sum']
function (..., na.rm = FALSE)  .Primitive("sum")
sage: ans = R['sum'](v)
sage: ans
<RVector - Python:0x4368368 / R:0x4e63188>
sage: print ans
[1] 55
sage: ans[0]
55
sage: R['sum'](v)[0]    # [0] since result is a vector of length 1
55
sage: R['mean'](v)[0]
```

```
5.5
sage: R['sd'](v)[0]
3.0276503540974917
sage: sd = R['sd']
sage: timeit('sd(v)')
625 loops, best of 3: 280 s per loop
sage: timeit("R['sd'](v)")
625 loops, best of 3: 237 s per loop
<p>You can also pass in keywords:</p>
sage: rsort = R['sort']
sage: print rsort(v, decreasing=True)
 [1] 10  9  8  7  6  5  4  3  2  1
<p>GOTCHA: In R variable names with dots in them are allowed, but in Python they are not.  
sage: v = R('c(1,NA,2,3)')
sage: print v
[1]  1 NA  2  3
sage: print R['sum']
function (..., na.rm = FALSE)  .Primitive("sum")
sage: rsum = R['sum']
sage: print rsum(v)
[1] NA
<p>Directly in R, we would just type na.rm=TRUE.  In Python this does not make sense.</p>
sage: print R('sum( c(1,NA,2,3), na.rm=TRUE )')
[1] 6
sage: print rsum(v, na.rm=True)   # boom!
Traceback (most recent call last):
...
SyntaxError: keyword can't be an expression
sage: f(*[5,,7])
33
<p>So we use **kwds, which works fine:</p>
sage: a = {'na.rm':True}
sage: print R['sum'](v, **a)
[1] 6
sage: def f(a, b, c):
...       return a + 2*b + 3*c
...
sage: args = (5,)
sage: kwds = {'b':7, 'c':13}
sage: f(*args, **kwds)
58
sage: def g(*scott, **alex):
...       print scott, alex
...       return f(*scott, **alex)
sage: g(1,2,c=3)
(1, 2) {'c': 3}
14
sage: f( *(3, 8), **{'c':2})
25
sage: f( 2, *(5,), **{'c':1})
```

```
15
<h2>Plotting using Rpy2:</h2>
<ol>
<li>Call the R.png function to tell R where the output image should be saved (and what size it
<li>Draw plots on the canvas until done.</li>
<li>Tell R to turn the plotting device off, which causes the output file to be written.  <
</ol>
<p>IMPORTANT: This must all happen in the same notebook cell.  Otherwise the output file g
sage: x = robjects.IntVector(range(50))
sage: y = R.rnorm(len(x))     # normal random numbers
sage: # 300r = "raw Python int" (no preparser)
sage: R.png('sage.png', width=600r, height=300r)
sage: R.plot(x, y, xlab="x", ylab="rnorm", col="red")
sage: _ = R['dev.off']()   # "_ =" to suppress printing
<p>Interact works, of course.</p>
sage: @interact
sage: def _(points=(10..1000)):
...       x = robjects.IntVector(range(points)); y = R.rnorm(int(points))
...       R.png('sage.png', width=600r, height=300r)
...       R.plot(x, y, xlab="x", ylab="rnorm", col="blue")
...       R['dev.off']()
<p><strong>Warning again -- Do NOT do this:</strong> call dev.off in a separate cell!</p>
sage: # intentionally broken!
sage: x = robjects.IntVector(range(50))
sage: y = R.rnorm(len(x))     # normal random numbers
sage: R.png('sage.png', width=600r, height=300r)
sage: R.plot(x, y, xlab="runif", ylab="foo/bar", col="red")
<RObject - Python:0x434b488 / R:0x42a6758>
sage: R['dev.off']()
<RVector - Python:0x434b128 / R:0x551b408>
<h2>A More Nontrivial Example</h2>
<p> </p>

<p>This is how we would do this directly in R, which we can use from Sage by using the "%r" mod
sage: %r
sage: ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
sage: trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
sage: group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
sage: weight <- c(ctl, trt)
sage: anova(lm.D9 <- lm(weight ~ group))
sage: summary(lm.D90 <- lm(weight ~ group - 1))# omitting intercept
Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1 0.6882 0.68820  1.4191  0.249
Residuals 18 8.7293 0.48496

Call:
lm(formula = weight ~ group - 1)
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-1.0710 -0.4938  0.0685  0.2462  1.3690

Coefficients:
         Estimate Std. Error t value Pr(>|t|)
groupCtl   5.0320     0.2202   22.85 9.55e-15 ***
groupTrt   4.6610     0.2202   21.16 3.62e-14 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared: 0.9818,Adjusted R-squared: 0.9798
F-statistic: 485.1 on 2 and 18 DF,  p-value: < 2.2e-16
<p>Next, we do the same computation, but via rpy2 (which is unfortunately more complicated):</p
sage: ctl = robjects.FloatVector([4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14])
sage: trt = robjects.FloatVector([4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69])
sage: group = R.gl(2r, 10r, 20r, labels = ["Ctl","Trt"])
sage: weight = ctl + trt
sage: robjects.globalEnv["weight"] = weight
sage: robjects.globalEnv["group"] = group
sage: lm_D9 = R.lm("weight ~ group")
sage: print(R.anova(lm_D9))
Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1 0.6882 0.68820  1.4191  0.249
Residuals 18 8.7293 0.48496
sage: lm_D90 = R.lm("weight ~ group - 1")
sage: v = R.summary(lm_D90)
sage: print(v)
Call:
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
{
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    mf <- match.call(expand.dots = FALSE)
    m <- match(c("formula", "data", "subset", "weights", "na.action",
        "offset"), names(mf), 0L)
    mf <- mf[c(1L, m)]
    mf$drop.unused.levels <- TRUE
    mf[[1L]] <- as.name("model.frame")
    mf <- eval(mf, parent.frame())
    if (method == "model.frame")
        return(mf)
```

```
    else if (method != "qr")
        warning(gettextf("method = '%s' is not supported. Using 'qr'",
            method), domain = NA)
    mt <- attr(mf, "terms")
    y <- model.response(mf, "numeric")
    w <- as.vector(model.weights(mf))
    if (!is.null(w) && !is.numeric(w))
        stop("'weights' must be a numeric vector")
    offset <- as.vector(model.offset(mf))
    if (!is.null(offset)) {
        if (length(offset) != NROW(y))
            stop(gettextf("number of offsets is %d, should equal %d (number of observations)",
                length(offset), NROW(y)), domain = NA)
    }
    if (is.empty.model(mt)) {
        x <- NULL
        z <- list(coefficients = if (is.matrix(y)) matrix(, 0,
            3) else numeric(0L), residuals = y, fitted.values = 0 *
            y, weights = w, rank = 0L, df.residual = if (is.matrix(y)) nrow(y) else length(y))
        if (!is.null(offset)) {
            z$fitted.values <- offset
            z$residuals <- y - offset
        }
    }
    else {
        x <- model.matrix(mt, mf, contrasts)
        z <- if (is.null(w))
            lm.fit(x, y, offset = offset, singular.ok = singular.ok,
                ...)
        else lm.wfit(x, y, w, offset = offset, singular.ok = singular.ok,
            ...)
    }
    class(z) <- c(if (is.matrix(y)) "mlm", "lm")
    z$na.action <- attr(mf, "na.action")
    z$offset <- offset
    z$contrasts <- attr(x, "contrasts")
    z$xlevels <- .getXlevels(mt, mf)
    z$call <- cl
    z$terms <- mt
    if (model)
        z$model <- mf
    if (ret.x)
        z$x <- x
    if (ret.y)
        z$y <- y
    if (!qr)
        z$qr <- NULL
    z
}(formula = "weight ~ group - 1")
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-1.0710 -0.4938  0.0685  0.2462  1.3690

