Here Be Dragons

Writing Reliable Python Extensions in C

Paul Ross, Lead Technologist, AHL

PyCon 2016 Portland, Oregon

Man AHL

https://twitter.com/manahltech

- London based systematic hedge fund since 1987
- $19.2bn Funds Under Management (2016-03-31)
- We are active in 400+ markets in 40+ countries
- We take ~2bn market data points each day
    - https://github.com/manahl/arctic
- 125 people, 22 first languages. And Python!

Why Write in C?

- Blinding performance
- Interface with C/C++ libraries
- Leaner resources
- Flee the GIL

Here Be Dragons

Here Be Dragons

Here Be Dragons

C

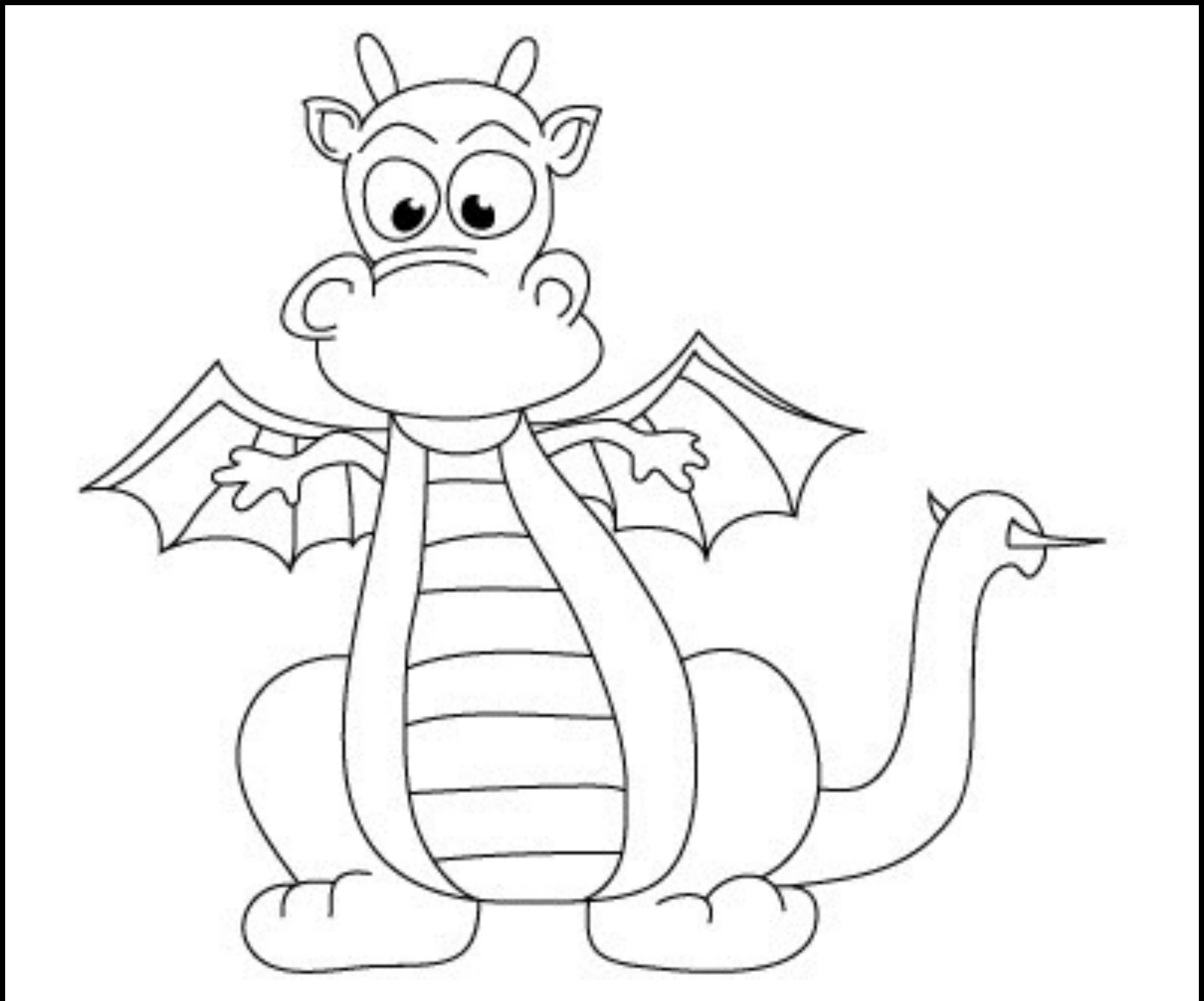
Here Be Dragons

CPython code!

C

Here Be Dragons

CPython code!



