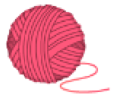
Python's Bytecode



Python Mauritius UserGroup (pymug)

More info: mscc.mu/python-mauritius-usergroup-pymug/

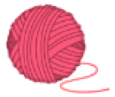
Why	Where
codes	github.com/pymug
share events	twitter.com/pymugdotcom
ping professionals	linkedin.com/company/pymug
all info	pymug.com
tell friends by like	facebook.com/pymug

Abdur-Rahmaan Janhangeer

Python

Help people get into OpenSource

Python's Bytecode



Overview

Traditionally

```
-------     ---------     ---------------
| src | --> | parse | --> | interpreter |
-------     ---------     ---------------
```

Now

```
-------
| src |
-------
   |
   v
------------
| compiler |
------------
   |
   V
--------------------
| virtual machine |
--------------------
```

A Virtual Machine is just a program

Compilation [1]

```
[ parse tree]
   ⬇
[ ast ]
   ⬇
[ bytecode generation ]
   ⬇
[ bytecode optimisation ]
   ⬇
[ flow control graph ]
   ⬇
[ code object generation ]
```

Hands-on Bytecode

Same

```
$ python3.10 main.py
```

```
$ python3.10 __pycache__/main.cpython-310.pyc
```

> 💡 `-m compileall` is for creating cached bytecode files when installing libraries

.pyc -> rb, code obj -> marshall.load(f)

dis.dis(code obj)

```python
import marshal
import sys
import dis

header_size = 8
if sys.version_info >= (3, 6):
    header_size = 12
if sys.version_info >= (3, 7):
    header_size = 16
with open("__pycache__/main.cpython-310.pyc", "rb") as f:
    metadata = f.read(header_size)
    code_obj = marshal.load(f)
    dis.dis(code_obj)
```

```
  1           0 LOAD_CONST               0 (1)
              2 STORE_NAME               0 (x)

  2           4 LOAD_CONST               1 (2)
...
```

```
>>> help(compile)
Help on built-in function compile in module builtins:

compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1, *, _feature_version=-1)
    Compile source into a code object that can be executed by exec() or eval().

    The source code may represent a Python module, statement or expression.
    The filename will be used for run-time error messages.
    The mode must be 'exec' to compile a module, 'single' to compile a
    single (interactive) statement, or 'eval' to compile an expression.
    The flags argument, if present, controls which future statements influence
    the compilation of the code.
    The dont_inherit argument, if true, stops the compilation inheriting
    the effects of any future statements in effect in the code calling
    compile; if absent or false these statements do influence the compilation,
    in addition to any features explicitly specified.
```

```python
src = '''
x = 1
y = 2

print(x+y)
'''

c = compile(src, '', "exec")
exec(c)
# exec(src)
```

```
>>> help(C)
Help on code object:

class code(object)
 |  code(argcount, posonlyargcount, kwonlyargcount,
 nlocals, stacksize, flags, codestring, constants,
 names, varnames, filename, name, firstlineno,
 linetable, freevars=(), cellvars=(), /)
 |
 |  Create a code object.  Not for the faint of heart.
...
```

Bytecode instructions ready to be executed

```
>>> help(exec)
Help on built-in function exec in module builtins:

exec(source, globals=None, locals=None, /)
    Execute the given source in the context of globals
    and locals.

    The source may be a string representing one or more
    Python statements
    or a code object as returned by compile().
    The globals must be a dictionary and locals can be any
    mapping,
    defaulting to the current globals and locals.
    If only globals is given, locals defaults to it.
```

```
>>> c.co_code
b'd\x00Z\x00d\x01Z\x01e\x02e\x00e
\x01\x17\x00\x83\x01\x01\x00d\x02S\x00'
>>> type(c.co_code)
<class 'bytes'>
```

```
>>> [c for c in c.co_code]
[
100, 0,
90, 0,
100, 1,
90, 1,
101, 2,
101, 0,
101, 1,
23, 0,
131, 1,
1, 0,
100, 2,
83, 0
]
```

```
LOAD_CONST 2
```

```
LOAD_CONST 2 op arg

  opcode
```

