Bytecode

and .pyc files

Konrad Gawda

#118
2025.02

Konrad Gawda

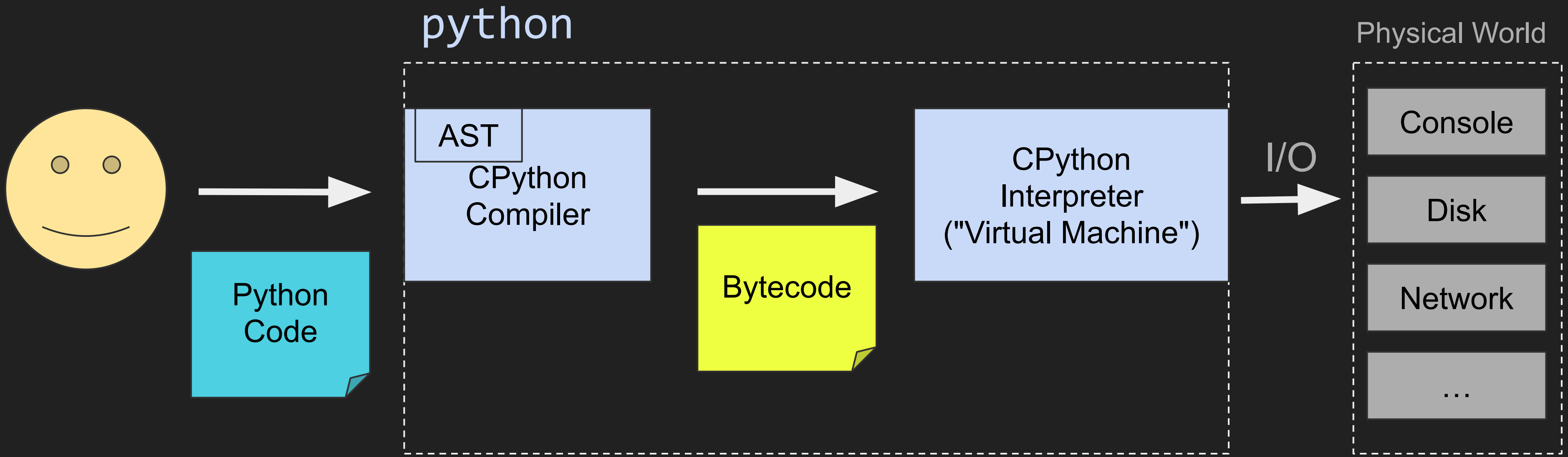
Python - programmer and trainer

Cloud Evangelist - [Integrated Computing]

Videocast host - [Porozmawiajmy o Chmurze]

```
1212           2 LOAD_FAST                0 (self)
               4 LOAD_ATTR                1 (NULL|self + is_absolute)
              24 CALL                     0
              32 POP_JUMP_IF_FALSE        2 (to 38)

1213          34 LOAD_FAST                0 (self)
              36 RETURN_VALUE

1214     >>   38 LOAD_FAST                0 (self)
              40 LOAD_ATTR                2 (drive)
              60 POP_JUMP_IF_FALSE       38 (to 138)

1216          62 LOAD_FAST                0 (self)
              64 LOAD_ATTR                4 (_flavour)
              84 LOAD_ATTR                7 (NULL|self + abspath)
             104 LOAD_FAST                0 (self)
             106 LOAD_ATTR                2 (drive)
             126 CALL                     1
             134 STORE_FAST               1 (cwd)
             136 JUMP_FORWARD            70 (to 278)

1218     >>  138 LOAD_GLOBAL              9 (NULL + os)
             148 LOAD_ATTR               10 (getcwd)
             168 CALL                     0
             176 STORE_FAST               1 (cwd)

1224         178 LOAD_FAST                0 (self)
             180 LOAD_ATTR               12 (root)
             200 POP_JUMP_IF_TRUE        38 (to 278)
             202 LOAD_FAST                0 (self)
             204 LOAD_ATTR               14 (_tail)
             224 POP_JUMP_IF_TRUE        26 (to 278)

1225         226 LOAD_FAST                0 (self)
             228 LOAD_ATTR               17 (NULL|self + with_segments)
             248 LOAD_FAST                1 (cwd)
             250 CALL                     1
```