Coefficients:
         Estimate Std. Error t value Pr(>|t|)
groupCtl   5.0320     0.2202   22.85 9.55e-15 ***
groupTrt   4.6610     0.2202   21.16 3.62e-14 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared: 0.9818,Adjusted R-squared: 0.9798
F-statistic: 485.1 on 2 and 18 DF,  p-value: < 2.2e-16
sage: print(lm_D9.names)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "contrasts"     "xlevels"       "call"          "terms"
[13] "model"
sage: print(lm_D9.r['coefficients'])
$coefficients
(Intercept)    groupTrt
      5.032      -0.371
<p>You could also use rpy2 as follows to do this computation:</p>
sage: R("""
sage: ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
sage: trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
sage: group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
sage: weight <- c(ctl, trt)
sage: print(anova(lm.D9 <- lm(weight ~ group)))
sage: print(summary(lm.D90 <- lm(weight ~ group - 1)))
sage: """)
Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value Pr(>F)
group      1 0.6882 0.68820  1.4191  0.249
Residuals 18 8.7293 0.48496

Call:
lm(formula = weight ~ group - 1)

Residuals:
    Min      1Q  Median      3Q     Max
-1.0710 -0.4938  0.0685  0.2462  1.3690

Coefficients:
         Estimate Std. Error t value Pr(>|t|)
groupCtl   5.0320     0.2202   22.85 9.55e-15 ***
groupTrt   4.6610     0.2202   21.16 3.62e-14 ***
```

```
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared: 0.9818,Adjusted R-squared: 0.9798
F-statistic: 485.1 on 2 and 18 DF,  p-value: < 2.2e-16
<h2>Data Frames</h2>
<p>In R a "data frame" is an array of values with labeled rows and columns (like part of a spre
<p>You can create a data frame using the data.frame R function:</p>
sage: d = {'value': robjects.IntVector((24,25,26)),
...        'letter': robjects.StrVector(('x', 'y', 'z'))}
...
sage: dataf = R['data.frame'](**d)
sage: print(dataf)
  letter value
1      x    24
2      y    25
3      z    26
sage: type(dataf)
<class 'rpy2.robjects.RDataFrame'>
<p>Get each column:</p>
sage: print dataf.r['letter']
  letter
1      x
2      y
3      z
sage: print dataf.r['value']
  value
1    24
2    25
3    26
<p>Labels for the rows:</p>
sage: print dataf.rownames()
[1] "1" "2" "3"
<p>Labels for the columns:</p>
sage: print dataf.colnames()
[1] "letter" "value"
<h2>Converting Between Numpy and RPy2</h2>
<p>If you are using rpy2 and Sage together to deal with large real-world data sets, then it is
<p>NOTE: The rpy2 documentation suggests doing "import rpy2.robjects.numpy2ri" but this is brok
sage: import numpy
sage: a = numpy.array([[1,2],[3,4]], dtype=float)
sage: v = numpy.arange(5)
sage: print R(v)
Traceback (most recent call last):
...
ValueError: Nothing can be done for the type <type 'numpy.ndarray'> at the moment.
sage: print(robjects.FloatVector(v))
[1] 0 1 2 3 4
sage: import rpy2.robjects.numpy2ri
```

```
sage: print R(numpy.array([[1,2],[3,4]], dtype=float))
[1] 4
<p>... CRAP, this seems to be just totally broken in rpy2.  Maybe it is fixed in a newer v
```

Chapter 10

Abstract Algebra

10.1 Groups, Rings and Fields

```
<p>The first page of "abstract mathematics" that I ever saw, accidentally misfiled in a the com
<p><img src="data/burton.png" alt="" /></p>
<h2>Groups</h2>
<p>A group is a set $G$ equipped with a binary operation $G \times G \to G$ that we write as a
<ol>
<li><strong>Associativity</strong>:  $(a\cdot b)\cdot c = a\cdot(b\cdot c)$</li>
<li><strong>Existence of identity</strong>: There is $1\in G$ such that $1\cdot a = a\cdot 1 =
<li><strong>Existence of inverse</strong>: For each $a\in G$ there is $a^{-1} \in G$ such that
</ol>
<h3>Examples</h3>
<p>We construct objects in Sage that have a binary operation satisfying the above properties.</

<h3>The Integers</h3>
sage: G = Integers()       # the operation is +
sage: G
Integer Ring
sage: G(2) + G(5)
7
<h3>The Integers Modulo 12 (Clock Arithmetic)</h3>
sage: G = Integers(12); G   # operation is "+"
Ring of integers modulo 12
sage: list(G)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
<p>If it is 7am, what time will it be 10 hours from now?  Answer: 5pm.</p>
sage: G(3) + G(10)
1
sage: G.addition_table()
+  a b c d e f g h i j k l
 +------------------------
a| a b c d e f g h i j k l
b| b c d e f g h i j k l a
c| c d e f g h i j k l a b
d| d e f g h i j k l a b c
```

