multigrad

June 8, 2014

Fun with Python bytecode

The Python programming language is now present everywhere. While significantly slower than other more low-level languages like C/C++, its ease of use, its power and its various libraries make it an ideal choice for many projects. Yet for some cases, performance does matter. For these, many solutions have been proposed. One of the most common is the use of an underlying, heavily-optimized, C library along with Python bindings (numpy, scipy, opencv, etc.). Another is the addition of performance related features to the interpreter, like JIT or stackless mode with Pypy. However, what to do when you can't apply neither of them, yet you still want more performance without sacrificing the Python zen and simplicity?

One of the answers, at least in the present case, is to roll up our sleeves, fasten our seat belts, and dive into the mysteries of the Python bytecode.

Wait, what?

You have probably already heard that Python, unlike Fortran, C++ or Go, was an *interpreted* language. While not completely wrong, it is not entirely true either. You *could* create a Python interpreter that works in a purely interpreted way, but it is not how it is done in most current Python implementations. The Python code is first *compiled* into a series of simpler instructions. Thereafter, *these* instructions are actually interpreted by a virtual machine. The main reason for that design choice is speed: directly interpreting high-level Python code each time it is executed (think for instance of a loop or a function executed many times) would be ineffective. Moreover, some different Python syntax turn out to give the same result. For instance, using a *if/elif* statement produces *exactly* the same result than a *if* inside the *else* clause of another condition. A list comprehension has the same behavior than a traditional loop – except for some underlying details in the current implementations, but it *could* be exactly the same thing. The use of a simplified, intermediate language allows the actual interpreter to be simpler, easier to understand and maintain, and faster, while being entirely hidden to the user in most cases. With CPython, the most visible effect is the creation of **.pyc* files, used to store the bytecode in order to avoid a recompilation step each time the file is executed.

It is to be noted that this design is not Python-specific, since many other prominent languages use it too (Java, Ruby, etc.). All the designs are more or less identical, the steps involved being a) the parsing of the high-level language into an intermediate, simpler representation, and b) the interpretation of this intermediate representation to actually run the program. One could note that Python makes use of a fancier design, by translating the Python code into Abstract Syntax Tree or AST before create the bytecode, but it is out of the scope of this post. However, those interested by the CPython internals could refer to the Cpython compiler developers information.

In Cpython 3.4 (the reference interpreter), the bytecode is based on a stack representation, and 101 different *opcodes*. Python 2.7 uses 18 more, mainly for slicing operations, which where separated from the item-based operations, and *print* and *exec* statements, which became functions in Python 3. For the rest of this post, we will use CPython 3.4. While the explanations provided here also mostly apply to other versions and interpreters, they may not be totally accurate.

Let's take a look at an actual bytecode snippet. Suppose you have the following simple function :

```
def myMathOperator(a, b, c):
    return a * (b + c)**2
```

mathoperator.py hosted with ❤ by **GitHub** **view raw**

We could observe the bytecode generated by Python by using the __code__ attribute :

```
In [9]: myMathOperator.__code__.co_code
Out[9]: b'|\x00\x00|\x01\x00|\x02\x00\x17d\x01\x00\x13\x14S'
```

Blog Arc
▼ 2014
► Jul
▼ Jur
Fun
by
► Ma
► Apr
► 2013

Contribu
f.m. de
Marc-
Félix-
Yannic
Olivier

Ok... While expected (since the bytecode is basically a sequence of bytes), the result is not exactly *readable*. Fortunately, there is a helper module in the Python library which can render a bit more understandable code. This module is named *dis*, for disassembly. Let's see what it can do :

```
In [12]: import dis
In [13]: print(dis.Bytecode(myMathOperator.__code__).dis())
2 0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 LOAD_FAST 2 (c)
9 BINARY_ADD
10 LOAD_CONST 1 (2)
13 BINARY_POWER
14 BINARY_MULTIPLY
15 RETURN_VALUE
```

Well, that's definitely better! The dis module is able to translate the bytecode into its opcodes and their arguments, which are far more readable. However, to find what does a bytecode do, we must place ourselves in a *stack* context. Most opcodes modify the stack by either adding or removing elements (or both), or by changing the value of the top of stack pointer. For instance, *LOAD_FAST* is described as:

Pushes a reference to the local co_varnames[var_num] onto the stack.

