bytecode Documentation

Release 0.5

Victor Stinner

January 09, 2017

Contents

`bytecode` is a Python module to generate and modify bytecode.

- bytecode project homepage at GitHub (code, bugs)
- bytecode documentation (this documentation)
- Download latest bytecode release at the Python Cheeseshop (PyPI)

CHAPTER 1

Table Of Contents

1.1 Bytecode Usage

1.1.1 Installation

Install bytecode:

```
python3 -m pip install bytecode
```

`bytecode` requires Python 3.4 or newer.

1.1.2 Hello World

Abstract bytecode

Example using abstract bytecode to execute `print('Hello World!')`:

```
from bytecode import Instr, Bytecode

bytecode = Bytecode([Instr("LOAD_NAME", 'print'),
                     Instr("LOAD_CONST", 'Hello World!'),
                     Instr("CALL_FUNCTION", 1),
                     Instr("POP_TOP"),
                     Instr("LOAD_CONST", None),
                     Instr("RETURN_VALUE")])
code = bytecode.to_code()
exec(code)
```

Output:

```
Hello World!
```

Concrete bytecode

Example using concrete bytecode to execute `print('Hello World!')`:

```
from bytecode import ConcreteInstr, ConcreteBytecode

bytecode = ConcreteBytecode()
bytecode.names = ['print']
```

```
bytecode.consts = ['Hello World!', None]
bytecode.extend([ConcreteInstr("LOAD_NAME", 0),
                 ConcreteInstr("LOAD_CONST", 0),
                 ConcreteInstr("CALL_FUNCTION", 1),
                 ConcreteInstr("POP_TOP"),
                 ConcreteInstr("LOAD_CONST", 1),
                 ConcreteInstr("RETURN_VALUE")])
code = bytecode.to_code()
exec(code)
```

Output:

```
Hello World!
```

1.1.3 Simple loop

Bytecode of `for x in (1, 2, 3): print(x)`:

```
from bytecode import Label, Instr, Bytecode

loop_start = Label()
loop_done = Label()
loop_exit = Label()
code = Bytecode([Instr('SETUP_LOOP', loop_exit),
                 Instr('LOAD_CONST', (1, 2, 3)),
                 Instr('GET_ITER'),
                 loop_start,
                     Instr('FOR_ITER', loop_done),
                     Instr('STORE_NAME', 'x'),
                     Instr('LOAD_NAME', 'print'),
                     Instr('LOAD_NAME', 'x'),
                     Instr('CALL_FUNCTION', 1),
                     Instr('POP_TOP'),
                     Instr('JUMP_ABSOLUTE', loop_start),
                 loop_done,
                     Instr('POP_BLOCK'),
                 loop_exit,
                     Instr('LOAD_CONST', None),
                     Instr('RETURN_VALUE')])

# the conversion to Python code object resolve jump targets:
# replace abstract labels with concrete offsets
code = code.to_code()
exec(code)
```

Output:

```
1
2
3
```

1.1.4 Conditional jump

Bytecode of the Python code `print('yes' if test else 'no')`:

```
from bytecode import Label, Instr, Bytecode

label_else = Label()
label_print = Label()
bytecode = Bytecode([Instr('LOAD_NAME', 'print'),
                     Instr('LOAD_NAME', 'test'),
                     Instr('POP_JUMP_IF_FALSE', label_else),
                         Instr('LOAD_CONST', 'yes'),
                         Instr('JUMP_FORWARD', label_print),
                     label_else,
                         Instr('LOAD_CONST', 'no'),
                     label_print,
                         Instr('CALL_FUNCTION', 1),
                     Instr('LOAD_CONST', None),
                     Instr('RETURN_VALUE')])
code = bytecode.to_code()

test = 0
exec(code)

test = 1
exec(code)
```

Output:

```
no
yes
```

Note: Instructions are only indented for readability.

1.2 Control Flow Graph (CFG)

To analyze or optimize existing code, `bytecode` provides a *`ControlFlowGraph`* class which is a control flow graph (CFG).

The control flow graph is used to perform the stack depth analysis when converting to code. Because it is better at identifying dead code than CPython it can lead to reduced stack size.