Bytecode

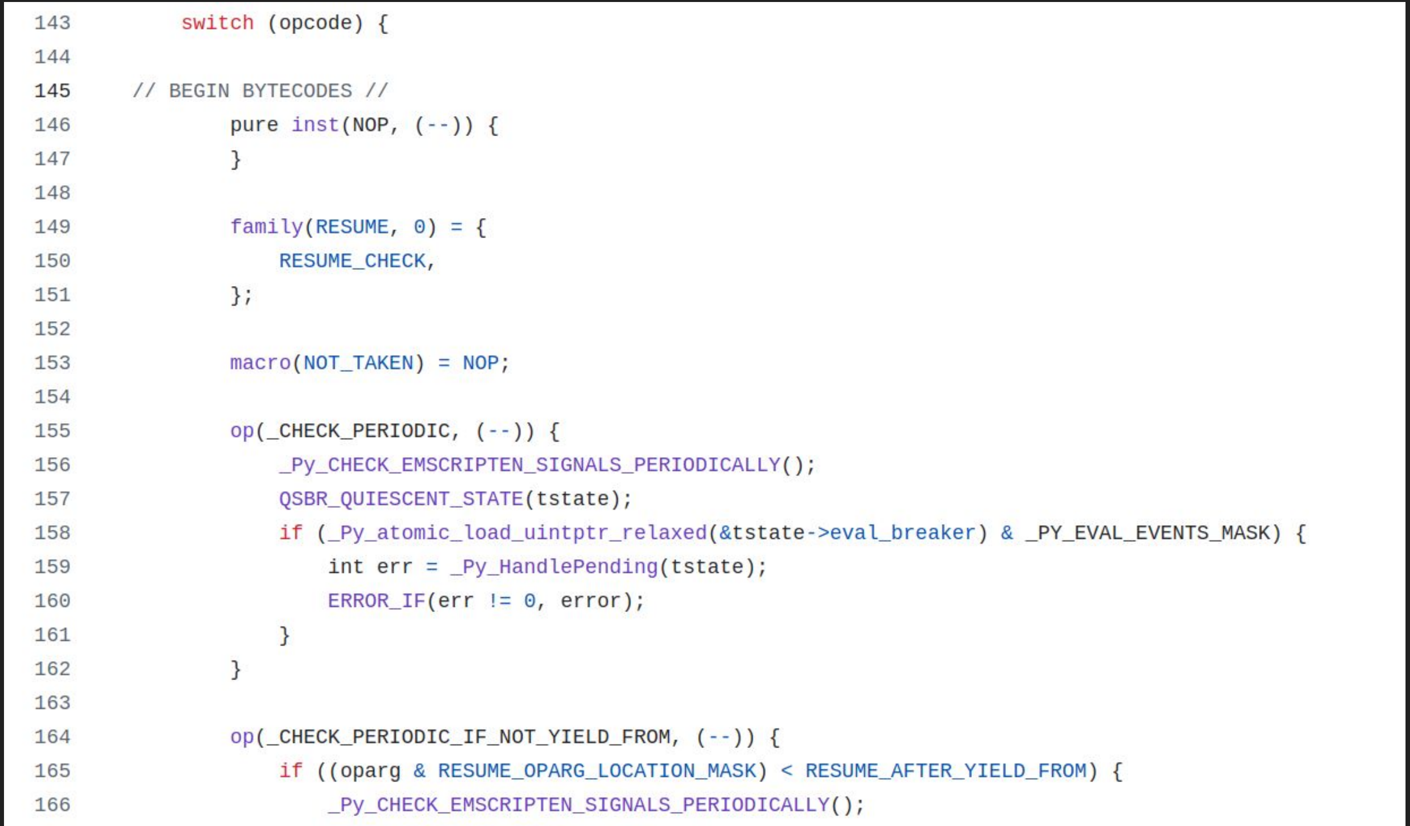

python
Physical World
AST
CPython Compiler
Python Code
Bytecode
CPython Interpreter ("Virtual Machine")
I/O
Console
Disk
Network
...

122 instructions (in Python 3.13)

```
NOP, POP_TOP, END_FOR, END_SEND, COPY, SWAP, CACHE, UNARY_NEGATIVE, UNARY_NOT, UNARY_INVERT,
GET_ITER, GET_YIELD_FROM_ITER, TO_BOOL, BINARY_OP, BINARY_SUBSCR, STORE_SUBSCR, DELETE_SUBSCR,
BINARY_SLICE, STORE_SLICE, GET_AWAITABLE, GET_AITER, GET_ANEXT, END_ASYNC_FOR, CLEANUP_THROW,
BEFORE_ASYNC_WITH, SET_ADD, LIST_APPEND, MAP_ADD, RETURN_VALUE, RETURN_CONST, YIELD_VALUE,
SETUP_ANNOTATIONS, POP_EXCEPT, RERAISE, PUSH_EXC_INFO, CHECK_EXC_MATCH, CHECK_EG_MATCH,
WITH_EXCEPT_START, LOAD_ASSERTION_ERROR, LOAD_BUILD_CLASS, BEFORE_WITH, GET_LEN, MATCH_MAPPING,
MATCH_SEQUENCE, MATCH_KEYS, STORE_NAME, DELETE_NAME, UNPACK_SEQUENCE, UNPACK_EX, STORE_ATTR,
DELETE_ATTR, STORE_GLOBAL, DELETE_GLOBAL, LOAD_CONST, LOAD_NAME, LOAD_LOCALS,
LOAD_FROM_DICT_OR_GLOBALS, BUILD_TUPLE, BUILD_LIST, BUILD_SET, BUILD_MAP, BUILD_CONST_KEY_MAP,
BUILD_STRING, LIST_EXTEND, SET_UPDATE, DICT_UPDATE, DICT_MERGE, LOAD_ATTR, LOAD_SUPER_ATTR,
COMPARE_OP, IS_OP, CONTAINS_OP, IMPORT_NAME, IMPORT_FROM, JUMP_FORWARD, JUMP_BACKWARD,
JUMP_BACKWARD_NO_INTERRUPT, POP_JUMP_IF_TRUE, POP_JUMP_IF_FALSE, POP_JUMP_IF_NOT_NONE,
POP_JUMP_IF_NONE, FOR_ITER, LOAD_GLOBAL, LOAD_FAST, LOAD_FAST_LOAD_FAST, LOAD_FAST_CHECK,
LOAD_FAST_AND_CLEAR, STORE_FAST, STORE_FAST_STORE_FAST, STORE_FAST_LOAD_FAST, DELETE_FAST,
MAKE_CELL, LOAD_DEREF, LOAD_FROM_DICT_OR_DEREF, STORE_DEREF, DELETE_DEREF, COPY_FREE_VARS,
RAISE_VARARGS, CALL, CALL_KW, CALL_FUNCTION_EX, PUSH_NULL, MAKE_FUNCTION, SET_FUNCTION_ATTRIBUTE,
BUILD_SLICE, EXTENDED_ARG, CONVERT_VALUE, FORMAT_SIMPLE, FORMAT_WITH_SPEC, MATCH_CLASS, RESUME,
RETURN_GENERATOR, SEND, HAVE_ARGUMENT, SETUP_FINALLY, SETUP_CLEANUP, SETUP_WITH, POP_BLOCK, JUMP,
JUMP_NO_INTERRUPT, LOAD_CLOSURE, LOAD_METHOD
```

