... how Python was shaped by leaky internals

Armin Ronacher
@mitsuhiko
Armin Ronacher
@mitsuhiko
Flask / Sentry / Lektor

http://lucumr.pocoo.org/
what is this about
The Leaky Interpreter

- Python is an insanely complex language
- You are being “lied” to in regards to how it works
- People however depend on the little details
- Which makes it very hard to evolve the language
the language you are told
MAGIC = 42

def add_magic(a):
    return a + MAGIC
MAGIC = 42

def add_magic(a):
    return a.__add__(MAGIC)
the language that is
0  LOAD_GLOBAL  0  (MAGIC)
3  LOAD_FAST  0  (a)
6  BINARY_ADD
7  RETURN_VALUE
TARGET_NOARG(BINARY_ADD)
{
    w = POP();
    v = TOP();
    if (PyInt_CheckExact(v) && PyInt_CheckExact(w)) {
        ...
    } else if (PyString_CheckExact(v) && PyString_CheckExact(w)) {
        ...
    } else {
        x = PyNumber_Add(v, w);
    }
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    if (x != NULL) DISPATCH();
    break;
}
PyObject *
PyNumber_Add(PyObject *v, PyObject *w)
{
    PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));
    if (result == Py_NotImplemented) {
        PySequenceMethods *m = v->ob_type->tp_as_sequence;
        Py_DECREF(result);
        if (m && m->sq_concat) {
            return (*m->sq_concat)(v, w);
        }
        result = binop_type_error(v, w, "+");
    }
    return result;
}
static PyObject *
binary_op1(PyObject *v, PyObject *w, const int op_slot)
{
    PyObject *x;
    binaryfunc slotv = NULL, slotw = NULL;

    if (v->ob_type->tp_as_number != NULL)
        slotv = NB_BINOP(v->ob_type->tp_as_number, op_slot);
    if (w->ob_type != v->ob_type && w->ob_type->tp_as_number != NULL) {
        slotw = NB_BINOP(w->ob_type->tp_as_number, op_slot);
        if (slotw == slotv) slotw = NULL;
    }

    if (slotv) {
        if (slotw && PyType_IsSubtype(w->ob_type, v->ob_type)) { … }
        x = slotv(v, w);
        if (x != Py_NotImplemented) return x;
        Py_DECREF(x); /* can't do it */
    }
    if (slotw) { … }
    Py_RETURN_NOTIMPLEMENTED;
}
So where is __add__?
slots :-(
What's a Slot?

- Slots are struct members in the PyTypeObject
- Each special method is wrapped and stored there
- `Foo.__add__` can be `FooType.tp_as_number.nb_add`
Weird Slots

- `FooType.tp_as_number.nb_add`
- `FooType.tp_as_sequence.nb_concat`

- Both correspond to `a+b (~__add__)`
Explaining Operators
• $a + b = a.__add__(b)$
• slightly more correct: `type(a).__add__(b)`

• Both wrong though
• are \( a \) and \( b \) integers? Then try fast add
• are \( a \) and \( b \) strings? Then try fast concat
• number addition:
  • does \( a \) implement number slots? resolve \texttt{nb_add} slot
  • does \( b \) implement number slots? resolve \texttt{nb_add} slot
  • based on type relationship use callback from \( a \) or \( b \)
• sequence concatenation:
  • does \( a \) implement sequence slots? invoke \texttt{sq_concat} slot

\[ a + b \]
a.__add__(b)

- Invoke attribute lookup flow on \texttt{type(a)}
- Ask to look up the \texttt{__add__} attribute
- Invoke the return value of the lookup with \texttt{b}
How do they do similar things?

- Depends on the type of the object
- C types expose slot wrappers to Python
- Python objects place Python functions in type slots
they are not equivalent!
one like the other
Python Objects

```python
class X(object):
    __add__ = lambda *x: 42

>>> X.__add__
<unbound method X.<lambda>>
```
C Objects

```python
>>> int.__add__
<slot wrapper '__add__' of 'int' objects>
```
python tries to "sync" them up
why do we care?
it's complex and canon
it makes optimizations impossible
PyPy needs to emulate all that
it shapes the language
The C API Leaks

Python 2.6.9 (unknown, Oct 23 2015, 19:19:20)  
[<GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)>] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import re  
>>> x = re.compile('foo')  
>>> x.__class__  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: __class__
>>> class X:
...     def __getattr__(self, name):
...         return getattr(42, name)
...
>>> a = X()
>>> a
42
>>> a + 23
65
so how did that work?
'instance' types forward all calls
UNICODE
We guaranteed too much

>>> u"foo"[0]
u'f'
UCS2 / UCS4 :'(
why did we end up here?
Two Pythons

• C Types and Python Classes evolved side-by-side
• Were later unified
• Optimizations always shine through :-(
• When it desyncs, it gets weird
Frames and Locals
Interpreter Internals

```python
>>> import sys
>>> sys._getframe().f_locals['foo'] = 42
>>> foo
42
```
Who uses getframe anyways

• Zope Interface
• warnings module
• inspect
• logging
• Debug Support (also Sentry)

• getframe and friends are everywhere
sys.modules :('((((('}
```python
import sys

def import_module(module):
    __import__(module)
    return sys.modules[module]
```
bad import API and pickle took away our chances of getting versioned modules
static types
type vs class

>>> int
<type 'int'>
>>> class X(int):
    ...   pass
    ...
...
>>> X
<class '__main__.X'>
Global Types

PyTypeObject PyInt_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "int",
    sizeof(PyIntObject),
    0,
    (destructor)int_dealloc,
    ...
    int_new,
    (freefunc)int_free,
};
C-Level Type Checks

#define PyInt_CheckExact(op) \ ((op)->ob_type == &PyInt_Type)
Consequences
hard to modernize:

getting rid of the GIL
hard to change internals

because all internals are exposed
can't be node.js: no multi version libraries
can't be fast:

expose interpreter logic too much
hard to be concurrent:

refcounts everywhere and exposed
hard to be parallel:

static types are shared :(
hard to be dynamic:

to be fast the interpreter needs to cheat
Shaped Expectations
What Python Programmers Want

• Refcounting or similar behavior
• Ability to access the interpreter state
• Lots and lots of metaprogramming
The Quirks gave birth to

- PDB
- ORMs
- Zope Interface and friends
- Many proxy objects
- Manhole
- Sentry :)