All-Singing All-Dancing Python Bytecode

Larry Hastings
larry@hastings.org

EuroPython 2013
July 2, 2013

Introduction

Intermediate

CPython

- 3.3.0
- **100%**
- roughly applicable elsewhere

What Is Bytecode?

Opcodes for VM

- Stack manipulation
- Flow control
- Arithmetic
- Pythonic

When Is Bytecode Used?

At all times.

Python → bytecode

bytecode  Python

Why Have Bytecode?

Manage complexity

Why Study Bytecode?

Core developer

otherwise … no good reason!

- "Understand what's *really* going on"

  Python → bytecode → C → assembler → microcode …
- Hand-tuned bytecode
- Granularity for GIL & threading

gunk

```
def gunk(a=1, *args, b=3):
  print(args)
  c = None
  return (a + b, c)
```

dis

```
>>> dis.dis(gunk)
```

```
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_FAST                2 (args)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_CONST               0 (None)
             13 STORE_FAST               3 (c)

  4          16 LOAD_FAST                0 (a)
             19 LOAD_FAST                1 (b)
             22 BINARY_ADD
             23 LOAD_FAST                3 (c)
             26 BUILD_TUPLE              2
             29 RETURN_VALUE
```

The Whole Picture

The opcodes

Runtime environment

Data and metadata

Opcodes and HAVE_ARGUMENT

101 opcodes

```
bytecode = list(bytes)


op = bytecode[i]
oparg = bytecode[i+1] | (bytecode[i+2] << 8)


dis.HAVE_ARGUMENT = 90
size = 1 if op < dis.HAVE_ARGUMENT else 3
```

The VM

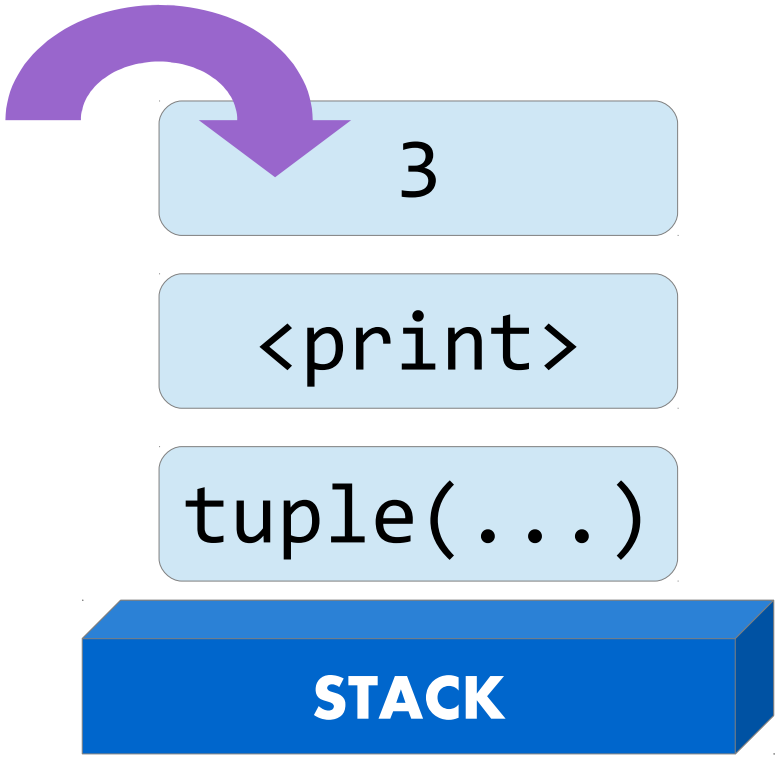
ip (JUMP_)

stack (LOAD_, STORE_, ...)