C

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 deallocated

Reference Counting

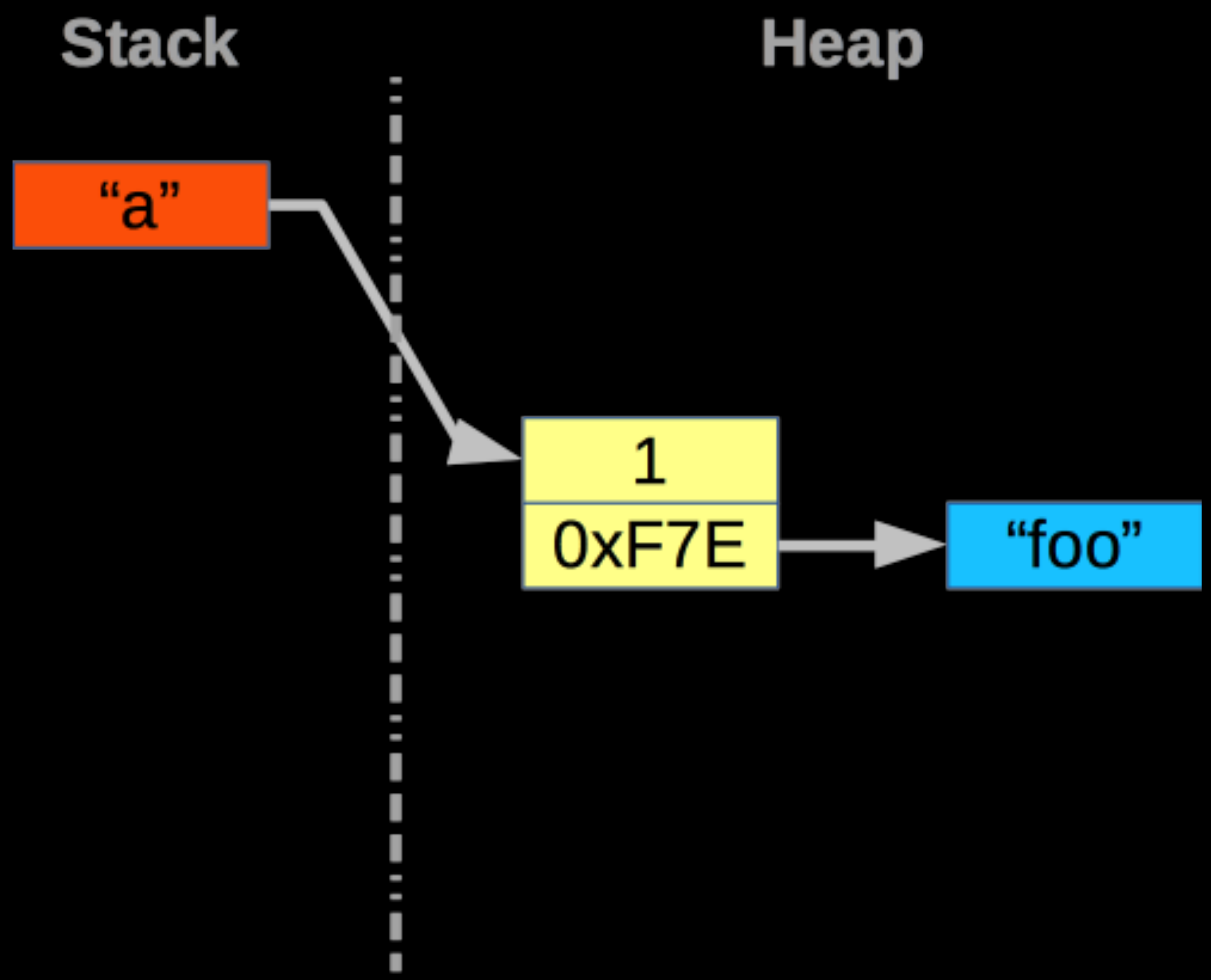
```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```

Reference Counting