Big CPython switch

https://github.com/python/cpython/blob/main/Python/bytecodes.c



```c
        switch (opcode) {

// BEGIN BYTECODES //
            pure inst(NOP, (--)) {
            }

            family(RESUME, 0) = {
                RESUME_CHECK,
            };

            macro(NOT_TAKEN) = NOP;

            op(_CHECK_PERIODIC, (--)) {
                _Py_CHECK_EMSCRIPTEN_SIGNALS_PERIODICALLY();
                QSBR_QUIESCENT_STATE(tstate);
                if (_Py_atomic_load_uintptr_relaxed(&tstate->eval_breaker) & _PY_EVAL_EVENTS_MASK) {
                    int err = _Py_HandlePending(tstate);
                    ERROR_IF(err != 0, error);
                }
            }

            op(_CHECK_PERIODIC_IF_NOT_YIELD_FROM, (--)) {
                if ((oparg & RESUME_OPARG_LOCATION_MASK) < RESUME_AFTER_YIELD_FROM) {
                    _Py_CHECK_EMSCRIPTEN_SIGNALS_PERIODICALLY();
```

```python
def add(a, b):
    return a + b
```

How it is exposed in Python

```
add.__code__.
```

```
_co_code_adaptive  # same as co_code
_varname_from_oparg <built-in method ...>
co_argcount 2
co_cellvars ()
co_code b'\x97\x00|\x00|\x01z\x00\x00\x00S\x00'
co_consts (None,)
co_exceptiontable b''
co_filename <stdin>
co_firstlineno 1
co_flags 3
co_freevars ()
co_kwonlyargcount 0
```

```
co_lines <built-in method ...>
co_linetable b'\x80\x00\xd8\t\n\x88Q\x89\x15\x80
co_lnotab b'\x02\x01' # DeprecationWarning
co_name add
co_names ()
co_nlocals 2
co_positions <built-in method ...>
co_posonlyargcount 0
co_qualname add
co_stacksize 2
co_varnames ('a', 'b')
replace <built-in method ...>
```

[151, 0, 124, 0, 124, 1, 122, 0, 0, 0, 83, 0]

dis — Disassembler for Python bytecode

```
import dis    # …not this
```

dis.show_code / dis.code_info

```
>>> dis.show_code(add)
Name:                   add
Filename:               <stdin>
Argument count:         2
Positional-only arguments: 0
Kw-only arguments:      0
Number of locals:       2
Stack size:             2
Flags:                  OPTIMIZED, NEWLOCALS
Constants:
   0: None
Variable names:
   0: a
   1: b
```

dis.dis

```
>>> dis.dis(add)
  1           0 RESUME                   0
  2           2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
```

dis.dis(add)

```
3.13, 3.14-rc
  1           RESUME                  0
  2           LOAD_FAST_LOAD_FAST     1 (a, b)
              BINARY_OP               0 (+)
              RETURN_VALUE

3.12, 3.11
  1           0 RESUME                  0
  2           2 LOAD_FAST               0 (a)
              4 LOAD_FAST               1 (b)
              6 BINARY_OP               0 (+)
             10 RETURN_VALUE

3.10, 3.9, 3.8, 3.7, 3.6, …
  2           0 LOAD_FAST               0 (a)
              2 LOAD_FAST               1 (b)
              4 BINARY_ADD
              6 RETURN_VALUE
```

dis.get_instructions

```
>>> for x in dis.get_instructions(add):
...  print(x)


Instruction(opname='RESUME', opcode=151, arg=0, argval=0, argrepr='', offset=0, start
Instruction(opname='LOAD_FAST', opcode=124, arg=0, argval='a', argrepr='a', offset=2,
Instruction(opname='LOAD_FAST', opcode=124, arg=1, argval='b', argrepr='b', offset=4,
Instruction(opname='BINARY_OP', opcode=122, arg=0, argval=0, argrepr='+', offset=6,
Instruction(opname='RETURN_VALUE', opcode=83, arg=None, argval=None, argrepr='', offs
```

```python
def f(x):
    y = 3.1415 * math.pow(x)
    return y
```

```
def f(x):
    y = 3.1415 * math.pow(x)
    return y
```

```
Python 3.12
[151, 0, 100, 1, 116, 1, 0, 0, 0, 0, 0, 0, 0, 0, 106, 2, 0, 0, 0, 0, 0,
0, 0, 0, 0, 124, 0, 171, 1, 0, 0, 0, 0, 0, 0, 122, 5, 0, 0, 125, 1, 124,
```

