py-py Interpreter

Run Python scripts with a Python byte code interpreter.

A practical example of Universal Turing machine

Written by Hitesh Kumar Rawat

Compiled from

http://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html

13 August 2017

Contents

1. Files included in the package
2. Core Python modules
   - 2.1 `argparse`
   - 2.2 `logging`
   - 2.3 `imp`
   - 2.4 `os`
   - 2.5 `sys`
   - 2.6 `tokenize`
   - 2.7 `dis`
   - 2.8 `inspect`
   - 2.9 `linecache`
   - 2.10 `operator`
   - 2.11 `six`
3. Getting started
4. Building an Interpreter from scratch
5. py-py Interpreter
6. Classes
   - 6.1 The VirtualMachine Class
   - 6.2 The Frame Class
   - 6.3 The Function Class
   - 6.4 The Block Class
7. The Instructions
8. Cons
9. Issues
10. Acknowledgement
11. Supplementary resources

Chapter1

Files included in the package

__init__.py

File required to make **Python** treat the directories as containing packages; this is done to prevent directories with a common name, such as string , from unintentionally hiding valid modules that occur later on the module search path.

__main__.py

The main script for the interpreter.

py_Obj.py

Implementations of Python fundamental objects for Byterun.

py_VM.py

A pure-Python Python bytecode interpreter. The heart of the interpreter.

Chapter2

Core Python modules

The core Python modules to implement the interpreter. The following:

argparse

The command-line parsing module in the Python standard library. Simple implementation in `__main__.py`

```
import argparse

parser = argparse.ArgumentParser(
    prog="A py-py Interpreter",
    description="Run Python scripts with a Python bytecode interpreter.",
)
parser.add_argument(
    '-m', dest='module', action='store_true',
     help="script is a module name, not a file name.",
)
parser.add_argument(
    '-v', '--verbose', dest='verbose', action='store_true',
    help="Trace the execution of the bytecode.",
)
parser.add_argument(
    'script',
    help="scriptname.py module to execute",
)
parser.add_argument(
    'args', nargs=argparse.REMAINDER,
    help="Arguments to pass to the program.",
)
args = parser.parse_args()

if args.module:
    run_fn = execfile.run_python_module
else:
run_fn = execfile.run_python_file
argv = [args.script] + args.args
run_fn(args.script, argv)
```

logging

Logging provides a set of convenience functions for simple logging usage. Specifically throwing Tracebacks to stdout in case of execution error.
Using:

```
def basicConfig(**kwargs)
```

Does basic configuration for the logging system.
The default behaviour is to create a StreamHandler which writes to sys.stderr, set a formatting for the BASIC_FORMAT format string. Note that DEBUG and WARNING are formatted strings.

Simple implementation in:
`__main__.py`

```
level = logging.DEBUG if args.verbose else logging.WARNING
logging.basicConfig(level=level)
```

imp

This module provides an interface to the mechanisms used to implement the **import** statement.

```
def find_module(name[, path])
```

Search for a module. If path is omitted or None, search for a built-in, frozen or special module and continue search in sys.path. Note that the module name cannot contain '.' to search for a sub module.

```
def new_module(name)
```

Create a new module and do not enter it in sys.modules. Name should include full package name.

Simple implementation in:
`execfile.py`

```
openfile, pathname, _ = imp.find_module(name, searchpath)
main_mod = imp.new_module('__main__')
```

os.path

Miscellaneous operating system interfaces.

```
def dirname(p)
```

Returns the directory components of a pathname.

```
def abspath(path)
```

Returns the absolute version of the path.

sys

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Simple implementation in:
execfile.py

```
try:
    # In Py 2.x, the builtins were in __builtin__
    BUILTINS = sys.modules['__builtin__']
except KeyError:
    # In Py 3.x, they're in builtins
    BUILTINS = sys.modules['builtins']
```

Grab the list of all default built-in modules.

```
def exec_info()
```

Return information about the most recent exception caught by an except clause in current stack frame or in older stack frame.

```
sys.argv
```

The list of command line arguments passed to a Python script.

```
sys.path
```

A list of strings that specifies the search path for modules. Initialized from the environment variable **PYTHONPATH**, plus an installation-dependent default.

tokenize

The tokenize module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing “pretty-printers,” including colorizers for on-screen displays.

```
tokenize.open(name)
```

Open a file in read only mode using the encoding detected by detect_encoding().

