Here Be Dragons

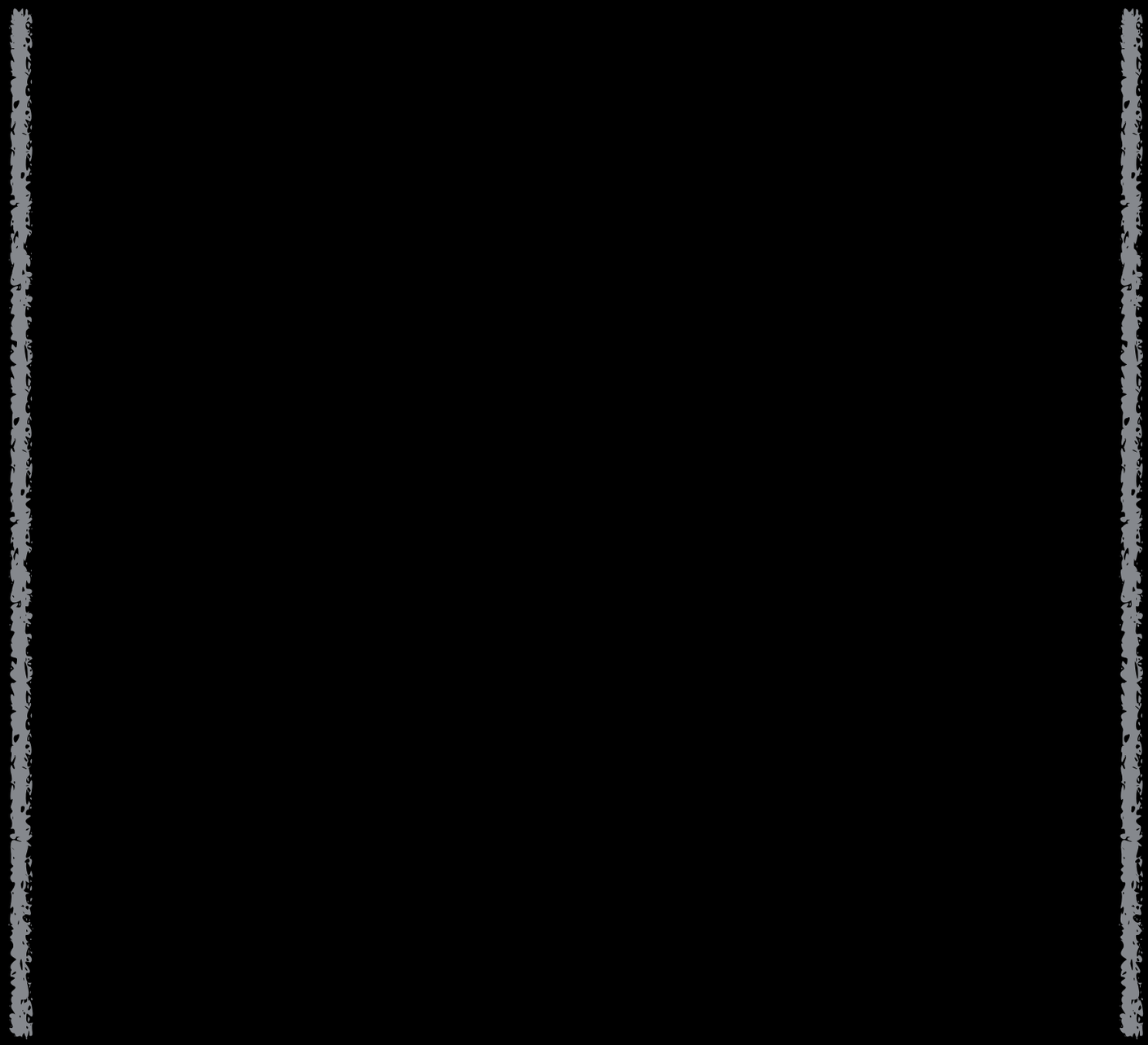
Writing Reliable Python Extensions in C

Paul Ross

Why?

- Blinding performance
- Leaner resources
- Interface with C/C++ libraries

Here Be Dragons



Here Be Dragons

Here Be Dragons

C

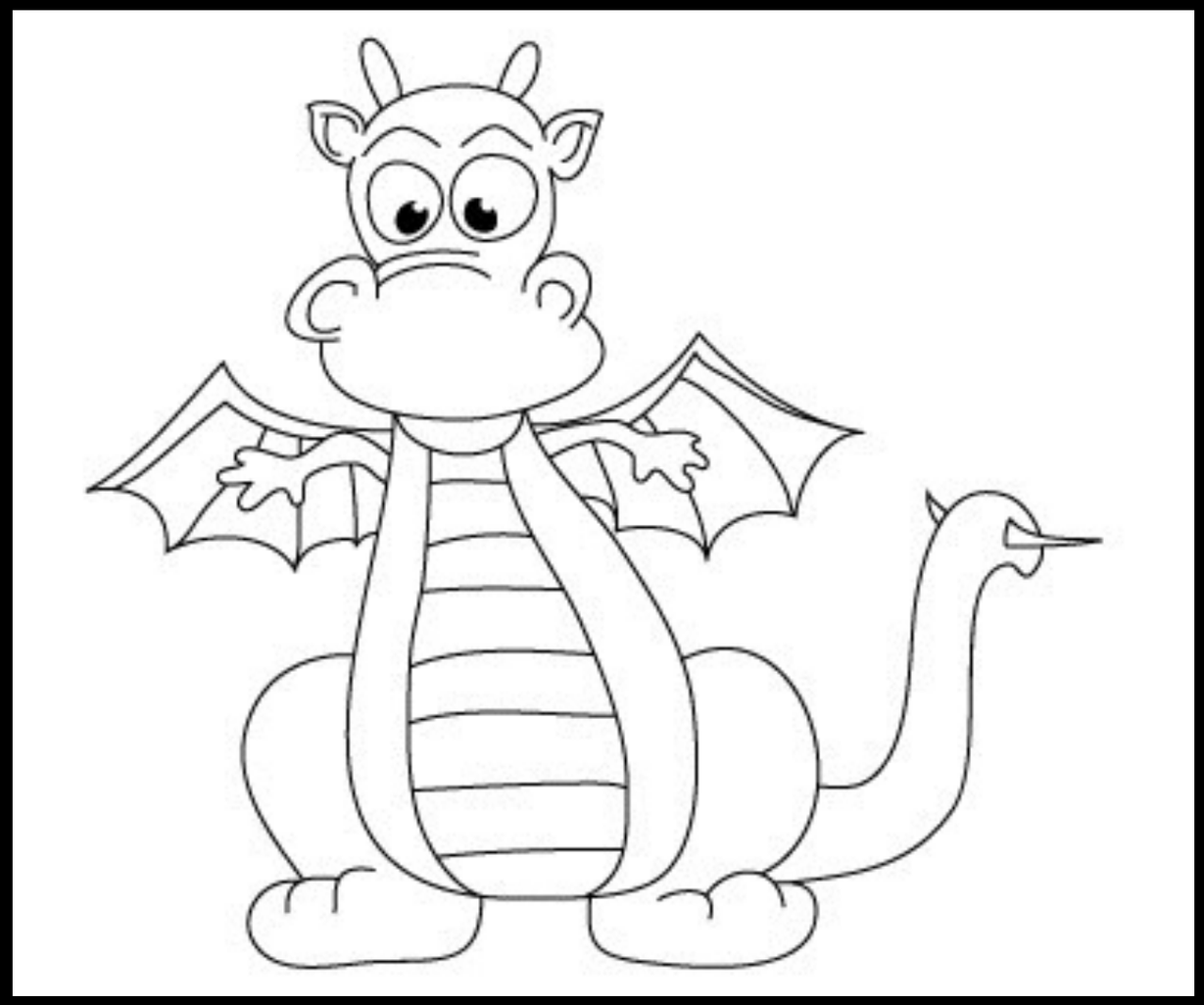
Here Be Dragons

CPython code!

C

Here Be Dragons

CPython code!



Automatic Memory Management

- Every Python object has a reference count
  - On creation this is set to 1
  - When it becomes 0 the object can be free'd