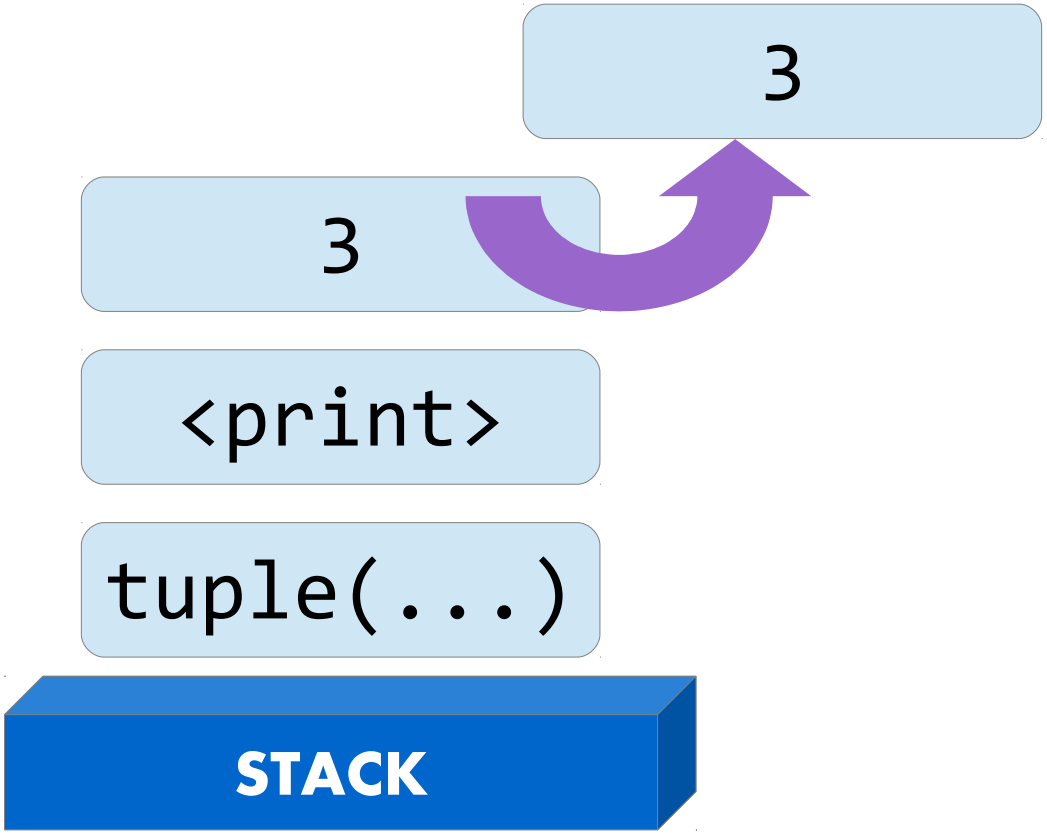
“fast locals” (LOAD_FAST, STORE_FAST)

