Python To Cython (Optimized) Convertor

by

Rahul Gupta

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

May, 2012

Python To Cython (Optimized) Convertor

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
Master of Technology (M.Tech)

by
Rahul Gupta

Supervised By
Dr. Amey Karkare

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

May, 2012

CERTIFICATE

It is certified that the work contained in this thesis entitled
"*Python To Cython (Optimized) Convertor*",
by *Rahul Gupta(Roll No. 10111030)*, has been carried out under my
supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Amey Karkare)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

May, 2012

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Dr. Amey karkare for his support and encouragement. He provided me freedom to experiment with new ideas and inculcated the techniques required to understand and progress my work. I am grateful for his patient guidance and advice in giving a proper direction to my efforts.

I would like to thank many friends with whom I had many enjoyable as well as thought-provoking discussions.

Last, but not the least, I would like to thank my parents who have been a constant source of support and encouragement. Their belief in me survived even as mine failed.

Rahul Gupta

Dedicated to
my parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Background

Before we get into details of the Python language let us know about the basic terminology involved in it.

Variables :: First of all, we would like to clarify what is meant by a 'variable' in Python. Actually Python is dynamically typed language, so unlike statically typed languages (like C or similar language) Python doesn't have variables in their original sense. For e.g.

```
xx = [2,4,6,9]
yy = xx
xx[0] = 3
```

In above code, xx is a name bound to list object. yy is a name which refers to the same object.So when we modifies the list xx, it is reflected in yy as well. That means after changing the list which xx refers to, if we print yy, it will also print the modified list as well. A Python compiler has to resolve what kind of variables are used in the code. The variables used in python can be local/global/declared global according to their scope or free/assigned etc. This information along with the name of the variable and other useful information is stored in the symbol table.

```
def outer(aa):
        def inner():
                global dd
                bb=1
                return aa + bb + cc + dd
        return
```

Local : bb is local to inner *Global* : cc is global variable *Declared Global* :: dd is declared global *Free* : aa is free to inner *Assigned* : bb is assigned in inner *Parameter* : aa is passed as parameter to outer *Referenced* : aa,bb,cc,dd are refrenced variables *Imported* : When a variable is imported from some other module, then it

is called imported variable. In the above example there is no imported variable.

Parse Tree :: Python's parser is a LL(1) Parser.The parser itself is created from a grammar specification defined in the file Grammar/Grammar with numeric value of grammer rules stored in *Include/graminit.h* in the standard Python distribution. The numeric values for types of tokens are stored in *Include/token.h* The parse Tree made up of node *structs and is defined in *Include/node.h*. The parse trees stored in the ST objects created by these modules are the actual output from the internal parser when created by the expr() or suite() functions.

Abstract Syntax Tree :: The Abstract Syntax Tree is a high level representation of a program, with syntactic details of the source text removed. It is tree representation of the abstract syntactic structure of the program, where each node denotes construct occurring in the the source code. AST is specified using ASDL(Zephyr Abstract Syntax Definition Language). In Python source code, it is defined in *Parser/Python.asdl*. Each AST node represents the structure of statements , expressions and several other important specialized types.(like list , exception handlers etc.). Most definition in the AST corresponds to a particular source and is independent of it's realization in any particular language. Each programming language has it's own way of representing ASDL, standard tools and a set of code top generate a specific Abstract Syntax Tree. For eg. Consider the following code :

```
rg = 13
print rg
```

The AST generated for the code written in the example is ::

```
"
Module(
        None,
        Stmt(
                [
                        Assign(
                                [
                                        AssName(
                                                        'rg', 'OP\_ASSIGN'
                                                )
                                ],
                                Const(13)
                               ),
                        Printnl(
                                [
                                        Name('rg')
                                ],
                        None
                               )
                ]
            )
      )
"
```

Symbol-table ::

Control Flow Graph :: A Control Flow Graph(CFG) is a directed graph that models the flow of program using some basic blocks containing Intermediate Representation(Python Bytecodes) within the blocks. Or in other words ,we can say that, It is representation of all paths that might get traversed during program execution. In a control flow graph, each block is represented by a node. Direct edges are used to represent jumps in the Control flow. We have two Specially designated blocks in a CFG :the entry block and the exit block. The entry block control enters into the flow graph, and the exit block is that basic block through which all control flow leaves. So, it means that a basic block is a chunk of code which starts at the entry point and runs to an exit point or end of the block. CFGs are usuallu just one step away from the final code output. Code is generated directly from the basic blocks when we do a post-order depth-first search on the CFG.

ByteCodes :: The result of PyAST_Compile() is a PyCodeObject. It is defined in Includecode.h . So, Now we have executable Python bytecode. These code-objects(byte code) is executed in Pythonceval.c

The core of python interpreter actually is nothing more than a classic compiler. When we type a python command, it scans our source code for tokens, which are further parsed into a tree representing the logical structure of the program. This tree is then transformed in bytecode, which then is finally executed by virtual machine. Now, the steps involved in working of standard compiler are :
(1)Parse source code into a parse tree (*Parser/pgen.c*)
(2)Transform parse tree into an Abstract Syntax Tree (*Python/ast.c*)
(3)Transform AST into a Control Flow Graph (*Python/compile.c*)
(4)Emit bytecode based on the Control Flow Graph (*Python/compile.c*)

So, now we are ready to start, let us start with an example. Suppose we have typed

```
\$ python -c 'print("Hello, world!")'
```

Now you will see the output on the screen as Hello World. (Here '-c' is used for executing a string. We can also use '-m' if we want to execute an module as an executable.)

```
"
[Rahul@localhost ~]\$ su
Password:

[root@localhost Rahul]# python -c 'print("Hello, world!")'
Hello, world!
[root@localhost Rahul]#
"
```