1.2.1 Example

Dump the control flow graph of the *conditional jump example*:

```
from bytecode import Label, Instr, Bytecode, ControlFlowGraph, dump_bytecode

label_else = Label()
label_print = Label()
bytecode = Bytecode([Instr('LOAD_NAME', 'print'),
                     Instr('LOAD_NAME', 'test'),
                     Instr('POP_JUMP_IF_FALSE', label_else),
                         Instr('LOAD_CONST', 'yes'),
                         Instr('JUMP_FORWARD', label_print),
                     label_else,
                         Instr('LOAD_CONST', 'no'),
```

```
                    label_print,
                        Instr('CALL_FUNCTION', 1),
                    Instr('LOAD_CONST', None),
                    Instr('RETURN_VALUE')])

blocks = ControlFlowGraph.from_bytecode(bytecode)
dump_bytecode(blocks)
```

Output:

```
block1:
    LOAD_NAME 'print'
    LOAD_NAME 'test'
    POP_JUMP_IF_FALSE <block3>
    -> block2

block2:
    LOAD_CONST 'yes'
    JUMP_FORWARD <block4>

block3:
    LOAD_CONST 'no'
    -> block4

block4:
    CALL_FUNCTION 1
    LOAD_CONST None
    RETURN_VALUE
```

We get 4 blocks:

- block #1 is the start block and ends with `POP_JUMP_IF_FALSE` conditional jump and is followed by the block #2
- block #2 ends with `JUMP_FORWARD` uncondition jump
- block #3 does not contain jump and is followed by the block #4
- block #4 is the final block

The start block is always the first block.

1.2.2 Analyze the control flow graph

The `bytecode` module provides two ways to iterate on blocks:

- iterate on the basic block as a sequential list
- browse the graph by following jumps and links to next blocks

Iterate on basic blocks

Iterating on basic blocks is a simple as this loop:

```
for block in blocks:
    ...
```

Example of a `display_blocks()` function:

```
from bytecode import UNSET, Label, Instr, Bytecode, BasicBlock, ControlFlowGraph

def display_blocks(blocks):
    for block in blocks:
        print("Block #%s" % (1 + blocks.get_block_index(block)))
        for instr in block:
            if isinstance(instr.arg, BasicBlock):
                arg = "<block #%s>" % (1 + blocks.get_block_index(instr.arg))
            elif instr.arg is not UNSET:
                arg = repr(instr.arg)
            else:
                arg = ''
            print("    %s %s" % (instr.name, arg))

        if block.next_block is not None:
            print("    => <block #%s>"
                  % (1 + blocks.get_block_index(block.next_block)))

        print()

label_else = Label()
label_print = Label()
bytecode = Bytecode([Instr('LOAD_NAME', 'print'),
                     Instr('LOAD_NAME', 'test'),
                     Instr('POP_JUMP_IF_FALSE', label_else),
                         Instr('LOAD_CONST', 'yes'),
                         Instr('JUMP_FORWARD', label_print),
                     label_else,
                         Instr('LOAD_CONST', 'no'),
                     label_print,
                         Instr('CALL_FUNCTION', 1),
                     Instr('LOAD_CONST', None),
                     Instr('RETURN_VALUE')])

blocks = ControlFlowGraph.from_bytecode(bytecode)
display_blocks(blocks)
```

Output:

```
Block #1
    LOAD_NAME 'print'
    LOAD_NAME 'test'
    POP_JUMP_IF_FALSE <block #3>
    => <block #2>

Block #2
    LOAD_CONST 'yes'
    JUMP_FORWARD <block #4>

Block #3
    LOAD_CONST 'no'
    => <block #4>

Block #4
    CALL_FUNCTION 1
    LOAD_CONST None
    RETURN_VALUE
```

Note: *SetLineno* is not handled in the example to keep it simple.

Browse the graph

Recursive function is a simple solution to browse the control flow graph.