A PyObject

```
typedef struct _object {
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

Reference Counting

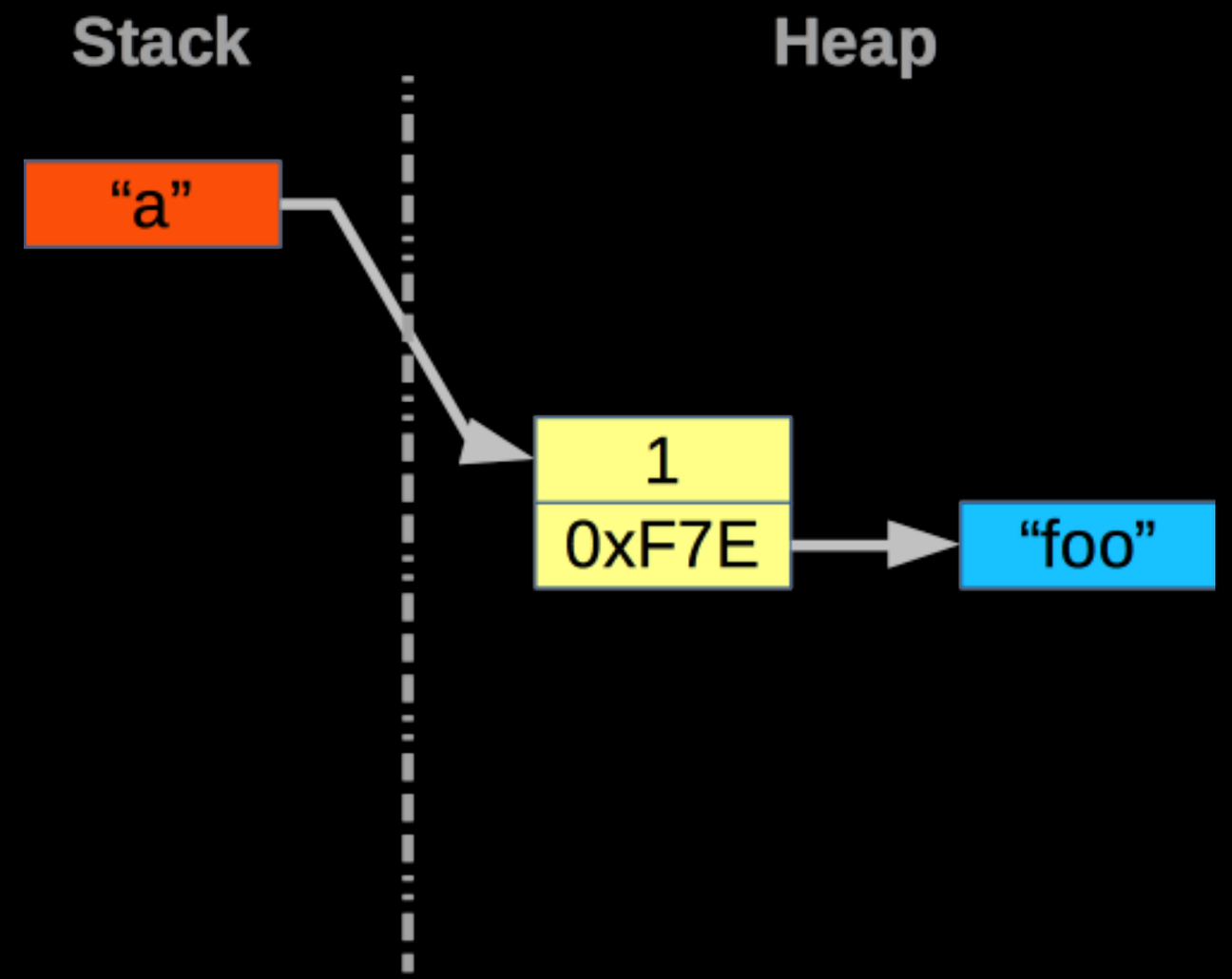
```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```

Reference Counting

```
>>> a = 'foo'
```

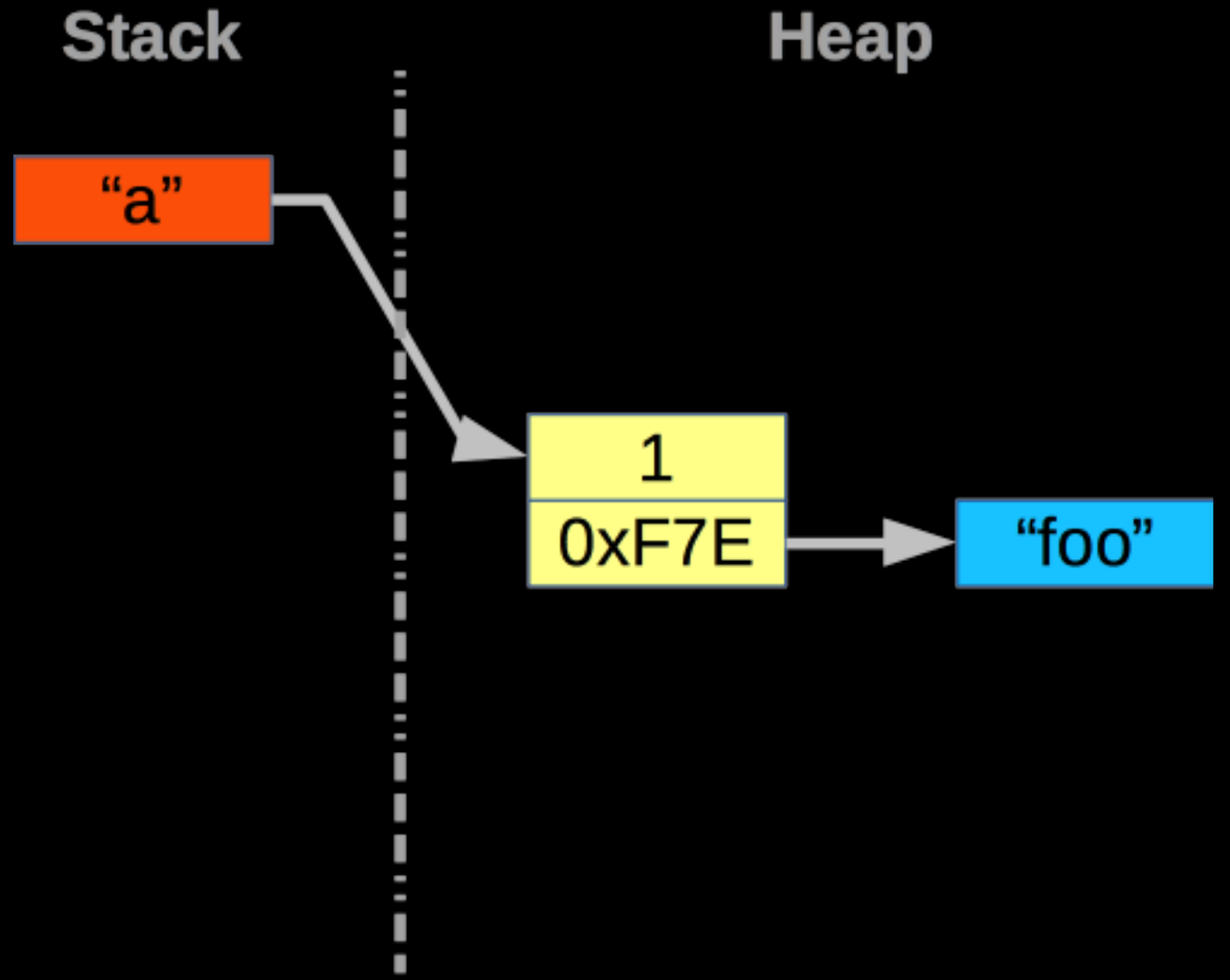
Reference Counting

```
>>> a = 'foo'
```



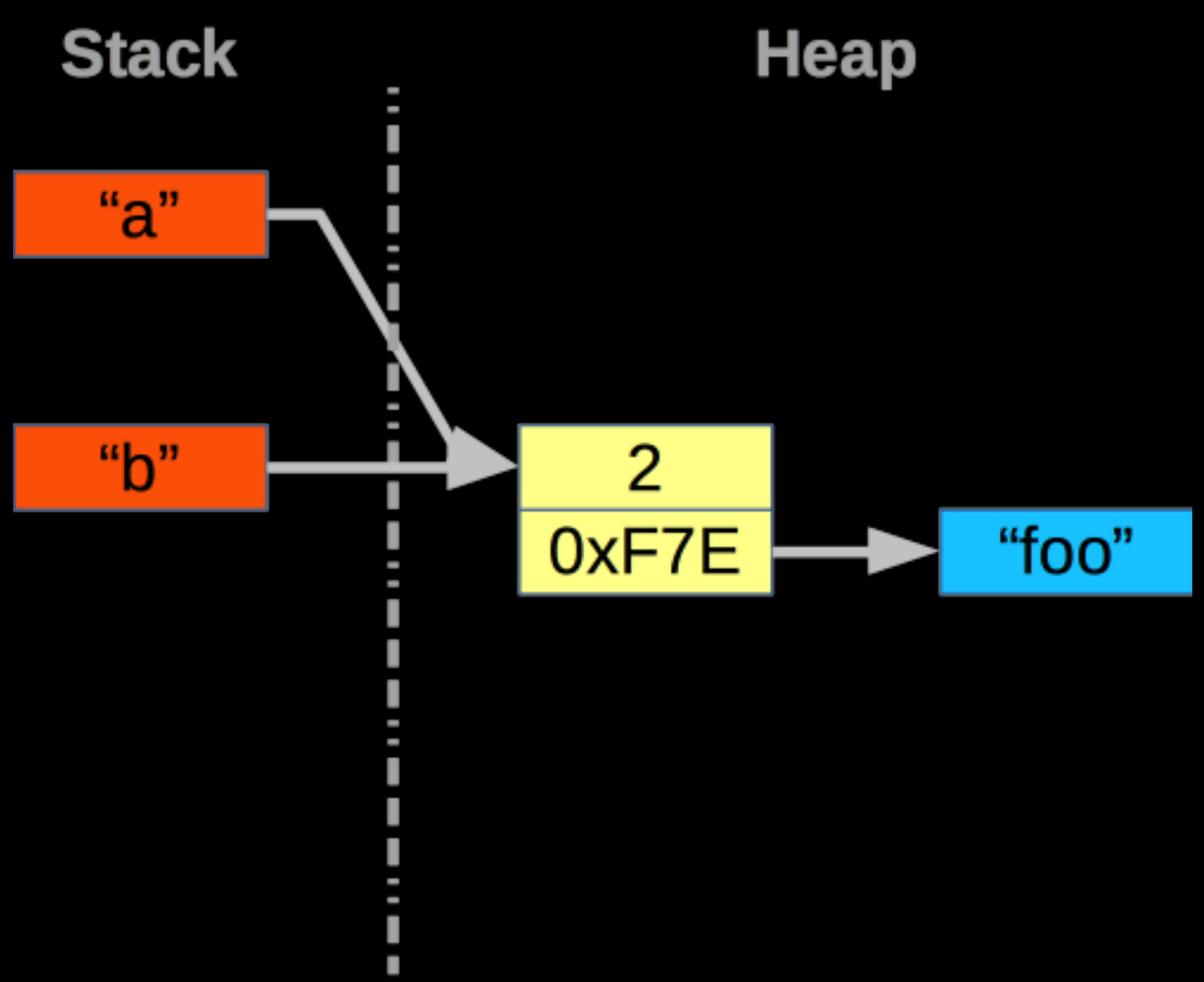
Reference Counting

```
>>> a = 'foo'
>>> b = a
```



