Project 1: A Python Interpreter for the Browser

Ian Gemp and Shaylyn Adams

October 19, 2014

1 Introduction

The goal of this project is to write a typescript program that will interpret python bytecode (.pyc) in the browser. Our general approach split this task into three parts: parsing the bytecode, interpreting the result, and porting to the browser. All preliminary work was to be done outside the browser using node.js with the last step being the transition to the browser using browserFS.

When python (.py) files are executed, they are compiled into bytecode (.pyc) and interpreted by the python interpreter (e.g. CPython, PyPy, etc.). The bytecode itself is conveniently universal across the interpreters of the same version (e.g. Python 2.7). In all subsequent executions of the python file (.py), the interpreter will run the bytecode (.pyc) directly in order to save time re-compiling the python file (.py) assuming the python file (.py) has not changed since its last compilation. In reality, the bytecode (.pyc) must be read and parsed in its entirety before the python implementation actually executes the source.

2 Parsing

The python bytecode (.pyc) adheres to a strict format, the "Marshall" format, that can be read and interpreted by the python implementation (e.g. CPython). The majority of the bytecode is structured in a way such that the interpreter can read in the bytecode stream as an alternating sequence of datum (singular piece of data) and datum types. Some of the datum types represent containers (e.g. objects, namespaces, etc.) and a natural hierarchy of data results.

2.1 Parsing Dependencies

Our program assumes the python bytecode was compiled with Python 2.7 although we believe the marshal format is consistent across the 2.x line.

2.2 Understanding Python Bytecode

As mentioned above, the majority of the python bytecode is structured as an alternating series of types and data chunks, however the first 8 bytes (64 bits) of bytecode are specially formatted. The first 4 bytes (32 bits) of bytecode represent a magic number corresponding to the marshal format version. The second 4 bytes are the modification timestamp of the python source file (.py). The next byte constitutes the beginning of the type/datum sequence. Each "type-byte" is stored in the bytecode as an unsigned 8 bit integer representing a single character. This character corresponds to a specific datum type, which dictates how the following bytes should be read in order to correctly parse the corresponding datum. For instance, the type-byte "s" indicates that the datum is a string. The 4 bytes (32 bits) following the type-byte are a signed integer representing the size of the string to be read. Once the string is parsed, the program moves to the following byte and expects to encounter another type-byte. The only way this rule is broken is if the program has moved beyond the length of the bytecode file, in which case parsing is complete. The diagram below depicts this example and the general algorithmic scheme for parsing the bytecode.

```
1) Read Magic Number
2) Read Timestamp
3) While Not End of File
4)     Parse (Type,Datum)
```

type	size (int32LE)	string					type	...
's'	5	h	e	l	l	o	'c'	...

The resulting parsed bytecode consists of a series of *code objects* containing *opcodes*, stored constants, argument names, and other general information. Code objects may appear nested within a list of constants creating a nested hierarchy of code objects.

2.3 Special Cases

An important object not given sufficient attention in python documentation is the interned list. The interned list is a global variable containing a list of names referred to by any number of code objects during bytecode execution. The parser must add certain strings to this list in order for the execution to function properly.

2.4 Our Implementation

3 Interpreter

Once the bytecode has been fully parsed, the python interpreter begins to proceed through the op codes, or instruction list, of the highest level code object (representing the source module). The interpreter maintains its place in the instruction list with a program counter pointing to the byte offset in the instruction list of the current instruction. Opcode execution consists of a series of Stack manipulations and completes by updating the program to point to the next instruction. The interpreter is finished when the program counter has moved beyond the end of the instruction list of the highest level code object.

3.1 Interpreter Dependencies

3.2 Understanding the Interpreter

3.3 Our Implementation

In our implementation, each opcode has a corresponding function which executes a series of operations by manipulating the global stack object. Upon completion of the opcode execution, the program counter is updated to point to the next instruction to be executed.

3.4 Special Cases

4 Conclusion