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

Reference Counting

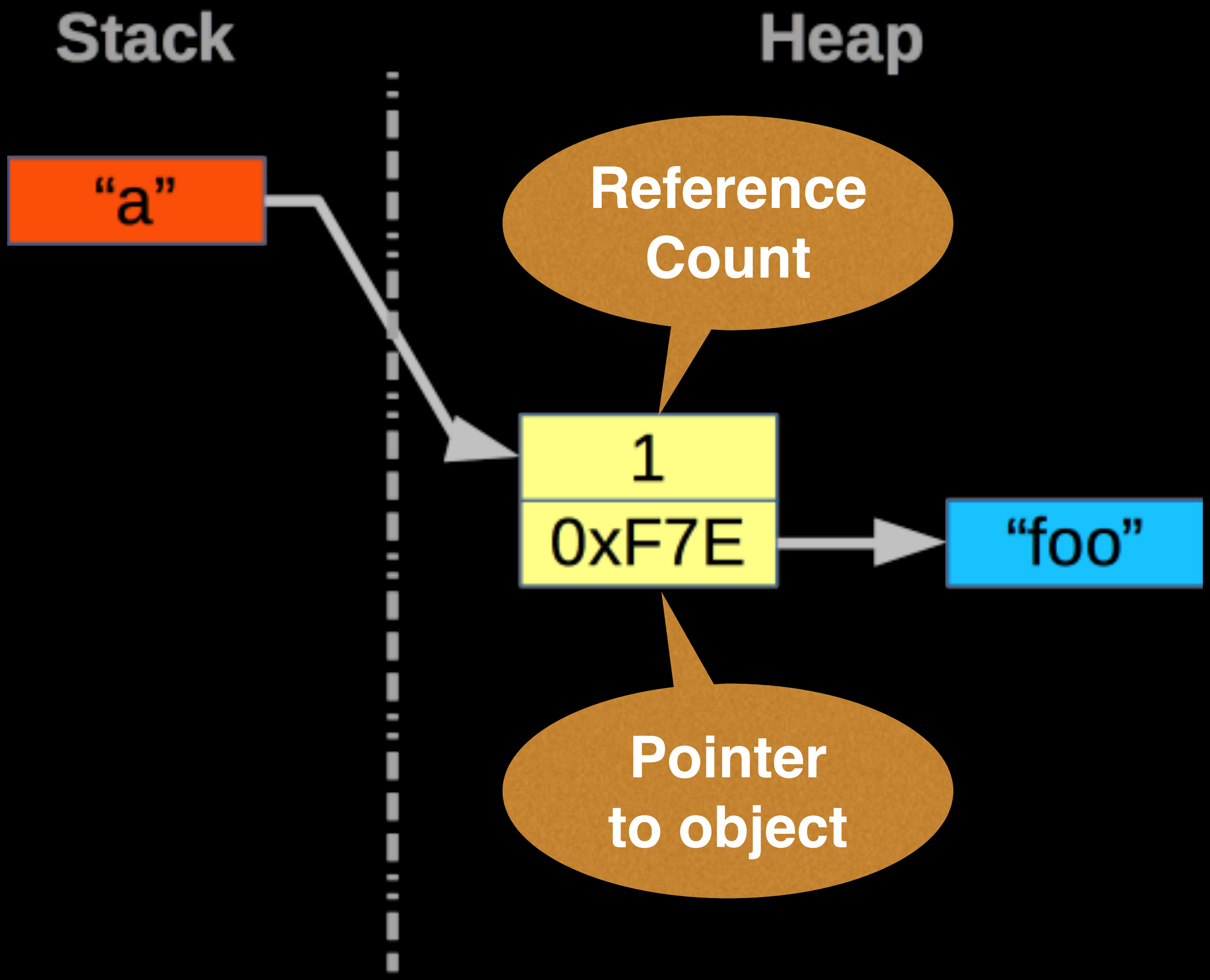
```
>>> a = 'foo'
```



© Paul Ross 2016

Reference Counting

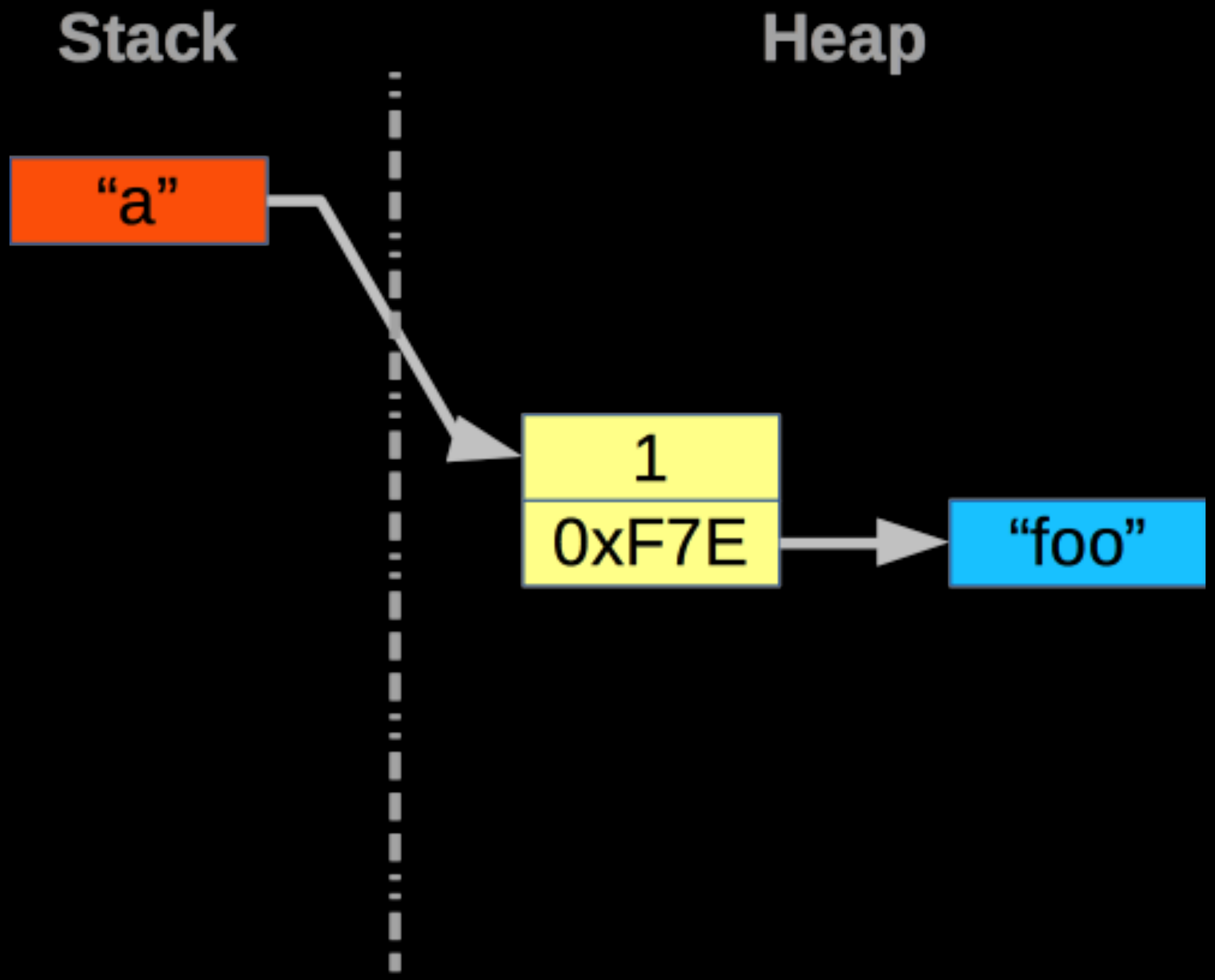