For now, we can forget the reference to co_varnames (we will talk about it later on), and just retain that this opcode fetches a variable and put it onto the stack. In our example, the stack is initially empty since it is the beginning of the function (we will represent it as [ ]). Supposing that the values of a, b, and c arguments are respectively *12*, *7* and *1*, then the stack will contain [12, 7, 1] after the first three statements execution.

On the next line, we reach a new opcode, *BINARY_ADD*. In the documentation, it is said that it:

Implements TOS = TOS1 + TOS

Ok, that's a bit less clear. Here, TOS means “Top Of the Stack”. So, basically, it takes the value at the top of the stack and the second value at the top (TOS1), add them, and push back the result of the top of the stack. Applying this operation to our [12, 7, 1] stack, we obtain [12, 8].

Moving on to the next opcode, we find a *LOAD_CONST*. Basically, it does the same job as *LOAD_FAST*, except that it loads constants and not variables (in our case, the constant loaded is the 2 used as exponent). So our stack now contains [12, 8, 2].

The next opcode, *BINARY_POWER* does, according to the documentation:

TOS = TOS1 ** TOS

As for the addition, we take the two top-most items, exponent the second by the first, and push back the result on the stack, which now contains [12, 64].

The next opcode is *BINARY_MULTIPLY*, which works similarly than *BINARY_POWER* and *BINARY_ADD*, and our stack now contains the product of 12 and 64, that is [768]. Finally, the *RETURN_VALUE* operation is said to:

Returns with TOS to the caller of the function.

That is, it pops the value on the top of the stack, and return it to the calling function. In our case, the answer (768) is effectively returned, and we're done for this example.
There are of course many other opcodes, but their behavior is essentially the same, popping and pushing values and references on a global stack. As one can see, interpreting (and even writing) Python bytecode is fairly simple when we get the twist.

A last twist before the next step: some opcodes need arguments. This is the case for *LOAD_FAST*, which needs the index of the variable to grab. This is achieved very simply, by using up to three bytes for each operation. The first byte is the opcode itself, and the two remaining can be used for parameter passing. If there is no parameter to pass (like *BINARY_ADD*), then only one byte is used. The attentive reader would have noticed the curious indexes in the bytecode snippet above: they precisely mark the byte index of each opcode.

Great, so we are now ready! Let's write bytecode instead of Python code!

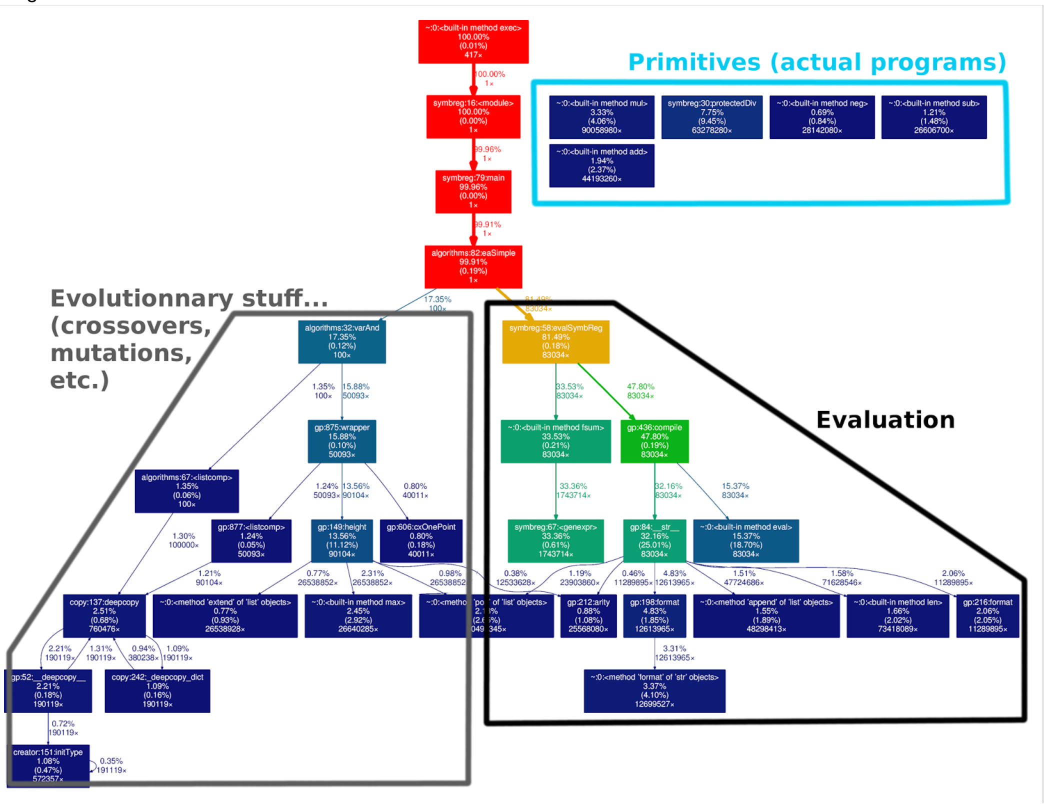
Wait, why?

