Bytecode

and .pyc files

Konrad Gawda

Swiss
Python
Summit

Konrad Gawda

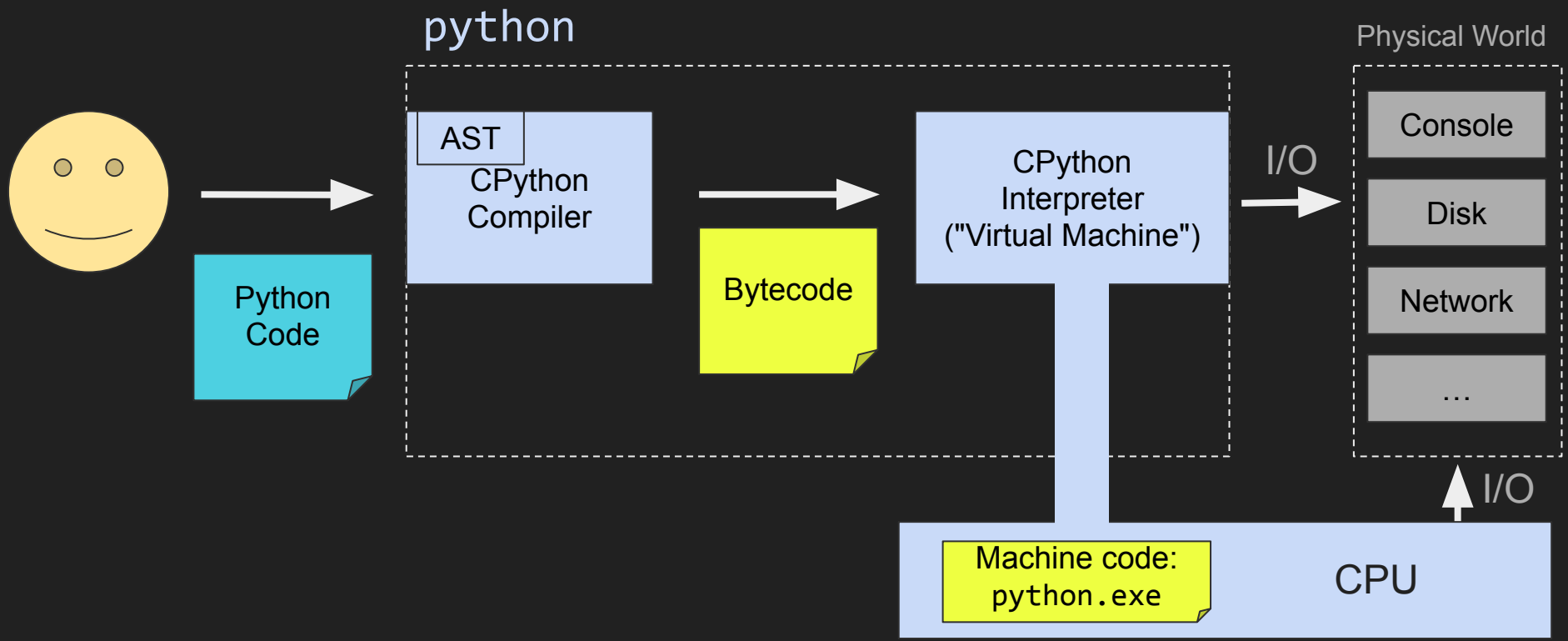
Python - programmer and trainer

Cloud Evangelist - Integrated Computing

Videocast host

orange™

```
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
```


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

Big CPython switch

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

```
143         switch (opcode) {
144
145     // BEGIN BYTECODES //
146             pure inst(NOP, (--)) {
147             }
148
149             family(RESUME, 0) = {
150                 RESUME_CHECK,
151             };
152
153             macro(NOT_TAKEN) = NOP;
154
155             op(_CHECK_PERIODIC, (--)) {
156                 _Py_CHECK_EMSCRIPTEN_SIGNALS_PERIODICALLY();
157                 QSBR_QUIESCENT_STATE(tstate);
158                 if (_Py_atomic_load_uintptr_relaxed(&tstate->eval_breaker) & _PY_EVAL_EVENTS_MASK) {
159                     int err = _Py_HandlePending(tstate);
160                     ERROR_IF(err != 0, error);
161                 }
162             }
163
164             op(_CHECK_PERIODIC_IF_NOT_YIELD_FROM, (--)) {
