First Attempt at Using PARI/GP in Python

Jianlin Su

July 22, 2014

BoJone likes Python very much and also loves number theory, so he enjoys playing with number theory using Python. Usually, I like to write some number theory functions myself; after all, Python supports large integer high-precision arithmetic, which is excellent. However, in many practical applications, I still hope to have a ready-made number theory function library to call. Previously, I tried the HugeCalc library from the Mathematics R&D Network, but due to various unfamiliarities, it came to nothing. Later, a user named Wuxin on the forum recommended PARI/GP. After a small trial, I successfully called it in Python. Now I no longer have to worry about number theory calculation problems in Python, haha

First, here is an official introduction to PARI/GP:

> PARI/GP is a widely used computer algebra system designed for fast computations in number theory (factorizations, algebraic number theory, elliptic curves...), but also contains a large number of other useful functions to compute with mathematical entities such as matrices, polynomials, power series, algebraic numbers, and many transcendental functions. PARI is also available as a C library to allow for even faster computations.
>
> Originally developed by Henri Cohen and his co-workers (Université Bordeaux I, France), PARI is now under the GPL and maintained by Karim Belabas with the help of many volunteers.
>
> - **PARI** is a C library, allowing for fast computations.
> - **gp** is an easy-to-use interactive shell giving access to the PARI functions.
> - **GP** is the name of the gp scripting language.
> - **gp2c**, the GP-to-C compiler, combines the best of both worlds by compiling GP scripts into the C language and transparently loading the resulting functions into gp. (gp2c-compiled scripts will typically run 3 to 4 times faster.) gp2c currently only understands a part of the GP language.

PARI/GP provides source code packages that can be compiled for use, and it also provides executable installation packages for Windows, which can be installed and used directly. This article only discusses calling PARI using Python on the Windows platform. It is estimated that it can be called similarly on Linux, but I have not personally tested it before writing this article. Calling it under Windows only requires the **libpari.dll** file in the PARI/GP installation directory. Readers can extract it after installing the package from the official website or download it from the attachment at the end of this article.

First, it should be noted that the Python version used to call PARI must be 32-bit. Both Python 2 and Python 3 are acceptable, but 64-bit will cause errors (you can install 32-bit Python on a 64-bit operating system). Of course, you can also use PyPy to call it. PyPy is based on 32-bit Python, and the latest version already supports Python 3; readers can choose

their favorite. Let's first look at the method in Python 2.x; the details in 3.x are slightly different and will be explained later.

A DLL is a Dynamic Link Library. After loading the DLL, you can use the functions inside it. This is somewhat similar to loading a module. The difference is that DLLs are generally written in C, so their execution efficiency is relatively high. Loading C/C++ functions in Python allows you to leverage the execution efficiency of C/C++ and the development efficiency of Python. To load a DLL in Python, you need to import the `ctypes` module. After importing, use the `ctypes.cdll.LoadLibrary()` function to load `libpari.dll` (there are similar link libraries in Linux with the `.so` suffix). The code is as follows:

```
import ctypes
from ctypes import *
pari = ctypes.cdll.LoadLibrary('libpari.dll')
```

Before calling PARI's functions, you must first initialize it using its `pari_init()` function, for example, `pari.pari_init(4000000, 2)`. The first parameter represents the number of bytes provided for PARI to use, which is the maximum memory usage of PARI; generally, it should not be less than 500,000. The second parameter is a pre-generated prime table. Pre-generating part of the prime table can slightly reduce the calculation load when dealing with prime-related problems later, but it is not mandatory; you can also set it to 0, and it can still complete prime-related operations.

Below is an example using PARI's `primes()` function. The `primes(n)` function outputs the first n prime numbers. The code is as follows:

```
import ctypes
from ctypes import *
pari = ctypes.cdll.LoadLibrary('libpari.dll') # Import libpari.dll

pari.pari_init(4000000, 2) # Initialization
pari.primes.restype = ctypes.POINTER(ctypes.c_long) # Purpose of these
    two lines is not entirely clear
pari.GENtostr.restype = ctypes.POINTER(ctypes.c_char) # Purpose of
    these two lines is not entirely clear

n = pari.primes(10) # Generate
y = pari.GENtostr(n) # Convert
print(ctypes.string_at(y)) # Output
```

Since I am just starting to try this, I don't quite understand the role of `restype`. Taking it literally, it should correspond the function's return type to a pointer. This code is based on some code found online. My tests also found that even without these two `restype` lines, the program can work normally; I'm not sure what the difference is. The process of the program is to first use the `primes()` function to generate the result. The result is of the `GEN` type, which cannot be recognized in Python. Use the `GENtostr()` function to convert it to a string, but this function returns an address, so `string_at()` must be used to read the output. In Python 2.7, the running result is (the output is a string):

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

The above code can also run normally in Python 3, but the output result is slightly different. This will be explained below.

The second example is the `isprime()` function, which is a primality testing function. This type of function is somewhat special; it does not use the C/C++ `int` type, so you cannot simply input a Python integer. It is estimated that PARI defines its own large integer type or stores large integers in other ways. We need to read the number from a string. The code is as follows:

```
import ctypes
from ctypes import *
pari = ctypes.cdll.LoadLibrary('libpari.dll') # Import libpari.dll

pari.pari_init(4000000, 2) # Initialization
pari.gp_read_str.restype = ctypes.POINTER(ctypes.c_int) # Purpose not
    entirely clear

n = pari.isprime(pari.gp_read_str(str(127))) # Test primality of 127
print(n)
```

The core code is the second to last line. First, convert 127 to a string, then use the `gp_read_str()` function to read from the string, and finally use the `isprime()` function to judge. The output result of this example is 1.

However, if you run this example in Python 3, an error will occur. The reason is that the string `str` in Python 3 is no longer the same as the `str` in Python 2.x. In Python 3, a new type `bytes` is defined. In fact, the `str` in 2.x is more like `bytes` in 3. Therefore, in Python 3, the code should be changed to:

```
pari.isprime(pari.gp_read_str(b'127'))
```

Or (using the `encode()` function of `str` to convert to `bytes` type):

```
pari.isprime(pari.gp_read_str(str(127).encode()))
```

Similarly, in Python 3, the object returned by `ctypes.string_at(y)` is not `str` but `bytes`. Readers may notice that the output result has an extra `b` prefix for this reason.

That's all for the introduction for now. If there are new discoveries, I will continue to share them with everyone I hope experts passing by will provide more guidance.

Download: libpari.zip