Example to a recursive `display_block()` function:

```
from bytecode import UNSET, Label, Instr, Bytecode, BasicBlock, ControlFlowGraph

def display_block(blocks, block, seen=None):
    # avoid loop: remember which blocks were already seen
    if seen is None:
        seen = set()
    if id(block) in seen:
        return
    seen.add(id(block))

    # display instructions of the block
    print("Block #%s" % (1 + blocks.get_block_index(block)))
    for instr in block:
        if isinstance(instr.arg, BasicBlock):
            arg = "<block #%s>" % (1 + blocks.get_block_index(instr.arg))
        elif instr.arg is not UNSET:
            arg = repr(instr.arg)
        else:
            arg = ''
        print("    %s %s" % (instr.name, arg))

    # is the block followed directly by another block?
    if block.next_block is not None:
        print("    => <block #%s>"
              % (1 + blocks.get_block_index(block.next_block)))

    print()

    # display the next block
    if block.next_block is not None:
        display_block(blocks, block.next_block, seen)

    # display the block linked by jump (if any)
    target_block = block.get_jump()
    if target_block is not None:
        display_block(blocks, target_block, seen)

label_else = Label()
label_print = Label()
bytecode = Bytecode([Instr('LOAD_NAME', 'print'),
                     Instr('LOAD_NAME', 'test'),
                     Instr('POP_JUMP_IF_FALSE', label_else),
                         Instr('LOAD_CONST', 'yes'),
                         Instr('JUMP_FORWARD', label_print),
                     label_else,
                         Instr('LOAD_CONST', 'no'),
                     label_print,
                         Instr('CALL_FUNCTION', 1),
```

```
                    Instr('LOAD_CONST', None),
                    Instr('RETURN_VALUE')])

blocks = ControlFlowGraph.from_bytecode(bytecode)
display_block(blocks, blocks[0])
```

Output:

```
Block #1
    LOAD_NAME 'print'
    LOAD_NAME 'test'
    POP_JUMP_IF_FALSE <block #3>
    => <block #2>

Block #2
    LOAD_CONST 'yes'
    JUMP_FORWARD <block #4>

Block #4
    CALL_FUNCTION 1
    LOAD_CONST None
    RETURN_VALUE

Block #3
    LOAD_CONST 'no'
    => <block #4>
```

Block numbers are no displayed in the sequential order: block #4 is displayed before block #3.

Note: Dead code (unreachable blocks) is not displayed by `display_block`.

1.3 Bytecode API

- Constants: *__version__*, *UNSET*
- Abstract bytecode: *Label*, *Instr*, *Bytecode*
- Line number: *SetLineno*
- Arguments: *CellVar*, *Compare*, *FreeVar*
- Concrete bytecode: *ConcreteInstr*, *ConcreteBytecode*
- Control Flow Graph (CFG): *BasicBlock*, *ControlFlowGraph*
- Base class: *BaseBytecode*

1.3.1 Constants

__version__
Module version string (ex: `'0.1'`).

UNSET
Singleton used to mark the lack of value. It is different than `None`.

1.3.2 Functions

dump_bytecode(*bytecode*, *, *lineno=False*)
Dump a bytecode to the standard output. *ConcreteBytecode*, *Bytecode* and *ControlFlowGraph* are accepted for *bytecode*.

If *lineno* is true, show also line numbers and instruction index/offset.

This function is written for debug purpose.

1.3.3 Instruction classes

Instr

class **Instr**(*name: str*, *arg=UNSET*, *, *lineno: int=None*)
Abstract instruction.

The type of the *arg* parameter (and the *arg* attribute) depends on the operation:

- If the operation has a jump argument (*has_jump()*, ex: JUMP_ABSOLUTE): *arg* must be a *Label* (if the instruction is used in *Bytecode*) or a *BasicBlock* (used in *ControlFlowGraph*).
- If the operation has a cell or free argument (ex: LOAD_DEREF): *arg* must be a *CellVar* or *FreeVar* instance.
- If the operation has a local variable (ex: LOAD_FAST): *arg* must be a variable name, type str.
- If the operation has a constant argument (LOAD_CONST): *arg* must not be a *Label* or *BasicBlock* instance.
- If the operation has a compare argument (COMPARE_OP): *arg* must be a *Compare* enum.
- If the operation has no argument (ex: DUP_TOP), *arg* must not be set.
- Otherwise (the operation has an argument, ex: CALL_FUNCTION), *arg* must be an integer (int) in the range 0..2,147,483,647.

To replace the operation name and the argument, the *set()* method must be used instead of modifying the *name* attribute and then the *arg* attribute. Otherwise, an exception is be raised if the previous operation requires an argument and the new operation has no argument (or the opposite).

Attributes:

arg
Argument value.

It can be *UNSET* if the instruction has no argument.

lineno
Line number (int >= 1), or None.

name
Operation name (str). Setting the name updates the *opcode* attibute.

opcode
Operation code (int). Setting the operation code updates the *name* attribute.

stack_effect
Operation effect on the stack size as computed by dis.stack_effect().

Changed in version 0.3: The op attribute was renamed to *opcode*.

