Byterun: A (C)Python interpreter in Python

Allison Kaptur

github.com/akaptur
akaptur.com
@akaptur

Byterun
with Ned Batchelder

Based on
pyvm2 by Paul Swartz (z3p)
from http://www.twistedmatrix.com/users/z3p/

“Interpreter”

1. Lexing
2. Parsing
3. Compiling
4. Interpreting

The Python virtual machine:

A bytecode interpreter

Bytecode: the internal representation of a python program in the interpreter

Why write an interpreter?

```
>>> if a or b:
...     pass
```

Testing

```
def test_for_loop(self):
    self.assert_ok("""\
        out = ""
        for i in range(5):
            out = out + str(i)
        print(out)
        """)
```

A problem

```
def test_for_loop(self):
    self.assert_ok("""\
        g = (x*x for x in range(5))
        h = (y+1 for y in g)
        print(list(h))
        """)
```

A simple VM

- LOAD_VALUE
- ADD_TWO_VALUES
- PRINT_ANSWER

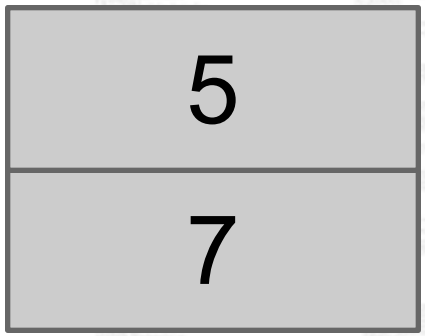
A simple VM

```
"7 + 5"

["LOAD_VALUE",
 "LOAD_VALUE",
 "ADD_TWO_VALUES",
 "PRINT_ANSWER"]
```

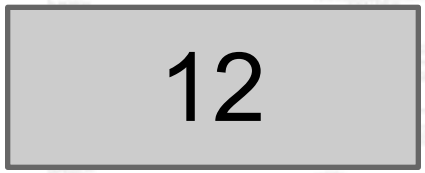
A simple VM

Before

After LOAD_ VALUE



After ADD_TWO_ VALUES



After PRINT_ ANSWER

A simple VM

```
what_to_execute = {
  "instructions":
    [("LOAD_VALUE", 0),
     ("LOAD_VALUE", 1),
     ("ADD_TWO_VALUES", None),
     ("PRINT_ANSWER", None)],

    "numbers": [7, 5] }
```

```
class Interpreter(object):
    def __init__(self):
        self.stack = []

    def value_loader(self, number):
        self.stack.append(number)

    def answer_printer(self):
        answer = self.stack.pop()
        print(answer)

    def two_value_adder(self):
        first_num = self.stack.pop()
        second_num = self.stack.pop()
        total = first_num + second_num
        self.stack.append(total)
```