```
>>> a = 'foo'
```



© Paul Ross 2016

Reference Counting

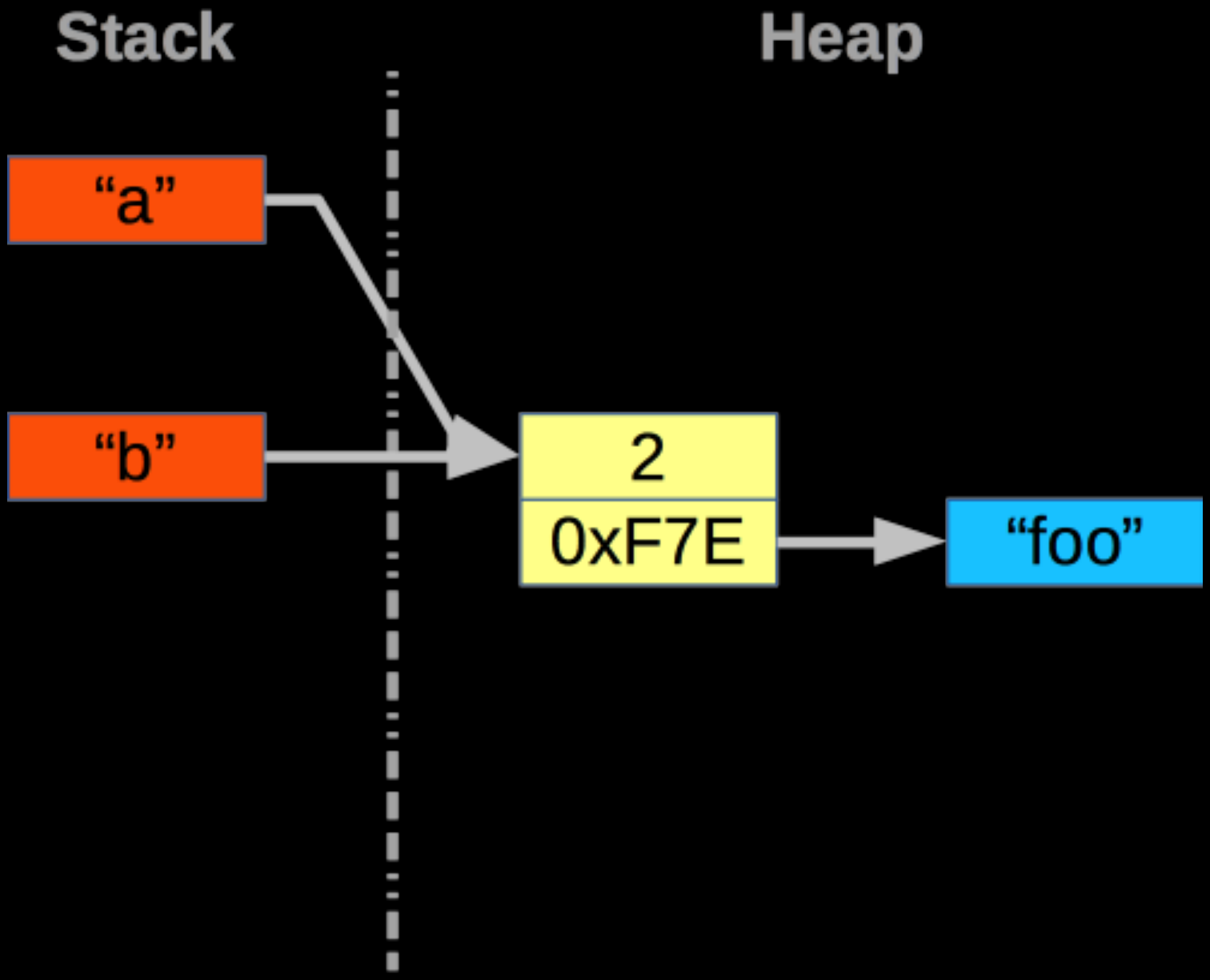
```
>>> a = 'foo'
>>> b = a
```



© Paul Ross 2016

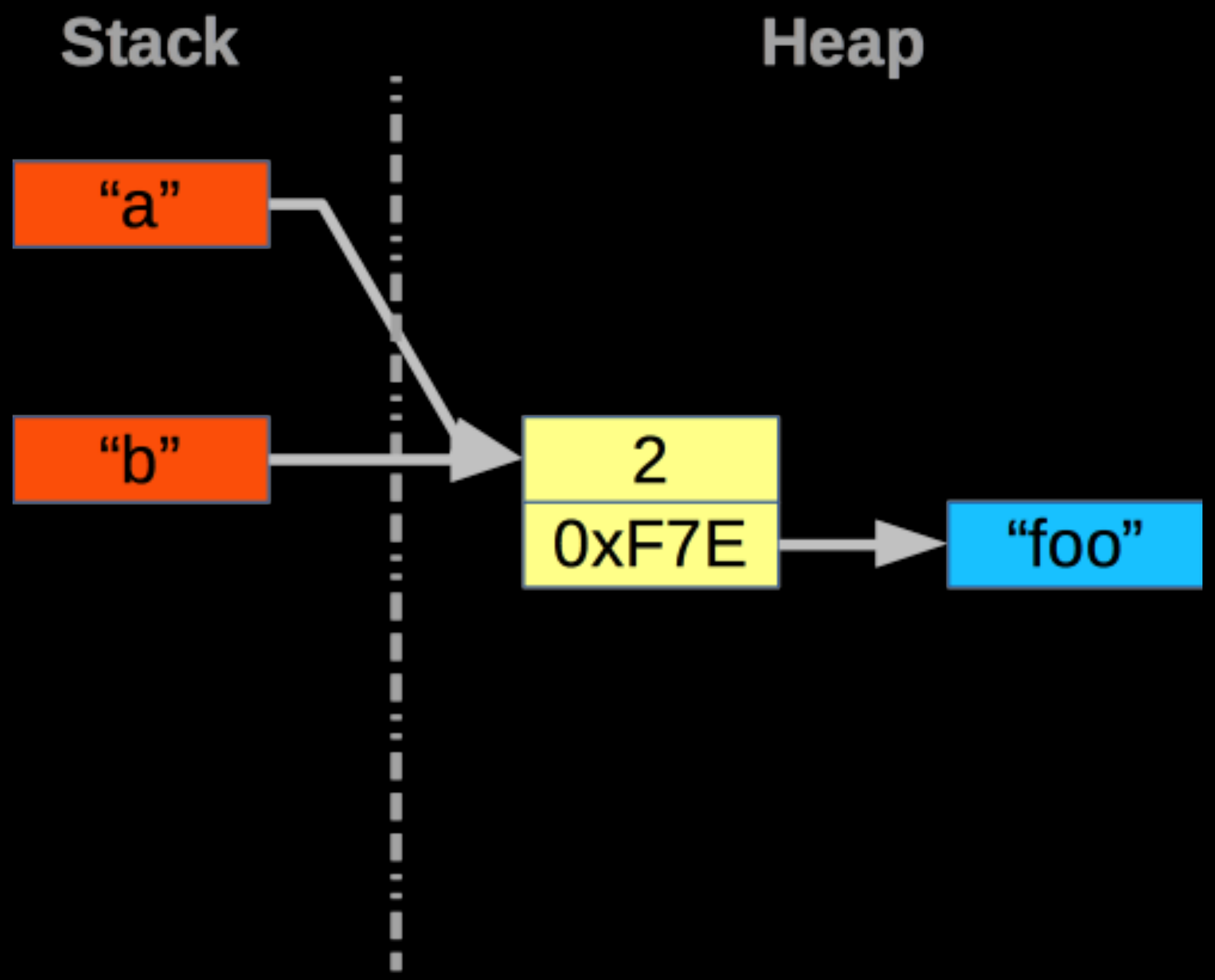