Simple implementation in:
`execfile.py`

```
try:
    # pass 'open' reference to open_source
    open_source = tokenize.open     # pylint: disable=E1101
except:
    def open_source(fname):
        """Open a source file the best way."""
        return open(fname, "rU")
```

dis

The **dis** module supports the analysis of CPython bytecode by disassembling it.

(For testing purposes only)

```
dis.dis(name_of_the_instance)
```

Disassemble the *bytesource* object. *bytesource* can denote either a module, a class, a method, a function, or a code object. For a module, it disassembles all functions. For a class, it disassembles all methods. For a single code sequence, it prints one line per bytecode instruction. If no object is provided, it disassembles the last traceback.

A simple off-topic example:

```
>>> def cond():
...     x = 3
...     if x < 5:
...         return 'yes'
...     else:
...         return 'no'
...
```

```
>>> dis.dis(cond)
  2           0 LOAD_CONST               1 (3)
              3 STORE_FAST               0 (x)

  3           6 LOAD_FAST                0 (x)
              9 LOAD_CONST               2 (5)
             12 COMPARE_OP               0 (<)
             15 POP_JUMP_IF_FALSE       22

  4          18 LOAD_CONST               3 ('yes')
             21 RETURN_VALUE

  6     >>   22 LOAD_CONST               4 ('no')
             25 RETURN_VALUE
             26 LOAD_CONST               0 (None)
             29 RETURN_VALUE
```

Let's look at the first instruction `LOAD_CONST` as an example. The number in the first column (2) shows the line number in our Python source code. The second column is an index into the bytecode, telling us that the `LOAD_CONST` instruction appears at position zero. The third column is the instruction itself, mapped to its human-readable name. The fourth column, when present, is the argument to that instruction. The fifth column, when present, is a hint about what the argument means.

```
dis.opname[name]
```

Sequence of operation names, indexable using the bytecode.

```
dis.HAVE_ARGUMENT
```

This is not really an opcode. It identifies the dividing line between opcodes which don't take arguments `< HAVE_ARGUMENT` and those which do `>= HAVE_ARGUMENT`.

```
dis.hasconst
```

Sequence of bytecodes that have a constant parameter.

```
dis.hasfree
```

Sequence of bytecodes that access a free variable.

```
dis.hasname
```

Sequence of bytecodes that access an attribute by name.

```
dis.hasjrel
```

Sequence of bytecodes that have a relative jump target.

```
dis.hasjabs
```

Sequence of bytecodes that have an absolute jump target.

```
dis.haslocal
```

Sequence of bytecodes that access a local variable.

Simple implementation of `dis` in
`py_VM.py`

```
if byteCode >= dis.HAVE_ARGUMENT:
            arg = f.f_code.co_code[f.f_lasti:f.f_lasti+2]
            f.f_lasti += 2
            intArg = byteint(arg[0]) + (byteint(arg[1]) << 8)
            if byteCode in dis.hasconst:
                arg = f.f_code.co_consts[intArg]
            elif byteCode in dis.hasfree:
                if intArg < len(f.f_code.co_cellvars):
                    arg = f.f_code.co_cellvars[intArg]
                else:
                    var_idx = intArg - len(f.f_code.co_cellvars)
                    arg = f.f_code.co_freevars[var_idx]
            elif byteCode in dis.hasname:
                arg = f.f_code.co_names[intArg]
            elif byteCode in dis.hasjrel:
                arg = f.f_lasti + intArg
            elif byteCode in dis.hasjabs:
                arg = intArg
            elif byteCode in dis.haslocal:
                arg = f.f_code.co_varnames[intArg]
            else:
                arg = intArg
            arguments = [arg]

        return byteName, arguments, opoffset
```

inspect

It helps to examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

linecache

The linecache module is used within other parts of the Python standard library when dealing with Python source files. The implementation of the cache holds the contents of files, parsed into separate lines, in memory. The API returns the requested line(s) by indexing into a list, and saves time over repeatedly reading the file and parsing lines to find the one desired.

```
def checkcache(filename=None)
```

Discard cache entries that are out of date.