Methods:

copy()
Create a copy of the instruction.

is_final() → bool
Is the operation a final operation?

Final operations:

- RETURN_VALUE
- RAISE_VARARGS
- BREAK_LOOP
- CONTINUE_LOOP
- unconditional jumps: *is_uncond_jump()*

has_jump() → bool
Does the operation have a jump argument?

More general than *is_cond_jump()* and *is_uncond_jump()*, it includes other operations. Examples:

- FOR_ITER
- SETUP_EXCEPT
- CONTINUE_LOOP

is_cond_jump() → bool
Is the operation an conditional jump?

Conditional jumps:

- JUMP_IF_FALSE_OR_POP
- JUMP_IF_TRUE_OR_POP
- POP_JUMP_IF_FALSE
- POP_JUMP_IF_TRUE

is_uncond_jump() → bool
Is the operation an unconditional jump?

Unconditional jumps:

- JUMP_FORWARD
- JUMP_ABSOLUTE

set(*name: str*, *arg=UNSET*)
Modify the instruction in-place: replace *name* and *arg* attributes, and update the *opcode* attribute.

Changed in version 0.3: The *lineno* parameter has been removed.

ConcreteInstr

class ConcreteInstr(*name: str*, *arg=UNSET*, *, *lineno: int=None*)
Concrete instruction Inherit from *Instr*.

If the operation requires an argument, *arg* must be an integer (`int`) in the range `0..2,147,483,647`. Otherwise, *arg* must not by set.

Concrete instructions should only be used in *ConcreteBytecode*.

Attributes:

arg
Argument value: an integer (`int`) in the range `0..2,147,483,647`, or *`UNSET`*. Setting the argument value can change the instruction size (*`size`*).

size
Read-only size of the instruction in bytes (`int`): between `1` byte (no agument) and `6` bytes (extended argument).

Static method:

static disassemble(*code: bytes*, *offset: int*) → ConcreteInstr
Create a concrete instruction from a bytecode string.

Methods:

get_jump_target(*instr_offset: int*) → int or None
Get the absolute target offset of a jump. Return `None` if the instruction is not a jump.

The *instr_offset* parameter is the offset of the instruction. It is required by relative jumps.

assemble() → bytes
Assemble the instruction to a bytecode string.

Compare

class Compare
Enum for the argument of the `COMPARE_OP` instruction.

Equality test:

- `Compare.EQ` (`2`): `x == y`
- `Compare.NE` (`3`): `x != y`
- `Compare.IS` (`8`): `x is y`
- `Compare.IS_NOT` (`9`): `x is not y`

Inequality test:

- `Compare.LT` (`0`): `x < y`
- `Compare.LE` (`1`): `x <= y`
- `Compare.GT` (`4`): `x > y`
- `Compare.GE` (`5`): `x >= y`

Other tests:

- `Compare.IN` (`6`): `x in y`
- `Compare.NOT_IN` (`7`): `x not in y`
- `Compare.EXC_MATCH` (`10`): used to compare exceptions in `except :` blocks

Label

class Label
Pseudo-instruction used as targets of jump instructions.

Label targets are "resolved" by *`Bytecode.to_concrete_bytecode`*.

Labels must only be used in *Bytecode*.

SetLineno

class **SetLineno** (*lineno: int*)
Pseudo-instruction to set the line number of following instructions.

lineno must be greater or equal than 1.

lineno
Line number (int), read-only attribute.

1.3.4 Bytecode classes

BaseBytecode

class **BaseBytecode**
Base class of bytecode classes.

Attributes:

argcount
Argument count (int), default: 0.

cellvars
Names of the cell variables (list of str), default: empty list.

docstring
Documentation string aka "docstring" (str), None, or *UNSET*. Default: *UNSET*.

If set, it is used by *ConcreteBytecode.to_code()* as the first constant of the created Python code object.

filename
Code filename (str), default: '<string>'.

first_lineno
First line number (int), default: 1.

flags
Flags (int).

freevars
List of free variable names (list of str), default: empty list.

kwonlyargcount
Keyword-only argument count (int), default: 0.

name
Code name (str), default: '<module>'.

Changed in version 0.3: Attribute kw_only_argcount renamed to *kwonlyargcount*.

Bytecode

class **Bytecode**
Abstract bytecode: list of abstract instructions (*Instr*). Inherit from *BaseBytecode* and list.