Reference Counting

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



Reference Counting

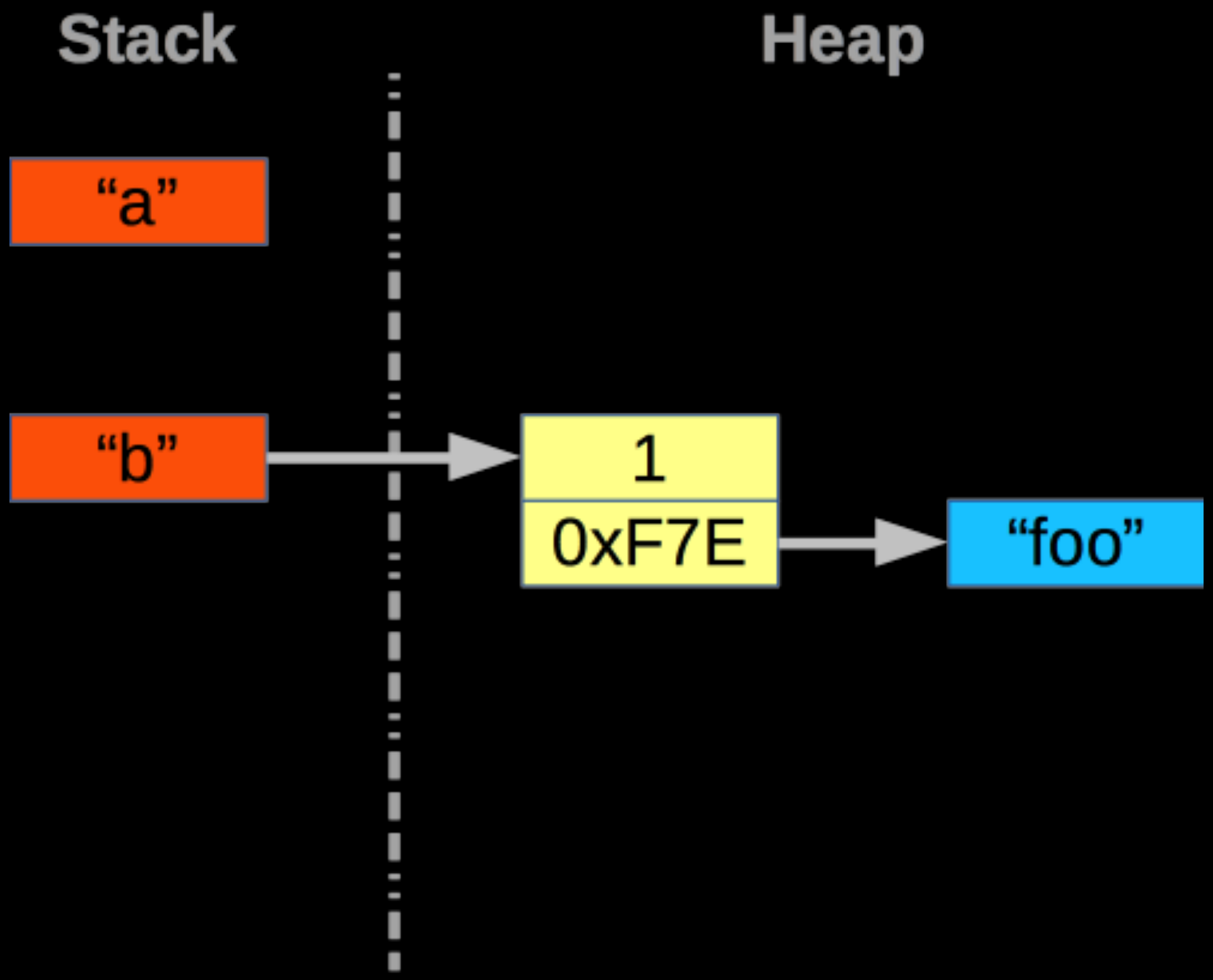
```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
```



© Paul Ross 2016