Simple implementation in:
py_VM.py

```
linecache.checkcache(filename)
```

Some context

```
def getline(filename, line_no, module_globals=None)
```

Get line *lineno* from file named *filename*.

Simple implementation in:
py_VM.py

```
line = linecache.getline(filename, lineno, f.f_globals)
```

Some context

operator

The **operator** module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)`is equivalent to the expression `x+y`. The function names are those used for special class methods; variants without leading and trailing `__`are also provided for convenience.

Simple implementation in
py_VM.py

```
    ## Operators

    UNARY_OPERATORS = {
```

```
        'POSITIVE': operator.pos,
        'NEGATIVE': operator.neg,
        'NOT':      operator.not_,
        'CONVERT':  repr,
        'INVERT':   operator.invert,
    }

    def unaryOperator(self, op):
        x = self.pop()
        self.push(self.UNARY_OPERATORS[op](x))

    BINARY_OPERATORS = {
        'POWER':    pow,
        'MULTIPLY': operator.mul,
        'DIVIDE':   getattr(operator, 'div', lambda x, y: None),
        'FLOOR_DIVIDE': operator.floordiv,
        'TRUE_DIVIDE':  operator.truediv,
        'MODULO':   operator.mod,
        'ADD':      operator.add,
        'SUBTRACT': operator.sub,
        'SUBSCR':   operator.getitem,
        'LSHIFT':   operator.lshift,
        'RSHIFT':   operator.rshift,
        'AND':      operator.and_,
        'XOR':      operator.xor,
        'OR':       operator.or_,
    }
    def binaryOperator(self, op):
        x, y = self.popn(2)
        self.push(self.BINARY_OPERATORS[op](x, y))

    def inplaceOperator(self, op):
        x, y = self.popn(2)
        if op == 'POWER':
            x **= y
        elif op == 'MULTIPLY':
            x *= y
        elif op in ['DIVIDE', 'FLOOR_DIVIDE']:
            x //= y
        elif op == 'TRUE_DIVIDE':
            x /= y
        elif op == 'MODULO':
            x %= y
        elif op == 'ADD':
            x += y
        elif op == 'SUBTRACT':
            x -= y
```

```
        elif op == 'LSHIFT':
            x <<= y
        elif op == 'RSHIFT':
            x >>= y
        elif op == 'AND':
            x &= y
        elif op == 'XOR':
            x ^= y
        elif op == 'OR':
            x |= y
        else:           # pragma: no cover
            raise VirtualMachineError("Unknown in-place operator: %r" % op)
        self.push(x)

    COMPARE_OPERATORS = [
        operator.lt,
        operator.le,
        operator.eq,
        operator.ne,
        operator.gt,
        operator.ge,
        lambda x, y: x in y,
        lambda x, y: x not in y,
        lambda x, y: x is y,
        lambda x, y: x is not y,
        lambda x, y: issubclass(x, Exception) and issubclass(x, y),
    ]

    def byte_COMPARE_OP(self, opnum):
        x, y = self.popn(2)
        self.push(self.COMPARE_OPERATORS[opnum](x, y))
```

six

Six provides simple utilities for wrapping over differences between Python 2 and Python 3. It is intended to support codebases that work on both Python 2 and 3 without modification.

```
def exec_(_code_, _globs_=None, _locs_=None)
```

Execute code in namespace.

Simple execution in:
py_VM.py

```
 def byte_EXEC_STMT(self):
        stmt, globs, locs = self.popn(3)
        six.exec_(stmt, globs, locs)
```

Some context

```
def reraise(tp, value, tb=None)
```

Simple execution in:
py_VM.py

```
six.reraise(*self.last_exception)
```

Some context

Chapter 3

Getting started

Py-py Interpreter is a Python interpreter written in Python. This may strike you as odd, but it's no more odd than writing a C compiler in C. You could write a Python interpreter in almost any language.