Stack Machine Part 1

LOAD_x → stack

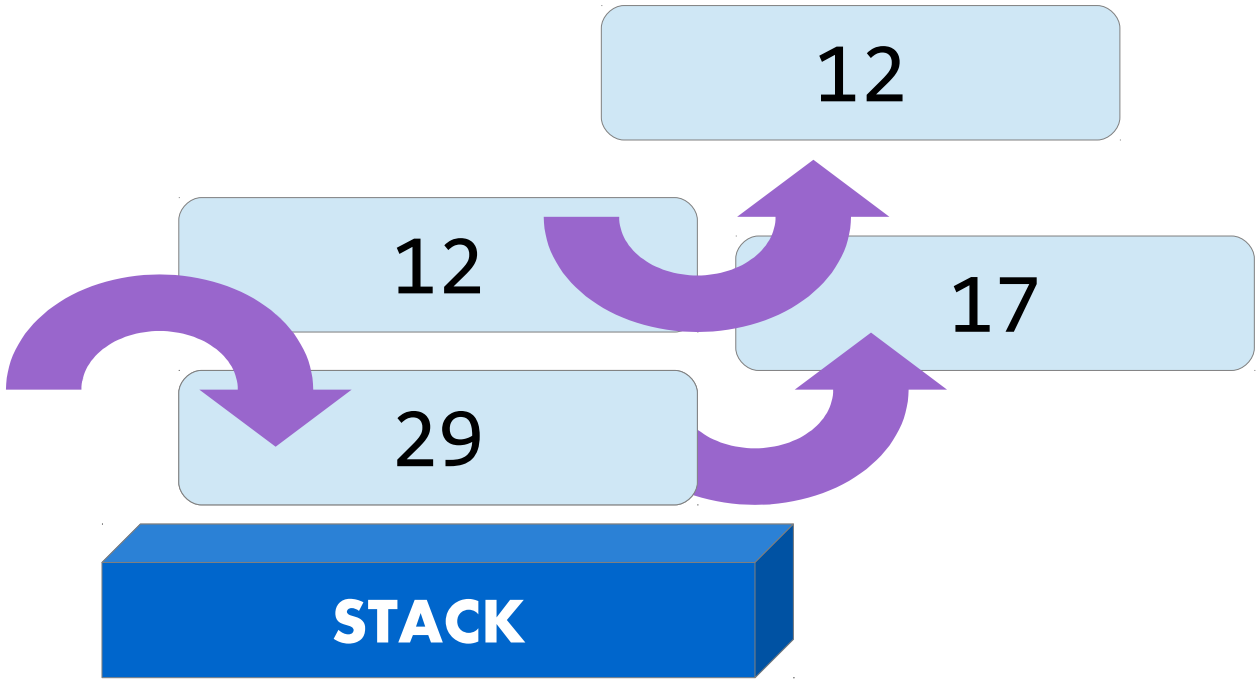


STORE_x ← stack



Stack Machine Part 2

BINARY_ADD



Bytecode Variable Types

Globals (+ builtins)	LOAD_GLOBAL
“Fast locals”	LOAD_FAST
“Locals” (“Slow locals”)	LOAD_NAME
Consts	LOAD_CONST
Object attributes	LOAD_ATTR
Cell	LOAD_DEREF

Free And Cell Variables

```
def foo():
    a = 1                 # local variable
    b = 2                 # cell variable
    def bar():
        nonlocal b        # free variable
        print(b)
```

Data And Metadata, Part 1

```
>>> type(gunk)          # types.FunctionType
<class 'function'>
>>> dir(gunk)
['__annotations__', '__call__', '__class__',
'__closure__', '__code__', '__defaults__',
'__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__',
'__globals__', '__gt__', '__hash__', '__init__',
'__kwdefaults__', '__le__', '__lt__', '__module__',
'__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
```

Data And Metadata, Part 2

```
>>> type(gunk.__code__)     # types.CodeType
<class 'code'>
>>> dir(gunk.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'co_argcount', 'co_cellvars', 'co_code',
'co_consts', 'co_filename', 'co_firstlineno',
'co_flags', 'co_freevars', 'co_kwonlyargcount',
'co_lnotab', 'co_name', 'co_names',
'co_nlocals', 'co_stacksize', 'co_varnames']
```

Why Have Both?

Function → Code 

Code → `marshal` Function → `marshal`

`__closure__` `__defaults__` `__globals__`

`__kwdefaults__`

Nested functions

__code__.co_code

```
>>> gunk.__code__.co_code
b't\x00\x00|\x02\x00\x83\x01\x00\x01d\x00
\x00}\x03\x00|\x00\x00|\x01\x00\x17|\x03\
x00f\x02\x00S'
>>> [x for x in gunk.__code__.co_code]
[116, 0, 0, 124, 2, 0, 131, 1, 0,
1, 100, 0, 0, 125, 3, 0, 124, 0, 0,
124, 1, 0, 23, 124, 3, 0, 102, 2, 0, 83]
```

The Simplest Useful Disassembler

```
import dis


def disassemble(callable):
    print("def", callable.__name__ + ":")
    program = callable.__code__.co_code
    i = 0
    while i < len(program):
        op = program[i]
        if op < dis.HAVE_ARGUMENT:
            oparg = ''
            i += 1
        else:
            oparg = program[i + 1] | (program[i + 2] << 8)
            i += 3
        print("   ", dis.opname[op], oparg)
```