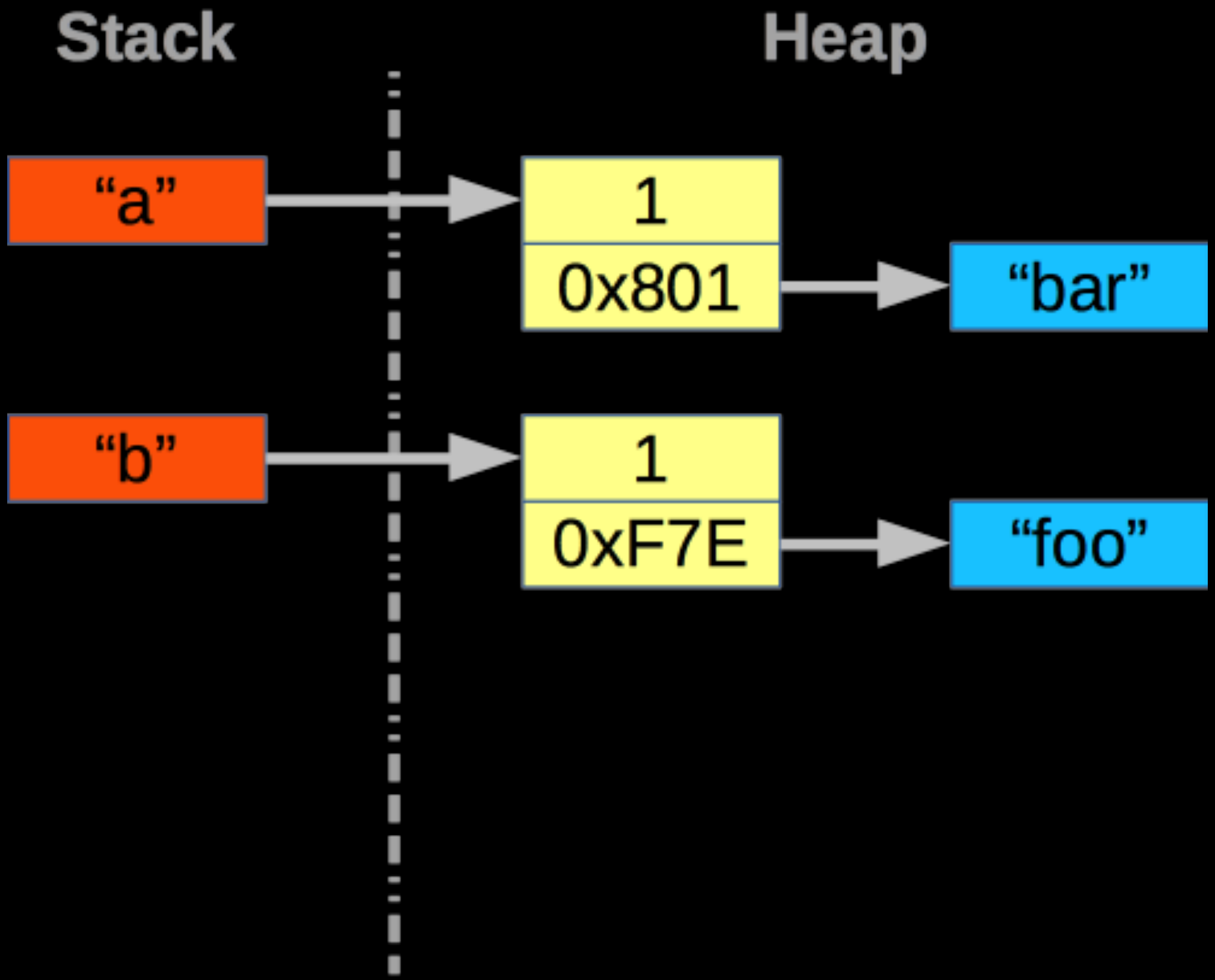
Reference Counting

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



Reference Counting

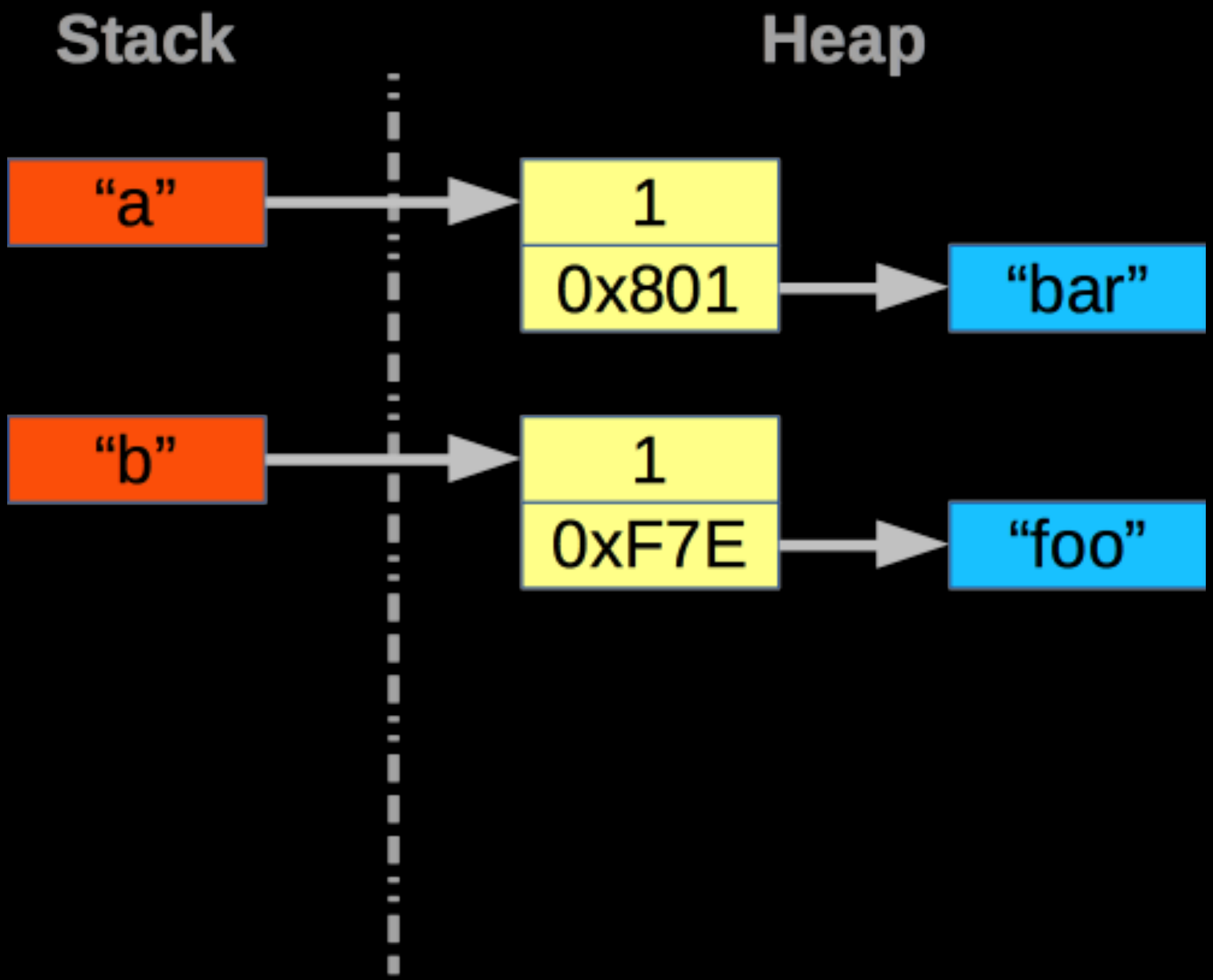
```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