A bytecode must only contain objects of the 4 following types:

- *Label*
- *SetLineno*
- *Instr*
- *ConcreteInstr*

It is possible to use concrete instructions (*ConcreteInstr*), but abstract instructions are preferred.

Attributes:

argnames
List of the argument names (`list` of `str`), default: empty list.

Static methods:

static from_code(*code*) → Bytecode
Create an abstract bytecode from a Python code object.

Methods:

to_concrete_bytecode() → ConcreteBytecode
Convert to concrete bytecode with concrete instructions.

Resolve jump targets: replace abstract labels (*Label*) with concrete instruction offsets (relative or absolute, depending on the jump operation).

to_code() → types.CodeType
Convert to a Python code object.

It is based on *to_concrete_bytecode()* and so resolve jump targets.

compute_stacksize() → int
Compute the stacksize needed to execute the code. Will raise an exception if the bytecode is invalid.

This computation requires to build the control flow graph associated with the code.

ConcreteBytecode

class ConcreteBytecode
List of concrete instructions (*ConcreteInstr*). Inherit from *BaseBytecode*.

A concrete bytecode must only contain objects of the 2 following types:

- *SetLineno*
- *ConcreteInstr*

Label and *Instr* must not be used in concrete bytecode.

Attributes:

consts
List of constants (`list`), default: empty list.

names
List of names (`list` of `str`), default: empty list.

varnames
List of variable names (`list` of `str`), default: empty list.

Static methods:

static from_code(*code*, *, *extended_arg=false*) → ConcreteBytecode
Create a concrete bytecode from a Python code object.

If *extended_arg* is true, create `EXTENDED_ARG` instructions. Otherwise, concrete instruction use extended argument (size of `6` bytes rather than `3` bytes).

Methods:

to_code() → types.CodeType
Convert to a Python code object.

On Python older than 3.6, raise an exception on negative line number delta.

to_bytecode() → Bytecode
Convert to abstract bytecode with abstract instructions.

compute_stacksize() → int
Compute the stacksize needed to execute the code. Will raise an exception if the bytecode is invalid.

This computation requires to build the control flow graph associated with the code.

BasicBlock

class BasicBlock
Basic block. Inherit from `list`.

A basic block is a straight-line code sequence of abstract instructions (*Instr*) with no branches in except to the entry and no branches out except at the exit.

A block must only contain objects of the 3 following types:

- *SetLineno*
- *Instr*
- *ConcreteInstr*

It is possible to use concrete instructions (*ConcreteInstr*) in blocks, but abstract instructions (*Instr*) are preferred.

Only the last instruction can have a jump argument, and the jump argument must be a basic block (*BasicBlock*).

Labels (*Label*) must not be used in blocks.

Attributes:

next_block
Next basic block (*BasicBlock*), or `None`.

Methods:

get_jump()
Get the target block (*BasicBlock*) of the jump if the basic block ends with an instruction with a jump argument. Otherwise, return `None`.

ControlFlowGraph

class ControlFlowGraph
Control flow graph (CFG): list of basic blocks (*BasicBlock*). A basic block is a straight-line code sequence of abstract instructions (*Instr*) with no branches in except to the entry and no branches out except at the exit. Inherit from *BaseBytecode*.

Labels (*Label*) must not be used in blocks.

This class is not designed to emit code, but to analyze and modify existing code. Use *Bytecode* to emit code.

Attributes:

argnames
List of the argument names (`list` of `str`), default: empty list.

Methods:

static from_bytecode(*bytecode: Bytecode*) → ControlFlowGraph
Convert a *Bytecode* object to a *ControlFlowGraph* object: convert labels to blocks.

Splits blocks after final instructions (*Instr.is_final()*) and after conditional jumps (*Instr.is_cond_jump()*).

add_block(*instructions=None*) → BasicBlock
Add a new basic block. Return the newly created basic block.

get_block_index(*block: BasicBlock*) → int
Get the index of a block in the bytecode.

Raise a `ValueError` if the block is not part of the bytecode.

New in version 0.3.

split_block(*block: BasicBlock*, *index: int*) → BasicBlock
Split a block into two blocks at the specific instruction. Return the newly created block, or *block* if index equals `0`.

to_bytecode() → Bytecode
Convert to a bytecode object using labels.

compute_stacksize() → int
Compute the stack size required by a bytecode object. Will raise an exception if the bytecode is invalid.

1.3.5 Cell and Free Variables

CellVar

class CellVar(*name: str*)
Cell variable used for instruction argument by operations taking a cell or free variable name.