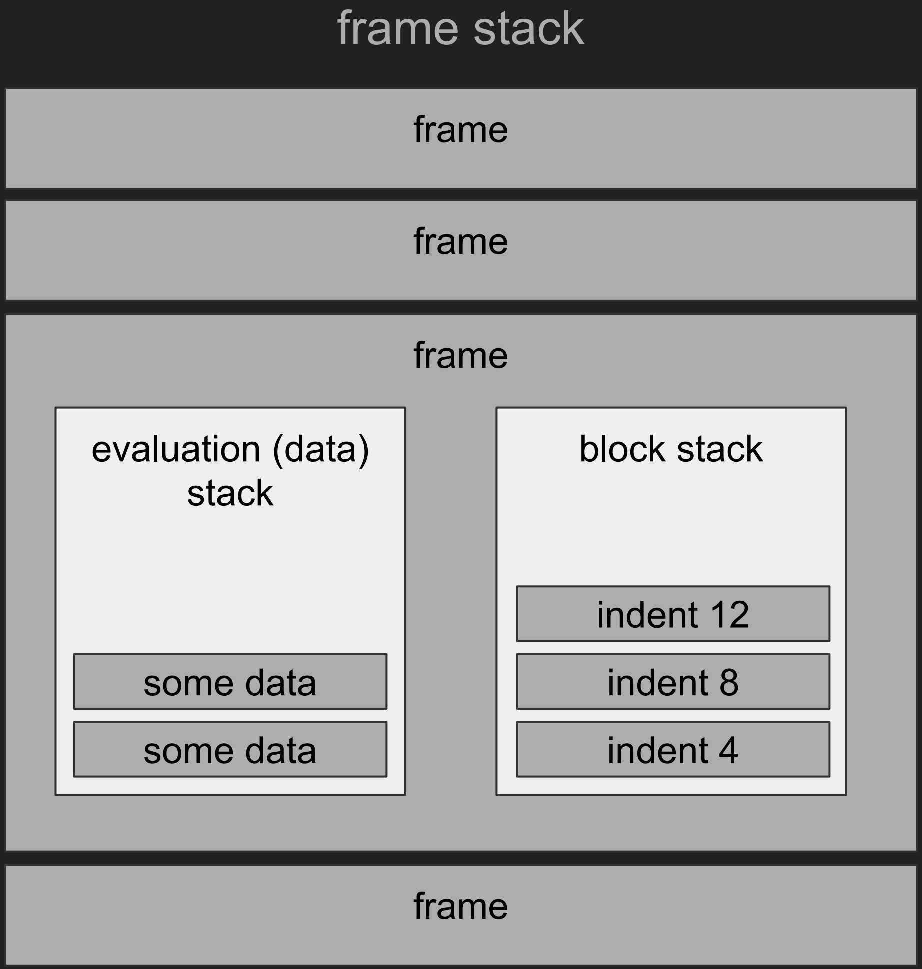
```
5       0 RESUME          0

6       2 LOAD_CONST      1 (3.1415)
        4 LOAD_GLOBAL     1 (NULL + math)
       14 LOAD_ATTR       2 (pow)
       34 LOAD_FAST       0 (x)
       36 CALL            1
       44 BINARY_OP       5 (*)
       48 STORE_FAST      1 (y)

7      50 LOAD_FAST       1 (y)
       52 RETURN_VALUE
```

```
_co_code_adaptive b'\x97\x00d\x01t\x01\x11\...
_varname_from_oparg <built-in method ...>
co_argcount 1
co_cellvars ()
co_code b'\x97\x00d\x01t\x01\x00\x00\x00...'
co_consts (None, 3.1415)
co_exceptiontable b''
co_filename /home/.../example.py
co_firstlineno 5
co_flags 3
co_freevars ()
co_kwonlyargcount 0
co_lines <built-in method ...
co_linetable b'\x80\x00\xd8\x08\x0e\x94...'
co_lnotab b'\x02\x010\x01'
co_name f
co_names ('math', 'pow')
co_nlocals 2
co_positions <built-in method ...>
co_posonlyargcount 0
co_qualname f
co_stacksize 4
co_varnames ('x', 'y')
```