At this point, a reasonable mind might say: why in the world would you bother to write bytecode, a limited and difficult language, instead of actual Python code? Well, because the compilation procedure is too slow and induces a significant overhead! To this answer, the same reasonable mind would probably say: who cares about the compilation time? It is not so slow (a few tenths of seconds in most cases), and, even if it was not, this must be done only once – since the next times, you can just reload the *.pyc and go happily!

In most cases, the clever guy would be right. But there's a specific use which has to be taken into account. Its name is GP, standing for *Genetic Programming*.

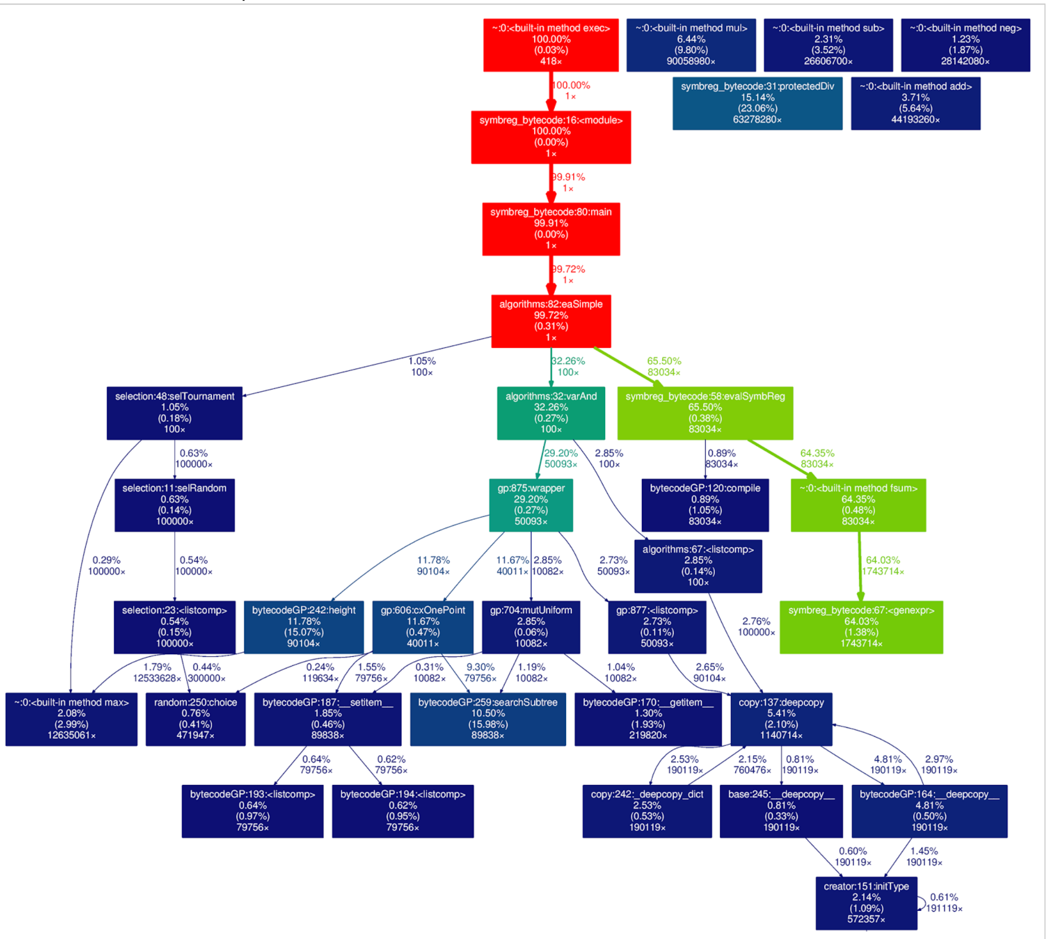
Genetic programming is an hyper-heuristics member of the bigger family of the *evolutionary algorithms*. Basically, it uses natural selection, mutations and other evolutionary concepts to *evolve* trees (in the computer sense). As any program can be represented as a tree, genetic programming is actually able to evolve *programs*, that is learn, alone, the best algorithm for a given problem, like getting out of a labyrinth, solving the Rubik Cube, etc. We do not intend to dive into deeper explanations on GP, but the interested reader could find introductory and advanced material about it at www.gp-field-guide.org.uk.