Attributes:

name
Name of the cell variable (`str`).

FreeVar

class FreeVar(*name: str*)
Free variable used for instruction argument by operations taking a cell or free variable name.

Attributes:

name
Name of the free variable (`str`).

1.3.6 Line Numbers

The line number can set directly on an instruction using the `lineno` parameter of the constructor. Otherwise, the line number if inherited from the previous instruction, starting at `first_lineno` of the bytecode.

`SetLineno` pseudo-instruction can be used to set the line number of following instructions.

1.4 Peephole Optimizer

Peephole optimizer: optimize Python code objects. It is implemented with the `BytecodeBlocks` class.

It is based on the peephole optimizer of CPython 3.6 which is written in C.

1.4.1 API

Content of the `bytecode.peephole_opt` module:

class `PeepholeOptimizer`

`optimize`(*code: types.CodeType*) → types.CodeType
Optimize a Python code object.

Return a new optimized Python code object.

class `CodeTransformer`
Code transformer for the API of the PEP 511 (API for code transformers).

`code_transformer`(*code*, *context*)
Run the *`PeepholeOptimizer`* optimizer on the code.

Return a new optimized Python code object.

1.4.2 Example

Code:

```
import dis
from bytecode.peephole_opt import PeepholeOptimizer

code = compile('print(5+5)', '<string>', 'exec')
print("Non-optimized:")
dis.dis(code)
print()

code = PeepholeOptimizer().optimize(code)
print("Optimized:")
dis.dis(code)
```

Output of Python 3.6 patched with the PEP 511 with `python -o noopt` (to disable the builtin peephole optimizer):

```
Non-optimized:
  1           0 LOAD_NAME                0 (print)
              3 LOAD_CONST               0 (5)
              6 LOAD_CONST               0 (5)
              9 BINARY_ADD
             10 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
```

```
             13 POP_TOP
             14 LOAD_CONST               1 (None)
             17 RETURN_VALUE

Optimized:
  1           0 LOAD_NAME                0 (print)
              3 LOAD_CONST               0 (10)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP
             10 LOAD_CONST               1 (None)
             13 RETURN_VALUE
```

1.4.3 Optimizations

Optimizations implemented in the peephole optimizer:

- Constant folding
  - unary operations: +a, -a, ~a
  - binary operations:
    - a + b, a - b, a * b, a / b, a // b, a % b, a ** b
    - a << b, a >> b, a & b, a | b, a ^ b
  - BUILD_TUPLE: convert to a constant
  - Replace BUILD_TUPLE <n> + UNPACK_SEQUENCE <n> and BUILD_LIST <n> + UNPACK_SEQUENCE <n> with ROT_TWO (2 arguments) or ROT_THREE+ROT_TWO (3 arguments). For BUILD_LIST, if inputs are LOAD_CONST, rewrite LOAD_CONST in the reverse order.
  - BUILD_LIST + COMPARE_OP(in/not in): convert list to tuple
  - BUILD_SET + COMPARE_OP(in/not in): convert set to frozenset
  - COMPARE_OP:
    - replace `not(a is b)` with `a is not b`
    - replace `not(a is not b)` with `a is b`
    - replace `not(a in b)` with `a not in b`
    - replace `not(a not in b)` with `a in b`
- Remove NOP instructions
- Dead code elimination
  - Remove unreachable code after a final operation (*`Instr.is_final()`*)
  - Remove unreachable blocks (`Block`)
- Replace UNARY_NOT+POP_JUMP_IF_FALSE with POP_JUMP_IF_TRUE
- Optimize jumps
  - Replace unconditional jumps to RETURN_VALUE with RETURN_VALUE
  - Replace jumps to unconditional jumps with jumps to the final target
  - Remove unconditional jumps to the following block

For tuples, constant folding is only run if the result has 20 items or less.

By design, only basic optimizations can be implemented. A peephole optimizer has a narrow view on the bytecode (a few instructions) and only a very limited knownledge of the code.

Note: `3 < 5` or `(1, 2, 3)[1]` are not optimized.

1.5 Comparison with byteplay and codetransformer

1.5.1 History of the bytecode API design

The design of the bytecode module started with a single use case: reimplement the CPython peephole optimizer (implemented in C) in pure Python. The design of the API required many iterations to get the current API.

bytecode now has a clear separation between concrete instructions using integer arguments and abstract instructions which use Python objects for arguments. Jump targets are labels or basic blocks. And the control flow graph abstraction is now an API well separated from the regular abstract bytecode which is a simple list of instructions.