Writing a Python interpreter in Python has both advantages and disadvantages. The biggest advantage to using Python is that we can more easily implement *just* the interpreter, and not the rest of the Python run-time, particularly the object system. For example, Byterun can fall back to "real" Python when it needs to create a class. Another advantage is that Byterun is easy to understand, partly because it's written in a high-level language (Python!) that many people find easy to read. (We also exclude interpreter optimizations in Byterun—once again favoring clarity and simplicity over speed.)

Chapter 4

Building an Interpreter from scratch

The Python interpreter is a *virtual machine*, meaning that it is software that emulates a physical computer. This particular virtual machine is a stack machine: it manipulates several stacks to perform its operations.

A Tiny Interpreter
Let us start with a minimal interpreter. This interpreter can only add numbers, and in its core, it has three functions.

- LOAD_VALUE
- ADD_TWO_VALUES
- PRINT_ANSWER

Now let's start to write the interpreter itself. The interpreter object has a stack, which we'll represent with a list. The object also has a method describing how to execute each instruction. For example, for LOAD_VALUE, the interpreter will push the value onto the stack.

```
class Interpreter:
    def __init__(self):
        self.stack = []

    def LOAD_VALUE(self, number):
        self.stack.append(number)

    def PRINT_ANSWER(self):
        answer = self.stack.pop()
        print(answer)

    def ADD_TWO_VALUES(self):
        first_num = self.stack.pop()
        second_num = self.stack.pop()
        total = first_num + second_num
        self.stack.append(total)
```

These three functions implement the three instructions our interpreter understands. The interpreter needs one more piece: a way to tie everything together and actually execute it.

```
what_to_execute = {
    "instructions": [("LOAD_VALUE", 0),  # the first number
                     ("LOAD_VALUE", 1),  # the second number
                     ("ADD_TWO_VALUES", None),
                     ("PRINT_ANSWER", None)],
    "numbers": [7, 5] }    # hardcoded values

def execute(self, what_to_execute):
        instructions = what_to_execute["instructions"]
        for each_step in instructions:
            instruction, argument = each_step
            argument = self.parse_argument(instruction, argument, what_to_execute)
            bytecode_method = getattr(self, instruction)
            if argument is None:
                bytecode_method()
            else:
                bytecode_method(argument)
```

Chapter 5

Classes

The VirtualMachine Class

Only one instance of **VirtualMachine** will be created each time the program is run, because we only have one Python interpreter. **VirtualMachine** stores the call stack, the exception state, and return values while they're being passed between frames. The entry point for executing code is the method run_code, which takes a compiled code object as an argument. It starts by setting up and running a frame. This frame may create other frames; the call stack will grow and shrink as the program executes. When the first frame eventually returns, execution is finished.

```
class VirtualMachineError(Exception):
    pass

class VirtualMachine(object):
    def __init__(self):
        self.frames = []   # The call stack of frames.
        self.frame = None  # The current frame.
        self.return_value = None
        self.last_exception = None

    def run_code(self, code, global_names=None, local_names=None):
        """ An entry point to execute code using the virtual machine."""
        frame = self.make_frame(code, global_names=global_names,
                                local_names=local_names)
        self.run_frame(frame)
```

See The Frame Class[1] for the next step.

Next, we'll add frame manipulation to the virtual machine. There are three helper functions for frames: one to create new frames (which is responsible for sorting out the namespaces for the new frame) and one each to push and pop frames on and off the frame stack. A fourth function, run_frame, does the main work of executing a frame. We'll come back to this soon.

```
class VirtualMachine(object):
    [... snip ...]
```

```
    # Frame manipulation
    def make_frame(self, code, callargs={}, global_names=None, local_names=None):
        if global_names is not None and local_names is not None:
            local_names = global_names
        elif self.frames:
            global_names = self.frame.global_names
            local_names = {}
        else:
            global_names = local_names = {
                '__builtins__': __builtins__,
                '__name__': '__main__',
                '__doc__': None,
                '__package__': None,
            }
        local_names.update(callargs)
        frame = Frame(code, global_names, local_names, self.frame)
        return frame

    def push_frame(self, frame):
        self.frames.append(frame)
        self.frame = frame

    def pop_frame(self):
        self.frames.pop()
        if self.frames:
            self.frame = self.frames[-1]
        else:
            self.frame = None

    def run_frame(self):
        pass
        # we'll come back to this shortly
```

See The Function Class[2] for the next step.