165                 if ((oparg & RESUME_OPARG_LOCATION_MASK) < RESUME_AFTER_YIELD_FROM) {
166                     _Py_CHECK_EMSCRIPTEN_SIGNALS_PERIODICALLY();
```

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
```

```
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)

```
3.14
  1           RESUME                                0
  2           LOAD_FAST_BORROW_LOAD_FAST_BORROW     1 (a, b)
              BINARY_OP                             0 (+)
              RETURN_VALUE

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, s
Instruction(opname='RETURN_VALUE', opcode=83, arg=None, argval=None, argrepr='', offs
```

```
def f(x):
    y = 0.5 * math.sqrt(x)
    return y
```

```
def f(x):
    y = 0.5 * math.sqrt(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,
```

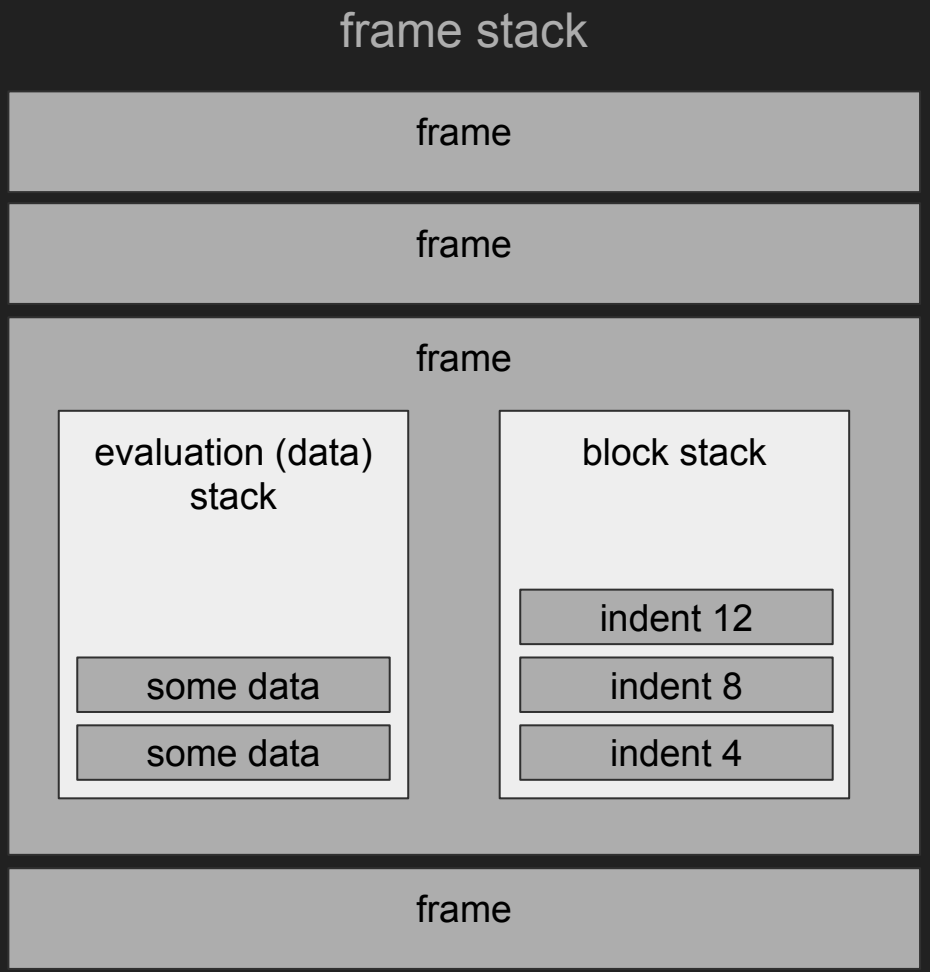
```
5      0 RESUME          0

6      2 LOAD_CONST      1 (0.5)
       4 LOAD_GLOBAL     1 (NULL + math)
      14 LOAD_ATTR       2 (sqrt)
      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, 0.5)
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', 'sqrt')
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

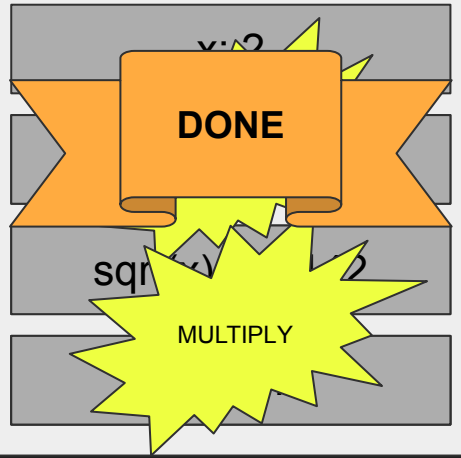


Python 3.12