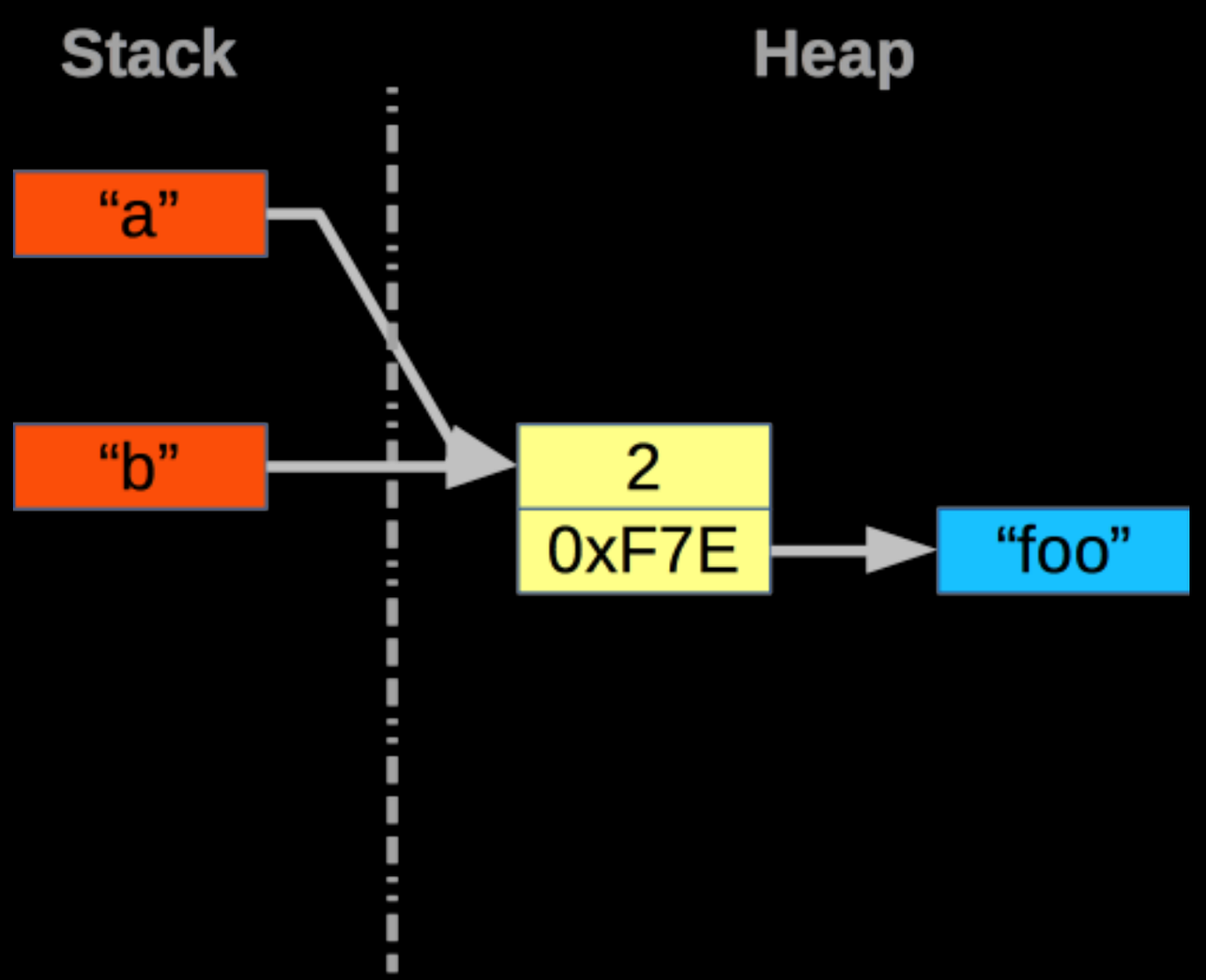
Reference Counting

```
>>> a = 'foo'
>>> b = a
```



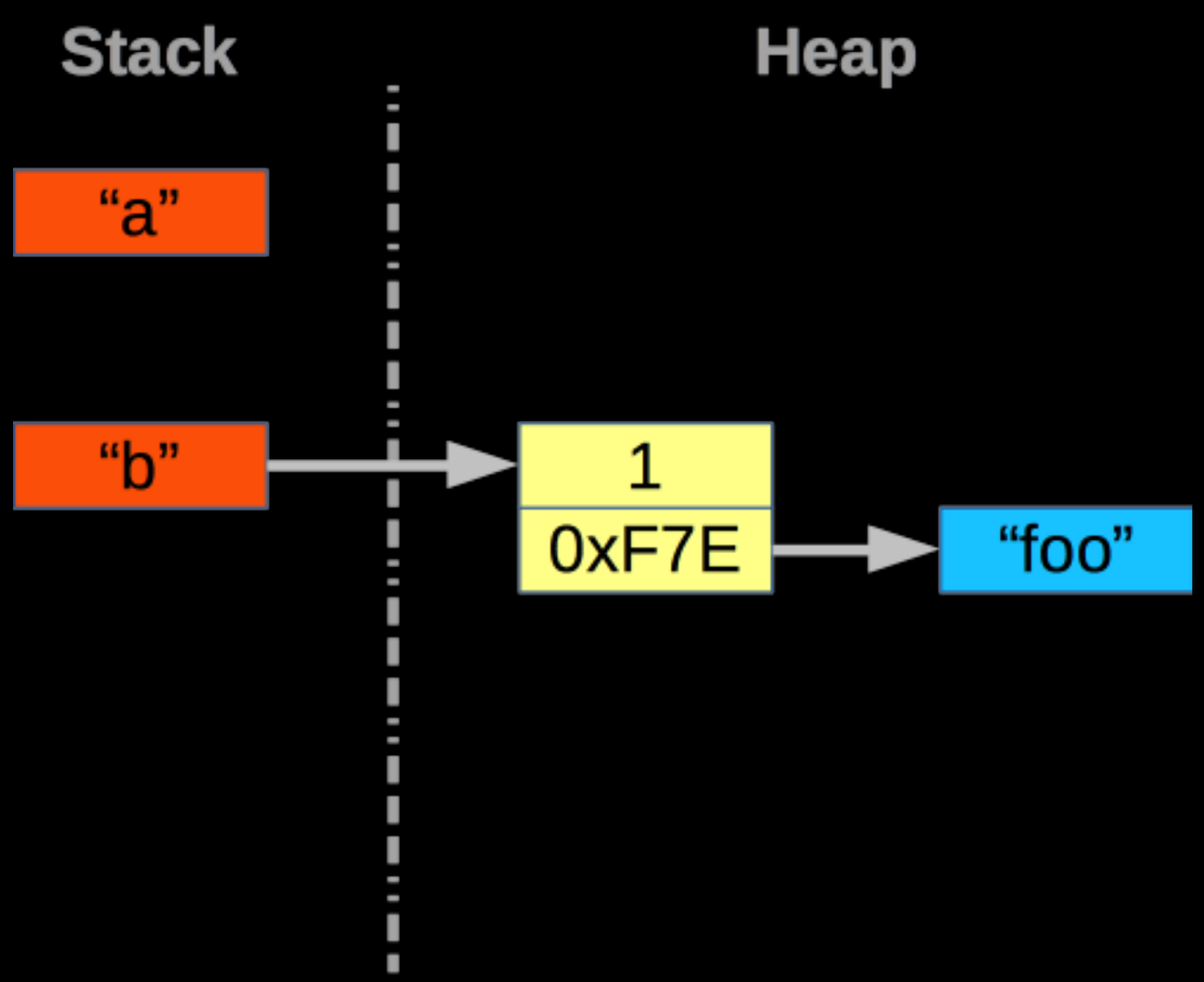
Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
```



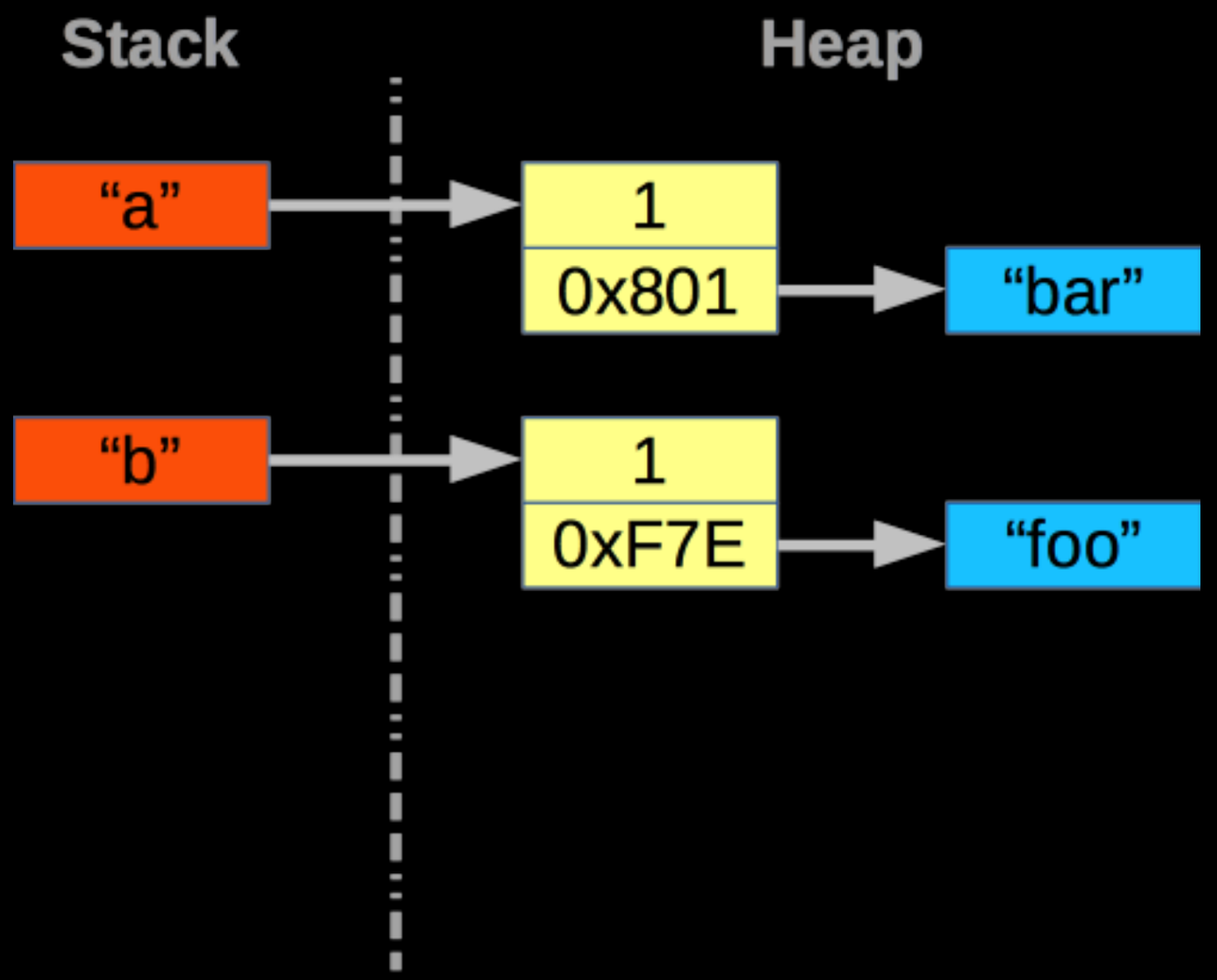
Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
```