LOAD_GLOBAL, LOAD_ATTR use `co_names[namei>>1]`

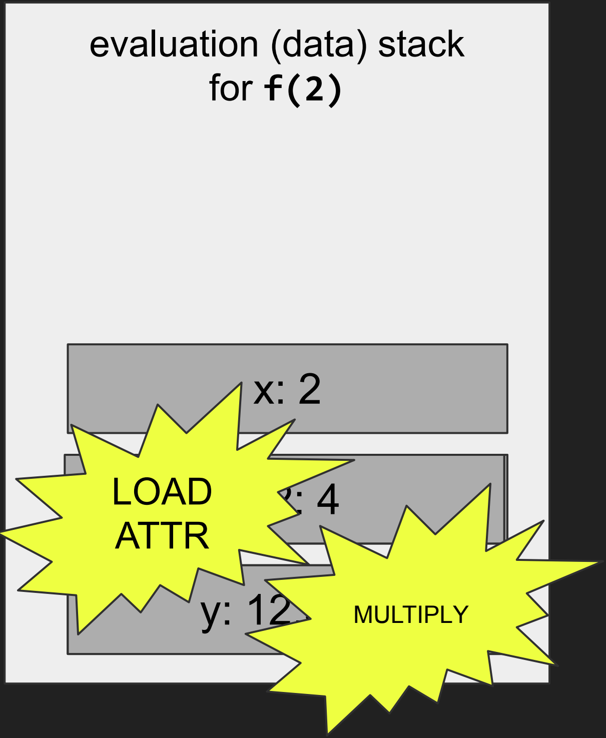
How interpreter sees it

- frame (call) stack
- evaluation (data) stack
- block stack

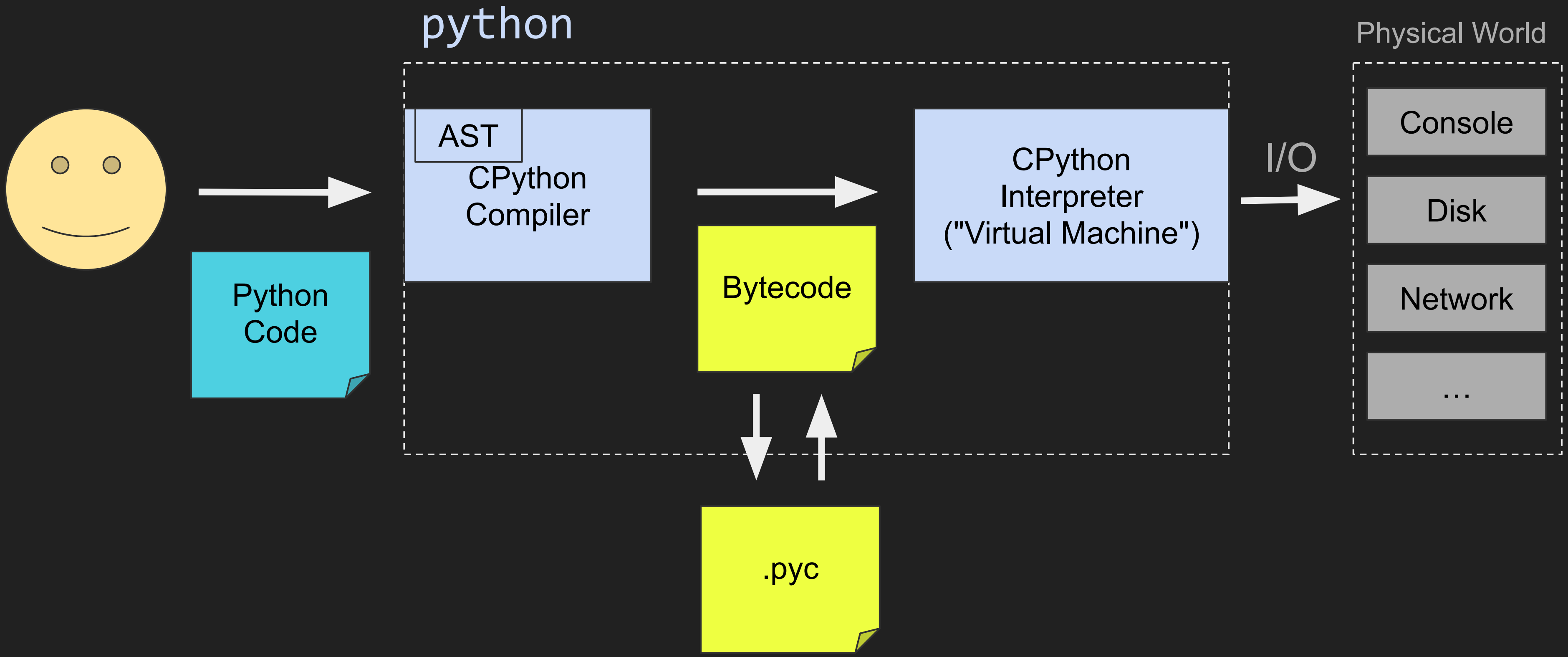


```
5        0 RESUME        ✅0

6        2 LOAD_CONST    ✅1 (3.1415)
         4 LOAD_GLOBAL   ✅1 (NULL + math)
        14 LOAD_ATTR     ✅2 (pow)
        34 LOAD_FAST     ✅0 (x)
        36 CALL          ✅1
        44 BINARY_OP     ✅5 (*)
        48 STORE_FAST    ✅1 (y)

7       50 LOAD_FAST     ✅1 (y)
        52 RETURN_VALUE  ✅
```

```python
def f(x):
    y = 3.1415 * math.pow(x)
    return y
```