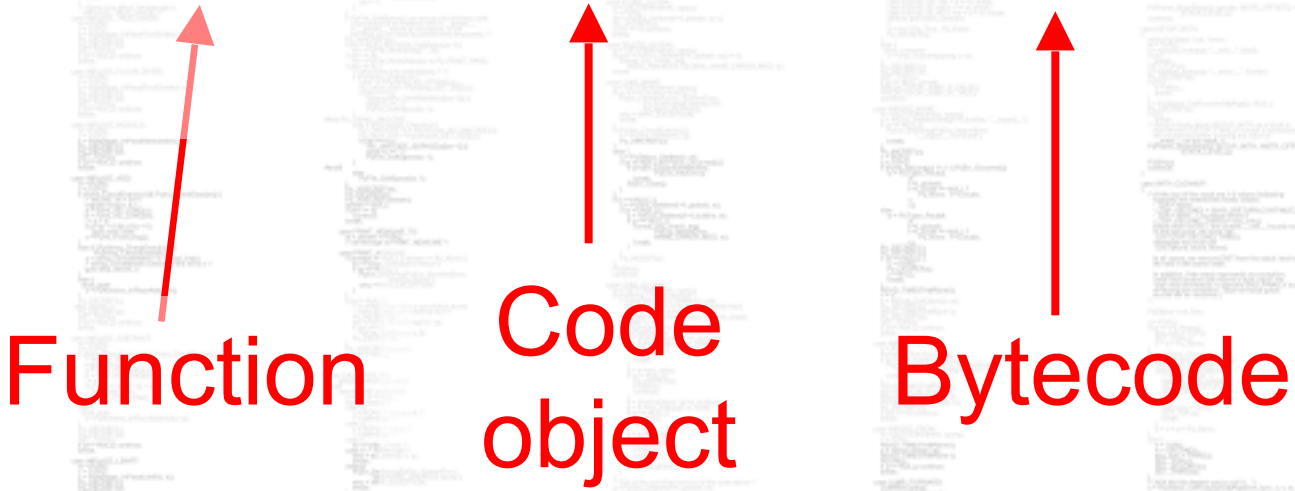
```
def run_code(self, what_to_execute):
    instrs = what_to_execute["instructions"]
    numbers = what_to_execute["numbers"]
    for each_step in instrs:
        instruction, argument = each_step
        if instruction == "LOAD_VALUE":
            number = numbers[argument]
            self.value_loader(number)
        elif instruction == "ADD_TWO_VALUES":
            self.two_value_adder()
        elif instruction == "PRINT_ANSWER":
            self.answer_printer()


interpreter = Interpreter()
interpreter.run_code(what_to_execute)
# 12
```

Bytecode: it's bytes!

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
```

Bytecode: it's bytes!

```
>>> def mod(a, b):
...         ans = a % b
...         return ans
>>> mod.func_code.co_code
```



Bytecode: it's bytes!

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod.func_code.co_code
'|\x00\x00|
\x01\x00\x16}\x02\x00|\x02\x00S'
```

Bytecode: it's bytes!

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod.func_code.co_code
'|\x00\x00|
\x01\x00\x16}\x02\x00|\x02\x00S'
>>> [ord(b) for b in
      mod.func_code.co_code]
[124, 0, 0, 124, 1, 0, 22, 125,
2, 0, 124, 2, 0, 83]
```

dis, a bytecode disassembler

```
>>> import dis
>>> dis.dis(mod)
  2    0   LOAD_FAST          0  (a)
       3   LOAD_FAST          1  (b)
       6   BINARY_MODULO
       7   STORE_FAST         2  (ans)

  3   10   LOAD_FAST          2  (ans)
      13   RETURN_VALUE
```

dis, a bytecode disassembler

```
>>> dis.dis(mod)
line ind     name              arg  hint
   2   0   LOAD_FAST            0  (a)
       3   LOAD_FAST            1  (b)
       6   BINARY_MODULO
       7   STORE_FAST           2  (ans)

   3  10  LOAD_FAST            2  (ans)
      13  RETURN_VALUE
```

Bytecode: it's bytes!

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod(7,5)
```