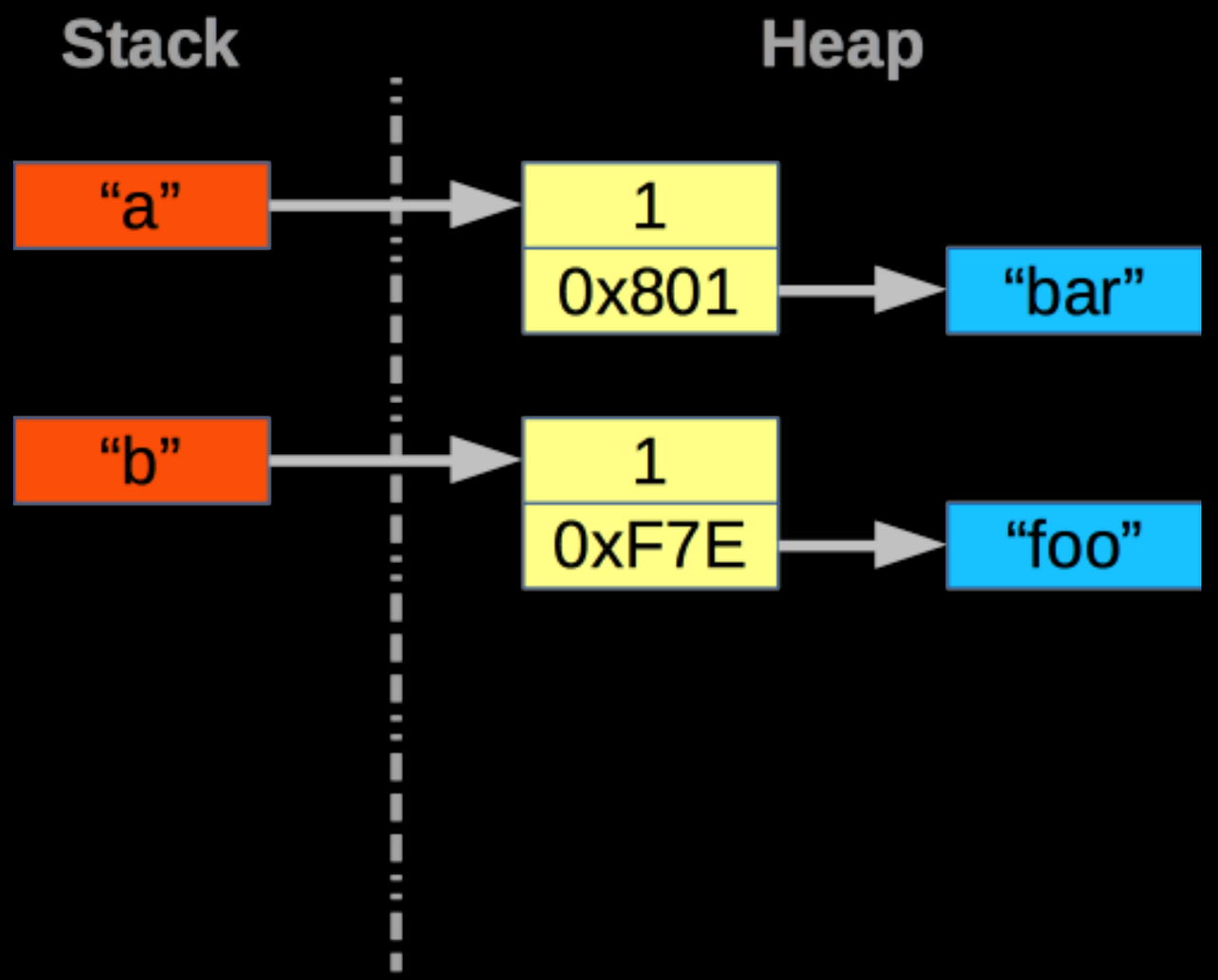
Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
```



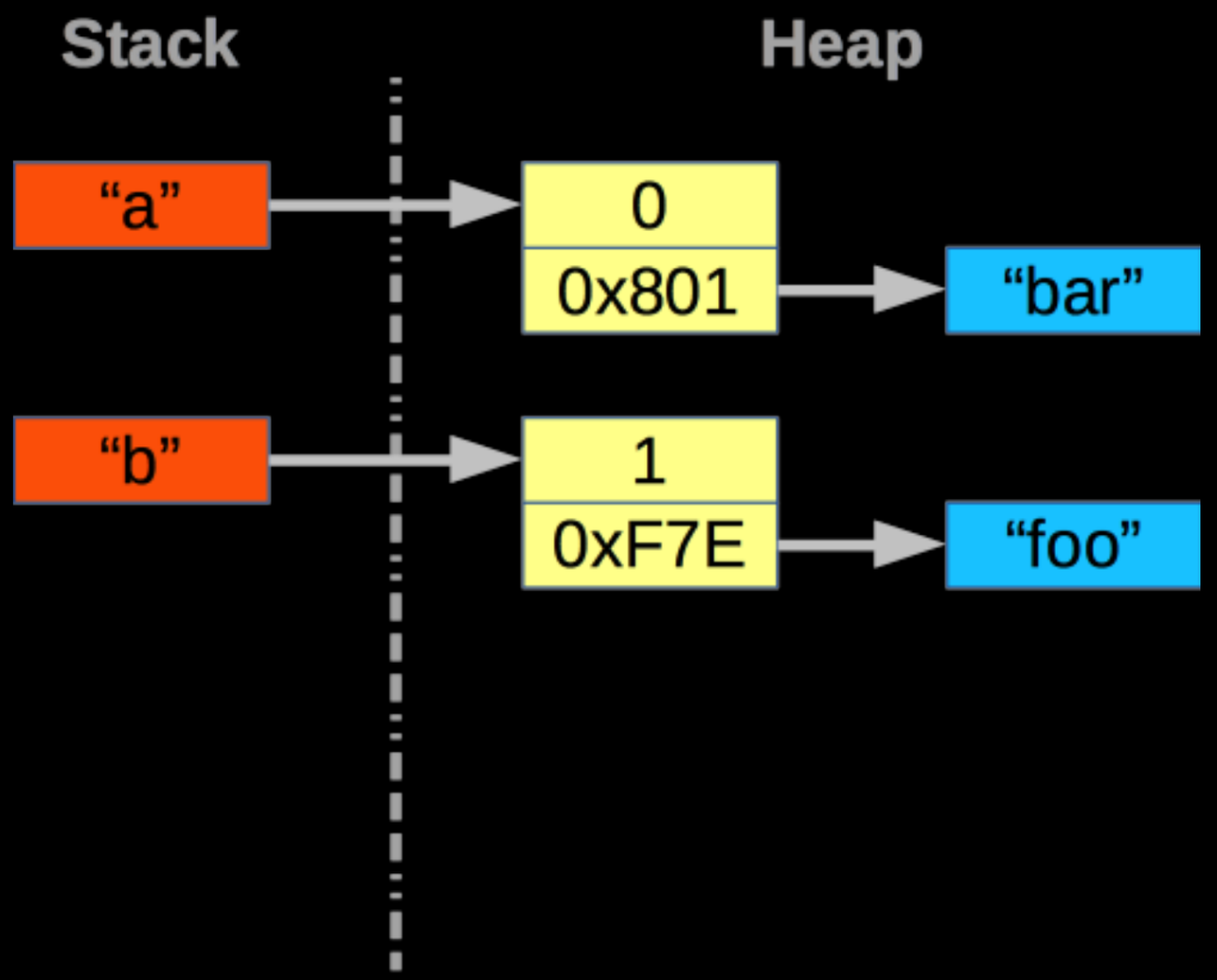
Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```



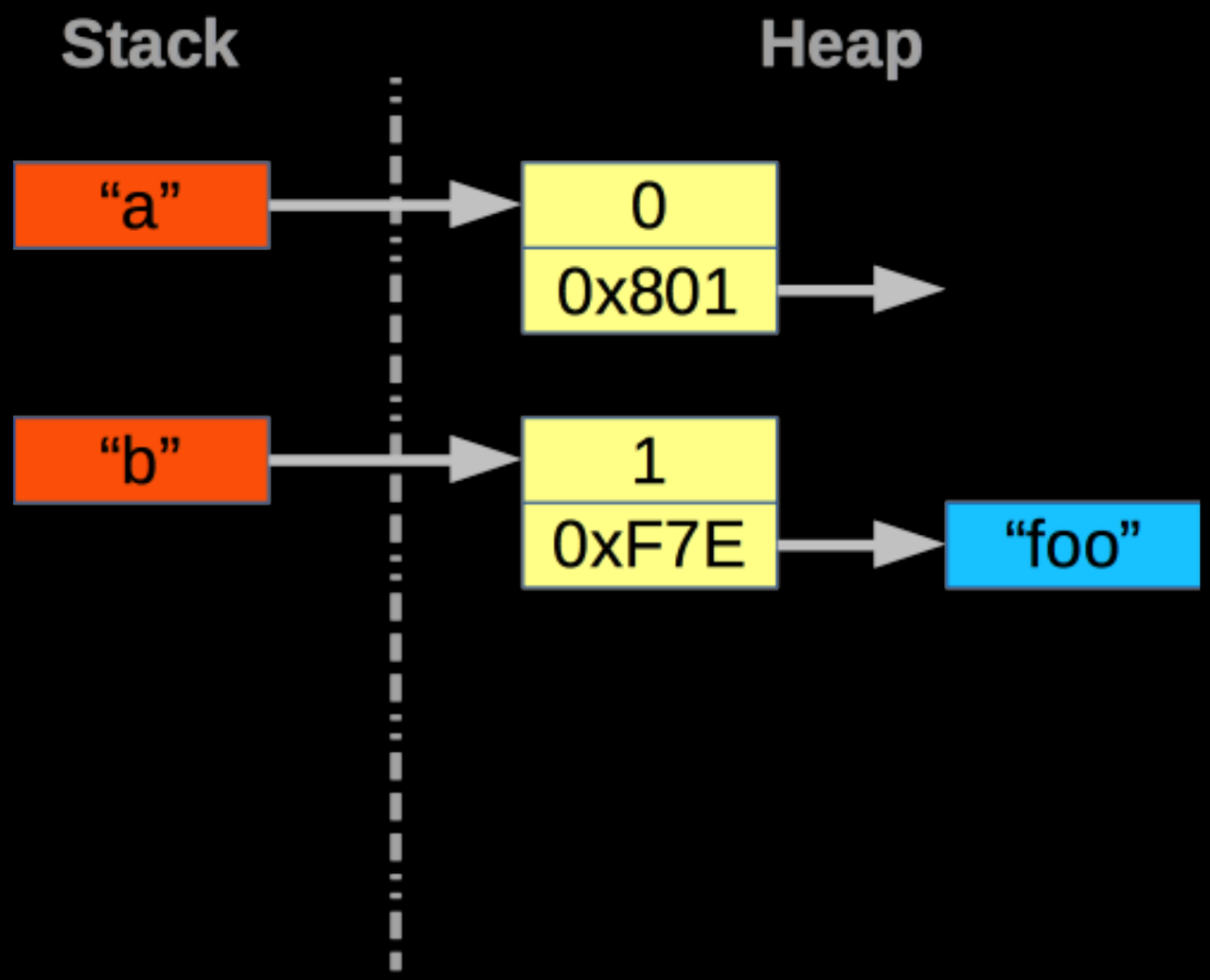
Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```



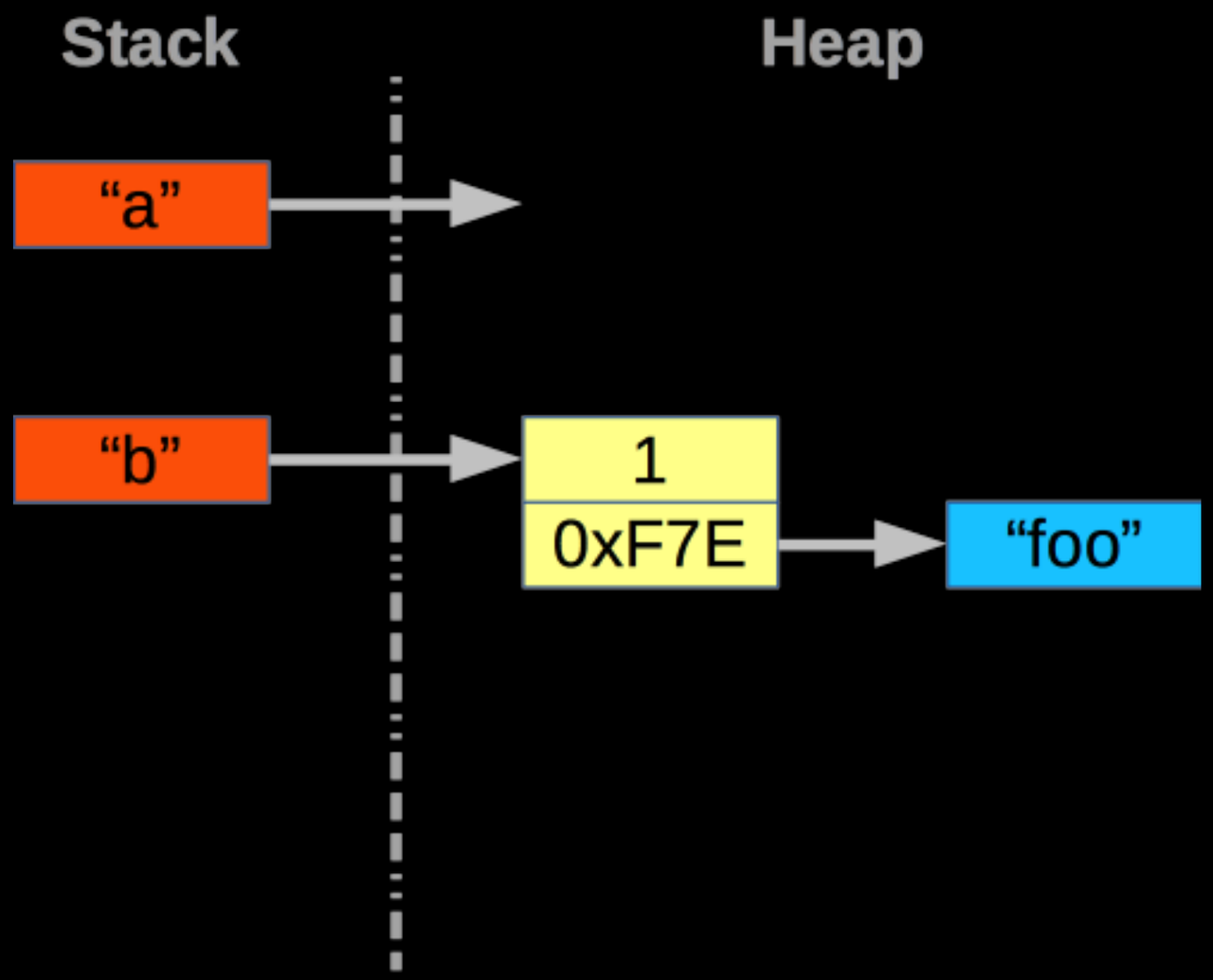
Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```



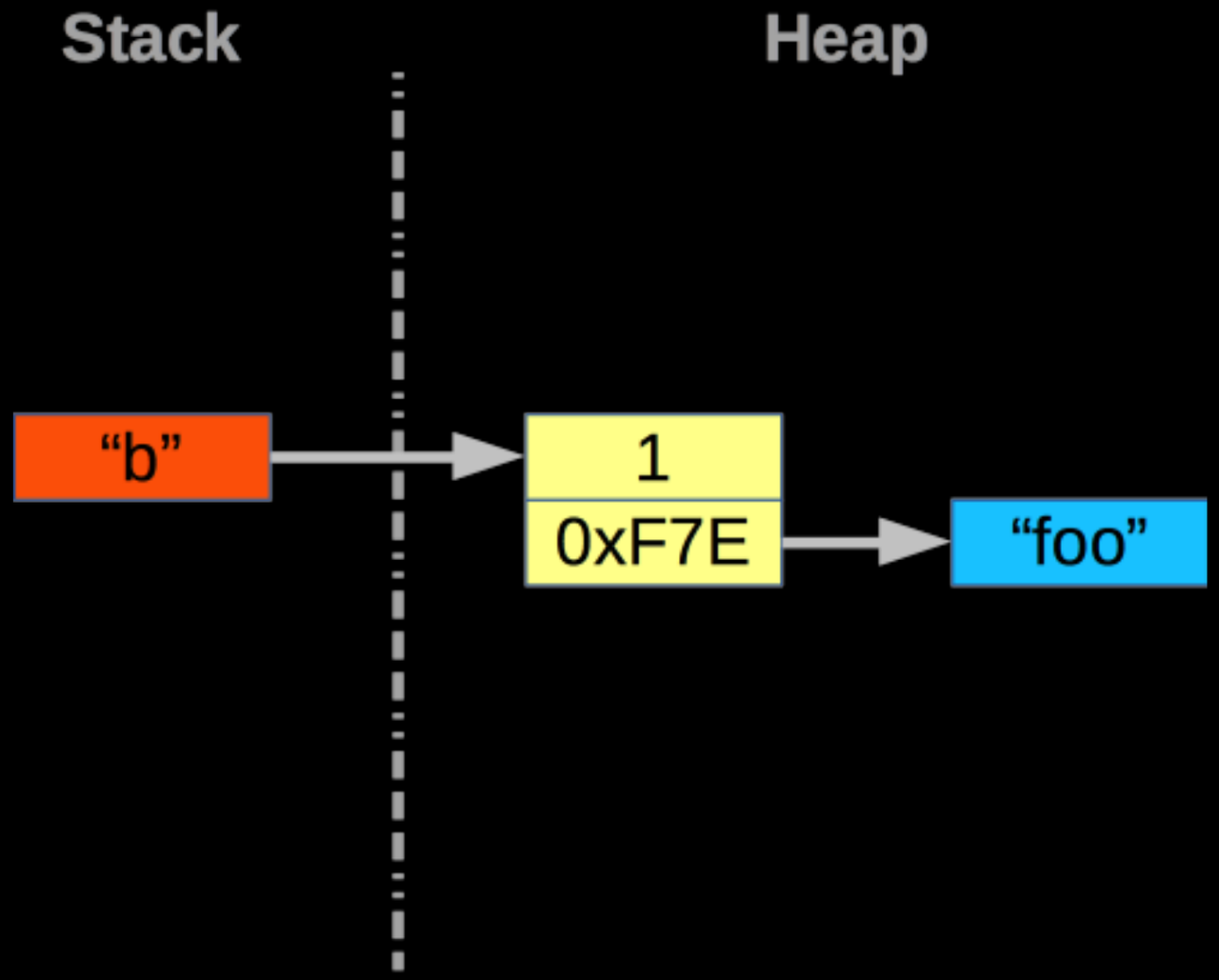
Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```



Reference Counting