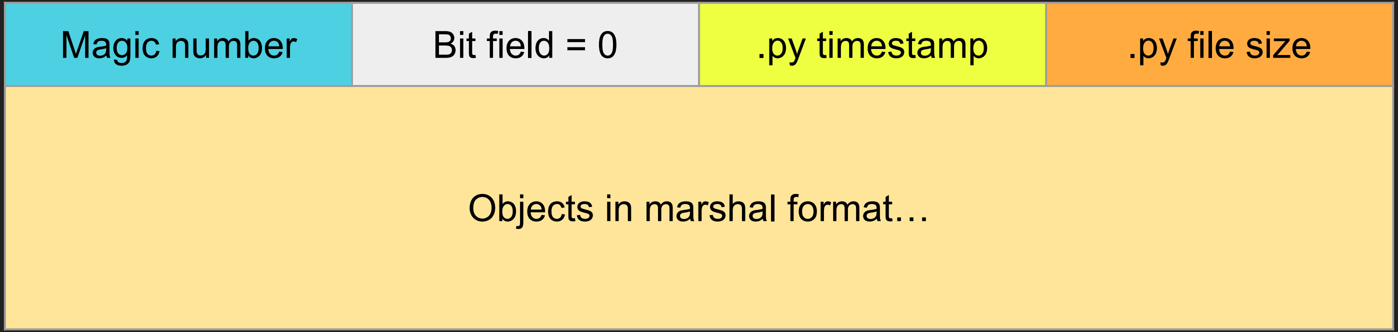


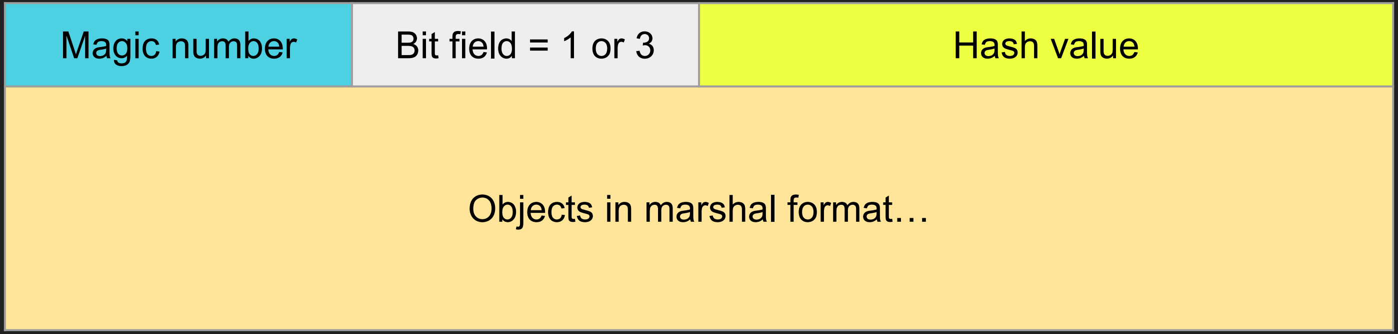
Bytecode in files



.pyc (cache files)

`importlib.util.MAGIC_NUMBER`
3.11: 0x**a70d0d0a**
3.12: 0x**cb0d0d0a**
3.13: 0x**f30d0d0a**
3.14-rc: 0x**1d0e0d0a**





Creation of **.pyc** and **__pycache__** (cache directories)

Created on import, on pip install. Also Python standard library comes with precompiled files.

```
python -m py_compile FILE
python -m compileall DIR_OR_FILE
```

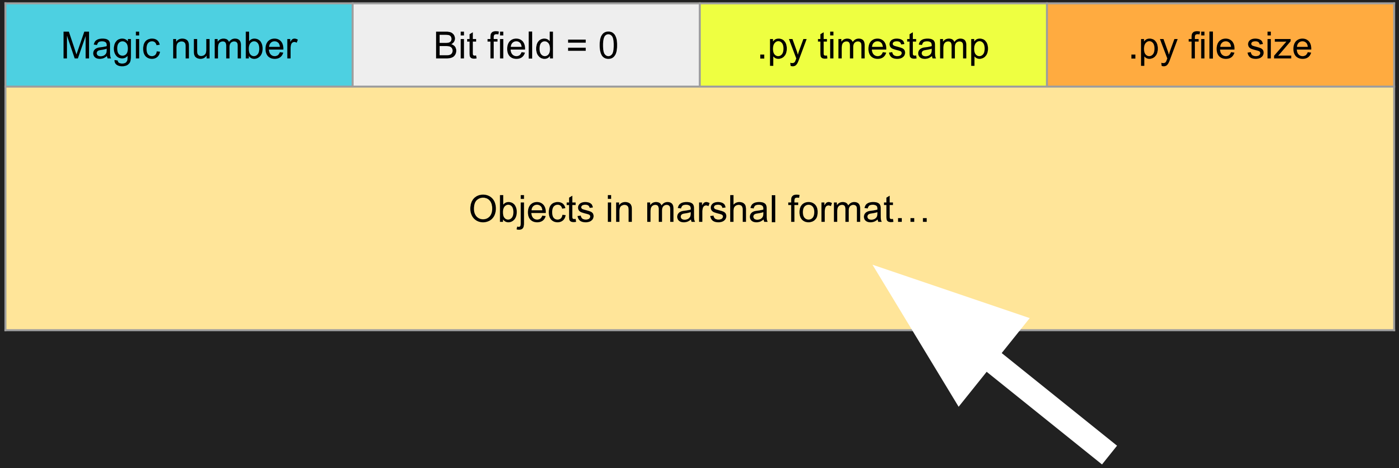
```
$ python -m compileall test.py
Compiling 'test.py'...
```

```
$ ls __pycache__
test.cpython-312.pyc
```

```
$ python __pycache__/test.cpython-312.pyc
Hello world!
```