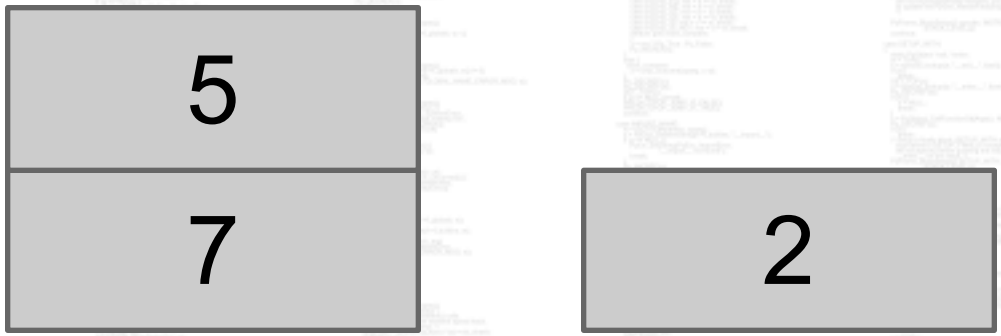
The Python interpreter

Before

After LOAD_ FAST

After BINARY_ MODULO

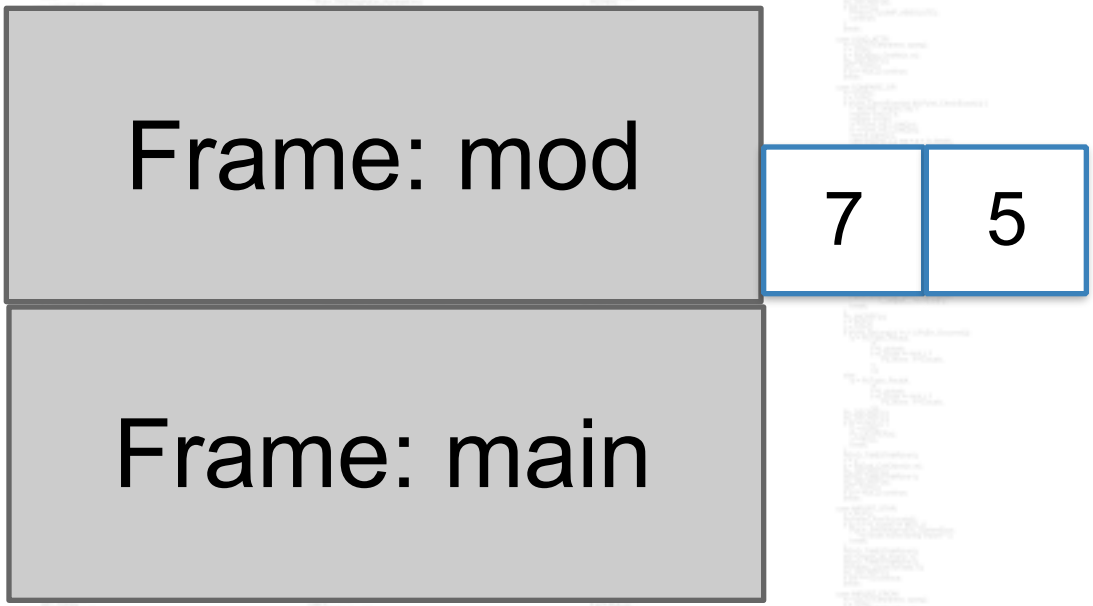
After STORE_ FAST



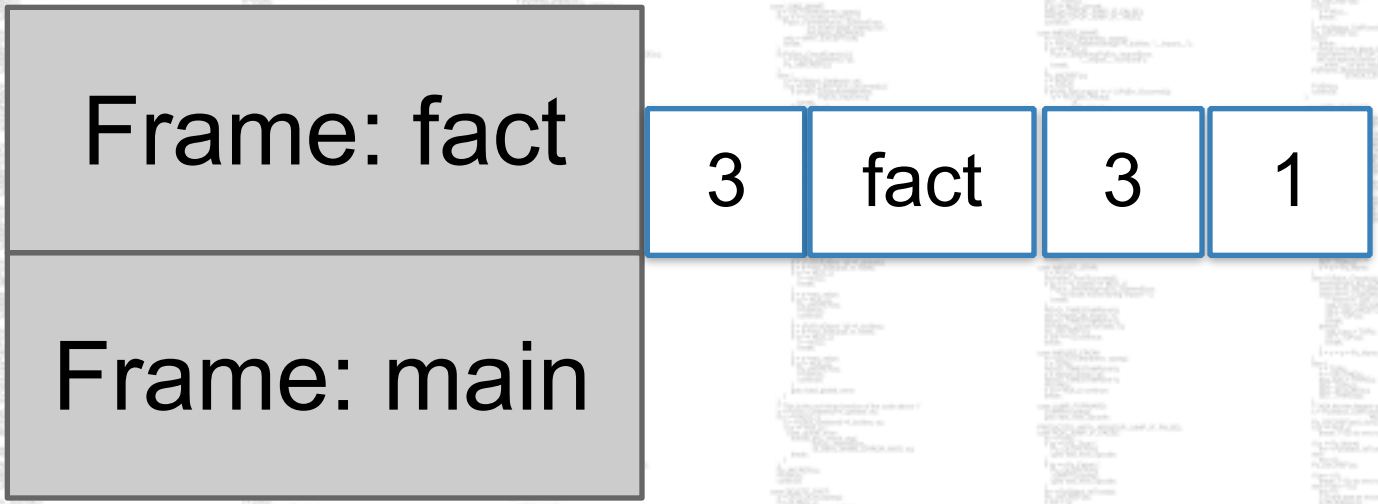
```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod(7,5)

>>> dis.dis(mod)
  2   0   LOAD_FAST        0  (a)
      3   LOAD_FAST        1  (b)
      6   BINARY_MODULO
      7   STORE_FAST       2  (ans)

  3  10  LOAD_FAST        2  (ans)
     13  RETURN_VALUE
```