```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```



Garbage Collection

- The GC is just there to resolve cyclic references
  - Only works with containers
- Its not a Unicorn
  - Will not reclaim lost C allocated memory
  - Will not reclaim lost PyObject references

Summary

1 coding pattern to keep the dragons at bay

2 things to avoid

3 kinds of `PyObject*` used in CPython

2 Things to Avoid

- Memory leaks
- Access after `free()`

C Memory Leaks

```
void leak() {
    char *p = malloc(1024);
}
```

C Access after `free()`

```
void access_after_free() {
    char *p = malloc(1024);

    p[8] = 'A';

    free(p);

    printf("%c", p[8]);
}
```

Py Memory Leaks

```
#include "Python.h"

void py_leak() {
    PyObject *pObj;
    pObj = PyBytes_FromString("Hello world\n");
    PyObject_Print(pObj, stdout, 0);


}
```

Py Memory Leaks - Fixed

```
#include "Python.h"

void py_leak() {
    PyObject *pObj;
    pObj = PyBytes_FromString("Hello world\n");
    PyObject_Print(pObj, stdout, 0);
    Py_DECREF(pObj);

}
```

Py Access after Free

```c
#include "Python.h"

void py_access_after_free() {
    PyObject *pObj;
    pObj = PyBytes_FromString("Hello world\n");
    PyObject_Print(pObj, stdout, 0);
    Py_DECREF(pObj);
    PyObject_Print(pObj, stdout, 0);
}
```

Py Access after Free
- Don't do this

```
Py_DECREF(pObj);

if (pObj->ob_refcnt > 0) {
    PyObject_Print(pObj, stdout, 0);
}
```

3 Reference Types

- **New** references occur when a `PyObject` is created
  - Example: creating a new list.
- **Stolen** references occur when a `PyObject` is created and assigned. Typically 'setters'
  - Example: setting the value in a list.
- **Borrowed** references are used when getting a `PyObject`
  - Example: accessing a member of a list.
  - If ***shared*** references mean more to you, great! Thats exactly what they are.

Programming by Contract

- **New** `PyObject*` Your job to free
  - Or give it to someone who will
- **Stolen** `PyObject*` Who ‘steals’ it will free it
  - Do not do so yourself
- **Borrowed** `PyObject*` The real owner can free it at any time
  - Unless you prevent them by registering your interest

New References

```
static PyObject *subtract_long(long a, long b) {
    PyObject *pA, *pB, *r;

    pA = PyLong_FromLong(a);          /* New reference pA */
    pB = PyLong_FromLong(b);          /* New reference pB */
    r = PyNumber_Subtract(pA, pB);  /* New reference r  */
    Py_DECREF(pA);                    /* I have to decref */
    Py_DECREF(pB);                    /* I have to decref */
    return r;                       /* Caller has to decref */
}
```

New References
- Don't do this

```
static PyObject *subtract_long(long a, long b) {
    return PyNumber_Subtract(PyLong_FromLong(a),
                             PyLong_FromLong(b));
}
```

Stolen References

```
static PyObject *make_tuple() {
    PyObject *r, *v;

    r = PyTuple_New(3);          /* New reference r */
    v = PyLong_FromLong(1024L); /* New reference v */
    PyTuple_SetItem(r, 0, v);

    v = PyLong_FromLong(2048L); /* New reference v */
    PyTuple_SetItem(r, 1, v);

    /* More common pattern */
    PyTuple_SetItem(r, 2, PyUnicode_FromString("three"));
    return r;    /* Callers responsibility to decref */
}
```

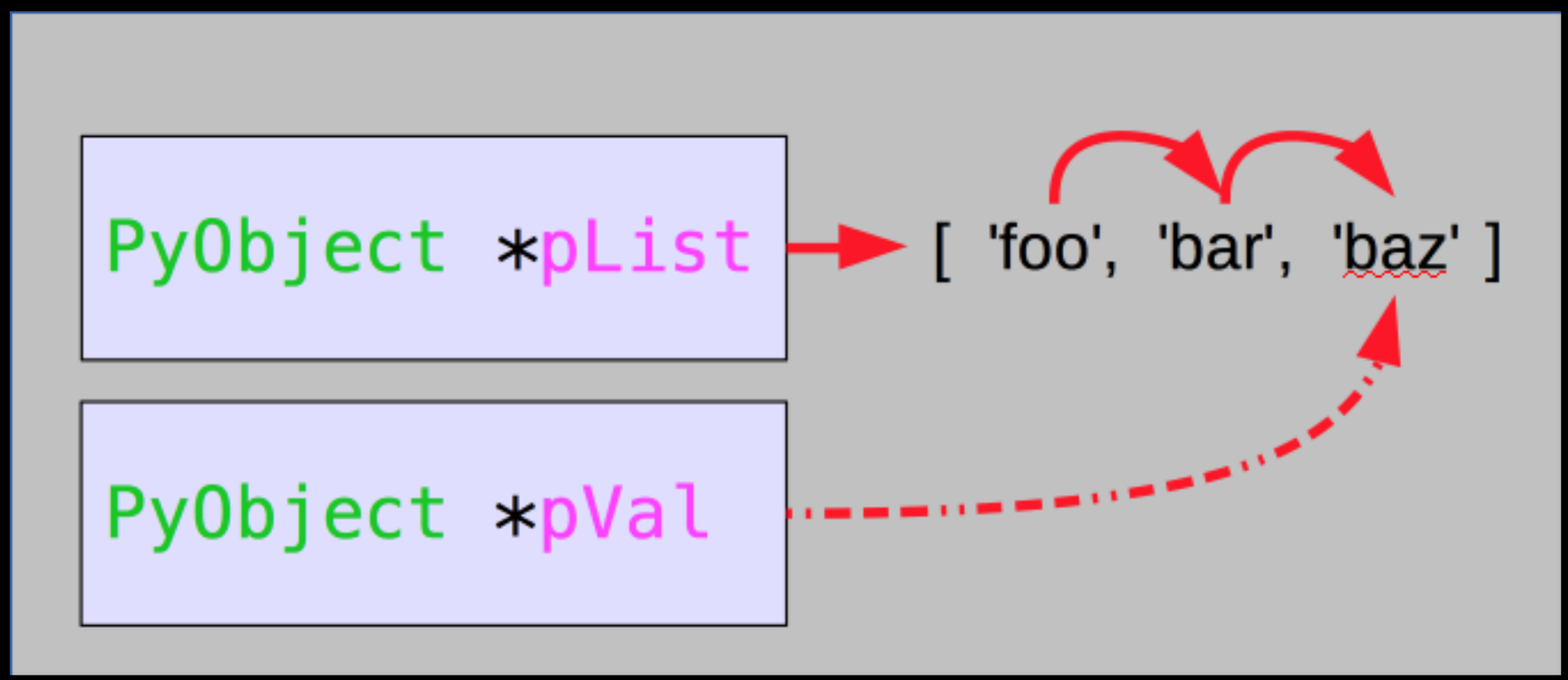
Stolen References
- Don't do this

```
PyObject *r, *v;

r = PyTuple_New(3);          /* New reference r */

v = PyLong_FromLong(1024L); /* New reference v */
PyTuple_SetItem(r, 0, v);   /* r 'steals' the reference */

Py_DECREF(v);
```

'Borrowed' References

- These are generally 'getters'

```
PyObject *pList = ...
PyObject *pVal  = PyList_GetItem(pList, 2);
```



'Borrowed' References

- Multiple pointers to the same object - Aaargh!
  - Which is responsible for freeing the object?
  - What happens to the other pointers when one frees the object?
- They can be the source of the most subtle bugs

'Borrowed' References

```
static PyObject *borrow_BAD(PyObject *pList) {
    PyObject *pLast;

    pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);

    function(pList);    /* Dragons ahoy me hearties! */

    PyObject_Print(pLast, stdout, 0);

    Py_RETURN_NONE;
}
```

Hmm… Suppose

```
static PyObject *borrow_BAD(PyObject *pList) {
    PyObject *pLast;

    pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);

    function(pList);    /* Dragons ahoy me hearties! */

    PyObject_Print(pLast, stdout, 0);

    Py_RETURN_NONE;
}
```

Hmm… Suppose

```
static PyObject *borrow_BAD(PyObject *pList) {
    PyObject *pLast;

    pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);

    function(pList);    /* Dragons ahoy me hearties! */

    PyObject_Print(pLast, stdout, 0);

    Py_RETURN_NONE;
}
```

This deleted the last item in the list

Borrowed Ref Dragon 0

```
>>> import cPyRefs
>>> l = ['foo', 'bar', 'baz']
>>> cPyRefs.borrow_bad(l) # SEGFAULT!
```

Borrowed Ref Dragon 1

```
>>> import cPyRefs
>>> l = ['foo', 'bar', 'baz']
>>> cPyRefs.borrow_bad(l) # SEGFAULT!
```

```
>>> import cPyRefs
>>> l = ['foo', 'bar', 'baz']
>>> a = l[-1]
>>> cPyRefs.borrow_bad(l) # Works fine!
```

Borrowed Ref Dragon 2

```
>>> import cPyRefs
>>> l = ['foo', 'bar', 'baz']
>>> cPyRefs.borrow_bad(l) # SEGFAULT!
```