```
e| e f g h i j k l a b c d
f| f g h i j k l a b c d e
g| g h i j k l a b c d e f
h| h i j k l a b c d e f g
i| i j k l a b c d e f g h
j| j k l a b c d e f g h i
k| k l a b c d e f g h i j
l| l a b c d e f g h i j k
<h3>Elliptic Curves</h3>
sage: E = EllipticCurve([0, 1, 1, -2, 0]); E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
sage: E(QQ)
Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Fi
sage: P, Q = E.gens(); P, Q
((-1 : 1 : 1), (0 : 0 : 1))
sage: P + Q + P + P + P + Q
(1809/1936 : -20033/85184 : 1)
sage: E = EllipticCurve(GF(7), [0, 1, 1, -2, 0]); E
Elliptic Curve defined by y^2 + y = x^3 + x^2 + 5*x over Finite Field of size 7
sage: E(GF(7))
Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 + x^2 + 5*x over Finite Fiel
sage: E.cardinality()
13
sage: plot(E, pointsize=40).show(figsize=[2.5,2.5], gridlines=True)
<html><font color='black'><img src='cell://sage0.png'></font></html>
<h3>The Group of all Permutations of $\{1,2,3,\ldots, n-1, n\}$:</h3>
sage: G = SymmetricGroup(3); G
Symmetric group of order 3! as a permutation group
sage: list(G)
[(), (2,3), (1,2), (1,2,3), (1,3,2), (1,3)]
sage: for g in G:
...       print g
()
(2,3)
(1,2)
(1,2,3)
(1,3,2)
(1,3)
sage: g(3)
1
sage: G = SymmetricGroup(12)
sage: G.cardinality()
479001600
sage: s = G([(1,5,3),(2,4)]); s
(1,5,3)(2,4)
sage: s(5)
3
sage: s.order()
6
sage: G.multiplication_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: show(G.cayley_graph())
<html><font color='black'><img src='cell://sage0.png'></font></html>
<h3>The Group of orientation preserving symmetries of the icosahedron...</h3>
sage: icosahedron().show(viewer='canvas3d')
sage: G = AlternatingGroup(5); G
Alternating group of order 5!/2 as a permutation group
sage: G.order()
60
<p>Advanced Functionality...</p>
sage: show(G.character_table())
<html><div class="math">\newcommand{\Bold}[1]{\mathbf{#1}}\left(\begin{array}{rrrrr}
1 & 1 & 1 & 1 & 1 \\
3 & -1 & 0 & \zeta_{5}^{3} + \zeta_{5}^{2} + 1 & -\zeta_{5}^{3} - \zeta_{5}^{2} \\
3 & -1 & 0 & -\zeta_{5}^{3} - \zeta_{5}^{2} & \zeta_{5}^{3} + \zeta_{5}^{2} + 1 \\
4 & 0 & 1 & -1 & -1 \\
5 & 1 & -1 & 0 & 0
\end{array}\right)</div></html>
sage: G.derived_series()
[Permutation Group with generators [(3,4,5), (1,2,3,4,5)]]
sage: G.is_solvable()
False
sage: G.upper_central_series()
[Permutation Group with generators [()]]
sage: var('x,a,b')
sage: show(solve(x^3+a*x+b==0,x)[0])
<html><div class="math">\newcommand{\Bold}[1]{\mathbf{#1}}x = \frac{{\left(-i \, \sqrt{3} + 1\r
sage: C = G.cayley_graph()
sage: G.cayley_graph().plot3d(engine='tachyon').show()
<h3>The General and Special Linear Groups (Invertible Matrices)</h3>
sage: G = GL(2, GF(5)); G   # 2x2 invertible matrices with entries modulo 5
General Linear Group of degree 2 over Finite Field of size 5
sage: G.gens()
[
[2 0]
[0 1],
[4 1]
[4 0]
]
sage: G.cardinality()
480
sage: G = SL(2, GF(5))   # determinant 1
sage: G.order()
```

```
120
sage: G.subgroup([G.gens()[0]])
Traceback (most recent call last):
...
AttributeError: 'SpecialLinearGroup_finite_field_with_category' object has no attribute 'subgro
sage: GG = gap(G)
sage: GG
SL(2,5)
sage: GG.Order()
120
<h3>Rubik's Cube Group</h3>
<p>See the <a href="http://www.sagemath.org/doc/reference/sage/groups/perm_gps/cubegroup.html"
sage: RubiksCube().plot3d().show(viewer='tachyon', figsize=2, zoom=.9)
sage: G = CubeGroup(); G
The PermutationGroup of all legal moves of the Rubik's cube.
sage: G.gens()
['(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27)', '(41,43,48,46)(42,45,47,4
sage: GG = PermutationGroup(G.gens())
sage: c = GG.cardinality(); c
43252003274489856000
sage: factor(c)
2^27 * 3^14 * 5^3 * 7^2 * 11
<h1>Rings and Fields</h1>
<p>An <strong>abelian group</strong> is a group $G$ where for every $a,b \in G$ we have $a\cdot
<p>An<strong> monoid</strong> is the same as a group, except we do not require the existence of
<p>A <strong>ring</strong> $R$ is a set with two binary operations, $+$ and $\cdot$ such that:<
<ol>
<li>$(R,+)$ is an abelian group,</li>
<li>$(R^*,\cdot)$ is an abelian monoid, where $R^*$ is the set of nonzero elements of $R$,</li>
<li>For all $a,b,c \in R$ we have $a\cdot (b+c) = a\cdot b + a\cdot c$.</li>
</ol>
<p>A <strong>field</strong> $K$ is a ring such that $(R^*, \cdot)$ is a group.</p>
<h2>Examples</h2>
<p>Like with groups, Sage (and mathematics!) comes loaded with numerous rings and fields.</p>
sage: ZZ
Integer Ring
sage: RR
Real Field with 53 bits of precision
sage: CC
Complex Field with 53 bits of precision
sage: RealField(200)
Real Field with 200 bits of precision
sage: AA
Algebraic Real Field
sage: Integers(12)
Ring of integers modulo 12
sage: GF(17)
Finite Field of size 17
sage: GF(9,'a')
Finite Field in a of size 3^2
```

```
sage: ZZ['x']
Univariate Polynomial Ring in x over Integer Ring
sage: QQ['x,y,z']
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: ZZ[sqrt(-5)]
Order in Number Field in a with defining polynomial x^2 + 5
sage: QQ[['q']]
Power Series Ring in q over Rational Field
<p>Just as for groups, there is much advanced functionality available for rings (e.g., Groebner
```

10.2 Exact Linear Algebra

Linear algebra is the study of matrices, vectors, solving linear systems of equations, vector spaces, and linear transformation. It is a topic that is loaded with interesting algorithms, and Sage is good at it. In this section, we will focus on *exact linear algebra*, in which all matrices and vectors that we consider have exact entries (e.g., rational numbers, numbers modulo p, polynomials over the rationals, etc.), as opposed to numerical linear algebra with floating point entries; thus, for this section, roundoff error and general numerical analysis are not directly relevant.

10.2.1 Documentation for Linear Algebra in Sage

- **Quick Reference Card:** There is a linear algebra quick reference card available at `http://wiki.sagemath.org/quickref`.
- **Sage reference manual:** The following chapters are particularly relevant:
  - Matrices: `http://sagemath.org/doc/reference/matrices.html`
  - Modules: `http://sagemath.org/doc/reference/modules.html`
- **Robert Beezer's book:** This is a free open source Undergraduate Linear Algebra Book, which is available here: `http://linear.ups.edu/`

10.2.2 Underlying Technology

The implementation of exact linear algebra in Sage is a combination of a large amount of code written in Cython from scratch with some C/C++ libraries. The Linbox C++ library `http://www.linalg.org/` is used for some matrix multiplication and characteristic and minimal polynomial computations, especially for very big matrices with entries in the rational numbers or a finite field. The IML library (see `http://www.cs.uwaterloo.ca/~astorjoh/iml.html`) is used behind the scenes for solving systems of linear equations over the rational numbers. The M4RI library is used for linear algebra over the field with two elements. Numpy is used in a few places, but only for numerical linear algebra. Most everything relies at some on the ATLAS basic linear algebra system (BLAS) at some level (see `http://math-atlas.sourceforge.net/`). Yes, even multiplying two matrices over the rational numbers is eventually done by multiplying matrices with floating point entries (via a block decomposition and reduction modulo primes)!

10.2.3 Matrices and Vectors

First we illustrate arithmetic with matrices