The Simplest Useful Disassembler

```
def disassemble:
    LOAD_FAST 0              LOAD_ATTR 3
    LOAD_ATTR 0              LOAD_CONST 3
    LOAD_ATTR 1              BINARY_ADD
    STORE_FAST 1             CALL_FUNCTION 2
    LOAD_CONST 1             POP_TOP
    STORE_FAST 2             SETUP_LOOP 129
    LOAD_GLOBAL 2            LOAD_FAST 2
    LOAD_CONST 2             LOAD_GLOBAL 4
    LOAD_FAST 0              ...
```

__code__ Argument Fields

```
>>> gunk.__code__.co_argcount
1
>>> gunk.__code__.co_kwonlyargcount
1
>>> gunk.__code__.co_nlocals
4
>>> gunk.__code__.co_varnames
('a', 'b', 'args', 'c')
```

Function Defaults

```
>>> gunk.__defaults__
(1,)
>>> gunk.__kwdefaults__
{'b': 3}
```

Globals And Const Tables

```
>>> gunk.__code__.co_names
('print', 'None')
>>> gunk.__code__.co_consts
(None,)
```

Line Numbers

```
>>> gunk.__code__.co_firstlineno
1
>>> gunk.__code__.co_lnotab
b'\x00\x01\n\x01\x06\x01'
>>> [x for x in gunk.__code__.co_lnotab]
[0, 1, 10, 1, 6, 1]
```

Metadata

```
>>> gunk.__globals__
{'__doc__': None, '__name__': '__main__', 'dis': <module
'dis' from '/home/larry/lib/python3.3/dis.py'>, ... }
>>> gunk.__module__
'__main__'
>>> gunk.__code__.co_filename
'<stdin>'
>>> gunk.__code__.co_name
'gunk'
>>> gunk.__code__.co_flags
71
>>> gunk.__code__.co_stacksize
2
```

Advanced Topics

```
>>> gunk.__annotations__
{}
>>> repr(gunk.__closure__)
'None'
>>> gunk.__code__.co_cellvars
()
>>> gunk.__code__.co_freevars
()
```

Modules Are Callables

```
def module():
    …
    LOAD_CONST     None
    RETURN_VALUE
```

Classes Are Callables, Part 1

```
def classname(__locals__):
    LOAD_FAST        0
    STORE_LOCALS                    # __prepare__
    LOAD_NAME        __name__
    STORE_NAME       __module__
    LOAD_CONST       None
    STORE_NAME       __qualname__

    …

    LOAD_CONST       None
    RETURN_VALUE
```

Classes Are Callables, Part 2

```
LOAD_BUILD_CLASS
LOAD_CONST          <code object 'classname'>
LOAD_CONST          'classname'
MAKE_FUNCTION       0
LOAD_CONST          'classname'
CALL_FUNCTION       2
```

Creating A Function By Hand

```
import types

code_object = types.CodeType(2, 0, 2, 2, 67,
    Bytes([124, 0, 0, 124, 1, 0, 23, 83]), (), (),
    ('a', 'b'), '<stdin>', 'add', 1, b'', (), ())
add = types.FunctionType(code_object, globals())

print(add(2, 3))
```

Readable & Hand-Coded, Part 1

```
import inspect
import dis
import types


op = dis.opmap.get
program = bytes([
    op('LOAD_FAST'), 0, 0,
    op('LOAD_FAST'), 1, 0,
    op('BINARY_ADD'),
    op('RETURN_VALUE'),
    ])
```

Readable & Hand-Coded, Part 2

```
argcount = 2
kwonlyargcount = localcount = 0
nlocals = argcount + kwonlyargcount + localcount
max_stack_depth = 2
flags = inspect.CO_OPTIMIZED | inspect.CO_NEWLOCALS |
  inspect.CO_NOFREE
constants = names = freevars = cellvars = ()
Varnames = ('a', 'b')
filename = '<stdin>'
name = 'add'
firstlineno = 1
lnotab = b''
```