call stack

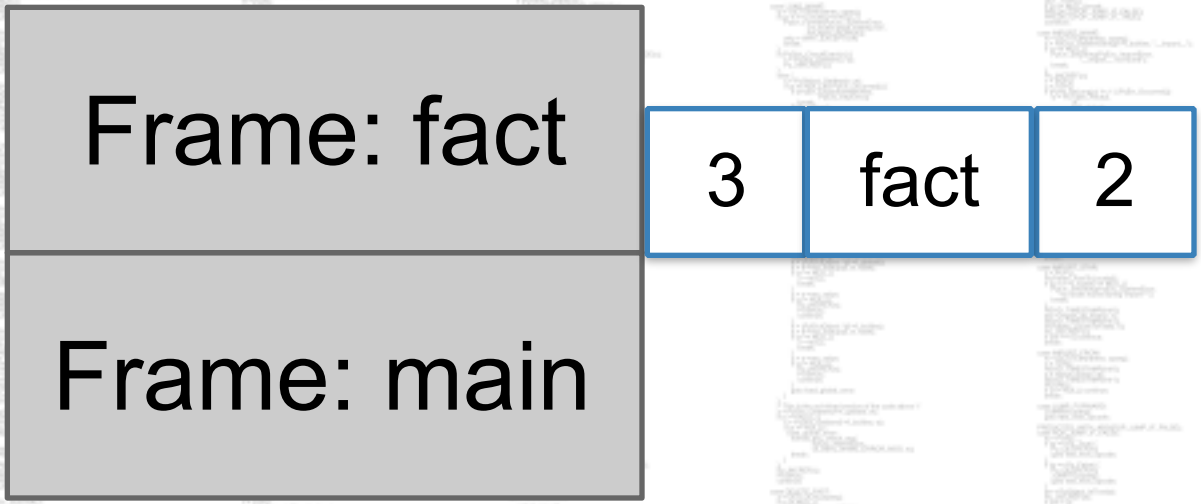
data stack ->



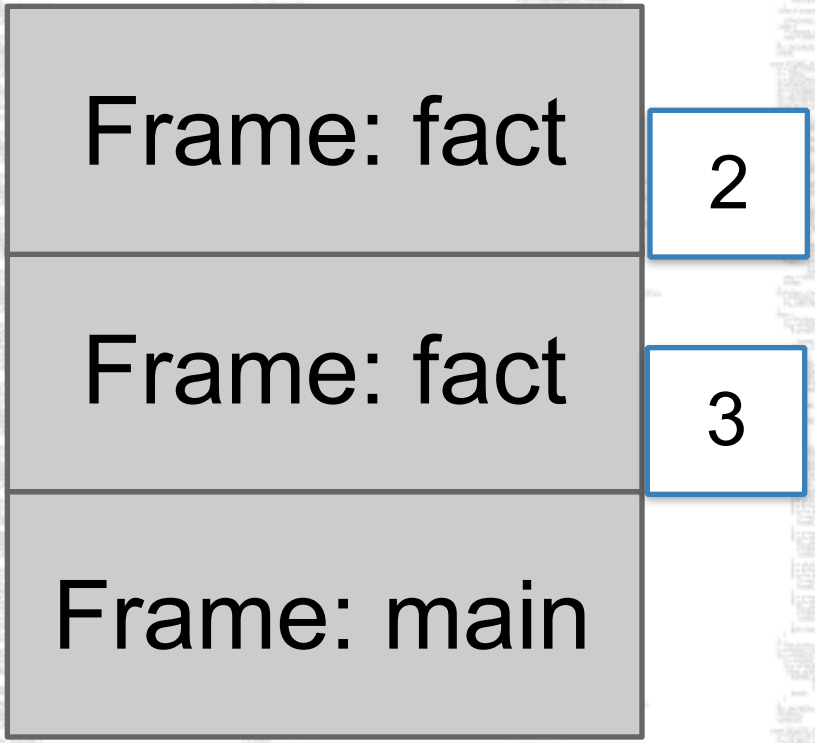
```
>>> def fact(n):
...     if n < 2: return 1
...     else: return n * fact(n-1)
>>> fact(3)
```