Next, back on the `VirtualMachine` object, we'll add some helper methods for data stack manipulation. The bytecodes that manipulate the stack always operate on the current frame's data stack. This will make our implementations of `POP_TOP`, `LOAD_FAST`, and all the other instructions that touch the stack more readable.

```
class VirtualMachine(object):
    [... snip ...]

    # Data stack manipulation
```

```
    def top(self):
        return self.frame.stack[-1]

    def pop(self):
        return self.frame.stack.pop()

    def push(self, *vals):    # Extend the frame stack
        self.frame.stack.extend(vals)

    def popn(self, n):
        """Pop a number of values from the value stack.
        A list of `n` values is returned, the deepest value first.
        """
        if n:
            ret = self.frame.stack[-n:]
            self.frame.stack[-n:] = []
            return ret
        else:
            return []
```

Before we get to running a frame, we need two more methods.

The first, parse_byte_and_args, takes a bytecode, checks if it has arguments, and parses the arguments if so. This method also updates the frame's attribute last_instruction, a reference to the last instruction executed. The meaning of the argument to each instruction depends on which instruction it is. For example, as mentioned above, for POP_JUMP_IF_FALSE, the argument to the instruction is the jump target. For BUILD_LIST, it is the number of elements in the list. For LOAD_CONST, it's an index into the list of constants.

The dismodule in the standard library exposes a cheatsheet explaining what arguments have what meaning, which makes our code more compact. For example, the list dis.hasname tells us that the arguments to LOAD_NAME, IMPORT_NAME, LOAD_GLOBAL, and nine other instructions have the same meaning: for these instructions, the argument represents an index into the list of names on the code object.

```
class VirtualMachine(object):
    [... snip ...]

    def parse_byte_and_args(self):
        f = self.frame
        opoffset = f.last_instruction
        byteCode = f.code_obj.co_code[opoffset]
```

```
        f.last_instruction += 1
        byte_name = dis.opname[byteCode]
        if byteCode >= dis.HAVE_ARGUMENT:
            # index into the bytecode
            arg = f.code_obj.co_code[f.last_instruction:f.last_instruction+2]
            f.last_instruction += 2   # advance the instruction pointer
            arg_val = arg[0] + (arg[1] * 256)
            if byteCode in dis.hasconst:   # Look up a constant
                arg = f.code_obj.co_consts[arg_val]
            elif byteCode in dis.hasname:  # Look up a name
                arg = f.code_obj.co_names[arg_val]
            elif byteCode in dis.haslocal: # Look up a local name
                arg = f.code_obj.co_varnames[arg_val]
            elif byteCode in dis.hasjrel:  # Calculate a relative jump
                arg = f.last_instruction + arg_val
            else:
                arg = arg_val
            argument = [arg]
        else:
            argument = []

        return byte_name, argument
```

The next method is `dispatch`, which looks up the operations for a given instruction and executes them. In the CPython interpreter, this dispatch is done with a giant switch statement that spans 1,500 lines! Luckily, since we're writing Python, we can be more compact. We'll define a method for each byte name and then use `getattr` to look it up. Each bytecode method will return either `None` or a string, called `why`, which is an extra piece of state the interpreter needs in some cases. These return values of the individual instruction methods are used only as internal indicators of interpreter state—don't confuse these with return values from executing frames.

```
class VirtualMachine(object):
    [... snip ...]

    def dispatch(self, byte_name, argument):
        """ Dispatch by bytename to the corresponding methods.
        Exceptions are caught and set on the virtual machine."""

        # When later unwinding the block stack,
        # we need to keep track of why we are doing it.
        why = None
        try:
            bytecode_fn = getattr(self, 'byte_%s' % byte_name, None)
            if bytecode_fn is None:
                if byte_name.startswith('UNARY_'):
```

```
                    self.unaryOperator(byte_name[6:])
                elif byte_name.startswith('BINARY_'):
                    self.binaryOperator(byte_name[7:])
                else:
                    raise VirtualMachineError(
                        "unsupported bytecode type: %s" % byte_name
                    )
            else:
                why = bytecode_fn(*argument)
        except:
            # deal with exceptions encountered while executing the op.
            self.last_exception = sys.exc_info()[:2] + (None,)
            why = 'exception'

        return why

    def run_frame(self, frame):
        """Run a frame until it returns (somehow).
        Exceptions are raised, the return value is returned.
        """
        self.push_frame(frame)
        while True:
            byte_name, arguments = self.parse_byte_and_args()

            why = self.dispatch(byte_name, arguments)

            # Deal with any block management we need to do
            while why and frame.block_stack:
                why = self.manage_block_stack(why)

            if why:
                break

        self.pop_frame()

        if why == 'exception':
            exc, val, tb = self.last_exception
            e = exc(val)
            e.__traceback__ = tb
            raise e

        return self.return_value
```