```
sage: A = matrix(QQ, 3, 4, [1..12]); B = matrix(QQ, 4,2, [1..8])
sage: A * B
[ 50  60]
[114 140]
[178 220]
```

The following arithmetic produces errors, as it should, since mathematically it makes no sense:

```
sage: A + B
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Full MatrixSpace
of 3 by 4 dense matrices over Rational Field' and 'Full MatrixSpace
of 4 by 2 dense matrices over Rational Field'
sage: B * A
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '*': 'Full MatrixSpace
of 4 by 2 dense matrices over Rational Field' and 'Full MatrixSpace
of 3 by 4 dense matrices over Rational Field'
```

Sage does let you add a scalar to a square matrix, which adds that scalar to each entry along the diagonal:

```
sage: A = matrix(QQ, 3, [1..9])
sage: A + 2/3
[ 5/3    2    3]
[   4 17/3    6]
[   7    8 29/3]
```

Next we consider the problem of solving linear systems. We can encode a linear system of equations as a matrix equation $Ax = v$, where the problem is to solve for the unknown x given A and v. In Sage, v can be either a vector or a matrix (and x will correspondingly be a vector or matrix). If there are infinitely many solutions for x, Sage returns exactly one.

```
sage: set_random_seed(1)
sage: A = random_matrix(QQ, 5, num_bound=100, den_bound=100); A
[ 59/78  13/14 -11/49 -47/75 -52/15]
[ 27/56 -40/51  10/53 -89/12  -3/16]
[ 82/61  -55/7 -74/45 -11/46   5/52]
[-43/32  79/37 -57/29 -48/29  43/15]
[ 67/47  12/23 -25/24  13/16  46/63]
sage: A.det()
-33309120911318572378640943486889/31089394772345027072747520000
sage: v = random_matrix(QQ, 5, 1, num_bound=100); v
[-76]
[ 98]
[-82]
```

```
[ 27]
[ 51]
sage: x = A.solve_right(v); x
[14237432503267641323564311584068 16/3330912091131857237864094348688 9]
[ 403480176009266931788705978326932/3330912091131857237864094348688 9]
[1021661231928866958567656117461050/3330912091131857237864094348688 9]
[-393424222265393565078003995300100/3330912091131857237864094348688 9]
[1153927117568938940697661220942640/3330912091131857237864094348688 9]
sage: A*x == v
True
```

You can also use the Matlab-style backslash notation for “solve right”:

```
sage: A \ v
[1423743250326764132356431158406816/33309120911318572378640943486889]
[ 403480176009266931788705978326932/33309120911318572378640943486889]
[1021661231928866958567656117461050/33309120911318572378640943486889]
[-393424222265393565078003995300100/33309120911318572378640943486889]
[1153927117568938940697661220942640/33309120911318572378640943486889]
```

We can also use the solve_left method to solve $xA = v$:

```
sage: v = random_matrix(QQ, 1, 5, num_bound=10^10); v
sage: x = A.solve_left(v)
sage: x*A == v
True
```

You can also solve linear sytems symbolically by using the solve command, as illustrated below. This is fine for relatively small systems (especially when you do not want to have to think about which field the coefficients lie in), but is dramatically less powerful for large systems.

```
sage: var('x1, x2, x3')
sage: e = [2*x1 + 3*x2 + 5*x3 == 1, -x1 + x2 + 15*x3 == 5, x1 + x2 + x3 == 1]
sage: S = solve(e, [x1, x2, x3]); S
[[x1 == (18/5), x2 == (-17/5), x3 == (4/5)]]
```

Here is how to “get at” the solution:

```
sage: S[0][0]
x1 == (18/5)
sage: S[0][0].lhs(), S[0][0].rhs()
(x1, 18/5)
```

Using matrices and exact linear algebra in Sage, we can solve the same system as follows:

```
sage: A = matrix(QQ, 3, [2,3,5, -1,1,15, 1,1,1])
sage: v = matrix(QQ, 3, 1, [1, 5, 1])
sage: x = A \ v; x
[ 18/5]
[-17/5]
[  4/5]
```

```
sage: A*x == v
True
```

Solving over the rational numbers using Sage matrices is quite powerful. For example:

```
sage: set_random_seed(1)
sage: A = random_matrix(QQ, 100, num_bound=10^10, den_bound=100)
sage: v = random_matrix(QQ, 100, 1, num_bound=10^10, den_bound=100)
sage: A[0]  # just the first row
(-9594630370/11, -2724596772/25, 1863701863/28, ... 164457253/5)
sage: x = A.solve_right(v)
sage: A*x == v
True
sage: len(x.str())
789999
```

On my 64-bit OS X dual core i7 2.7GHZ laptop, the timing to solve $Ax = v$ for exactly the above matrix in various software is as follows:

- Sage-4.6.2 (which uses the IML library): 0.45 seconds
- Magma 2.17-4: 1.39 seconds
- Mathematica 7.0: 10.5 seconds
- Maple 14: 18.2 seconds

The characteristic polynomial of a square matrix A is $f(x) = \det(A - x)$; it has the property that $f(A) = 0$.

```
sage: A = matrix(QQ, 5, [1..25]); A
[ 1  2  3  4  5]
[ 6  7  8  9 10]
[11 12 13 14 15]
[16 17 18 19 20]
[21 22 23 24 25]
sage: f = A.characteristic_polynomial(); f
x^5 - 65*x^4 - 250*x^3
sage: f.factor()
x^3 * (x^2 - 65*x - 250)
sage: f(A)
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
sage: R.<x> = QQ[]
sage: (x - A).det()
x^5 - 65*x^4 - 250*x^3
```

Internally, Sage using some very clever algorithm (from the Linbox C++ library) to compute the characteristic polynomial, so Sage is fairly fast at this operation.

```
sage: set_random_seed(0)
```

```
sage: A = random_matrix(QQ, 200)
sage: f = A.charpoly()  # a second or so
```

On my laptop, Magma and Sage both take 0.7 seconds to compute this characteristic polynomial. Mathematica takes 338 seconds (nearly 6 minutes).

```
sage: len(str(f))    # about 5-10 typed pages?
35823
```

Sage can also compute the kernel (the nullspace) and the image (column space) of a matrix.

```
sage: A = matrix(QQ, 3, 4, [1..12]); A
[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]
```

The right kernel V of A is the *vector space* of all vectors x such that $Ax = 0$. (The left kernel is the space of those vectors with $xA = 0$.)

```
sage: V = A.right_kernel(); V
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -3  2]
[ 0  1 -2  1]
sage: V.basis()   # vectors always get written as row vectors
[
(1, 0, -3, 2),
(0, 1, -2, 1)
]
sage: for v in V.basis(): print A * v
(0, 0, 0)
(0, 0, 0)
```

If you know linear algebra, you'll know that the echelon form of a matrix is used to compute the kernel.