Borrowed Ref Dragon 2

```
>>> import cPyRefs
>>> l = ['foo', 'bar', 'baz']
>>> cPyRefs.borrow_bad(l) # SEGFAULT!
```

```
>>> import cPyRefs
>>> l = [1, 2, 3]
>>> cPyRefs.borrow_bad(l) # Works fine!
```

Borrowed Ref Dragon 2

```
>>> import cPyRefs
>>> l = ['foo', 'bar', 'baz']
>>> cPyRefs.borrow_bad(l) # SEGFAULT!
```

```
>>> import cPyRefs
>>> l = [1, 2, 3]
>>> cPyRefs.borrow_bad(l) # Works fine

>>> import cPyRefs
>>> l = [800, 801, 802]
>>> cPyRefs.borrow_bad(l) # Kaboom!
```

Run-time Errors
+ Data Dependent Errors

Run-time Errors
+ Data Dependent Errors

The Problem

```
static PyObject *borrow_BAD(PyObject *pList) {
    PyObject *pLast;

    pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);
    function(pList);    /* Dragons ahoy me hearties! */

    PyObject_Print(pLast, stdout, 0);


    Py_RETURN_NONE;
}
```

The Fix

```
static PyObject *borrow_BAD(PyObject *pList) {
    PyObject *pLast;

    pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);
>   Py_INCREF(pLast);   /* Protect myself */

    function(pList);    /* Dragons tamed! */


    PyObject_Print(pLast, stdout, 0);

>   Py_DECREF(pLast);   /* I'm done */
>   pLast = NULL;

    Py_RETURN_NONE;
}
```

1 Pattern For Reliable C

- Borrowed references incref'd and decref'd correctly.
- A single place for clean up code
  - No early returns
- Exception consistency. Either:
  - An exception is set **and** NULL is returned.
  - Or: no Exception set **and** non-NULL returned.

Writing Pythonic Python

```
def simple(obj):
    ret = None;
    try:
        # Do fabulous stuff here
        # On error raise an exception
    except ... as err:
        # Handle exceptions
    finally:
        # And we are out
    return ret;
```

Writing Pythonic C

Writing Pythonic C

```
static PyObject *simple(PyObject *arg1) {
    PyObject *ret = NULL;

    goto try;
try:
    /* Do fabulous stuff here */
    /* On error “goto except;” */
    goto finally;
except:
    /* Handle exceptions */
finally:
    /* And we are out. */
    return ret;
}
```

Function Entry

```
static PyObject *func(PyObject *arg1) {
    /* Create any local PyObject* as NULL */
    PyObject *obj_a    = NULL;
    /* Create the PyObject* return value as NULL */
    PyObject *ret      = NULL;

    goto try; /* Pythonic 'C' ;-) */
try:
```

try:

```
try:
    assert(! PyErr_Occurred());
    /* Inc the reference count of the arguments. */
    assert(arg1);
    Py_INCREF(arg1);

    /* Your code here */

    /* Local object creation; borrowed or new. */
    obj_a = ...;
    /* If an error ... */
    if (! obj_a) {
        PyErr_SetString(PyExc_ValueError, "Ooops.");
        goto except;
    }
```

try:

```
/* Return object creation, ret will either be a
 * new reference or a borrowed reference
 * INCREF'd */
ret = ...;
if (! ret) {
        PyErr_SetString(PyExc_ValueError,
                        "Ooops again.");
        goto except;
    }
    /* If success then check exception is clear,
     * goto finally; with non-NULL return value. */
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
```

except:

```
except:
    /* Failure so Py_XDECREF the return value */
    Py_XDECREF(ret);
    /* Check a Python error set somewhere above */
    assert(PyErr_Occurred());
    /* Signal failure */
    ret = NULL;
    /* Fall through to finally: */
finally:
```

finally:

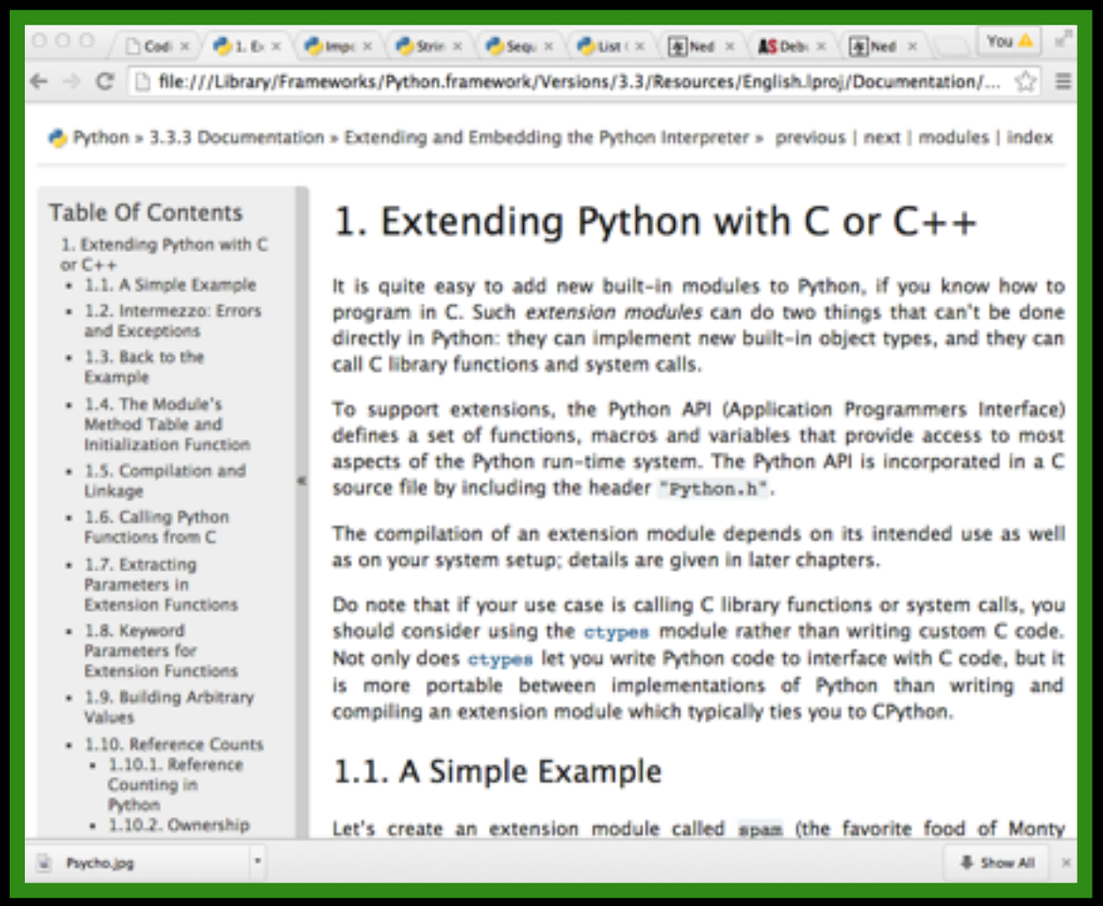
```
finally:
    /* All _local_ PyObjects are Py_XDECREF'd here.
     * For new references this will free them.
     * For borrowed references this
     * will return them to their previous state. */
    Py_XDECREF(obj_a);
    /* Decrement the ref count of given arguments
     * if they have been incremented. */
    Py_DECREF(arg1);
    /* And return... */
    return ret;
}
```

All this and more…

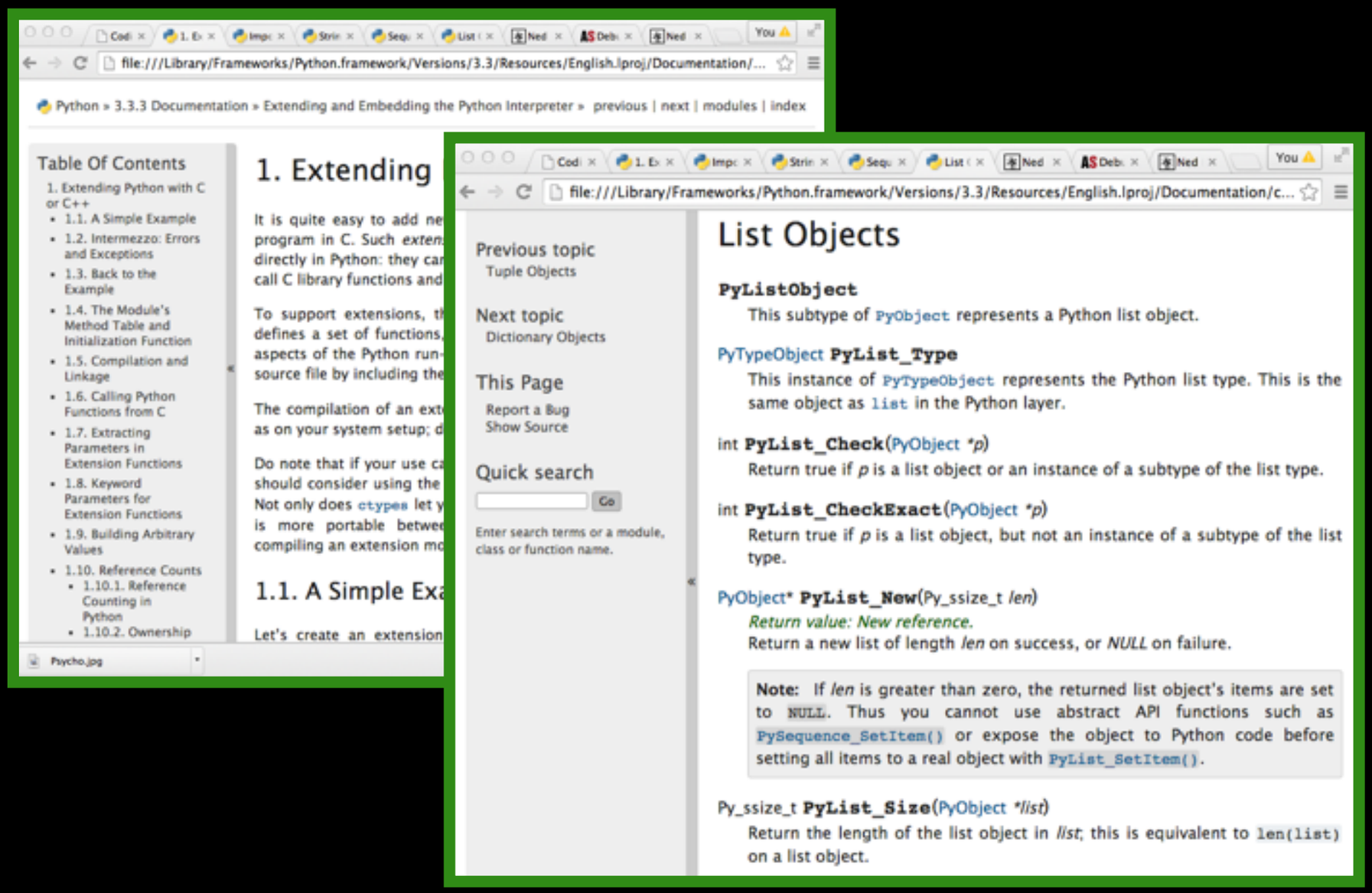
https://github.com/paulross

In “PythonExtensionPatterns”