As a DEAP contributor and user, I frequently have to work with GP and its DEAP implementation (DEAP is a generic evolutionary algorithms framework in Python, which implements GP among others). Here, the compilation time becomes a problem. GP uses a *population* of several hundreds, and even thousands of different programs. This is mandatory to keep a sufficient level of diversity. But in order to see the results produced by each of these programs, we must execute them, and, as they change during the evolution, we have to compile them each time. Consider the next figure, produced with *gprof2dot* and the Python *profile* module, which represents the execution of a typical evolution with DEAP. One can see that the compilation time represented by the gp.compile function, in light green, takes almost 50% of the *total* computation time. In other words, we spend half our computational effort just to compile the programs!



A few solution have been proposed to address the problem. The compilation of the individuals with LLVM, as shown here, is one of them. Since it compiles to bare x86 bytecode, it indeed reduces the execution time, but at the price of an increase in compilation time, the penalty being more severe if we also add compiler optimizations. Also, the use of an optimized Python interpreter like Pypy is not of great help with GP – it actually *decreases* performance by a factor 2. Shortly, this can be explained by the JIT design itself: just-in-time compilers are efficient with long loop and repetitive code segments, since the whole idea is to make up for the compilation time by an important speedup in upcoming executions of the same code. With GP, the code is constantly changing, so the JIT either does nothing (because it detects that it would not be of any use) or deteriorate the processing time by trying to compile every individual at each evaluation. Finally, the use of an heavily optimized C implementation of the basic functions used in the GP program is not helpful, since Python still have to compile the high level code *before* calling this C implementation!

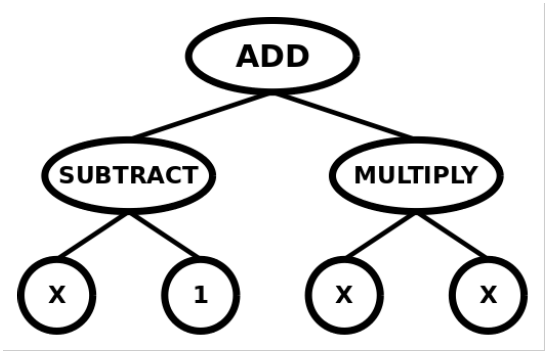
That's where our bytecode hobby comes on stage. What if we directly evolve Python bytecode? Well (spoiler alert), the results obtained with a preliminary implementation are given in the next figure. It should be noted that we use the exact same problem, configuration, and random seed for our standard and optimized implementations. The results are impressive: the compile time is now negligible (less than 1% of the total execution time), and the computation time is divided by more than two (47 seconds vs. 107 for the standard implementation), without altering the results in any way! It should be noted that this is the maximum speedup achievable, since the standard compilation procedure took about half of the program execution time. The small additional edge is a side effect caused by more effective implementations of other compile related methods.