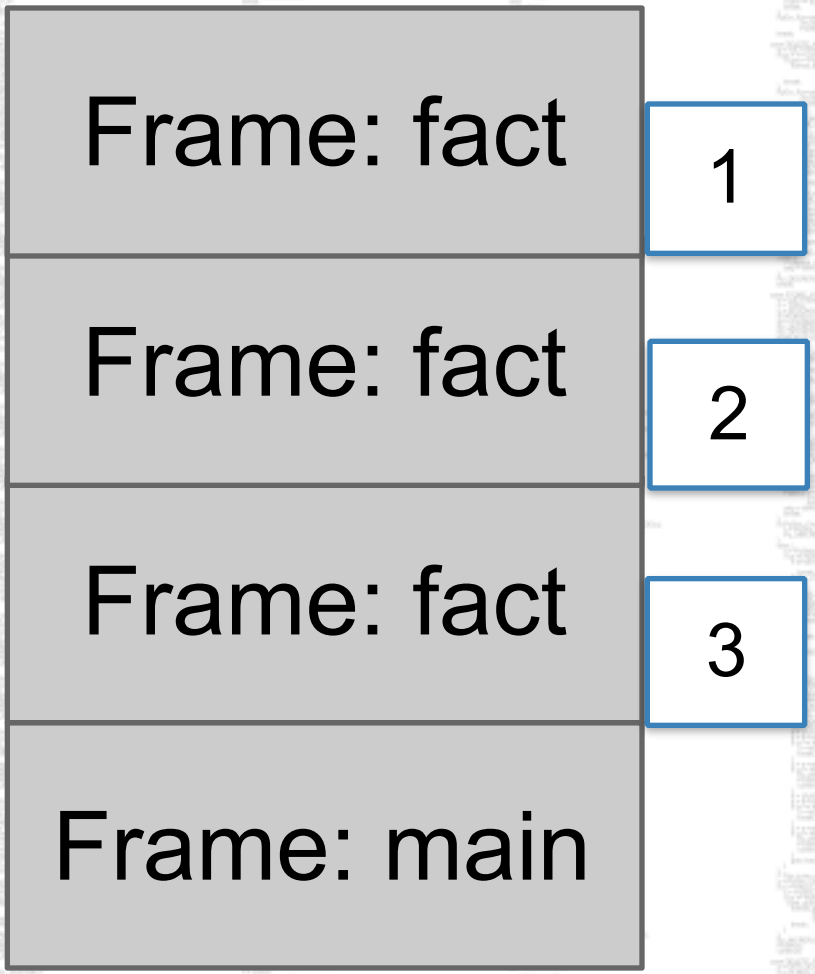


```
>>> def fact(n):
...     if n < 2: return 1
...     else: return n * fact(n-1)
>>> fact(3)
```