`if > dis.HAVE_ARGUMENT, has args`

```
>>> import dis
>>> [(dis.opname[c] if i%2==0 else c)
        for i, c in enumerate(c.co_code)]
[
    'LOAD_CONST', 0,
    'STORE_NAME', 0,
    'LOAD_CONST', 1,
    'STORE_NAME', 1,
    'LOAD_NAME', 2,
    'LOAD_NAME', 0,
    'LOAD_NAME', 1,
    'BINARY_ADD', 0,
    'CALL_FUNCTION', 1,
    'POP_TOP', 0,
    'LOAD_CONST', 2,
    'RETURN_VALUE', 0
]
```

```
>>> def func():
...     x = 1
...     y = 1
...     print(x+y)
...
>>> dis.dis(func)
  2           0 LOAD_CONST               1 (1)
              2 STORE_FAST               0 (x)

  3           4 LOAD_CONST               1 (1)
              6 STORE_FAST               1 (y)

  4           8 LOAD_GLOBAL              0 (print)
             10 LOAD_FAST                0 (x)
             12 LOAD_FAST                1 (y)
             14 BINARY_ADD
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 LOAD_CONST               0 (None)
             22 RETURN_VALUE
```

2 3 4 line nums

0 2 4 6 opcode index, used for jumps

```
>>> func.__code__.co_names
('print',)
>>> func.__code__.co_varnames
('x', 'y')
>>> func.__code__.co_consts
(None, 1)
```

free variables: used in a code block but not defined there, not applied to global vars

```
inspect.stack() -> [
    FrameInfo(frame, filename, lineno,
    function, code_context, index), ...]
```

values and results live on the stack

`BINARY_ADD` pops two values from the stack

operates on them

places back

cpython/Include/opcode.h

some 191

Frames: contextual info about stack and interpreter states. Attached to a thread.

Stack of frames possible.

Each module, func and class has a frame [2]

Generators switch frames, need a data stack for each frame

Running

cpython/Programs/python.c has main (or wmain)

calls `Py_BytesMain` or `Py_Main` from `modules/main.c`, both calling same thing with different args

```
switch (opcode) {
    // ...
case TARGET(BINARY_ADD): {
            PyObject *right = POP();
            PyObject *left = TOP();
            PyObject *sum;
            /* NOTE(haypo): Please don't try to micro-optimize int+int on
               CPython using bytecode, it is simply worthless.
               See http://bugs.python.org/issue21955 and
               http://bugs.python.org/issue10044 for the discussion. In short,
               no patch shown any impact on a realistic benchmark, only a minor
               speedup on microbenchmarks. */
            if (PyUnicode_CheckExact(left) &&
                     PyUnicode_CheckExact(right)) {
                sum = unicode_concatenate(tstate, left, right, f, next_instr);
                /* unicode_concatenate consumed the ref to left */
            }
            else {
                sum = PyNumber_Add(left, right);
                Py_DECREF(left);
            }
            Py_DECREF(right);
            SET_TOP(sum);
            if (sum == NULL)
                goto error;
            DISPATCH();
        }
```

Bytecodes not same for all versions

VM not a platform

Working of common opcodes

```
BINARY_ADD

[1, 2]
  ⬇
[]
  ⬇
[3]
```

```
LOAD_CONST


[]
  ⬇
[5]
```

```
STORE_FAST

[5]
  ⬇
[]
```

```
x = 1

    1           0 LOAD_CONST               1 (1)
                2 STORE_FAST               0 (x)
```

```
if x < 2:
    return True

  2           0 LOAD_CONST               1 (1)
              2 LOAD_CONST               2 (2)
              4 COMPARE_OP               0 (<)
              6 POP_JUMP_IF_FALSE        6 (to 12)

  3           8 LOAD_CONST               3 (True)
             10 RETURN_VALUE

  2     >>   12 LOAD_CONST               0 (None)
             14 RETURN_VALUE
```

```
x = 10
while x < 20:
        x += 2
```

```
  2           0 LOAD_CONST               1 (10)
              2 STORE_FAST               0 (x)

  3           4 LOAD_FAST                0 (x)
              6 LOAD_CONST               2 (20)
              8 COMPARE_OP               0 (<)
             10 POP_JUMP_IF_FALSE       16 (to 32)

  4     >>   12 LOAD_FAST                0 (x)
             14 LOAD_CONST               3 (2)
             16 INPLACE_ADD
             18 STORE_FAST               0 (x)

  3          20 LOAD_FAST                0 (x)
             22 LOAD_CONST               2 (20)
             24 COMPARE_OP               0 (<)
             26 POP_JUMP_IF_TRUE         6 (to 12)
             28 LOAD_CONST               0 (None)
             30 RETURN_VALUE
        >>   32 LOAD_CONST               0 (None)
             34 RETURN_VALUE
```

Refs

- [1] Inside The Python VM, Obi Ike-Nwosu
- [2] A Python Interpreter Written in Python, Allison Kaptur, Ned Batchelder
- [3] Understanding Python Bytecode, Reza Bagheri https://www.linkedin.com/in/reza-bagheri-71882a76/