~:0:<built-in method exec> 100.00% (0.03%) 418×
~:0:<built-in method mul> 6.44% (9.80%) 90058980×
~:0:<built-in method sub> 2.31% (3.52%) 26606700×
~:0:<built-in method neg> 1.23% (1.87%) 28142080×
100.00% 1×
symbreg_bytecode:31:protectedDiv 15.14% (23.06%) 63278280×
~:0:<built-in method add> 3.71% (5.64%) 44193260×
symbreg_bytecode:16:<module> 100.00% (0.00%) 1×
99.91% 1×
symbreg_bytecode:80:main 99.91% (0.00%) 1×
99.72% 1×
algorithms:82:eaSimple 99.72% (0.31%) 1×
1.05% 100×
32.26% 100×
65.50% 83034×
selection:48:selTournament 1.05% (0.18%) 100×
algorithms:32:varAnd 32.26% (0.27%) 100×
symbreg_bytecode:58:evalSymbReg 65.50% (0.38%) 83034×
0.63% 100000×
29.20% 50093×
2.85% 100×
0.89% 83034×
64.35% 83034×
selection:11:selRandom 0.63% (0.14%) 100000×
gp:875:wrapper 29.20% (0.27%) 50093×
bytecodeGP:120:compile 0.89% (1.05%) 83034×
~:0:<built-in method fsum> 64.35% (0.48%) 83034×
algorithms:67:<listcomp> 2.85% (0.14%) 100×
0.29% 100000×
0.54% 100000×
11.78% 90104×
11.67% 40011×
2.85% 10082×
2.73% 50093×
64.03% 1743714×
selection:23:<listcomp> 0.54% (0.15%) 100000×
bytecodeGP:242:height 11.78% (15.07%) 90104×
gp:606:cxOnePoint 11.67% (0.47%) 40011×
gp:704:mutUniform 2.85% (0.06%) 10082×
gp:877:<listcomp> 2.73% (0.11%) 50093×
2.76% 100000×
symbreg_bytecode:67:<genexpr> 64.03% (1.38%) 1743714×
1.79% 12533628×
0.44% 300000×
0.24% 119634×
1.55% 79756×
0.31% 10082×
9.30% 79756×
1.19% 10082×
1.04% 10082×
2.65% 90104×
~:0:<built-in method max> 2.08% (2.99%) 12635061×
random:250:choice 0.76% (0.41%) 471947×
bytecodeGP:187:__setitem__ 1.85% (0.46%) 89838×
bytecodeGP:259:searchSubtree 10.50% (15.98%) 89838×
bytecodeGP:170:__getitem__ 1.30% (1.93%) 219820×
copy:137:deepcopy 5.41% (2.10%) 1140714×
0.64% 79756×
0.62% 79756×
2.53% 190119×
2.15% 760476×
0.81% 190119×
4.81% 190119×
2.97% 190119×
bytecodeGP:193:<listcomp> 0.64% (0.97%) 79756×
bytecodeGP:194:<listcomp> 0.62% (0.95%) 79756×
copy:242:_deepcopy_dict 2.53% (0.53%) 190119×
base:245:__deepcopy__ 0.81% (0.33%) 190119×
bytecodeGP:164:__deepcopy__ 4.81% (0.50%) 190119×
0.60% 190119×
1.45% 190119×
creator:151:initType 2.14% (1.09%) 572357×
0.61% 191119×

Ok, great, so that was easy! Another problem solved!

Wait, how?

In order to understand the intricacies of the solution, we have to learn a bit about how GP is implemented in DEAP. As we said earlier, GP works by evolving *trees*. Since there's no tree implementation in the Python standard library, DEAP includes one specifically targeted on GP. It works by storing the trees into an underlying list, along with information about how many children each node have. For instance, the mathematical function *f(x) = (x-1) + x*x* can be represented by the following tree:



and will be stored in the following list:

```
['add', 'subtract', 'x', 1, 'multiply', 'x', 'x']
```

Because DEAP knows the number of arguments of each function (or, in genetic programming gibberish, their *arity*), it can reconstitute the tree. Thereafter, when comes the evaluation step, the following procedure is used:

1. The tree is converted into its string representation. Basically, it is the program that one would write if he wanted to produce the result of the tree. For instance, the string representation of the previous tree would be *add(subtract(x, 1), multiply(x, x))*.
2. This string is passed to *eval()*. This function is a powerful (and dangerous!) tool which allows to execute arbitrary code from its string representation. For instance, the following code would write “Hello world!” to the standard output:

   ```
   eval('print(“Hello world!”)')
   ```

   It is in this step that the Python code is actually compiled into its executable form.
3. The compiled function is returned, so it can be evaluated.

The reader would have understood that the last step is mandatory and cannot be suppressed, which is why we focus on the first two steps with our bytecode hack.