Now we will try to analyze what is happening at the back end while we have just typed a single command.
First python's binary is executed and then the main function starts executing *Modules/python.c : main* which calls *Modules/main.c : Py_Main*.
This is followed by basic initialization is done and then *Python/pythonrun.c:Py_Initialize* is called. Now this function is responsible for converting 'a process' to 'a process with python interpreter' in it.It initializes the python interpreter. It initializes the table of loaded modules(sys.modules), and also creates the fundamental modules like __builtin__ , __main__ and sys. Apart from initializing, it also creates two very important python data structures : Interpreter state and Thread state. This Thread State is a data structure is associated with each python thread and it points to the stack of the current executing frame.
Now, in our example a single string is executed , since we have invoked python with -c option. To execute this single string *Python/pythonrun.c: PyRun_SimpleStringFlags*

is called. This function creates the __main__ namespace. After the namespace is created, the string is executed in it.
We'll now have to transform the string into some code on which machine can work on. For that, lets have a look on Parser/Compiler stage of Py_Run_SimpleStringFlags. It tokenize and creates a Concrete Syntax Tree(CST) or Parse Tree from the code. Then it transforms the CST into an Abstract Syntax Tree(AST) and finally compile the AST into a code object using *Python/ast.c: PyAST_FromNode*. Now, this code object can be think of as binary string of machine code that Python's Virtual Machine can operate on.

```
"
[root@localhost Thesis]# python
Python 2.6.4 (r264:75706, Jun  4 2010, 18:20:16)
[GCC 4.4.4 20100503 (Red Hat 4.4.4-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import ast
>>> str = """print("Hello World")"""
>>> str_ast = ast.parse(str)
>>> str_ast
<_ast.Module object at 0x9a81fcc>
>>> ast.dump(str_ast)
""
Module(
        body=[
                Print(
                        dest=None,
                        values=[
                                Str(
                                        s='Hello World'
                                   )
                               ],
                        nl=True
                     )
             ]
       )
""
>>>
"
```

Now we have an empty main and a code object. Now this line : *Python/pythonrun.c: run_mod, v = PyEval_EvalCode(co, globals, locals)* receives a code object and a namespace for globals and for locals.

```
"
[Rahul@localhost ~]\$ python
Python 2.6.4 (r264:75706, Jun  4 2010, 18:20:16)
[GCC 4.4.4 20100503 (Red Hat 4.4.4-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> str = """print("Hello World")"""
>>> str
'print("Hello World")'
>>> locals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__':
None, 'str': 'print("Hello World")', '__package__': None}
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__':
None, 'str': 'print("Hello World")', '__package__': None}
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'str']
>>> import base64
>>> dir(base64)
['EMPTYSTRING', 'MAXBINSIZE', 'MAXLINESIZE', '__all__', '__builtins__', '__doc__',
'__file__', '__name__', '__package__', '_b32alphabet', '_b32rev', '_b32tab',
'_translate', '_translation', '_x', 'b16decode', 'b16encode', 'b32decode', 'b32encode',
'b64decode', 'b64encode', 'binascii', 'decode', 'decodestring', 'encode', 'encodestring'
'k', 're', 'standard_b64decode', 'standard_b64encode', 'struct', 'test', 'test1',
'urlsafe_b64decode', 'urlsafe_b64encode', 'v']
>>>
"
```

It also creates a frame object from these and executes it. After creating frame object and placing it on the top of the stack pointed by thread state, *Python/ceval.c: PyEval_EvalFrameEx* evaluates it opcode by opcode.

Now, this frame then goes to *PyEval_EvalFrameEx*. It extracts the opcode after opcode from the code and runs a C code which matches with the respective opcode.

After execution of this frame, *PyRun_SimpleStringFlags* returns. Some mundane cleaning is then done by *main* function and the process is exited successfully.

Chapter 3

The Problem

3.0.1 Benchmarking

We know that our main aim was to speed-up Python. We want to use it's dynamic features, rich libraries, simplicity and along with that want to increase it's speed. So, In order to know the possible reasons why Python is slower than C started with benchmarking the following algorithms.

3.0.2 Problems Used for Benchmarking

Josephous Problem

Longest Common Subsequence

Pollard's rho Algorithm

3.0.3 Computer Algebra System Software

These are the results

- Python is interpreted, while C is compiled
- Python has no primitives data types
- List can handle more than one type of data,, so additional field is required
- But development time in python is less than as compared to C

Chapter 4

Approach

Introduction

Chapter 5

Implementation Details

Introduction

Chapter 6

Experimentation Results

Josephous Problem				
Time(in microseconds)	**C**	**Python**	**Cython(dynamic)**	**Cython(static)**
Array initialization	1	38.912	26.732	–
Computing win position	1.249	2.457	2.86	–
Total Time taken	1.304	48.332	31.129	–

Longest Common Subsequence				
Time(in microseconds)	**C**	**Python**	**Cython (dynamic)**	**Cython (static)**
Algorithmic Time	0.132	78	90	–
Total Time taken	0.198	135	145	–

Pollard's rho Algorithm				
Time(in microseconds)	**C**	**Python**	**Cython(dynamic)**	**Cython(static)**
Initialization	2	3	–	–
Algorithmic time	13	60	–	–
Total Time taken	73	392	–	–

Computer Algebra System Software				
Time(in microseconds)	**C**	**Python**	**Cython (dynamic)**	**Cython (static)**
Algorithmic Time	–	–	–	–
Total Time taken	–	–	–	–

Chapter 7

Conclusion

Introduction