```
sage: A.echelon_form()
[ 1  0 -1 -2]
[ 0  1  2  3]
[ 0  0  0  0]
```

The column space (or image) of A (viewed as acting from the right) is the vector space of linear combinations of the colums of A:

```
sage: V = A.column_space(); V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: V.basis()
[
(1, 0, -1),
```

```
(0, 1, 2)
]
```

10.2.4 Vector Spaces

When we computed the kernel of (the linear transformation defined by) a matrix above, the result is a vector space, which is a certain set of vectors. There is a class in Sage that represents such objects. For example, the vector space $\mathbb{Q}^3$ is the set of all 3-tuples of rational numbers:

```
sage: V = QQ^3; V
Vector space of dimension 3 over Rational Field
```

Let's construct two of the coordinate planes as subspaces of V.

```
sage: Wxy = V.span([ (1,0,0), (0,1,0) ]); Wxy
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
sage: Wyz = V.span([ (0,1,0), (0,0,1) ]); Wyz
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[0 1 0]
[0 0 1]
```

We can compute in Sage the *intersection* of these two subspaces, which is geometrically the y axis:

```
sage: Wxy.intersection(Wyz)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
```

We can also compute the sum, which is the set of all sums $v + w$, where $v \in W_{xy}$ and $w \in W_{yz}$.

```
sage: Wxy + Wyz
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

If we want to consider a subspace W spanned by a particular list of vectors with that basis, use the `span_of_basis` method.

```
sage: W = V.span_of_basis([ (1,2,3),  (4,5,6) ]); W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 3]
[4 5 6]
```

```
sage: W.basis()
[
(1, 2, 3),
(4, 5, 6)
]
```

Given a vector we can ask if it is in W or not, and if so, ask for its coordinates in terms of our basis for W.

```
sage: x = V([1,8,5])
sage: x in W
False
sage: x = V([7,8,9])
sage: x in W
True
sage: W.coordinates(x)
[-1, 2]
sage: # sometimes getting a vector back is more useful
sage: W.coordinate_vector(x)
(-1, 2)
```

We can also define linear transformations (lienar maps) between vector spaces by specifying where each basis vector goes.

```
sage: phi = Hom(W, V)([3*V.1 - V.2, V.2 - 3*V.1]); phi
Free module morphism defined by the matrix
[ 0  3 -1]
[ 0 -3  1]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
User ...
Codomain: Vector space of dimension 3 over Rational Field
```

Let's apply this linear transformation φ to some vectors:

```
sage: W.0
(1, 2, 3)
sage: phi(W.0)
(0, 3, -1)
sage: phi(W.1)
(0, -3, 1)
sage: phi(W.0 + W.1)
(0, 0, 0)
sage: phi.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[  1 7/5 9/5]
sage: phi.image()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[   0    1 -1/3]
```

Chapter 11

Databases

In this chapter, we will explain how to store and manipulate data that arises when using Sage.

Good news! You're using Sage, hence Python, and there is a huge range of excellent database technology available. Many object oriented, relational, and noSQL databases have excellent Python interfaces and support, and the Python language supports object serialization. With Sage you have far more powerful and scalable tools available for storing data to disk, indexing it, and manipulating it, than with any other *mathematics* software platform out there.

The main topics we will discuss in this chapter are pickling Python objects, using the filesystem to write and read files, and using SQLite (which is included with Sage) to create a database.

11.1 Saving and Loading Python Objects

11.1.1 save and load

The `save` and `load` commands are the most important thing you will learn in this section. Everything else in this section just enhances your depth of understanding.

First we make a complicated object Sage object, consisting of a list with entries a pair of a rational and int, then a matrix, and finally a symbolic expression.

```
sage: A = [(2/3, int(5)), matrix(QQ, 1, 4, [1,2,-5/3,8]), sin(x^2)]
```

You can save this one object to a file on disk:

```
sage: save(A, '/tmp/A.sobj')
```

You can then load it back from disk:

```
sage: load('/tmp/A.sobj')
[(2/3, 5), [   1    2 -5/3    8], sin(x^2)]
```

Finally, we should cleanup our "mess":

```
sage: os.unlink('/tmp/A.sobj')
```

In the notebook, you can also just save A to the current cell, then click to download it to your computer, and possibly load it into another copy of Sage elsewhere.

```
sage: save(A, 'A.sobj')
```

The rest of this section will give you a bit more depth of understanding about how this works.

11.1.2 pickle: Python object serialization

The `save` and `load` commands from Section 11.1.1 above are implemented using Python's pickling mechanism. *Pickling* refers to turning almost any object X into a single string `s`. You can then save `s` somewhere, and (hopefully) load it later. This process is known as object serialization (see `http://en.wikipedia.org/wiki/Serialization`), and is also very important for parallel distributed computation.

To illustrate pickling, first we create the Python `int` 2011, and turn it into a string using the `dumps` function that is defined in the builtin Python `pickle` module.[1]

```
sage: import pickle
sage: s = pickle.dumps(int(2011))
sage: s
'I2011\n.'
sage: type(s)
<type 'str'>
sage: print s
I2011
.
```

The `loads` function turns our pickled string `s` back into an object:

```
sage: n = pickle.loads(s); n
2011
sage: type(n)
<type 'int'>
```

The `explain_pickle` command, which was written for Sage by Carl Witty, attempts to produce Sage code that, when evaluated *in Sage*, produces the same result as unpickling the pickle.

```
sage: explain_pickle(s)
2011r
```

Next, let's pickle a more complicated data structure:

```
sage: s = pickle.dumps([20r, long(11)]); s
'(lp0\nI20\naL11L\na.'
sage: print s
(lp0
I20
aL11L
```

[1]There is also a `cPickle` module in Python that is a faster version of pickle, and is supposed to be a drop in replacement.

```
a.
sage: explain_pickle(s)
[20r, long(11)]
sage: pickle.loads(s)
[20, 11L]
```

Pickling also deals sensibly with references, e.g., in the following notice that the integer n is only pickled once, not 5 times:

```
sage: n = 93574
sage: v = [n,n,n,n,n]; s = pickle.dumps(v); s
"(lp0\ncsage.rings.integer\nmake_integer\np1\n(S'2rc6'\np2\ntp3\nRp4\nag4\nag4\nag4\nag4
sage: explain_pickle(s)
pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
si = pg_make_integer('2rc6')
[si, si, si, si, si]
```

You might notice in the above he pickle of a Sage integer is even more complicated, since the pickle stores the callable that can be used to recreate the integer, along with binary data that efficiently represents the integer (*not* in base 10!). The representation is not in a base 10, since base conversion is potentially slow, and all numbers are stored internally in base 2.

```
sage: s = pickle.dumps(2011); s
"csage.rings.integer\nmake_integer\np0\n(S'1ur'\np1\ntp2\nRp3\n."
sage: print s
csage.rings.integer
make_integer
p0
(S'1ur'
p1
tp2
Rp3
.
sage: explain_pickle(s)
pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
pg_make_integer('1ur')
```

How fast is pickling and unpickling a big Sage integer?

```
sage: n = ZZ.random_element(10^1000)  # a 1000 digit Sage Integer
sage: timeit('s = pickle.dumps(n)')
sage: s = pickle.dumps(n)
sage: timeit('k = pickle.loads(s)')
625 loops, best of 3: 45.9  s per loop
625 loops, best of 3: 34.4  s per loop
```

