Afterword

Why Python?

Advantages of Python

The primary implementation language of Sage is Python (see [Py]), though code that must be fast is implemented in a compiled language. Python has several advantages:

- **Object saving** is well-supported in Python. There is extensive support in Python for saving (nearly) arbitrary objects to disk files or a database.
- Excellent support for **documentation** of functions and packages in the source code, including automatic extraction of documentation and automatic testing of all examples. The examples are automatically tested regularly and guaranteed to work as indicated.
- **Memory management**: Python now has a well thought out and robust memory manager and garbage collector that correctly deals with circular references, and allows for local variables in files.
- Python has **many packages** available now that might be of great interest to users of Sage: numerical analysis and linear algebra, 2D and 3D visualization, networking (for distributed computations and servers, e.g., via twisted), database support, etc.
- **Portability:** Python is easy to compile from source on most platforms in minutes.
- **Exception handling:** Python has a sophisticated and well thought out system of exception handling, whereby programs gracefully recover even if errors occur in code they call.
- **Debugger:** Python includes a debugger, so when code fails for some reason, the user can access an extensive stack trace, inspect the state of all relevant variables, and move up and down the stack.
- **Profiler:** There is a Python profiler, which runs code and creates a report detailing how many times and for how long each function was called.
- **A Language:** Instead of writing a **new language** for mathematics as was done for Magma, Maple, Mathematica, Matlab, GP/PARI, GAP, Macaulay 2, Simath, etc., we use the Python language, which is a popular computer language that is being actively developed and optimized by hundreds of skilled software engineers. Python is a major open-source success story with a mature development process (see [PyDev]).

The Pre-Parser: Differences between Sage and Python

Some mathematical aspects of Python can be confusing, so Sage behaves differently from Python in several ways.

- **Notation for exponentiation:** ** versus ^. In Python, ^ means "xor", not exponentiation, so in Python we have

```
>>> 2^8
10
>>> 3^2
1
>>> 3**2
9
```

This use of ^ may appear odd, and it is inefficient for pure math research, since the "exclusive or" function is rarely used. For convenience, Sage pre-parses all command lines before passing them to Python, replacing instances of ^ that are not in strings with **:

```
sage: 2^8
256
sage: 3^2
9
sage: "3^2"
'3^2'
```

The bitwise xor operator in Sage is ^^. This also works for the inplace operator ^^=:

```
sage: 3^^2
1
sage: a = 2
sage: a ^^= 8
sage: a
10
```

- **Integer division:** The Python expression `2/3` does not behave the way mathematicians might expect. In Python, if `m` and `n` are ints, then `m/n` is also an int, namely the quotient of `m` divided by `n`. Therefore `2/3=0`. There has been talk in the Python community about changing Python so `2/3` returns the floating point number `0.6666...`, and making `2//3` return `0`.

  We deal with this in the Sage interpreter, by wrapping integer literals in `Integer( )` and making division a constructor for rational numbers. For example:

```
sage: 2/3
2/3
sage: (2/3).parent()
Rational Field
sage: 2//3
0
sage: int(2)/int(3)
0
```

- **Long integers:** Python has native support for arbitrary precision integers, in addition to C-int's. These are significantly slower than what GMP provides, and have the property that they print with an `L` at the end to

distinguish them from int's (and this won't change any time soon). Sage implements arbitrary precision integers using the GMP C-library, and these print without an `L`.

Rather than modifying the Python interpreter (as some people have done for internal projects), we use the Python language exactly as is, and write a pre-parser for IPython so that the command line behavior of IPython is what a mathematician expects. This means any existing Python code can be used in Sage. However, one must still obey the standard Python rules when writing packages that will be imported into Sage.

(To install a Python library, for example that you have found on the Internet, follow the directions, but run `sage -python` instead of `python`. Very often this means typing `sage -python setup.py install`.)

I would like to contribute somehow. How can I?

If you would like to contribute to Sage, your help will be greatly appreciated! It can range from substantial code contributions to adding to the Sage documentation to reporting bugs.

Browse the Sage web page for information for developers; among other things, you can find a long list of Sage-related projects ordered by priority and category. The Sage Developer's Guide has helpful information, as well, and you can also check out the `sage-devel` Google group.

How do I reference Sage?

If you write a paper using Sage, please reference computations done with Sage by including

```
[Sage] William A. Stein et al., Sage Mathematics Software (Version 4.3).
       The Sage Development Team, 2009, http://www.sagemath.org.
```

in your bibliography (replacing 4.3 with the version of Sage you used). Moreover, please attempt to track down what components of Sage are used for your computation, e.g., PARI?, GAP?, Singular? Maxima? and also cite those systems. If you are in doubt about what software your computation uses, feel free to ask on the `sage-devel` Google group. See *Univariate Polynomials* for further discussion of this point.

If you happen to have just read straight through this tutorial, and have some sense of how long it took you, please let us know on the `sage-devel` Google group.

Have fun with Sage!