Converting Python Functions to Dynamically Compiled C

Ilan Schnell (ilanschnell@gmail.com) – *Enthought*, USA

Applications written in Python often suffer from the lack of speed, compared to C and other languages which can be compiled to native machine code. In this paper we discuss ways to write functions in pure Python and still benefit from the speed provided by C code compiled to machine code. The focus is to make it as easy as possible for the programmer to write these functions.

Motivation

There are various tools (SWIG, Pyrex, Cython, boost, ctypes, an others) for creating and/or wrapping C functions into Python code. The function is either written in C or some special domain-specific language, other than Python. What these tools have in common are several inconvenience for the scientific programmer who quickly wants to accomplish a certain task:

- Learning the tool, although there are excellent references and tutorials online, the overhead (in particular for the casual programmer) is still significant.
- Dealing with additional files.
- The source code becomes harder to read, because the function which needs some speedup is no longer in the same Python source.
- If the application need to be deployed, there is usually an extra build step.

Overview of CPython

Firstly, when referring to Python, we refer to the language *not the implementation*. The most-widely used implementation of the Python programming language is CPython (Classic Python, although sometimes also referred to as C Python, since implemented in C). CPython consists of several components, most importantly a bytecode compiler and a bytecode interpreter. The bytecode compiler translates Python source code into Python bytecode. Python bytecode consists of a set of instructions for the bytecode interpreter. The bytecode interpreter (also called Python virtual machine) is executing bytecode instructions. Python bytecode is really an implementation detail of CPython, and the instruction set is not stable, i.e. the bytecode changes with every major Python version. One could perfectly write a Python interpreter which does not use bytecode at all. However, there are at least two good reasons for having bytecode as an intermediate step.

- Speed: A Python program only needs to be translated to bytecode when it is first loaded into the interpreter.
- Design: Having bytecode as an internal intermediate step simplifies the design of the (entire) interpreter, since each component (bytecode compiler and bytecode interpreter) can be individually maintained, debugged and tested.

The PyPy project

In this section, we give a brief overview of the PyPy project. The project started by writing a Python bytecode interpreter in Python itself, and grew to implement an entire Python interpreter in Python. Compared to the CPython implementation, Python takes the role of the C Code. The clear advantage of this approach is that the description of the interpreter is shorter and simpler to read, as many implementation details vanish. The obvious drawback of this approach is that this interpreter will be unbearably slow as long as it is run on top of CPython. To get to a more useful interpreter again, the PyPy project translates the high-level description of Python to a lower level one. This is done by translating the Python implementation on the Python interpreter to C source. In order to translate Python to C, the PyPy virtual machine is written in RPython. RPython is a restricted subset of Python, and the PyPy project includes a RPython translator, which can produce output in C, LLVM, and other languages.

Using the PyPy translator

The following piece of Python code shows how the translator in PyPy can be used to create a compile decorator, i.e. a decorator functions which lets the programmer easily compile a Python function:

```
from pypy.translator.interactive import Translation

class compdec:
    def __init__(self, func):
        self.func = func
        self.argtypes = None

    def __call__(self, *args):
        argtypes = tuple(type(arg) for arg in args)
        if argtypes != self.argtypes:
            self.argtypes = argtypes
            t = Translation(self.func)
            t.annotate(argtypes)
            self.cfunc = t.compile_c()

        return self.cfunc(*args)

@compdec
def is_prime(n):
    if n < 2:
        return False
    for i in xrange(2, n):
        if n%i == 0:
            return False
    return True

print sum(is_prime(n) for n in xrange(100000))
```

There are several things to note about this code:

- A decorator is only syntactic sugar.
- The decorator function is in fact a class which upon initialization receives the function to be compiled.
- When the compiled function is called, the special *__call__* method of the instance is invoked.
- A decorated function is only compiled when invoked with a new set of arguments types.
- The function which is compiled, (*is_prime* in the example) must restrict it's features to RPython, e.g. it can not contain dynamic features like Python's *eval* function.

In the above decorator example all the hard work is done by PyPy. Which includes:

- The translation of the RPython function to C.
- Invoking the C compiler to create a C extension module.
- Importing the compiled function back into the Python interpreter.

The advantage of the above approach is that the embedded function uses Python syntax and is therefore an internal part of the application. Moreover, a function can be written without even having PyPy in mind, and the compile decorator can be applied later when necessary.

A faster numpy.vectorize

Using the PyPy translator, we have implemented a function called *fast_vectorize*. It is designed to behave as NumPy's vectorize function, i.e. create a generic function object (ufunc) from a Python function. The argument types (signature) of the function need to be provided, and it is possible to provides several signatures for the same function. For each signature, the PyPy translator is invoked, to generate a C version of the function for the given signature. The UFunc object is created using the function *PyUFunc_FromFuncAndData* available through NumPy's C-API, the support code necessary to put all the pieces together is generated, and at the end *scipy.weave* is used to create the UFunc object. The figure gives a high level overview of *fast_vectorize* internals.

Here are some benchmarks in which the simple function is evaluated for a numpy array of size 10 Million using different methods:

```
def f(x):
    return 4.2 * x*x - x + 6.3
```

These benchmarks were obtained on a 2.4GHz Linux system:

	Method	Runtime (sec)	Speed vs.
1	numpy.vectorize	8.674	69.9
2	x as numpy.array	0.467	3.8
3	fast_vectorize	0.124	1.0
4	inlined	0.076	0.61

Remarks:

1. *numpy.vectorize* is slow, everything is implemented in Python, and no UFunc object is created.
2. When x is used as the array itself, the calculation is more memory-intensive, since for every step in the calculation a copy of the array is being made. For example, first 4.2 is multiplied to the array x, which results in a new array ($tmp = 4.2 * x$) which is then multiplied with the array x, which again results in a new array, and so on.
3. Here, the function f is translated to a Ufunc object which performs all steps of the calculation in compiled C code. Also worth noting is that the (inner) loop over 10 Million elements is a C loop.
4. In this example, the C function has been inlined into the inner C loop. For simplicity and clarity, this has been available in *fast_vectorize*. Also, if the function is more complicated than the one in the benchmark, the performance increase will be less significant.

When caluculation the simple quadratic function as a numpy array (2), it is also possible to rewrite the function in such a way that fewer arrays are created by doing some operations in-place, however this only created a modest speedup to 10%. A more significant speedup is achieved by rewriting the function as `f(x) = (a*x + b) * x + c`. It should be mentioned that whenever one is trying to optimize some numerical code one should *always* try to first optimize the mathematical expression or the algorithms being used *before* trying to optimize the execution of some particular piece of code.

There are many interesting applications for the PyPy translator, apart from the generation of UFunc objects. I encourage everyone interested in this subject to take a closer look at the PyPy project.

Note I am working on putting the *fast_vectorize* function in SciPy.

Acknowledgments

The author would like to thank SciPy community and Enthought for offering suggestions, assistance, and for making this work possible. In particular, I would like to thank Travis Oliphant for questioning me about how including compiled functions into Python can be done is an easy manner, and his support throughout this project. Also, I would like to thank Eric Jones for sharing his knowledge about *weave*. Thanks to Gael Varoquaux for suggesting the name *fast_vectorize*.

References

- PyPy: http://codespeak.net/pypy/dist/pypy/doc/home.html
- SWIG: http://www.swig.org/
- Pyrex: http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/
- Cython: http://cython.org/
- boost: http://www.boost.org/
- vectorize: http://scipy.org/Numpy_Example_List_With_Doc#vectorize