It takes much longer (ten times longer!) to pickle a Python int. Part of this might be base 2 to base 10 conversion overhead?

```
sage: n = int(n) # same 1000 digit Python int
sage: timeit('s = pickle.dumps(n)')
sage: s = pickle.dumps(n)
```

```
sage: timeit('k = pickle.loads(s)')
625 loops, best of 3: 476   s per loop
625 loops, best of 3: 72.9   s per loop
```

References to Other Math Software

Not every object can be serialized in Sage. For example, as we discussed in Chapter 6, some Sage objects are wrappers around objects defined in another mathematical software package, e.g., Maxima, Singular, GAP, Magma, Mathematica, etc. In some case, such objects are difficult or impossible serialize. However, in most cases math software does provide some form of serialization of objects, and in some cases Sage automatically makes use of it. For example,

```
sage: import pickle; s = pickle.dumps(a); s
"csage.interfaces.expect\nreduce_load\np0\n(csage.interfaces.gp\nreduce_load_GP\np1\n(tR
sage: pickle.loads(s)
[1, 2/3, 1.5000000000000000000000000000000000]
```

In GP/PARI, object data structures are all fairly straightforward, so the print representation of most objects can simply be evaluated to get them back using the `eval` command.

In Magma, object data structures are very complicated and there is no way to serialize most of them (as far as the author knows). There also was no `eval` command in Magma until fairly recently, but fortunately there is one now. (On very simple input, the `eval` in Magma is roughly 10 times slower to call than the `eval` command in PARI and Python, so watch out.)

You can also pickle objects of classes you define...

```
class Foo:
    def __init__(self, x):
        self.x = x
    def __repr__(self):
        return 'Foo x=%s'%self.x
```

```
sage: f = Foo('2010')
sage: s = pickle.dumps(f); s
"(i__main__\nFoo\np0\n(dp1\nS'x'\np2\nS'2010'\np3\nsb."
sage: C = pickle.loads(s); type(C)
<type 'instance'>
sage: C
Foo x=2010
```

BIG FAT WARNING: The *code* of the Python modules (code or compiled .so's) that define the objects is *NOT* stored in the pickled form of the object. (This is pretty obvious with the integer example above!) If the relevant Python modules don't exist in the right place, then the pickle will simply be broken.

This means that if somebody decides to rename or move some code in Sage, it can easily render pickles useless. So be careful. We do have something called "the pickle jar", which helps ensure that in Sage itself this doesn't cause too much trouble. This large "pickle jar" contains hundreds of objects, and testing that they unpickle is part of Sage's test suite.

Example: All of the state of the Sage notebook used to be stored as pickles of Python classes that are part of the source code of the notebook. I wanted to move the code of the Sage notebook out of the Sage library, and make the notebook a separate project. This was nearly impossible because of how I had designed those pickles. Tim Dumol and I spent over a week writing and testing code to load notebook pickles, them convert the data structures to very simple data structures (e.g., dictionaries, strings) that didn't use any special classes, then resave them. The resulting new saved pickles can be read by any Python independently of Sage or the notebook. This makes it possible to move the notebook code out of the Sage library. However, it is still there (just waiting to confuse you!), in case somebody tries to load an old Sage Notebook instance using a new version of Sage, since we want to migrate the old notebook pickles to the new format. (This code and capability will be removed soon, since it was over a year ago that the notebook was removed from the Sage library.)

Customization: You can fully customize how any class gets pickled, including Cython classes (where you pretty much have to customize them). This can make pickling more robust and potentially faster. Also, careful thought about customizing how objects get pickled can make them more robust in case you change your mind later (the matrix code in Sage is particularly good this way). The example below illustrates how two seemingly similar classes can have massively difference pickling performance, depending on whether somebody cared to write some fast pickling code.

Moral: For longterm use of data, using pickles is very dangerous and should be avoided if possible. For shortterm use (over the course of a few minutes, weeks or months), using pickles is incredibly useful. Think of pickles like a jar of pickles that you buy from the store (and open). They have to be refrigerators and they have an expiration date. But they last a while.

```
sage: A = random_matrix(Integers(10^100), 200)
sage: time s = pickle.dumps(A)
Time: CPU 6.26 s, Wall: 6.26 s
```

Here B is exactly the same matrix as A, except the entries are viewed as being in $\mathbb{Z}$ instead of $\mathbb{Z}/10^{100}\mathbb{Z}$. Yet it pickles 60 times more quickly (somebody should fix this!).

```
sage: B = A.change_ring(ZZ)
sage: time t = pickle.dumps(B)
Time: CPU 0.11 s, Wall: 0.11 s
sage: 6.26/.11
56.9090909090909
```

Pickles in Sage

Sage has some convenience functions for working with pickles:

load, save, loads, dumps

There is also `save` and `dumps` method on any classes that derives from SageObject.

The main thing that the load/save/loads/dumps functions in Sage do, over the pickle methods, is they transparently by default do *in memory zlib compression*. Also, save and load combine pickling with actually writing the pickle string out to a file. Also, load can load many other types of objects, for example load a pickle off of a webpage. We illustrate all this below.

```
sage: A = matrix(ZZ, 4, 20, [1..80]); A
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]
[41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60]
[61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80]
sage: len(pickle.dumps(A))
489
sage: # the sage dumps method compresses by default -- here we get a factor of 2 savings
sage: len(dumps(A))
282
```