First, one useful simplification: the resulting program is merely only a sequence of calls to different functions. We never actually have to understand what is going *inside* these functions. This leads to the fact that our implementation does not restrain at all the function choice. If it works in Python, it will work with our approach! Also, this reduce our solution complexity, since we only have to call a few opcodes. Basically, we will need the following:

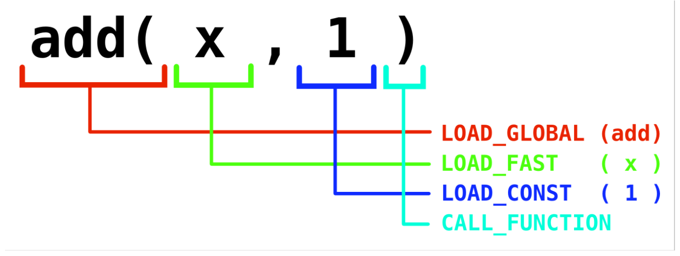
- *LOAD_FAST* : this will be needed to access to our program arguments.
- *LOAD_CONST* : similarly, this opcode is required to use constant values (numerical or not).
- *LOAD_GLOBAL* : this one will be used to retrieve references to the function objects of our tree. As its name suggests, instead of grabbing a symbol from the local dictionary like *LOAD_FAST*, it uses the global dictionary.
- *CALL_FUNCTION* : obviously, this is mandatory to actually call the functions previously loaded. It takes as parameter the number of arguments to pass to the function. All these arguments must be on the stack at the moment of the call, plus, under them, a reference to the function object itself. This particular structure will prove to be very useful hereinafter.
- *RETURN_VALUE* : this one will actually be needed only one time, to return the final result value.

Before we can go on with some code, it is the time to learn about how the functions, constants and arguments are actually described in a code object. The *LOAD_** functions have an almost common description:

Loads the {global/local/constant} named {co_names/co_varnames/co_consts}[arg] onto the stack

Fair enough, but what are these co_names, co_varnames or co_consts fields? Well, they are simply tuple objects containing all the needed symbols. For instance, if *co_names = (add, divide, subtract)* and that we want the subtract function, we will write *LOAD_GLOBAL (2)*, that is put the index of the wanted function in the *co_names* tuple as operation argument. As explained in the first section, the argument is simply the value of the two bytes following the opcode in the bytecode.

For the sake of simplicity, we will consider in the following part that we already have these tuples. Let's now see how to convert a list representation of our tree to its bytecode representation. The important point is to realize the similarities between the Python bytecode and our list representation. The following figure shows it quite straightly.



As one can see, the conversion is merely an iteration through our list with the appending of a corresponding *LOAD_** for each node, the relative order between the nodes staying the same (for the record, this order is called *depth-first*). The only tricky part is to add the *CALL_FUNCTION* opcode after the last argument. For this purpose, we keep a list of the number of arguments of each added node. Whenever we bump into a terminal node (a leaf), we decrement the argument count of its parent node. If this count gets to 0, then we know that we are done with this function, and we add the appropriate *CALL_FUNCTION*. Using this algorithm, we provide a simplified conversion function.

```python
def convertToBytecode(self, content):
    arities, numargs = [], []
    b = bytearray()

    for node in content:
        if node.arity > 0:
            # Non-terminal node
            symbolpos = self.co_names.index(node.name)
            arities.append(node.arity)
            numargs.append(node.arity)
            b.extend([opcode.opmap['LOAD_GLOBAL'], symbolpos & 0xFF, symbolpos >> 8])

        else:
            # Terminal node
            if isconstant(node):
                # Constant
                constpos = self.co_consts.index(node.value)
                b.extend([opcode.opmap['LOAD_CONST'], constpos & 0xFF, constpos >>8])
            else:
                # Variable
                argpos = self.co_varnames.index(node.name)
                b.extend([opcode.opmap['LOAD_FAST'], argpos & 0xFF, argpos >> 8])

            try:
                arities[-1] -= 1
                while arities[-1] == 0:
                    b.extend([opcode.opmap['CALL_FUNCTION'], numargs[-1], 0])
                    arities.pop()
                    numargs.pop()
                    arities[-1] -= 1

            except IndexError:  # We have no pending child (end of the tree)
                pass

    b.append(opcode.opmap['RETURN_VALUE'])        # Final return

    return bytes(b)
```

conv_to_bytecode.py hosted with ❤ by **GitHub** **view raw**

This function takes a list as argument, and return its corresponding bytecode. Note that a bytecode must be of type *bytes*, but this type is not suitable for our manipulations, because it is non-mutable. Therefore, we use a *bytearray*, a mutable equivalent of *bytes* object. The *opcode* module contains various tools to assist the bytecode creation. In this case, we use the opmap dictionary, which allows us to write the opcode in plain text ('CALL_FUNCTION' being far more readable than 0x83).