```
  5      0 RESUME       ✅0

  6      2 LOAD_CONST   ✅1 (0.5)
         4 LOAD_GLOBAL  ✅1 (NULL + math)
        14 LOAD_ATTR    ✅2 (sqrt)
        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 ✅
```

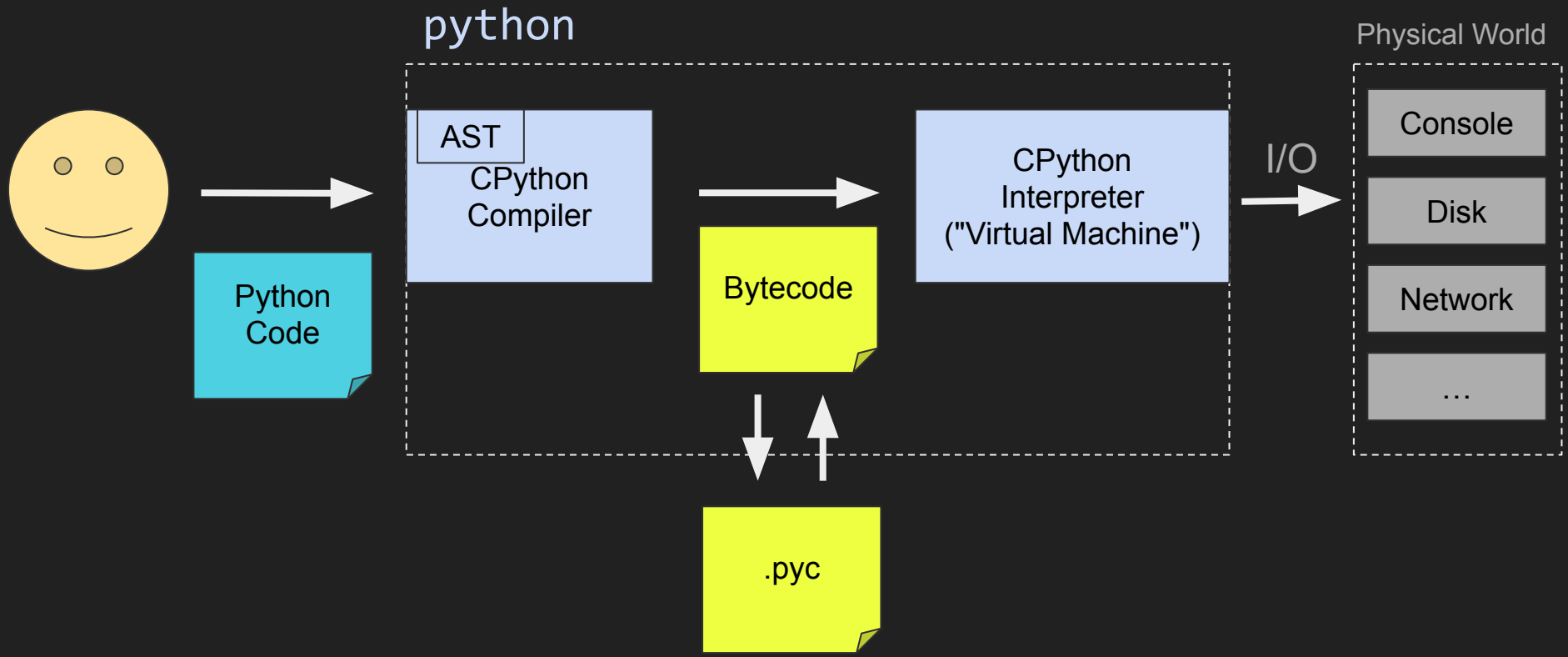
