Project 1: A Python Interpreter for the Browser

Ian Gemp and Shaylyn Adams

October 27, 2014

1 Introduction

The goal of this project was to write a typescript program to interpret python programs in the form of bytecode files (.pyc) within 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 involving the transition to the browser using browserFS.

When python (.py) files are executed, they are first compiled into bytecode (.pyc) and then interpreted by the python interpreter (e.g. CPython, PyPy, etc.). The bytecode itself is conveniently universal across 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) consists of serialized code object(s) that adhere to a specific "Marshal" format. These code objects can be read and implemented by the python interpreter (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 program was compiled with Python version 2.7 and this is ensured through a check of the 'magic number'. The 'magic number' is a unique number that refers to a specific Python version for the interpreter (in our case Python 2.7 is 03f3 0d0a).
- The long data type is not implemented, however, integers, floats, and complex (32 and 64 bit) are handled.

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 follow a strict format. The first 4 bytes (32 bits) of the bytecode always represent the magic number corresponding to the marshal format version. Similarly, the second 4 bytes make up the modification timestamp of the python source file (.py). The following 9th 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, "c" for code objects, "i" for 32-bit integers, etc..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 will consist of a series of *code objects* containing *opcodes*, stored constants, argument names, and other general information like flags. Code objects may appear nested within a list of constants creating a nested hierarchy of code objects and executable op codes.

2.3 Our Implementation

Numeric data types read in by the parser are immediately converted to the appropriate class object (integer, float, complex) which we defined in typescript to allow for the correct manipulations of each type since typescript only provides the generic class of 'number'. The typical builtin arithmetic functions are maintained as methods of the numeric objects. When arithmetic operations are called by the interpreter (e.g. +), a typescript method (e.g. add) is called which implements the data type's specific rules for the operation. If that operation is not implemented by that data type, the inverse method is called by the operand. For instance, `<int>.__add__(<float>)` returns 'NotImplemented' and so its inverse, `<float>.__add__(<int>)`, which is in fact supported. If both calls fail, then an error is thrown.

The largest parsing function is devoted to parsing the code object which contains a number of properties, most important of which include the list of opcodes, constants, names, variable names, free variables, and cell variables. The opcode list contains the sequence of instructions to be executed by the interpreter. The variables and constants are utilized and manipulated by the opcodes and appropriate function calls.

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 the correct order for the execution to function properly.

3 Interpreter

Once the bytecode has been fully parsed, the python interpreter begins to proceed through the op codes, or instruction list, starting with the highest level code object (representing the source module). The interpreter maintains its place in the instruction list via a program counter that refers to the byte offset in the instruction list of the current instruction. Opcode execution consists of a series of stack manipulations and completes after updating the program counter 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

- Iterator objects are not implemented (i.e. for loops), however, while loops function as expected.
- With blocks are not implemented.
-There is no difference between inplace and binary arithmetic operations in our code. - Imports are not implemented although our interface only allows a single .pyc file to be run at once, so it's not to be expected.
- Sets and maps are not implemented.

- A few other opcodes are not implemented as well, but we have yet to use some of these in our own extensive work with python so do not expect them to be major obstacles/downfalls.

3.2 Understanding the Interpreter

The interpreter functions in a manner that is very similar to the parser. The interpreter's role is to proceed through the list of opcode instructions which consists of alternating sequences of operators and operands. The operands only exist for certain operators and provide information such as the number of arguments to pop of the stack, where to find certain elements in the code objects variable lists, etc. The diagram below depicts an example along with the general algorithmic scheme for the interpreter.

```
1) While Not End of Instruction Set
2)     Read Op Code Operator
3)     Read Op Code Operand (if any)
4)     Execute Operation (most likely manipulates stack)
```

operator	operand (int16)	operator	operand (int8,int8)	...
100 (`LOAD_CONST`)	0 (index in consts)	131 (`CALL_FUNCTION`)	(# args,# kwargs)	...