1.5.2 byteplay and codetransformer

The byteplay and codetransformer are clear inspiration for the design of the bytecode API. Sadly, byteplay and codetransformer API have design issues (at least for my specific use cases).

Free and cell variables

Converting a code object to bytecode and then back to code must no modify the code object. It is an important requirement.

The LOAD_DEREF instruction supports free variables and cell variables. byteplay and codetransformer use a simple string for the variable name. When the bytecode is converted to a code object, they check if the variable is a free variable, or fallback to a cell variable.

The CPython code base contains a corner case: code having a free variable and a cell variable with the same name. The heuristic produces invalid code which can lead to a crash.

bytecode uses *`FreeVar`* and *`CellVar`* classes to tag the type of the variable. Trying to use a simple string raise a `TypeError` in the *`Instr`* constructor.

Note: It's possible to fix this issue in byteplay and codetransformer, maybe even with keeping support for simple string for free/cell variables for backward compatibility.

Line numbers

codetransformer uses internally a dictionary mapping offsets to line numbers. It is updated when the `.steal()` method is used.

byteplay uses a pseudo-instruction `SetLineno` to set the current line number of the following instructions. It requires to handle these pseudo-instructions when you modify the bytecode, especially when instructions are moved.

In FAT Python, some optimizations move instructions but their line numbers must be kept. That's also why Python 3.6 was modified to support negative line number delta in `code.co_lntotab`.

bytecode has a different design: line numbers are stored directly inside instructions (*`Instr.lineno`* attribute). Moving an instruction keeps the line number information by design.

bytecode also supports the pseudo-instruction *`SetLineno`*. It was added to simplify functions emitting bytecode. It's not used when an existing code object is converted to bytecode.

Jump targets

In codetransformer, a jump target is an instruction. Jump targets are computed when the bytecode is converted to a code object.

byteplay and bytecode use labels. Jump targets are computed when the abstract bytecode is converted to a code object.

Note: A loop is need in the conversion from bytecode to code: if the jump target is larger than 2**16, the size of the jump instruction changes (from 3 to 6 bytes). So other jump targets must be recomputed.

bytecode handles this corner case. byteplay and codetransformer don't, but it should be easy to fix them.

Control flow graph

The peephole optimizer has strong requirements on the control flow: an optimization must not modify two instructions which are part of two different basic blocks. Otherwise, the optimizer produces invalid code.

bytecode provides a control flow graph API for this use case.

byteplay and codetransformer don't.

Functions or methods

This point is a matter of taste.

In bytecode, instructions are objects with methods like *`is_final()`*, `has_cond_jump()`, etc.

The byteplay project uses functions taking an instruction as parameter.

1.6 ChangeLog

1.6.1 Version 0.6

- Add stack depth computation based on control flow graph analysis

1.6.2 Version 0.5

- Add the new bytecode format of Python 3.6.
- Remove the `BaseInstr` class which became useless. It was replaced with the *`Instr`* class.
- Documentation: Add a comparison with byteplay and codetransformer.
- Remove the BaseIntr class: Instr becomes the new base class.

- Fix PEP 8 issues and check PEP 8 on Travis CI.

1.6.3 2016-04-12: Version 0.4

Peephole optimizer:

- Reenable optimization on `JUMP_IF_TRUE_OR_POP` jumping to `POP_JUMP_IF_FALSE <target>`.

1.6.4 2016-03-02: Version 0.3

New features:

- Add *`ControlFlowGraph.get_block_index()`* method

API changes:

- Rename `Block` class to *`BasicBlock`*
- Rename `BytecodeBlocks` class to *`ControlFlowGraph`*
- Rename `BaseInstr.op` to `BaseInstr.opcode`
- Rename `BaseBytecode.kw_only_argcount` attribute to *`BaseBytecode.kwonlyargcount`*, name closer to the Python code object attribute (`co_kwonlyargcount`)
- *`Instr`* constructor and its `set()` method now validates the argument type
- Add *`Compare`* enum, used for `COMPARE_OP` argument of *`Instr`*
- Remove *lineno* parameter from the `BaseInstr.set()` method
- Add *`CellVar`* and *`FreeVar`* classes: instructions having a cell or free variable now require a *`CellVar`* or *`FreeVar`* instance rather than a simple string (`str`). This change is required to handle correctly code with duplicated varible names in cell and free variables.
- *`ControlFlowGraph`*: remove undocumented `to_concrete_bytecode()` and `to_code()` methods