Readable & Hand-Coded, Part 3

```
code_object = types.CodeType(
    argcount, kwonlyargcount, nlocals,
    max_stack_depth, flags, program,
    constants, names, varnames,
    filename, function_name,
    firstlineno, lnotab,
    freevars, cellvars
    )
add = types.FunctionType(code_object, globals())


print(add(2, 3))
```

Maynard

Maynard vs. gunk

```
def gunk:
    arg a 1

    kwonly b 3

    args args

    global print
    global None
```

```
const const_None None
local c


load_global print
load_fast args
call_function 1
pop_top
load_const const_None
store_fast c
…
```

Class Disassembly With Maynard

```
def foo():
  class H:
    a = 3


maynard.disassemble(foo)
def foo:
    const const_None None
    const const_index1 <code object H at ...>


maynard.disassemble(foo.__code__.co_consts[1])
```

Perth

Toy FORTH on Python VM

integer, float, string literals

```
: ; { + - if then else . cr
```

recursion

```
: fib { n } n 1 <= if 1 else
  n 1 - fib n 2 - fib + then ;
```

Bring It All Together

A Python VM

... in Python

fib

```
def fib(n):
    if n <= 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

The Simplest Possible VM, Part 1

```
def vm(fn, *args):
    code = fn.__code__
    constants = code.co_consts
    names = code.co_names
    program = code.co_code
    nlocals = code.co_nlocals

    globals_dict = fn.__globals__
    builtins_dict = globals_dict['__builtins__']

    ip = 0
    locals = list(args) + [uninitialized] * (nlocals - len(args))
    stack = []
```

The Simplest Possible VM, Part 2

```
while True:
  op = program[ip]
  ip += 1
```

```
if op >= dis.HAVE_ARGUMENT:
  low  = program[ip]
  high = program[ip + 1]
  oparg = (high << 8) | low
  ip += 2
```

```
if op == op_load_const:
  stack.append(
    constants[oparg])
```

```
elif op == op_load_fast:
  stack.append(locals[oparg])
```

```
elif op == op_load_global:
  name = names[oparg]
  if name in globals_dict:
    stack.append(
      globals_dict[name])
  else:
    stack.append(
      builtins_dict[name])
```

The Simplest Possible VM, Part 3

```
elif op == op_binary_add:
  w = stack.pop()
  v = stack.pop()
  stack.append(v + w)
```

```
elif op == op_binary_subtract:
  w = stack.pop()
  v = stack.pop()
  stack.append(v - w)
```

```
elif op == op_pop_jump_if_false:
  if not stack.pop():
    ip = oparg
```

The Simplest Possible VM, Part 4

```
elif op == op_compare_op:
    w = stack.pop()
    v = stack.pop()
    if oparg == Py_LT:
        value = v < w
    elif oparg == Py_LE:
        value = v <= w
    else:
        sys.exit('unhandled compare_op oparg', oparg)
    stack.append(value)
```

The Simplest Possible VM, Part 5

```
elif op == op_call_function:
    assert oparg < 255, \
      "can't handle keyword arguments"
    args = [stack.pop() for i in range(oparg)]
    callable = stack.pop()
    value = vm(callable, *args)
    stack.append(value)
elif op == op_return_value:
    assert len(stack) == 1
    return stack[0]
```

It Works!

```
>>> for n in range(10):
...  print("fib(", n, ") =", fib(n), " = ", vm(fib, n))


fib( 0 ) = 1  =  1
fib( 1 ) = 1  =  1
fib( 2 ) = 2  =  2
fib( 3 ) = 3  =  3
fib( 4 ) = 5  =  5
fib( 5 ) = 8  =  8
fib( 6 ) = 13  =  13
fib( 7 ) = 21  =  21
fib( 8 ) = 34  =  34
fib( 9 ) = 55  =  55
```

If You Experiment With It Yourself

```
zsh: segmentation fault (core dumped)
% _
```

Resources

```
import dis, inspect, __future__
```

Maynard

https://bitbucket.org/larry/maynard/

https://pypi.python.org/pypi/maynard/

`Python/ceval.c`

ByteRun

https://github.com/nedbat/byterun/

The End

Larry Hastings
larry@hastings.org

radiofreepython.com