The Python GMP Module

An efficient BIGNUM module for Python
Edition 1.0
For version 0.95 of the `gmp` module
Based on version 2.0 of the GMP library
20 August 1996

by Niels Mller

The Python GMP module

1 Introduction

The python module gmp is an interface to GMP library routines for arbitrary large integer numbers. The library can also handle rational numbers and arbitrary precision floats, but this module can not.

Comments and bug-reports can be sent to the author, Niels Mller <nisse@lysator.liu.se>.

The manual is written for gmpmodule version 0.95.

1.1 Why use GMP?

There's mainly two reasons you may want to use gmp integer objects instead of ordinary long python integers.

- In many cases, gmp is much more efficient. For example modular exponentiation is an order of magnitude faster with GMP on the machines I have tried.
- The gmp module contains some additional functions, for example fac, gcdext and is_prime that you may find handy.

2 Installing

I assume that you already have Python installed. The instructions in this document describes installation with `python` version 1.3, but later versions should not be very different.

First, you have to install the GNU Multiple Precision Library, version 2.0 or better. As of this writing, the latest version is 2.0.2, avilable at any GNU mirror, for example as '`ftp://ftp.lysator.liu.se/pub/gnu/gmp-2.0.2.tar.gz`'. Unpack it and follow the installation instructions.

Copy the file '`gmpmodule.c`' into the '`Modules`' directory in the Python source tree. Then you have to edit the '`Setup`' file in the same directory. Add two lines

```
GMP=/usr/local           # directory prefix used when installing GMP
gmp gmpmodule.c -I$(GMP)/include -L$(GMP)/lib -lgmp
```

It's a good idea to comment out the obsolete `mpz` module from the '`Setup`' file at the same time.

3 Basic GMP usage

3.1 Construction

There's only one constructor, `mpz`, that can take different types and number of arguments.

mpz, *number* Function
mpz, *digits, base* Function
mpz, *string* Function

Possible arguments are

- *number* may be an `int`, `long` or a `mpz` object.
- *digits* is a string of digits in *base*. Possible bases are the numbers 2 – 36, inclusive.
- *string* is a string interpreted as digits in base 256, least significant octet first.

For the reverse operations, use the `digits()` and `binary()` methods (see Chapter 4 [Methods], page 5).

3.2 Basic arithmetic

The standard arithmetic operations can be used for `mpz` numbers the same way as ordibary python integers. Arguments are converted to `mpz` objects as needed. The operations are `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `abs`, `divmod`, `int`, `hex`, `long`, `oct`, `pow`, `repr` and `str`.

The formatting conventions for `oct` *and* `hex` *are non-optimal, and might change in the future.* I recommend using the `digits()` method instead.

4 More methods and functions

Some of the features of the `gmp` module are available as methods of `mpz` objects, some as functions, and many as both methods and functions. In the latter case, arguments are converted to `mpz` objects when needed.

binary *number* Function
binary Method

Converts the number to a string of base 256 digits, least significant octet first. The returned value is suitable as argument to the constructor `mpz` (see Chapter 3 [Basic usage], page 4).

digits (*number* [*base*]) Function
digits [*base*] Method

Converts the number to a string of digits, in the indicated base. Possible bases are 2 – 36 inclusive. *base* zero or omitted is equivalent to 10. This function is the inverse to the two arg form of the `mpz` constructor (see Chapter 3 [Basic usage], page 4).

divm *a b m* Function

Computes x such that $b * x = a$ (modulo m). If b is not invertible modulo m a `ZeroDivisionError` is raised.

fac *i* Function

Computes the factorial of i, where i must be a (short) int.

gcd *m n* Function

Returns the greatest common divisor of the numbers m and n.

gcdext *m n* Function

Returns the a tuple (g, s, t) such that g is the greatest common divisor of the numbers m and n, and $g = s * m + t * n$.

is_square *number* Function
is_square Method

Returns 1 if and only if *number* is a square, and 0 otherwise.

is_prime (*number* [*count*]) Function

is_prime [*count*] Method

Returns 1 if *number* is probably prime, and 0 otherwise. The meaning of *probably* depends on the *count* argument. Default is 25, and the larger the *count* value, the more thoroughly the *number* is investigated.

set_debug *level* Function

Set the debug level. The previous debug level is returned. Nonzero level turns debug output on, zero turns it off.

sqrt *number* Function

sqrt Method

Computes the greatest integer less than or equal to the square root of *number*, which must be non-negative.

sqrtrem *number* Function

sqrtrem Method

Similar to the previous function. Returns a tuple (*x*, *rem*), where *x* is greatest integer less than or equalt to the square root, and *rem* = *number* - *x*.

5 Index

Table of Contents