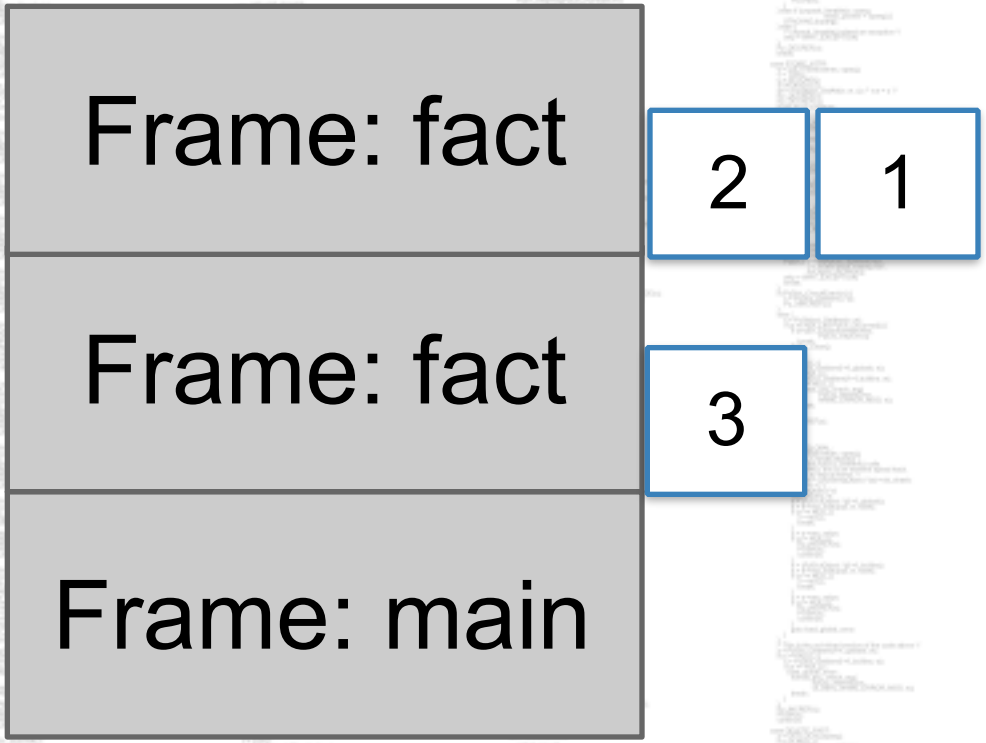


```
>>> def fact(n):
...     if n < 2: return 1
...     else: return n * fact(n-1)
>>> fact(3)
```