Bugfixes:

- Fix support of *`SetLineno`*

Peephole optimizer:

- Better code for LOAD_CONST x n + BUILD_LIST + UNPACK_SEQUENCE: rewrite LOAD_CONST in the reverse order instead of using ROT_TWO and ROT_THREE. This optimization supports more than 3 items.
- Remove JUMP_ABSOLUTE pointing to the following code. It can occur after dead code was removed.
- Remove NOP instructions
- Bugfix: catch IndexError when trying to get the next instruction.

1.6.5 2016-02-29: Version 0.2

- Again, the API is deeply reworked.
- The project has now a documentation: bytecode documentation
- Fix bug #1: support jumps larger than 2^16.
- Add a new *bytecode.peephole_opt module*: a peephole optimizer, code based on peephole optimizer of CPython 3.6 which is implemented in C

- Add *dump_bytecode()* function to ease debug.
- *Instr*:
  - Add *Instr.is_final()* method
  - Add *Instr.copy()* and `ConcreteInstr.copy()` methods
  - *Instr* now uses variable name instead of integer for cell and free variables.
  - Rename `Instr.is_jump` to *Instr.has_jump()*
- *ConcreteInstr* is now mutable
- Redesign the `BytecodeBlocks` class:
  - `Block` have no more label attribute: jump targets are now directly blocks
  - Rename `BytecodeBlocks.add_label()` method to `BytecodeBlocks.split_block()`
  - Labels are not more allowed in blocks
  - `BytecodeBlocks.from_bytecode()` now splits blocks after final instructions (*Instr.is_final()*) and after conditional jumps (*Instr.is_cond_jump()*). It helps the peephole optimizer to respect the control flow and to remove dead code.
- Rework API to convert bytecode classes:
  - BytecodeBlocks: Remove `to_concrete_bytecode()` and `to_code()` methods. Now you first have to convert blocks to bytecode using `to_bytecode()`.
  - Remove `Bytecode.to_bytecode_blocks()` method, replaced with `BytecodeBlocks.from_bytecode()`
  - Remove `ConcreteBytecode.to_concrete_bytecode()` and `Bytecode.to_bytecode()` methods which did nothing (return `self`)
- Fix *ConcreteBytecode* for code with no constant (empty list of constants)
- Fix argnames in *ConcreteBytecode.to_bytecode()*: use CO_VARARGS and CO_VARKEYWORDS flags to count the number of arguments
- Fix const_key() to compare correctly constants equal but of different types and special cases like `-0.0` and `+0.0`

1.6.6 2016-02-26: Version 0.1

- Rewrite completely the API!

1.6.7 2016-02-23: Release 0.0

- First public release

1.7 TODO list

- BaseBytecode.flags: use a class for flags?
- Remove Bytecode.cellvars and Bytecode.freevars?
- Remove Bytecode.first_lineno? Compute it on conversions.

- Add instruction constants/enums? Example:

```
from bytecode import instructions as i

bytecode = Bytecode([i.LOAD_NAME('print'),
                     i.LOAD_CONST('Hello World!'),
                     i.CALL_FUNCTION(1),
                     i.POP_TOP(),
                     i.LOAD_CONST(None),
                     i.RETURN_VALUE()])
```

Should we support instructions without parenthesis for instruction with no parameter? Example with POP_TOP and RETURN_VALUE:

```
from bytecode import instructions as i

bytecode = Bytecode([i.LOAD_NAME('print'),
                     i.LOAD_CONST('Hello World!'),
                     i.CALL_FUNCTION(1),
                     i.POP_TOP,
                     i.LOAD_CONST(None),
                     i.RETURN_VALUE])
```

- Nicer API for function arguments in bytecode object? Bytecode has argcount, kwonlyargcount and argnames. 4 types of parameters: indexed, `*args`, `**kwargs` and `*, kwonly=3`. See inspect.signature()

CHAPTER 2

See also

- codetransformer
- byteplay
- byteasm: an “assembler” for Python 3 bytecodes.
- BytecodeAssembler
- PEP 511 – API for code transformers

Symbols

A

B

C

D

F

G

H

I

K

L

N

O

P

S

T

U

V