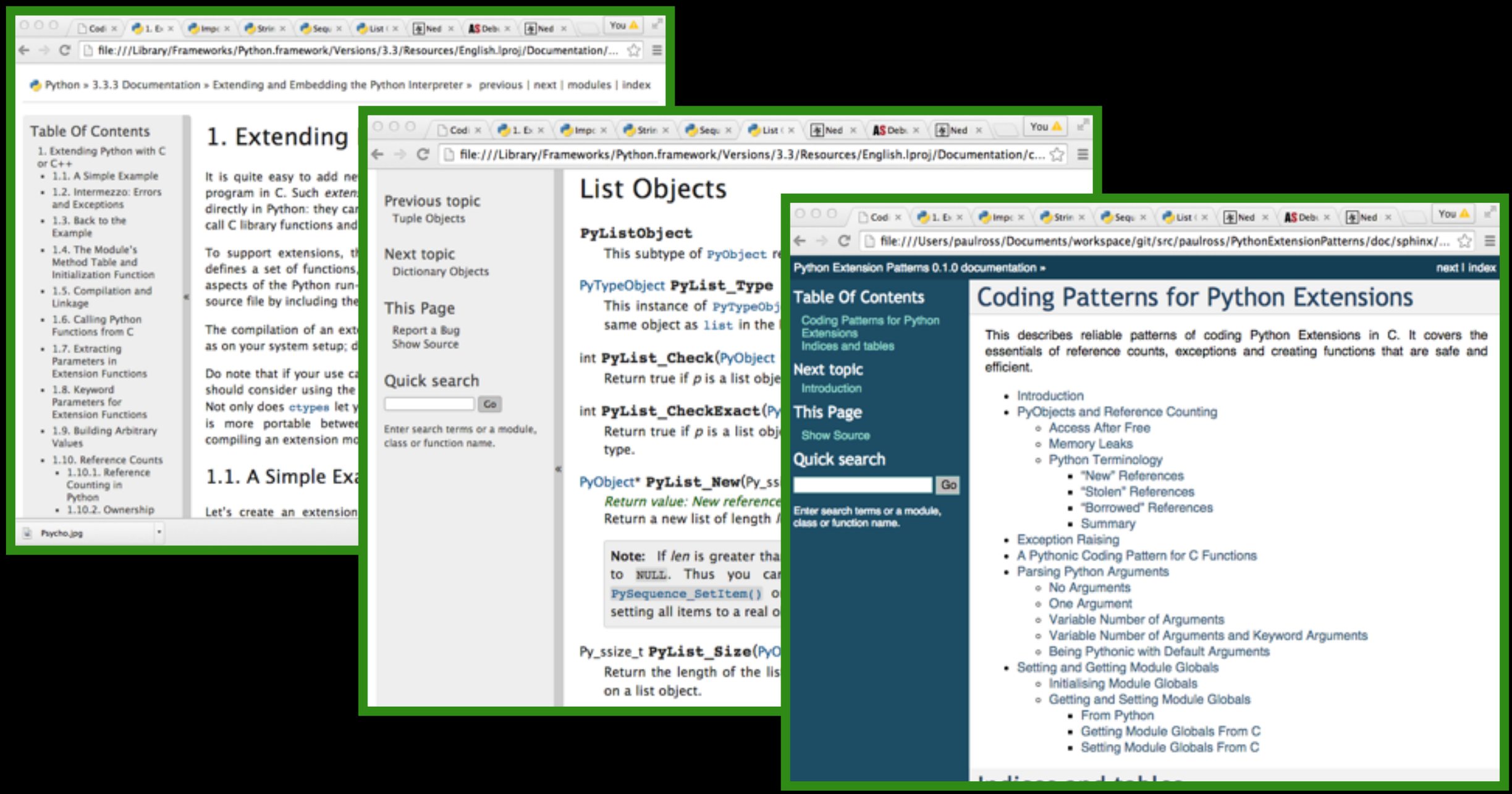
The Documentation is Excellent - Use it!



The Documentation is Excellent - Use it!



The Documentation is Excellent - Use it!



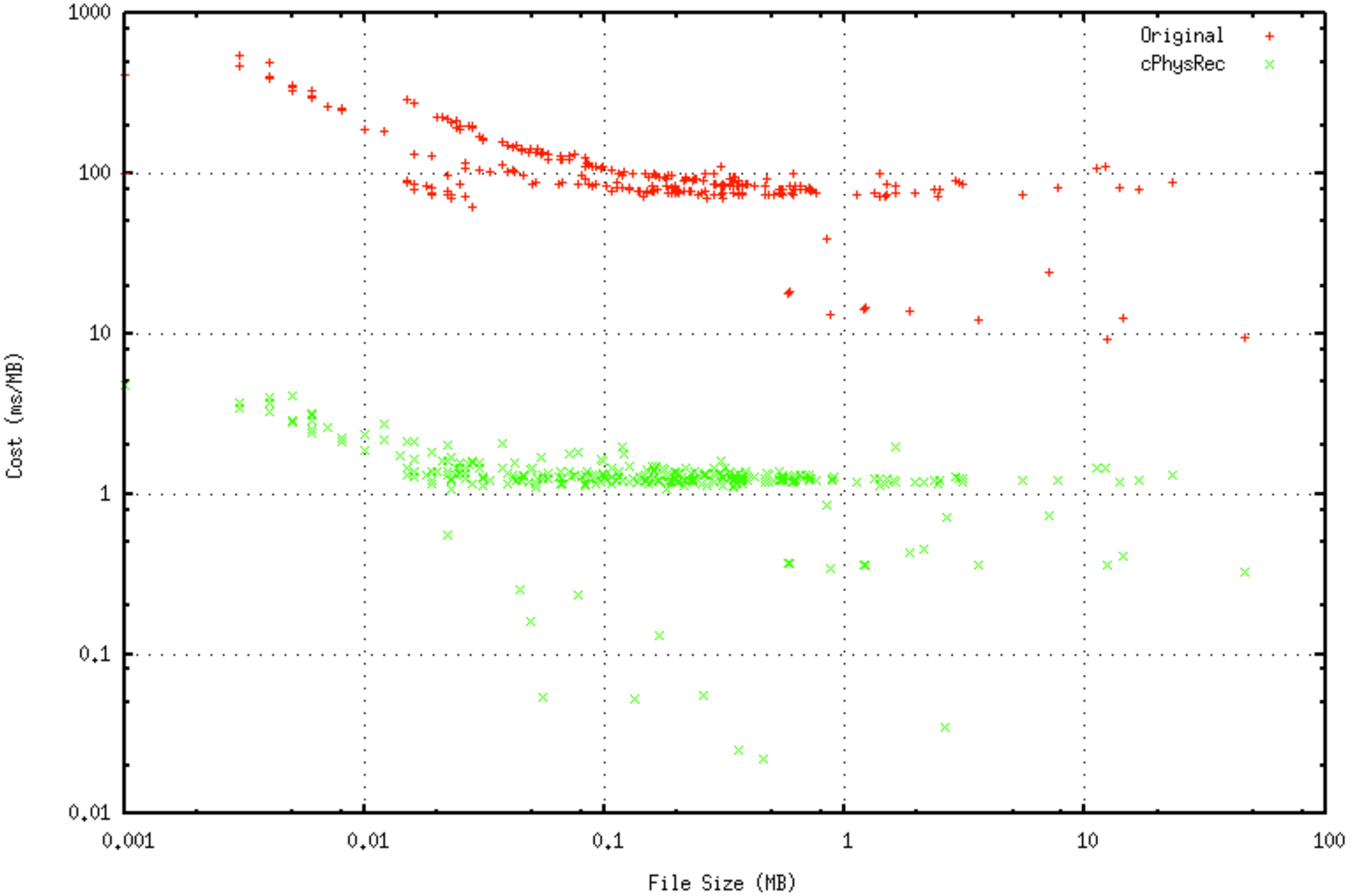
https://github.com/paulross

War Story ~ Mandatory

- Proprietary binary files of oilfield data
  - Self describing, variable format, sequentially written
- Make them random access by creating an index
  - The index is built with a sequence of `seek()/read()` operations
  - `read()` is about 1% to 2% of the original file size
- Originally written in Python. Typ. 10-100ms/Mb
- How fast can we go?

Gnuplot

Improvement in indexing cost using C extension cPhysRec.



0.000272545, 0.00274264

Summary

1 coding pattern to keep the dragons at bay

2 things to avoid

- `malloc()` with no `free()`
- Access after `free()`

3 kinds of references to `PyObject*`

- **New**: its yours
- **Stolen**: its theirs
- **Borrowed**: you are sharing something that is really theirs - let them know!

That's It

https://github.com/paulross/PythonExtensionPatterns