We now have to make Python understand that he has to *execute* this bytecode. First, we must create a complete code object with it. While the bytecode obviously plays an important role in a code object, there are plenty of other information we have to give to Python:

```
code = types.CodeType(
    len(self.co_vars),          # argcount
    0,                          # kwonlyargcount
    len(self.co_vars),          # nlocals
    200,                        # stacksize
    67,                         # flags
    bytes(bytecode),            # codestring
    self.co_consts,             # consts
    self.co_names,              # names
    self.co_vars,               # varnames
    "DEAP-Bytecode-Compiler",   # filename
    "Prim",                     # name
    1,                          # firstlineno
    bytes(),                    # lnotab
    (),                         # freevars
    ()                          # cellvars
)
```

codetype_ex.py hosted with ❤ by **GitHub** **view raw**

That's a bunch of information! Most of these parameters are self-explanatory, but a few deserve more explanations. The *stacksize* parameter controls how many things can be put onto the stack at the same time. For instance, a recursive call will add elements on the stack each time. While its exact value is not really important, one must take care to not exceed it, under threat of a segfault of the Python interpreter! The *flags* argument control various things about the way the code is handle, yet there is not much documentation about it. The value of 67 (64+2+1) comes from the actual value given by Python to the executable objects in the standard GP implementation.

Now that we have the code object, we must associate it with a function. There are various ways to do it, but one of the most obvious in Python 3 is to use the *types* module again, but to create a function object this time:

```
func = types.FunctionType(code, self.pset.context)
```

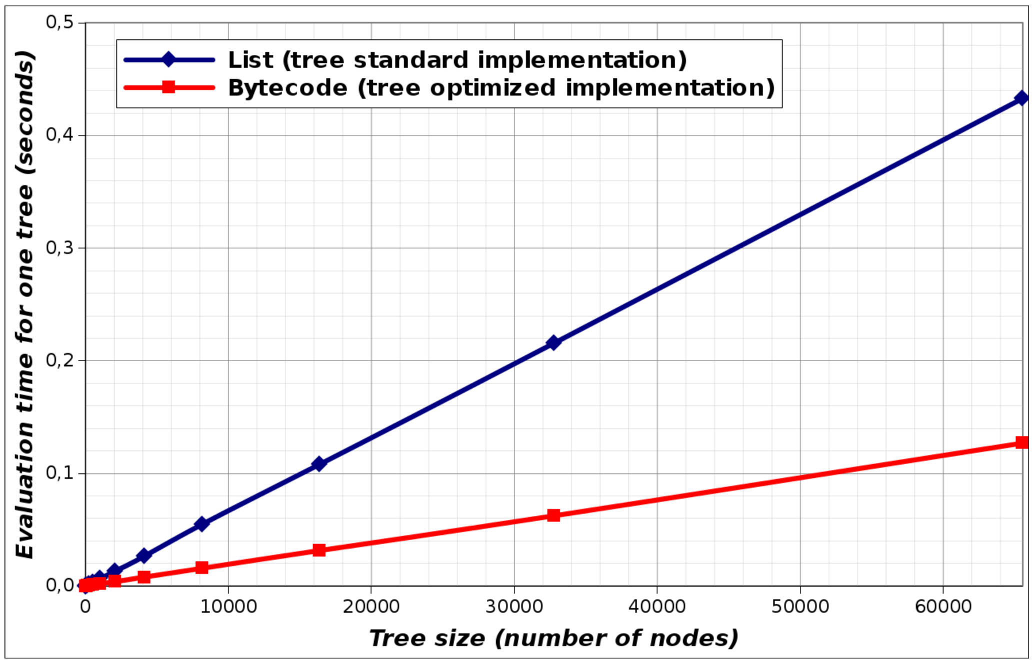
functiontype.py hosted with ❤ by **GitHub** **view raw**

The second argument (a dictionary) is mandatory to tell Python which function actually corresponds to each symbol. Fortunately, its generation was already implemented in DEAP, since the same mechanism applies when using the eval function.

Another interesting thing about this approach is that it allows for easy genetic manipulation, like crossovers or mutations. In a few words, these operations change the content of the tree by modifying or exchanging branches. In the standard list implementation, a branch replacement could be done with a simple slicing operation, since each branch is stored in a contiguous way. Well, that rationale also applies to our bytecode representation! In order to identify a subtree, we just need to obtain the position of the *LOAD_GLOBAL* of its root node, and look for the corresponding *CALL_FUNCTION* thereafter. These two positions then give us the indices needed for the slice construction.