The Frame Class

Next we'll write the `Frame` object. The frame is a collection of attributes with no methods. As mentioned above, the attributes include the code object created by the compiler; the local,

global, and builtin namespaces; a reference to the previous frame; a data stack; a block stack; and the last instruction executed.

Then, we'll add frame manipulation to the virtual machine. There are three helper functions for frames: one to create new frames (which is responsible for sorting out the namespaces for the new frame) and one each to push and pop frames on and off the frame stack. A fourth function, `run_frame`, does the main work of executing a frame.

The Function Class

The important thing to notice is that calling a function—invoking the `__call__` method—creates a new `Frame` object and starts running it.

```
def __init__(self, name, code, globs, defaults, closure, vm)
```

Now run the frame:

```
def __call__(self, *args, **kwargs):
     """When calling a Function, make a new frame and run it."""
     callargs = inspect.getcallargs(self._func, *args, **kwargs)
     # Use callargs to provide a mapping of arguments: values to pass into the new
     # frame.
     frame = self._vm.make_frame(
         self.func_code, callargs, self.func_globals, {}
     )
     return self._vm.run_frame(frame)
```

The Block Class

A block is used for certain kinds of flow control, specifically exception handling and looping. The block is responsible for making sure that the data stack is in the appropriate state when the operation is finished. For example, in a loop, a special iterator object remains on the stack while the loop is running, but is popped off when it is finished. The interpreter must keep track of whether the loop is continuing or is finished.

```
def manage_block_stack(self, why):
    [... snip ...]
if block.type == 'loop' and why == 'continue':
            self.jump(self.return_value)
            why = None
            return why
```

```
    [... snip ...]
if block.type == 'loop' and why == 'break':
            why = None
            self.jump(block.handler)
            return why
    [... snip ...]
if (block.type in ['setup-except', 'finally'] and why == 'exception'):
            self.push_block('except-handler')
            exctype, value, tb = self.last_exception
            self.push(tb, value, exctype)
            self.push(tb, value, exctype) # yes, twice
            why = None
            self.jump(block.handler)
            return why
    [... snip ...]
elif block.type == 'finally':
            if why in ('return', 'continue'):
                self.push(self.return_value)

            self.push(why)

            why = None
            self.jump(block.handler)
            return why
```

To keep track of this extra piece of information, the interpreter sets a flag to indicate its state. We implement this flag as a variable called `why`, which can be `None` or one of the strings `"continue"`, `"break"`, `"exception"`, or `"return"`. This indicates what kind of manipulation of the block stack and data stack should happen. To return to the iterator example, if the top of the block stack is a `loop` block and the `why` code is `continue`, the iterator object should remain on the data stack, but if the `why` code is `break`, it should be popped off.

Chapter 7

The Instructions

All that is left is to implement dozens of methods for instructions.
Stack manipulation:

```
def byte_LOAD_CONST(self, const):
```

Loads a constant from the data stack.

```
def byte_POP_TOP(self):
```

Pops the top value from the stack.