Avoid unneeded imports

Simple Dockerfile? Consider:
`python -m compileall .`


Magic number
Bit field = 0
.py timestamp
.py file size
Objects in marshal format…

Marshal???

Marshal format

Docs:

Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does).

```
>>> with open("__pycache__/test.cpython-312.pyc", "rb") as f:
...     f.seek(16)
...     print(marshal.load(f))
...
16
<code object <module> at 0x7be3bf4f79f0, file "test.py", line 1>
```

Other useful things

Optimization

```
$ python -m compileall -o 1 -o 2 test.py
Compiling 'test.py'...

$ ls __pycache__
test.cpython-312.opt-1.pyc  test.cpython-312.opt-2.pyc
```

Levels:

- -1: use default
- 0: no optimization; `__debug__` is true
- 1: asserts are removed, `__debug__` is false → `python -O main.py`
- 2: docstrings are removed too → `python -OO main.py`

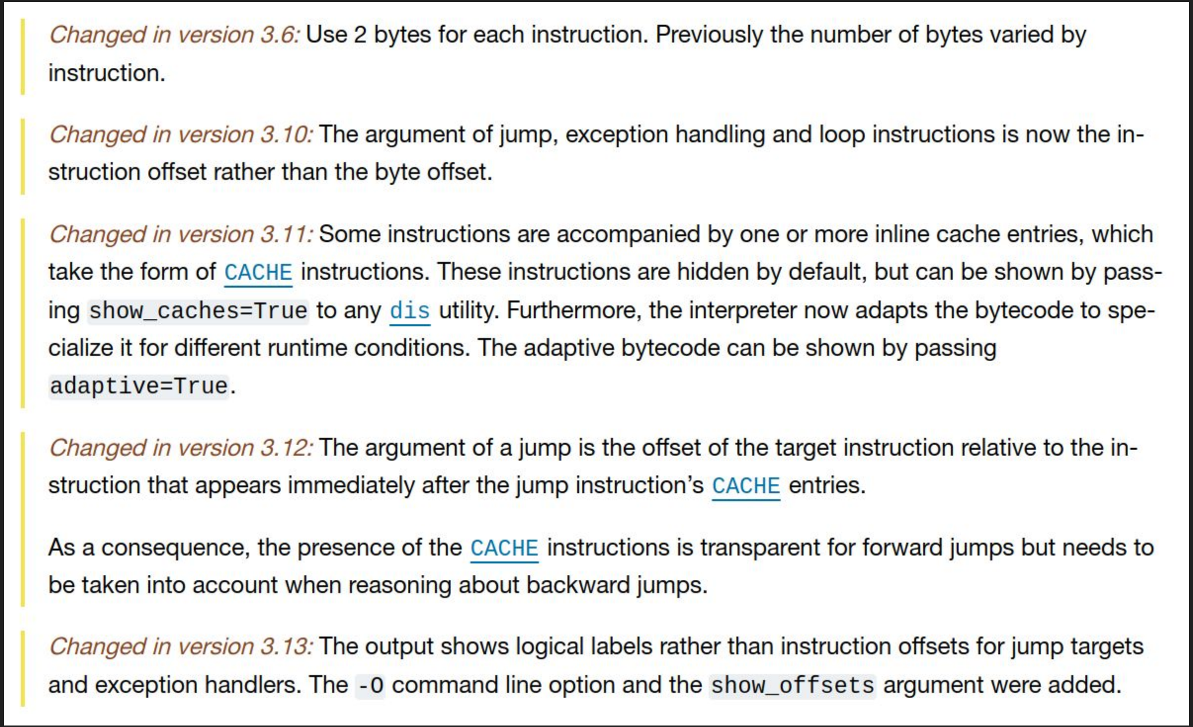
Exception handling

```
>>> [][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> dis.dis()
  0           0 RESUME                   0

  1           2 BUILD_LIST               0
              4 LOAD_CONST               0 (0)
    -->       6 BINARY_SUBSCR
             10 CALL_INTRINSIC_1         1 (INTRINSIC_PRINT)
             12 POP_TOP
             14 RETURN_CONST             1 (None)
```

`dis.dis()`
If no object is provided, this function disassembles the last traceback.

`dis.distb(tb=None, …)`
Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

History of bytecode in Python versions
Evolution of bytecode over Python versions



Changed in version 3.6: Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

Changed in version 3.10: The argument of jump, exception handling and loop instructions is now the instruction offset rather than the byte offset.

