Using GMP in Python (gmpy2)

Jianlin Su

October 28, 2014

Previously, the author wrote "First Trial of Using PARI/GP in Python", which briefly introduced the method of calling PARI/GP in Python. PARI/GP is a relatively powerful number theory library, "designed for fast computations in number theory (factorization, algebraic number theory, elliptic curves...)." It can be called by programming languages like C/C++ or Python, and it is also a self-contained scripting language. However, if one only needs high-precision large number arithmetic, GMP seems to better meet our needs.

Readers familiar with C/C++ will know GMP (full name: GNU Multiple Precision Arithmetic Library). It is an open-source high-precision arithmetic library that includes not only high-precision operations for ordinary integers, real numbers, and floating-point numbers but also random number generation. In particular, it provides a very complete set of arithmetic interfaces for number theory, such as the Miller-Rabin primality test algorithm, large prime generation, the Euclidean algorithm, finding the inverse of elements in a field, Jacobi symbols, Legendre symbols, etc.[Source]. Although calling GMP in C/C++ is not overly complex, being able to use GMP in Python, which is known for its high development efficiency, is undoubtedly a pleasant prospect. This is exactly what gmpy2 is for.

Introduction to gmpy2 gmpy2 is a Python extension library and a wrapper for GMP; its predecessor was gmpy. After adjustments and encapsulation by the author, the use of gmpy2 has been greatly simplified. According to my current experience, using gmpy2 in Python has at least the following advantages compared to using GMP in C/C++:

1. Simplified function names: The author has simplified a large number of GMP function names. For example, for probabilistic primality testing, the command in C/C++ is `mpz_probab_prime_p`, while in gmpy2, it is simply `is_prime`.
2. Convenient operator overloading: In C/C++, to add two `mpz` integers, we use `mpz_add(z_i, z_i, z_o)`. In gmpy2, we only need to use the `+` sign. More generally, adding an `mpz` integer to an `int` integer also only requires the `+` sign. Similar overloading exists for subtraction, multiplication, division, modulo, exponentiation, etc.

My study is not yet deep, so I can only offer these partial comments. Of course, as a call from a third-party language, the efficiency of gmpy2 will generally be lower than calling GMP directly in C/C++, but the difference is very small because gmpy2 is essentially a pre-compiled C library. For a detailed gmpy2 tutorial, please see:
`http://gmpy2.readthedocs.org/en/latest/`

Basic Usage This article only provides a brief introduction. The following code runs in Python 3.4. To initialize a large integer, you only need:

```python
import gmpy2
n=gmpy2.mpz(1257787) # Initialization
gmpy2.is_prime(n) # Probabilistic primality test
```

This is parallel to C/C++. In fact, the parameter in the parentheses can be an integer or a string. gmpy2 integrates not only the large integer `mpz` but also the high-precision floating-point number `mpfr`. The usage is parallel to C/C++. Below are some basic calculations:

```
a+2 # Sum, result is an mpz
a-2 # Difference, result is an mpz
a*2 # Product, result is an mpz
a/2 # Quotient, result is an mpfr
a//2 # Quotient, result is an mpz
a**2 # Square, result is an mpz
a%2 # Modulo, result is an mpz
```

The overloading of these operators is both intuitive and convenient. It goes without saying that replacing the number 2 with an `mpz` type variable is also possible. In fact, the operation `a+2` first converts 2 into an `mpz` 2 and then performs `mpz` addition.

Comparison with Python's Built-in Features Python itself supports high-precision large integer arithmetic, which is sufficient for cases where the numbers are relatively small. However, it is not "fast." Readers only need to run the following two lines of code separately:

```
gmpy2.mpz(1257787)**123456 # gmpy2 calculation
1257787**123456 # Python built-in calculation
```

to feel the difference (of course, if the reader's configuration is good, there might not be a significant difference; in that case, please increase the exponent by ten times. My laptop's configuration is not high, please excuse me ^_^).

Attempting Large Number Factorization Below is a piece of code I wrote for large number factorization:

```
from gmpy2 import *
import time
start=time.clock()
n = mpz(632812179102577425839184065 71)
x = mpz(2)
y = x**2 + 1
for i in range(n):
    p = gcd(y-x,n)
    if p != 1:
        print(p)
        break
    else:
        y=(((y**2+1)%n)**2+1)%n
        x=(x**2+1)%n
end=time.clock()
print(end-start)
```

This code calls gmpy2, and the algorithm used is the Pollard rho method, running in its simplest form without any optimization. On my laptop, this algorithm factorized:

$$632812179102577425839184065 71 = 125778791843321 \times 503115167373251$$

in 84 seconds (stopping once one factor was obtained). Of course, this is not a remarkable achievement; this number is factored instantaneously in Mathematica and PARI/GP. This script is purely for practice and testing Python's calls to gmpy2.

Regarding the steps of the Pollard rho method:

1. Given a composite number n, let $x_0 = 2, y_0 = x_0^2 + 1$;
2. Calculate $p = \gcd(x_0 - y_0, n)$. If p is not 1, then the result is a factor of n, stop;
3. If $p = 1$, then update $x_{n+1} = x_n^2 + 1, y_{n+1} = (y_n^2 + 1)^2 + 1$, and repeat step 2.

In order to reduce the amount of calculation, the calculation of x_n, y_n at each step involves a modulo n operation. The above steps are only the simplest part of the calculation and do not handle various possible exceptions, such as $n \mid (x_n - y_n)$, nor have certain details been optimized. Please use with caution.

Reference URL:
`http://hi.baidu.com/pytvzcnbuolrsue/item/ae714592f1fda8d87b7f010a`