3.3 Our Implementation

In our implementation, each opcode has a corresponding function which executes a series of operations by manipulating the stack object. There are roughly 120 different opcodes of which half take some argument. Thus, collecting the correct arguments from the stack is critical. Once an opcode has been executed, the program counter is updated to point to the next instruction and the return value, if any, is placed on the stack.

We chose to treat the stack as a global object as opposed to maintaining a separate stack for each frame (i.e. code object). This made implementing the interpreter easier, however, it prohibits us from incorporating threading since a global stack requires operations to occur sequentially whereas threads can manipulate the stack asynchronously as long as they obey data dependencies.

The heaviest opcode is undoubtedly the call function opcode. This op code is responsible for a series of operations including retrieving arguments off the stack, passing arguments to the function to be called, executing the corresponding function's op code list, and finally pushing the returned value to the stack. It's interesting to note that python makes function calls to class objects directly instead of first retrieving the object's constructor ('__init__') as an attribute which is how it handles all other class method calls. For this reason, we check to see if the function being called is actually a function object or if it is in fact a class object. We also check to see if the function refers to a builtin such as float() or odd() which we have implemented directly in typescript. Once the function has been identified, its argument names list is retrieved for comparison against the arguments supplied on the stack. Keyword arguments, positional arguments, and default values are added to an argument list that will be used by the function. The function is then called, either by executing its list of opcodes or by calling the builtin directly. The returned value is then pushed onto the stack. In the case of the class object, that object is also pushed back onto the stack and represents an initialized class instance.

Since the interpreter level mechanisms behind Python's try-except, loops, and blocks are not explicitly documented, we designed our own implementation to handle exceptions and loops. In our interpreter in addition to the regular stack, there is a global *block stack* that maintains individual blocks. Blocks hold properties defining their type (e.g. 'except') as well as a boolean flag indicating the state of the program. They also contain the size of the block which is important when breaking out of a block. The program may raise an exception by constructing an error object and calling it's throw method. When an exception is thrown, the top of the block stack is temporarily popped and it's flag is set to true if the block is an exception block (e.g. type equals 'except'). If the block is not an exception block, the program is halted.

Otherwise, the program continues and is expected to encounter a pop block statement which pops a block, inspects it's flag, and updates the program counter to the start of the exception handler if the flag is true.

4 Browser Interface & Testing

The browser interface is very simple using a mix of basic html and javascript. The user may choose a single .pyc file to execute. The user has the option of viewing the interpreter operations in detail including opcode executions and stack manipulations or they can view just the output that is printed out and returned by the program. *Run Interpreter* clears any bytecode in local storage and then interprets the new bytecode while the *Clear Output* button will clear the entire text area.

The interpreter is located at `http://shaylyna.github.io/630Systems_Project1/`. In order to run the test suite, simply choose the file `testSuite.pyc`. This file tests a number of operations outlined below:

- Storing constants
- If else statements
- Printing
- Function creation & function calls
- Argument handling (args, kwargs, defaults)
- Document strings and other function properties (e.g. closure)
- Classes and property arguments
- Dictionaries and tuples
- While loops plus break and continue statements
- Builtins (not all of them)
- Try catch blocks
- Integer, Float, and Complex arithmetic

5 Conclusion

Our initial impression of python bytecode and the underlying mechanisms of the interpreter was complete bewilderment. Documentation is sparse at best and very few references are available online. Only a few rare enthusiasts have explored the machinery. As we put together the bare skeleton of the interpreter and became more comfortable with its high level operations, we began to see the elegance of the design. This impression lasted only temporarily. Once we began to implement more complex features (e.g. classes, blocks, etc.), we discovered that the supposedly elegant design is really just a simplistic container for more the obscure, subtle operations happening inside the interpreter. It appears as though the creators did their best to adhere to a simple design philosophy as long as they could until they were forced to abandon the approach for a more practical engineering effort. Some of their final design choices still elude us.

Although our learning curve was extremely shallow at first, we eventually ramped up to speed and started to understand the inner workings of the python bytecode and the interpreter. The principles learned from this project are extremely valuable and we have already begun to apply some of them in our work outside of class.