The following table and figure show the time required to evaluate trees of different lengths and the relative speedup. The bytecode implementation reaches its optimal speedup around 250 nodes, but provides a considerable gain even for smaller lengths. The speedup value can be easily understand if we refer to the evaluation procedure we describe at the beginning of this section: the first two steps that were executed in O(n) are now done in O(1). Of course, the result computation itself has not changed, and still has, in our test case, a O(n) complexity. Overall, in exact notation, we started from a Θ(3n) complexity, down to Θ(n+2) with the bytecode approach, which is coherent with the observed speedups.

# of nodes	Standard	Bytecode	Speedup
1	4,2674E-05	2,3882E-05	1,787
3	5,9971E-05	3,0232E-05	1,984
7	8,1913E-05	3,6364E-05	2,253
15	1,5890E-04	7,4422E-05	2,135
31	2,5088E-04	9,6281E-05	2,606
63	4,5313E-04	1,5996E-04	2,833
127	8,6494E-04	2,9974E-04	2,886
255	1,6832E-03	5,0897E-04	3,307
511	3,3082E-03	1,0151E-03	3,259
1023	6,5782E-03	1,9996E-03	3,290
2047	1,3001E-02	3,8225E-03	3,401
4095	2,6315E-02	7,8760E-03	3,341
8191	5,4996E-02	1,5786E-02	3,484
16383	1,0808E-01	3,1520E-02	3,429
32767	2,1563E-01	6,2349E-02	3,459
65535	4,3286E-01	1,2690E-01	3,411



Conclusion

We have provided a simple way to speed up a genetic programming evolution by directly evolving Python bytecode, without intermediate representations. This generally divides by two the computation time needed to perform an evolution, which is non negligible with real world problems. This is especially important when taking into account that as most stochastic methods, evolutionary algorithms needs to be run at least a couple of times to produce statistical significant results. The complete code is available at github.com/mgard/deap. However, it should be noted that this code includes many hacks to make it fully compliant with DEAP API (so a lambda user does not have to worry about which tree implementation he is using), and is still in development to further improve performance and reliability – one funny characteristic of our approach is that it is going so deep into Python internals that it could actually segfault the interpreter whenever an error occurs...

If the clever mind from section 3 did not run away already (which would probably be the reasonable thing to do instead of reading an article describing a weird and unclean optimization technique targeting a very narrow topic of an already narrow field on a specific language), he might notice one interesting thing in the second profiling figure. The evaluation itself now takes about two thirds of the total computation time. Moreover, even basic arithmetic functions like multiply or divide are called so often (almost 100 million times!) that they take up to 20% of the total execution time! That's clearly where we should focus in order to further improve performance. But how could we do it?

Well, in this post, we had fun with *Python* bytecode. Maybe it's time to take the next step, and use a different type of bytecode, even more low level. But that will be the subject of another story...

Posted by Marc-André Gardner at 1:08 PM

G+1 +44 Recommend this on Google

3 comments:

Padarn Wilson June 20, 2014 at 6:44 PM

Very cool. How complicated do these tree expressions usually get? Could you define a dictionary of common 'operations' which could be linked together as snippets of something compiled at a lower level? Keen to read more ...

Reply

Niklas June 21, 2014 at 3:56 AM

Thanks for the great post!
A question about the byte code from myMathOperator. The index goes till 15, but I can only see 11 hex numbers. Even more important, the byte string contains "|" which I didn't find explained in the python docs. Could you elaborate?

Reply

Marc-André Gardner June 21, 2014 at 4:02 PM

Hi Niklas,
You are right, this is a tricky part I didn't explain. Basically, it has to do with the way Python handles bytes objects when it comes to printing : all valid display ASCII characters (roughly any byte between 0x30 and 0xFF) are displayed as is, and the others (which do not représente displayable characters) are displayed as hex numbers (with the \x** syntax).
In this case, the actual opcode value for LOAD_FAST is 124 (0xFC), which is also the ASCII code for "|". The same rationale applies for RETURN_VALUE :
>>> opcode.opmap["RETURN_VALUE"] == ord("S")
True

If you make the count, you will see that there is 3 LOAD_FAST and 1 RETURN_VALUE in the bytecode, the rest being non display characters codes. That explains your difference of 4.

Reply

Enter your comment...

Comment as: Select profile...

Publish **Preview**

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)