Of course, the compressed version is unreadable to the eye since it is zlib compressed:

```
sage: print dumps(A)
xmN...
<p>Compared to:</p>
sage: print pickle.dumps(A)
csage.matrix.matrix0
unpickle
p0
(csage.matrix.matrix_integer_dense
Matrix_integer_dense
p1
csage.matrix.matrix_space
MatrixSpace
p2
(csage.rings.integer_ring
IntegerRing
p3
(tRp4
I4
I20
I00
tp5
Rp6
csage.structure.mutability
Mutability
p7
(I00
tp8
Rp9
(dp10
S'1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v 10 11 12 13 14 15 16 17
p11
I0
tp12
Rp13
.
```

```
<p>loads can parse both the compressed and uncompressed pickles (it figures out which is right
```

```
sage: loads(dumps(A))
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]
[41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60]
[61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80]
sage: loads(pickle.dumps(A))
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]
[41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60]
[61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80]
<p>Compression has a performance penalty:</p>
sage: timeit('loads(dumps(A))')
625 loops, best of 3: 192 s per loop
sage: timeit('loads(dumps(A,compress=False), compress=False)')
625 loops, best of 3: 130 s per loop
<p>We can save a pickle to a file and load it from a file:</p>
sage: save(A, 'A.sobj')
sage: save(A, '/tmp/A.sobj')
34
sage: load('/tmp/A.sobj')
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]
[41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60]
[61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80]
sage: os.unlink('/tmp/A.sobj')  # clean up
<p>We can load a pickle from a webpage too, which is pretty cool:</p>
sage: X = load('http://wiki.wstein.org/11/480a/5-25?action=AttachFile&do=get&target=A.sobj')
sage: X
Attempting to load remote file: http://wiki.wstein.org/11/480a/5-25?action=AttachFile&do=get&ta
Loading: [.]
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]
[41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60]
[61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80]
sage: X = load('http://wiki.wstein.org/11/480a/5-25?action=AttachFile&do=get&target=A.sobj', ve
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
[21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]
[41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60]
[61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80]
<p><strong>Conclusion: </strong></p>
<ul>
<li>Understanding object serialization is useful if you do some research computations, and want
<li>It requires very little thought to use.  <strong>save(obj, 'filename.sobj') </strong>and <s
<li>You could make a simple "database" that anybody can easily use over the web by: (1) putting
</ul>
<h2>Opening Files</h2>
<p>If you want to store a plain string to disk, and load it later, it is critical to master the
sage: file = open('/tmp/file', 'w'); file
<open file '/tmp/file', mode 'w' at 0x456f470>
sage: file.write("This is a line.")
```

```
sage: file.close()
sage: open('/tmp/file').read()
'This is a line.'
sage: file = open('/tmp/file'); file
<open file '/tmp/file', mode 'r' at 0x4b85ad0>
sage: file.seek(3)
sage: file.read(4)
's is'
sage: file.seek(0)
sage: file.read()
'This is a line.'
sage: file.close()
sage: os.unlink('/tmp/file')
<p>One can do a lot with a file, or a bunch of files in a directory.  Don't use a sophisticated
<p> </p>
<h2>Pickling + Files: @disk_cached_function</h2>
<p>Here's a nice decorator (written by Tom Boothby) that combines files with pickling.  </p>
sage: disk_cached_function?
<html><!--notruncate-->

<div class="docstring">

  <p><strong>File:</strong> /sagenb/flask/sage-4.6.2/local/lib/python2.6/site-packages/sage/mis
<p><strong>Type:</strong> &lt;type &#8216;classobj&#8217;&gt;</p>
<p><strong>Definition:</strong> disk_cached_function(f)</p>
<p><strong>Docstring:</strong></p>
<blockquote>
<p>Decorator for <tt class="xref py py-class docutils literal"><span class="pre">DiskCachedFunc
<p>EXAMPLES:</p>
<div class="highlight-python"><div class="highlight"><pre class="literal-block"><span class="gp
<span class="gp">sage: </span><span class="nd">@disk_cached_function</span><span class="p">(</s
<span class="gp">... </span><span class="k">def</span> <span class="nf">foo</span><span class="
<span class="gp">sage: </span><span class="n">x</span> <span class="o">=</span> <span class="n"
<span class="go">11</span>
<span class="gp">sage: </span><span class="nd">@disk_cached_function</span><span class="p">(</s
<span class="gp">... </span><span class="k">def</span> <span class="nf">foo</span><span class="
<span class="gp">sage: </span><span class="n">foo</span><span class="p">(</span><span class="mi
<span class="go">11</span>
<span class="gp">sage: </span><span class="n">foo</span><span class="o">.</span><span class="n"
<span class="gp">sage: </span><span class="n">foo</span><span class="p">(</span><span class="mi
<span class="go">1/200</span>
</pre></div>
</div>
</blockquote>


</div>
</html>
sage: if os.path.exists('/tmp/factor_cache'):
...      import shutil
```

```
...      shutil.rmtree('/tmp/factor_cache')
...
...
sage: @disk_cached_function('/tmp/factor_cache')
sage: def my_factor(n):
...       return factor(n)
sage: time my_factor(2^157+1)
3 * 15073 * 2350291 * 17751783757817897 * 96833299198971305921
Time: CPU 0.08 s, Wall: 0.08 s
sage: time my_factor(2^157+1)
3 * 15073 * 2350291 * 17751783757817897 * 96833299198971305921
Time: CPU 0.00 s, Wall: 0.00 s
sage: os.listdir('/tmp/factor_cache')
['my_factor-182687704666362864775460604089535377456991567873.sobj', 'my_factor-1826877046663628
sage: time my_factor(2^157+3)
5^3 * 557 * 2623880856967509727475197186205175977838299
Time: CPU 0.02 s, Wall: 0.02 s
sage: os.listdir('/tmp/factor_cache')
['my_factor-182687704666362864775460604089535377456991567875.sobj', 'my_factor-1826877046663628
sage: load('/tmp/factor_cache/%s'%os.listdir('/tmp/factor_cache')[0])
(((182687704666362864775460604089535377456991567875,), ()), 5^3 * 557 * 262388085696750972747 51
sage: load('/tmp/factor_cache/%s'%os.listdir('/tmp/factor_cache')[1])
((182687704666362864775460604089535377456991567875,), ())
<p>Clean our mess:</p>
sage: import shutil
sage: shutil.rmtree('/tmp/factor_cache')
<p><strong>Summary</strong>:</p>
<ul>
<li><strong>save/load:</strong> If you remember nothing else from today's lecture, remember the
<li><strong>open</strong>: It is easy to open and write to and read from files in Python.</li>
<li><strong>disk_cached_function:</strong> provides a function decorator that makes a function
</ul>
<h2>Next:</h2>
<ul>
<li><a href="http://www.sqlite.org/" target="_blank">SQLite</a>: a <em>relational database </em
<li>(Maybe) <a href="http://www.sqlalchemy.org/" target="_blank">SQLalchemy</a>: an <em>object
</ul>
```