```



© Paul Ross 2016

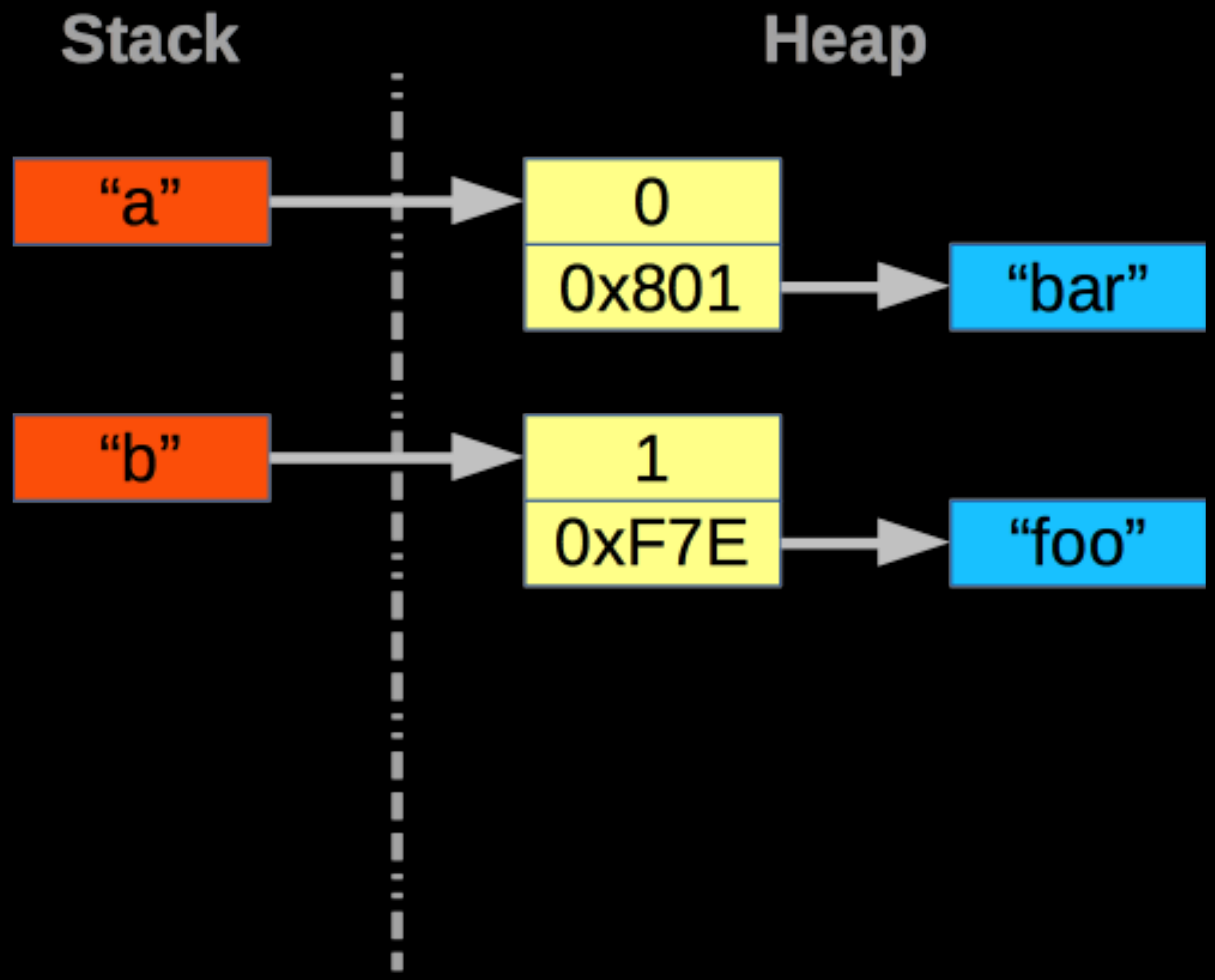
Reference Counting

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



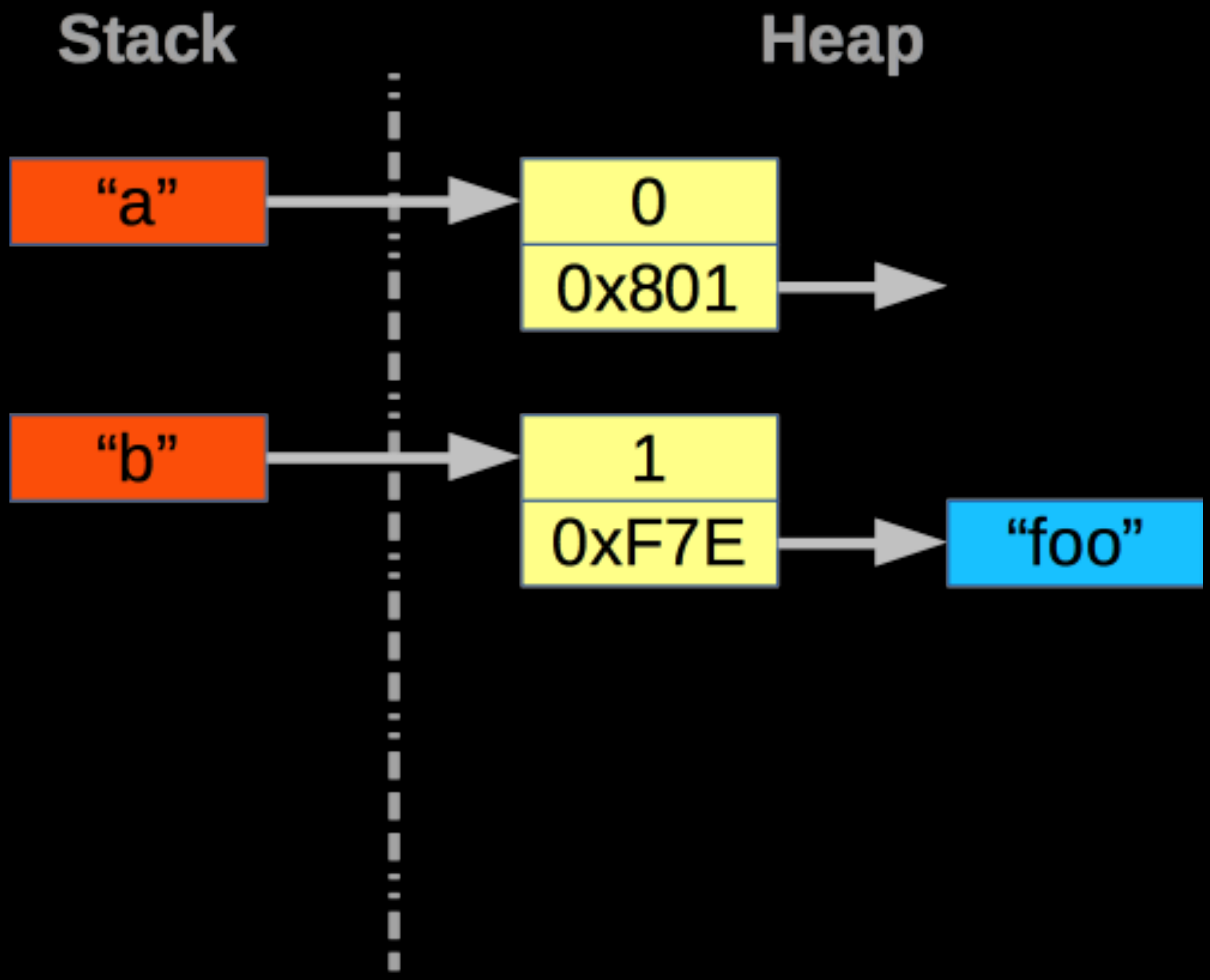