```python
def f(x):
    y = 0.5 * math.sqrt(x)
    return y
```

evaluation (data) stack
for **f(2)**



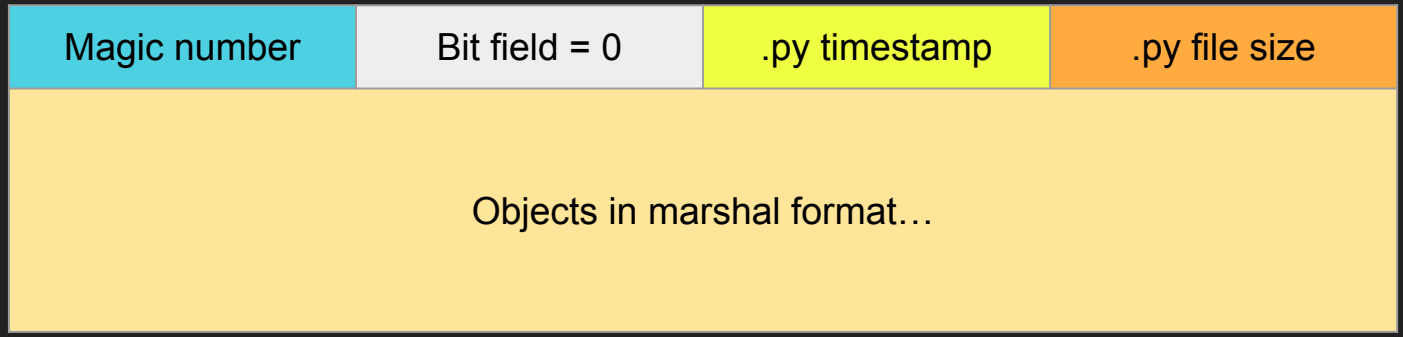
.pyc files

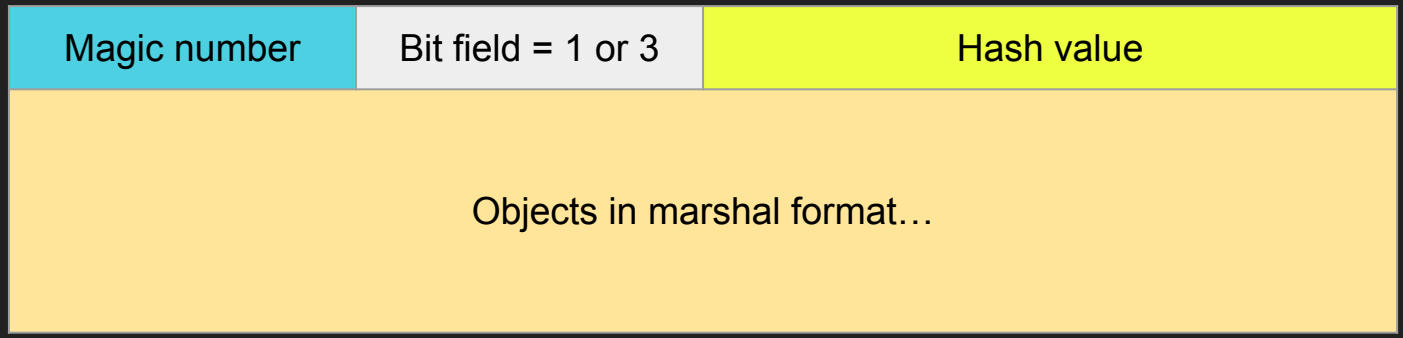
Bytecode in files



.pyc (cache files)