11.2 SQLite and SQLAlchemy

```
<p><strong>Using SQLite in Sage</strong></p>
<p>Check out <a href="http://www.sqlite.org/" target="_blank">the SQLite website.</a>  &nb
<ul>
<li>SQLite is surely the most widely deployed database in the world, in some sense.</li>
<li>SQLite is vastly simpler to use and administer than pretty much all other databases.</li>
<li>SQLite is extremely fast (if used correctly). </li>
<li>SQLite is <strong>public domain.  </strong>You can do absolutely anything you want wit
<li>Every copy of Sage comes with SQLite.</li>
<li>Learning about SQLite may server you well in non-Sage related projects, since it can be use
</ul>
```

```
<p> </p>
<p>Here's a complete example of using SQLite to make a database of integer factorizations.</p>
sage: # sqlite3 is a standard Python module
sage: import sqlite3
sage: # Make sure the database file isn't left over from a previous demo...
sage: file = '/tmp/sqlite0'
sage: if os.path.exists(file):
...       os.unlink(file)
sage: # open the database file -- zero configuration!
sage: db = sqlite3.connect(file)
sage: # get a "cursor"
sage: cursor = db.cursor()
sage: # start executing SQL commands
sage: cursor.execute("""CREATE TABLE factorizations
...            (number INTEGER, factorization TEXT, UNIQUE(number))""")
...
sage: cursor.execute("CREATE INDEX factorizations_idx ON factorizations(number)")
sage: # commit our changes -- SQL uses transactions
sage: db.commit()
sage: t = ('6', '[(2,1),(3,1)]')
sage: cursor.execute('INSERT INTO factorizations VALUES(?,?)', t)
<sqlite3.Cursor object at 0x4846298>
sage: db.commit()
<p>We can look at our new database on the command line, completely independently of Sage/Python
<pre><span style="background-color: #ffff99;">boxen:~ wstein\$ sage -sh
(sage subshell)\$ sqlite3 /tmp/sqlite1
SQLite version 3.4.2
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite&gt; .schema
CREATE TABLE factorizations
         (number TEXT, factorization TEXT, UNIQUE(number));
CREATE INDEX factorizations_idx ON factorizations(number);
sqlite&gt; select * from factorizations;
6|[(2,1),(3,1)]</span>
</pre>
<p>By the way, the UNIQUE above makes it so you can't enter another factorization of the same n
sage: t = ('6', '[(2,1),(3,1)]')
sage: cursor.execute('INSERT INTO factorizations VALUES(?,?)', t)
Traceback (most recent call last):
...
sqlite3.IntegrityError: column number is not unique
sage: %time
sage: for n in range(1,10000):
...       f = str(list(factor(n))).replace(' ','')
...       try:
...           t = (str(n), f)
...           z = cursor.execute('INSERT INTO factorizations VALUES(?,?)', t)
...       except:
...           print "Unable to insert factorization of %s"%n
```

```
Unable to insert factorization of 6
CPU time: 0.63 s,  Wall time: 0.63 s
sage: time db.commit()
Time: CPU 0.01 s, Wall: 0.00 s
sage: a = cursor.execute('SELECT * FROM factorizations ORDER BY number;')
sage: i = 0
sage: for x in a:
...       print x
...       i += 1
...       if i>10: break
(1, u'[]')
(2, u'[(2,1)]')
(3, u'[(3,1)]')
(4, u'[(2,2)]')
(5, u'[(5,1)]')
(6, u'[(2,1),(3,1)]')
(7, u'[(7,1)]')
(8, u'[(2,3)]')
(9, u'[(3,2)]')
(10, u'[(2,1),(5,1)]')
(11, u'[(11,1)]')
<p>We use the command line again (we <strong><em>do not</em></strong> have to exit or reload!)
<pre>sqlite&gt; SELECT * FROM factorizations where number&lt;10;
1|[]
2|[(2,1)]
3|[(3,1)]
4|[(2,2)]
5|[(5,1)]
6|[(2,1),(3,1)]
7|[(7,1)]
8|[(2,3)]
9|[(3,2)]</pre>
<p>Obviously, to use SQLite effectively, it helps enormously to know the SQL language.   F
<p> </p>
<p>Python documentation for the sqlite3 module: <a href="http://docs.python.org/library/sqlite3
<h2 style="text-align: center; ">SQLAlchemy</h2>
<p>Next we'll spend a few moments on <a href="http://www.sqlalchemy.org/" target="_blank">SQLAl
<ul>
<li>SQLAlchemy is the <strong><em>canonical</em></strong> "object relational database mapper" f
<li>SQLAlchemy abstracts away the database backend, so the same code/application can work with
<li>SQLAlchemy has a large test suite, good documentation, and is a high quality polished produ
<li>SQLAlchemy is MIT licensed (so very open source)</li>
</ul>
<p> </p>
<p><strong>WARNING:</strong> As of this writing (May 27, 2011) the version of SQLAlchemy in the
sage: import sqlalchemy
sage: sqlalchemy.__version__
'0.5.8'
<p>We will use the file /tmp/sqlite1 for our demo.  Make sure it is deleted.</p>
sage: file = '/tmp/sqlite1'
```

```
sage: if os.path.exists(file):
...       os.unlink(file)
<p>Create a SQLite engine, which SQLalchemy will use.  This is the only place below that S
sage: from sqlalchemy import create_engine
sage: engine = create_engine('sqlite:///%s'%file)  #, echo=True)
<p>Use SQLalchemy to declare a new Python class, which will get mapped to a table in the above
sage: from sqlalchemy.ext.declarative import declarative_base
sage: from sqlalchemy import Column
sage: Base = declarative_base()
sage: class IntFac(Base):
...
...       __tablename__ = 'factorizations'
...       number = Column(sqlalchemy.Integer, primary_key=True)
...       factorization = Column(sqlalchemy.String)
...
...       def __init__(self, number):
...           self.number = int(number)
...           self.factorization = str(list(factor(number))).replace(' ','')
...
...       def __repr__(self):
...           return '%s: %s'%(self.number, self.factorization)
<p>Make a particular session that connects to the database.</p>
sage: from sqlalchemy.orm import sessionmaker
sage: session = sessionmaker(bind=engine)()
<p>Create the tables.  In this case, there is exactly one, which corresponds to the IntFac
sage: Base.metadata.create_all(engine)
<p>Now create an integer factorization object.</p>
sage: f = IntFac(6); f
6: [(2,1),(3,1)]
<p>And add it to our session, so it will get tracked by the database.</p>
sage: session.add(f)
<p>Commit everything we have done so far.  After this commit, the database exists separate
sage: session.commit()
<pre>wstein@boxen:/tmp\$ ls -lh /tmp/sqlite1
-rw-r--r-- 1 sagenbflask sagenbflask 2.0K 2011-05-27 13:46 /tmp/sqlite1
wstein@boxen:/tmp\$ sqlite3 /tmp/sqlite1
SQLite version 3.4.2
Enter ".help" for instructions
sqlite&gt; .schema
CREATE TABLE factorizations (
number INTEGER NOT NULL,
factorization VARCHAR,
PRIMARY KEY (number)
);
sqlite&gt; select * from factorizations;
6|[(2,1),(3,1)]
</pre>
<p>We try a query on the session:</p>
sage: session.query(IntFac).first()
6: [(2,1),(3,1)]
```

```
<p>We try adding the factorization of 6 again.  This should give an error because number i
sage: session.add(IntFac(6))
sage: session.commit()
Traceback (most recent call last):
...
sqlalchemy.orm.exc.FlushError: New instance <IntFac at 0x596d790> with identity key (<class ’__
<p>Once an error occurs the only option is to rollback the whole transaction.</p>
sage: session.rollback()
<p>Let’s make a few thousand factorization (like we did above) and include them all in one tran
sage: time v = [IntFac(n) for n in [1..5] + [7..10000]]
Time: CPU 1.98 s, Wall: 1.98 s
<p>Using add_all should be more efficient than calling add many times. </p>
sage: time session.add_all(v)
Time: CPU 0.35 s, Wall: 0.36 s
sage: time session.commit()
Time: CPU 6.59 s, Wall: 6.59 s
<p>Now we have factorizations of all integers up to 10000.  We can do a query like above.<
sage: for X in session.query(IntFac).filter(’number<10’):
...       print X
1: []
2: [(2,1)]
3: [(3,1)]
4: [(2,2)]
5: [(5,1)]
6: [(2,1),(3,1)]
7: [(7,1)]
8: [(2,3)]
9: [(3,2)]
<p>And, we can do the same on the command line:</p>
<pre>sqlite&gt; select * from factorizations where number&lt;10;
1|[]
2|[(2,1)]
3|[(3,1)]
4|[(2,2)]
5|[(5,1)]
6|[(2,1),(3,1)]
7|[(7,1)]
8|[(2,3)]
9|[(3,2)]
</pre>
```

[[TODO: Add something about storing BLOBS = pickled objects in a database, e.g., my key:value store demo from 580d.]]

Bibliography