Names:

```
def byte_LOAD_NAME(self, name):
```

Return the top level variable name from the stack.

```
def byte_STORE_NAME(self, name):
```

Stores a variable name in the frame from the stack.

```
def byte_LOAD_FAST(self, name):
```

Return a local variable in scope of current frame. Else raise an error.

```
def byte_STORE_FAST(self, name):
```

Stores a variable name from expressions.

```
def byte_LOAD_GLOBAL(self, name):
```

Stores a global variable name in current stack frame.

For reference see operators[2].

```
def byte_LOAD_ATTR(self, attr):
```

Stores a global variable name in current stack frame.

```
def byte_STORE_ATTR(self, name):
```

Sets attribute for object code, variable name and its value.

Building lists, maps

```
def byte_BUILD_LIST(self, count):
```

Pops count number of elements from the stack. Creates and pushes a new list into data stack.

```
def byte_BUILD_MAP(self, size):
```

Pushes an empty dictionary into the data stack.

```
def byte_STORE_MAP(self):
```

Pops three elements from the stack. First the map itself, second, list of all values and finally list of all keys. Adds data in map and pushes it into data stack.

```
def byte_APPEND_LIST(self, count):
```

Pops the value to append from the stack. Then from current frame stack, gets the list at location `-count`. Appends the list.

Jumps and Conditionals

```
def byte_JUMP_FORWARD(self, jump):
```

Jump to jump location inside the loop in the local loop's stack frame.

```
def byte_JUMP_ABSOLUTE(self, jump):
```

Jump to jump location outside the loop or anywhere else in the local function's stack frame.

```
def byte_POP_JUMP_IF_TRUE(self, jump):
```

Jump to jump location in true cases and conditional expressions.

```
def byte_POP_JUMP_IF_FALSE(self, jump):
```

Jump to jump location in false cases and conditional expressions.

Iteratives

```
def byte_SETUP_LOOP(self, dest):
```

Create a block to let the VM know that a loop follows the given destination.

```
def byte_GET_ITER(self):
```

Pushes the iterative form of the list into the stack.

```
def byte_FOR_ITER(self, jump):
```

Gets the next iterative object, gets its value and pushes it into the stack. If None, then Jump to jump location.

```
def byte_BREAK_LOOP(self, jump):
```

Returns “break”.

```
def byte_POP_BLOCK(self):
```

Pops the loop’s block out of the stack when jump location is outside of the loop.

Functions:

```
def byte_MAKE_FUNCTION(self, argc):
```

Gets the name, code, defaults argc number of arguments from the stack and global variables. Creates an object of the class Frame with name, code, globals, defaults, closure, vm as arguments then pushes it into the stack.

```
def byte_CALL_FUNCTION(self, arg):
```

Gets the function and its frame. Executes and pushes the return value into the stack.

```
def byte_RETURN_VALUE(self):
```

Pops the return value out of the stack and creates a “return” block.

Chapter 8

Cons

py-py Interpreter is a compact Python interpreter that's easier to understand than CPython. Yet it is not the full implementation of cpython.c in python. The biggest disadvantage is speed: executing code via this interpreter is much slower than executing it in CPython, where the interpreter is written in C and carefully optimized. Also, CPython was used to import modules. However, Byterun was designed originally as a learning exercise, so speed is not important that important. The biggest advantage to using Python is that we can more easily implement *just* the interpreter, and not the rest of the Python run-time, particularly the object system.

Chapter 9

Issues

Following points very briefly summarizes the issues byterun is currently having.

- Does not support Python 3.6
- Does not support list comprehension
- CPython used for imported modules rather than byterun
- closure Frame initialization uses wrong outer frame

Chapter 10

Acknowledgement

Thanks to Ned Batchelder, Allison Kaptur, Michael Arntzenius, Leta Montopoli and Recurse Center community for originating this project, debugging the code and editing the prose, edits. Any errors are my own.

Closing Chapter

Supplementary resources

Python dependencies:
`argparse`
`logging`
`imp`
`os`
`sys`
`tokenize`
`dis`
`inspect`
`linecache`
`operator`
`six`

Suggested reading:
http://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html
Computational Complexity: A Modern Approach

Source code available on GitHub:
Author's repository : https://github.com/nedbat/byterun