Changed in version 3.11: Some instructions are accompanied by one or more inline cache entries, which take the form of `CACHE` instructions. These instructions are hidden by default, but can be shown by passing `show_caches=True` to any `dis` utility. Furthermore, the interpreter now adapts the bytecode to specialize it for different runtime conditions. The adaptive bytecode can be shown by passing `adaptive=True`.

Changed in version 3.12: The argument of a jump is the offset of the target instruction relative to the instruction that appears immediately after the jump instruction's `CACHE` entries.

As a consequence, the presence of the `CACHE` instructions is transparent for forward jumps but needs to be taken into account when reasoning about backward jumps.

Changed in version 3.13: The output shows logical labels rather than instruction offsets for jump targets and exception handlers. The `-O` command line option and the `show_offsets` argument were added.

```python
def add(a, b):
    return a + b
```

```
>>> dis.dis(add)
  1           RESUME                  0
  2           LOAD_FAST_LOAD_FAST     1 (a, b)
              BINARY_OP               0 (+)
              RETURN_VALUE
```

Bytecode instruction specialization

PEP 659: Specializing Adaptive Interpreter

```
>>> dis.dis(add, adaptive=True)
  1           RESUME                   0
  2           LOAD_FAST_LOAD_FAST      1 (a, b)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

```
>>> add(1, 1)
>>> add(1, 1)
>>> dis.dis(add, adaptive=True)
  1           RESUME                  0
  2           LOAD_FAST_LOAD_FAST     1 (a, b)
              BINARY_OP_ADD_INT       0 (+)
              RETURN_VALUE
```

```
>>> add(1.0, 1.0)
>>> # ... repeat many times
>>> dis.dis(add, adaptive=True)
  1           RESUME                    0
  2           LOAD_FAST_LOAD_FAST       1 (a, b)
              BINARY_OP_ADD_FLOAT       0 (+)
              RETURN_VALUE
```

Not so useful (?), but cool

Modify function on the fly?

```
>>> from types import CodeType
>>> help(CodeType)

Help on class code in module builtins:

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

```
from types import FunctionType

FunctionType(add.__code__, {})()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() missing 2 required positional arguments: 'a' and 'b'

FunctionType(add.__code__, {})(2, 3)
5

FunctionType(add.__code__.replace(), {})(2, 3)
5

FunctionType(add.__code__.replace(co_varnames=('x', 'y')), {})()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: add() missing 2 required positional arguments: 'x' and 'y'

FunctionType(add.__code__.replace(co_varnames=('x', 'y')), {})(2, 3)
5
```

```
list(add.__code__.co_code)
             [151, 0, 124, 0, 124, 1, 122, 0, 0, 0, 83, 0]

code = bytes([151, 0, 124, 0, 124, 1, 122, 1, 0, 0, 83, 0])

FunctionType(add.__code__.replace(co_code=code), {})(2, 3)
2

>>> for x in range(25):
...   code = bytes([151, 0, 124, 0, 124, 1, 122, x, 0, 0, 83, 0])
...   print(FunctionType(add.__code__.replace(co_code=code), {})(2, 3))

5
2
0
16
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for @: 'int' and 'int'
```

```
>>> for x in range(25):
...  code = bytes([151, 0, 124, 0, 124, 1, 122, x, 0, 0, 83, 0])
...  try:
...   print(FunctionType(add.__code__.replace(co_code=code), {})(2, 3))
...  except Exception as e:
...   print(e)

5
2
0
16
unsupported operand type(s) for @: 'int' and 'int'
6
2
3
8
0
-1
0.6666666666666666
1
5
2
0
16
unsupported operand type(s) for @=: 'int' and 'int'
6
2
3
8
0
-1
0.6666666666666666
```

```
>>> for x in range(128):
...  code = bytes([151, 0, 124, 0, 124, 1, 122, x, 0, 0, 83, 0])
...  try:
...   f = FunctionType(add.__code__.replace(co_code=code), {})
...   print(f"{x}: 2 {list(dis.get_instructions(f))[3].argrepr} 3 --> {f(2,3)}")
...  except Exception as e:
...   print(e)

0: 2 + 3 --> 5
1: 2 & 3 --> 2
2: 2 // 3 --> 0
3: 2 << 3 --> 16
unsupported operand type(s) for @: 'int' and 'int'
5: 2 * 3 --> 6
6: 2 % 3 --> 2
7: 2 | 3 --> 3
8: 2 ** 3 --> 8
9: 2 >> 3 --> 0
10: 2 - 3 --> -1
11: 2 / 3 --> 0.6666666666666666
12: 2 ^ 3 --> 1
13: 2 += 3 --> 5
14: 2 &= 3 --> 2
15: 2 //= 3 --> 0
16: 2 <<= 3 --> 16
unsupported operand type(s) for @=: 'int' and 'int'
18: 2 *= 3 --> 6
19: 2 %= 3 --> 2
20: 2 |= 3 --> 3
21: 2 **= 3 --> 8
22: 2 >>= 3 --> 0
23: 2 -= 3 --> -1
24: 2 /= 3 --> 0.6666666666666666
25: 2 ^= 3 --> 1
```

Thanks for your attention

Konrad Gawda

linkedin: konradgawda

orange™

Discord Python Polska

https://discord.gg/QEUyNcAx