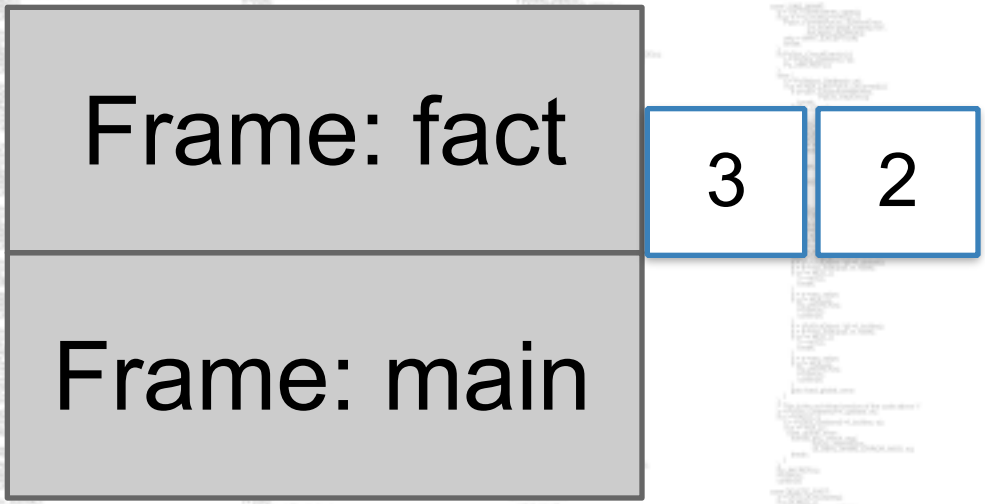



Frame: fact
1
Frame: fact
2
Frame: fact
3
Frame: main

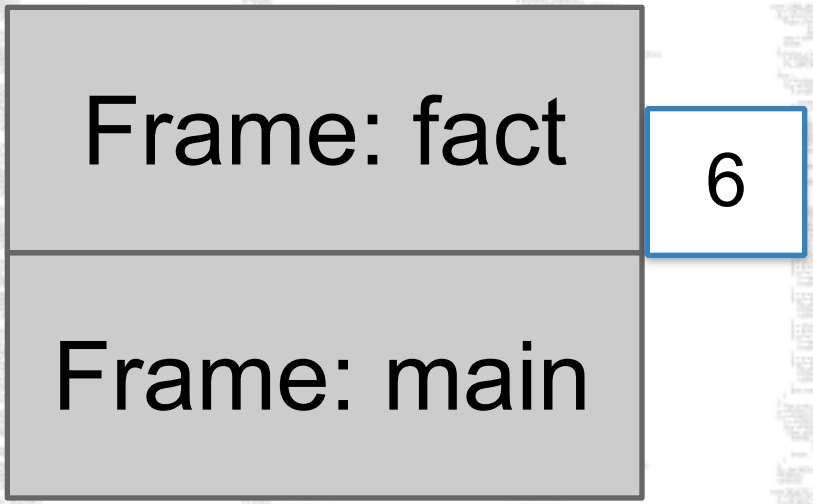
```
>>> def fact(n):
...     if n < 2: return 1
...     else: return n * fact(n-1)
>>> fact(3)
```



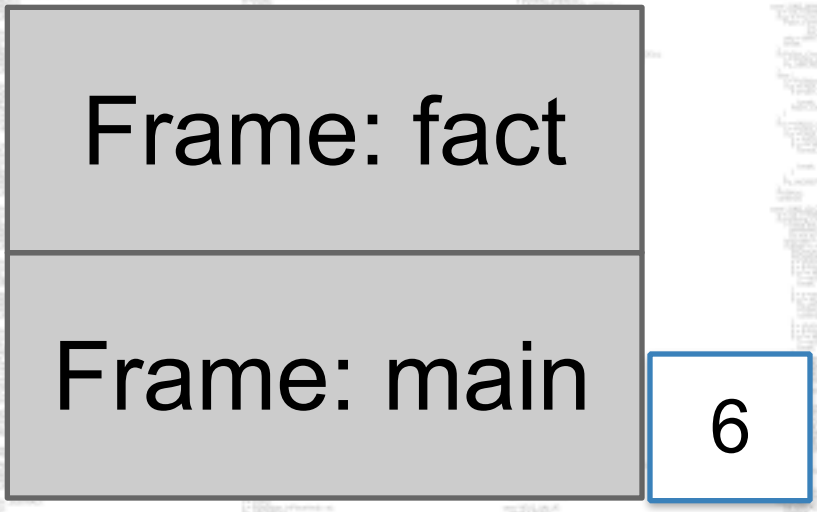
```
>>> def fact(n):
...     if n < 2: return 1
...     else: return n * fact(n-1)
>>> fact(3)
```



```
>>> def fact(n):
...     if n < 2: return 1
...     else: return n * fact(n-1)
>>> fact(3)
```



```
>>> def fact(n):
...     if n < 2: return 1
...     else: return n * fact(n-1)
>>> fact(3)
```



Python VM:

– A collection of frames

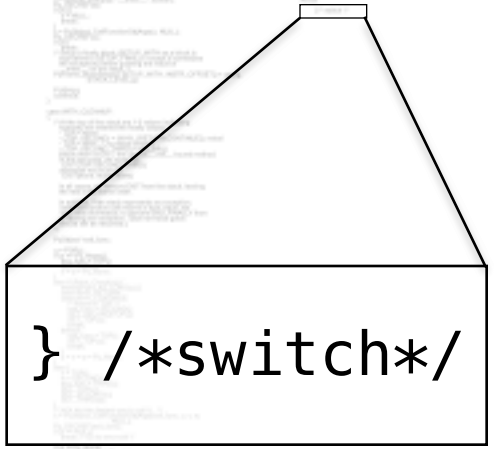
– Data stacks on frames

– A way to run frames

Instructions we need

```
>>> import dis
>>> dis.dis(mod)
  2     0   LOAD_FAST                0  (a)
        3   LOAD_FAST                1  (b)
        6   BINARY_MODULO
        7   STORE_FAST               2  (ans)

  3    10  LOAD_FAST                2  (ans)
       13  RETURN_VALUE
```

```
/* Main switch on opcode
*/
READ_TIMESTAMP(inst0);

switch (opcode) {
```



```
/* Turn this on if your compiler chokes on the big switch: */
/* #define CASE_TOO_BIG 1 */
```

```
#ifdef CASE_TOO_BIG
        default: switch (opcode) {
#endif
```

Instructions we need

```
>>> import dis
>>> dis.dis(mod)
  2     0   LOAD_FAST                0  (a)
        3   LOAD_FAST                1  (b)
        6   BINARY_MODULO
        7   STORE_FAST               2  (ans)

  3    10  LOAD_FAST                2  (ans)
       13  RETURN_VALUE
```

```
case LOAD_FAST:
    x = GETLOCAL(oparg);
    if (x != NULL) {
        Py_INCREF(x);
        PUSH(x);
        goto fast_next_opcode;
    }

format_exc_check_arg(PyExc_UnboundLocalError,
        UNBOUNDLOCAL_ERROR_MSG,
        PyTuple_GetItem(co->co_varnames,
oparg));
    break;
```

```
case BINARY_MODULO:
        w = POP();
        v = TOP();
        if (PyString_CheckExact(v))
            x = PyString_Format(v, w);
        else
            x = PyNumber_Remainder(v, w);
        Py_DECREF(v);
        Py_DECREF(w);
        SET_TOP(x);
        if (x != NULL) continue;
        break;
```

Back to our problem

```
g = (x*x for x in range(5))
h = (y+1 for y in g)
print(list(h))
```

It's "dynamic"

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod(15, 4)
3
```

“Dynamic”

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod(15, 4)
3
>>> mod(“%s%s”, (“Py”, “Con”))
```

“Dynamic”

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod(15, 4)
3
>>> mod(“%s%s”, (“Py”, “Con”))
PyCon
```

“Dynamic”

```
>>> def mod(a, b):
...     ans = a % b
...     return ans
>>> mod(15, 4)
3
>>> mod(“%s%s”, (“Py”, “Con”))
PyCon
>>> print “%s%s” % (“Py”, “Con”)
PyCon
```

dis, a bytecode disassembler

```
>>> import dis
>>> dis.dis(mod)
  2    0   LOAD_FAST          0  (a)
       3   LOAD_FAST          1  (b)
       6   BINARY_MODULO
       7   STORE_FAST         2  (ans)

  3   10   LOAD_FAST          2  (ans)
      13   RETURN_VALUE
```

```
case BINARY_MODULO:
        w = POP();
        v = TOP();
        if (PyString_CheckExact(v))
            x = PyString_Format(v, w);
        else
            x = PyNumber_Remainder(v, w);
        Py_DECREF(v);
        Py_DECREF(w);
        SET_TOP(x);
        if (x != NULL) continue;
        break;
```

```
>>> class Surprising(object):
…         def __mod__(self, other):
…             print “Surprise!”

>>> s = Surprising()
>>> t = Surprsing()
>>> s % t
Surprise!
```

“In the general absence of type information, almost every instruction must be treated as INVOKE_ARBITRARY_METHOD.”

- Russell Power and Alex Rubinsteyn, “How Fast Can We Make Interpreted Python?”

More

Great blogs
http://tech.blog.aknin.name/category/my-projects/pythons-innards/ by @aknin

http://eli.thegreenplace.net/ by Eli Bendersky

Contribute! Find bugs!
https://github.com/nedbat/byterun

Apply to the Recurse Center!
www.recurse.com/apply