```
importlib.util.MAGIC_NUMBER
3.11: 0xa70d0d0a
3.12: 0xcb0d0d0a
3.13: 0xf30d0d0a
3.14-rc: 0x1d0e0d0a
```





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

Created on **import**, on **pip install**. Also Python **standard library** comes with precompiled files. Or created manually:

```
python -m py_compile FILE
python -m compileall DIR_OR_FILES
```

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

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

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

How is it in **uv** today? :P

Avoid unneeded imports

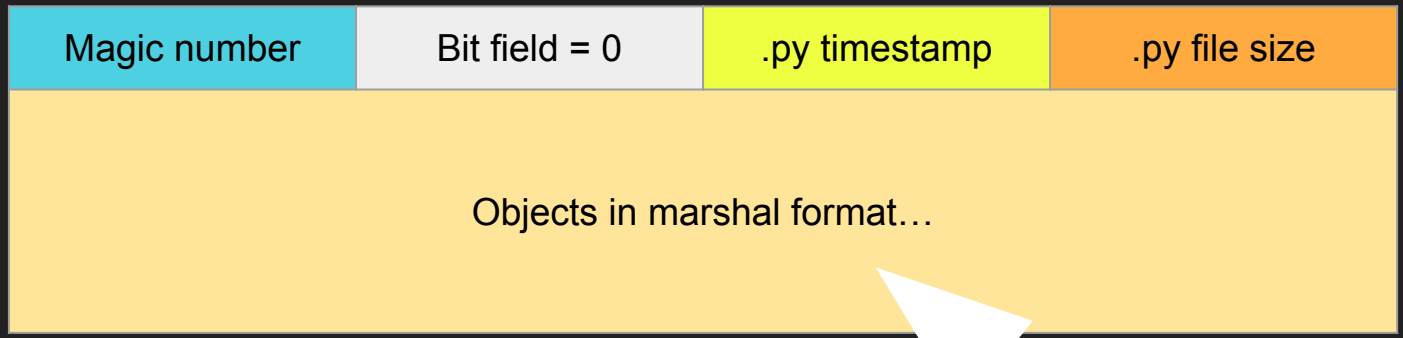
Simple Dockerfile? Consider:
`python -m compileall .`

But .pyc files take space…

Yes. Both .py and .pyc take space
Example case:
.py: 600 kB
.pyc: 800 kB

If in real need,
consider **removing .py** files

```
docker python main.pyc
```


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)
...     m = marshal.load(f)
16
>>> print(m)
<code object <module> at 0x7be3bf4f79f0, file "test.py", line 1>
```

```
>>> from types import FunctionType
>>> module_as_function = FunctionType(m, {})
>>> module_as_function()
0.7071067811865476
```

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:

- 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.

```
def iter_pizzas():
    pizzas = ['Margherita', 'Pepperoni', 'Capriciosa', 'Fungi', ...
    for x in pizzas:
        yield x
```

pizzas created on every function run

```
def iter_pizzas():
    pizzas = ('Margherita', 'Pepperoni', 'Capriciosa', 'Fungi', ...
    for x in pizzas:
        yield x
```

pizzas in `__code__.co_consts`
(not mutable, does not contain mutable)

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
```

```
...     f = FunctionType(add.__code__.repla
...     print(f"{x}: 2 {list(dis.get_instru
```

```
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
```


```
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
```

Thanks for your attention

Konrad Gawda

LinkedIn: konradgawda

Cloud: Integrated Computing Standard

orange™

Machine code… and JIT?