Reference Counting

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



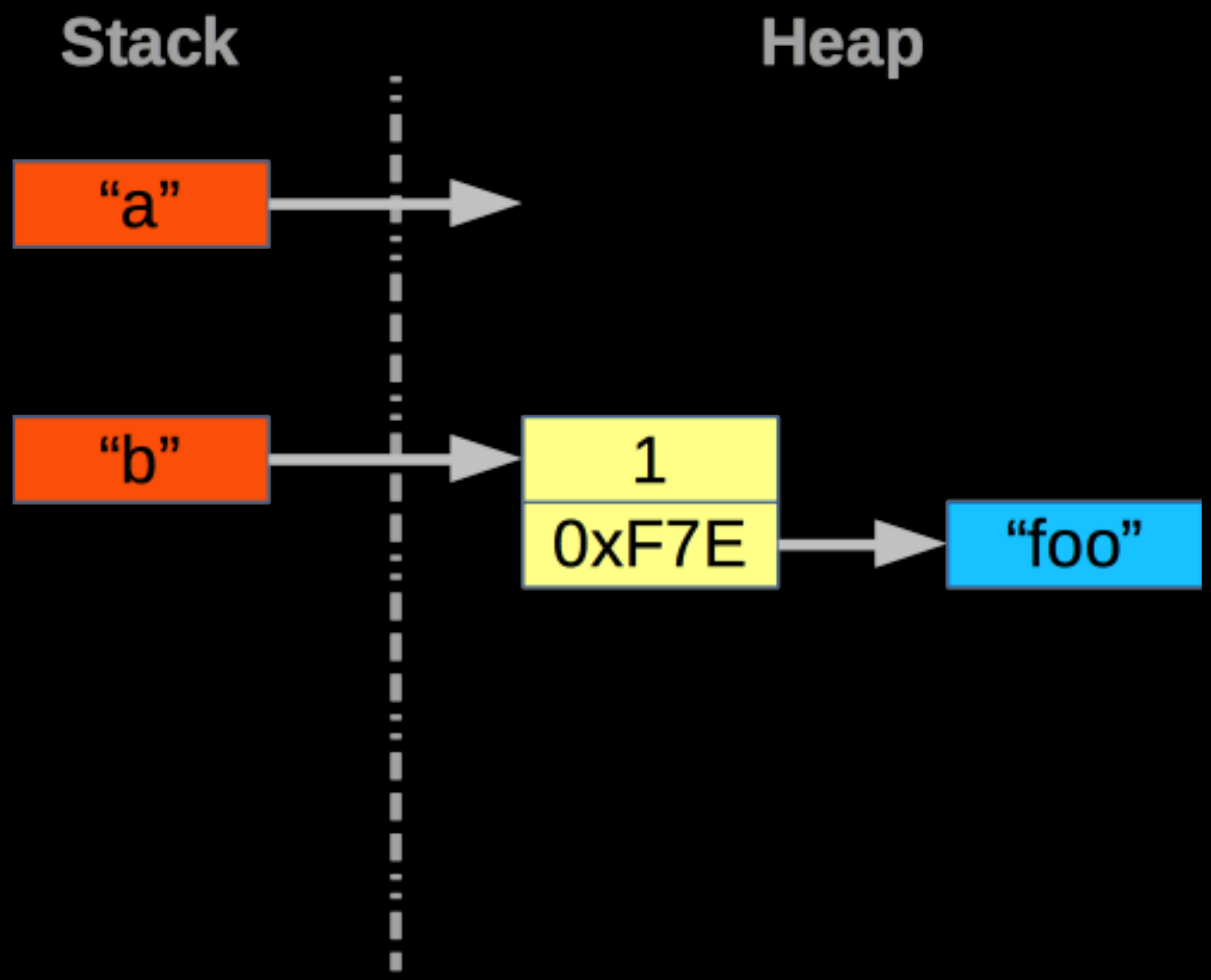
Reference Counting

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



Reference Counting

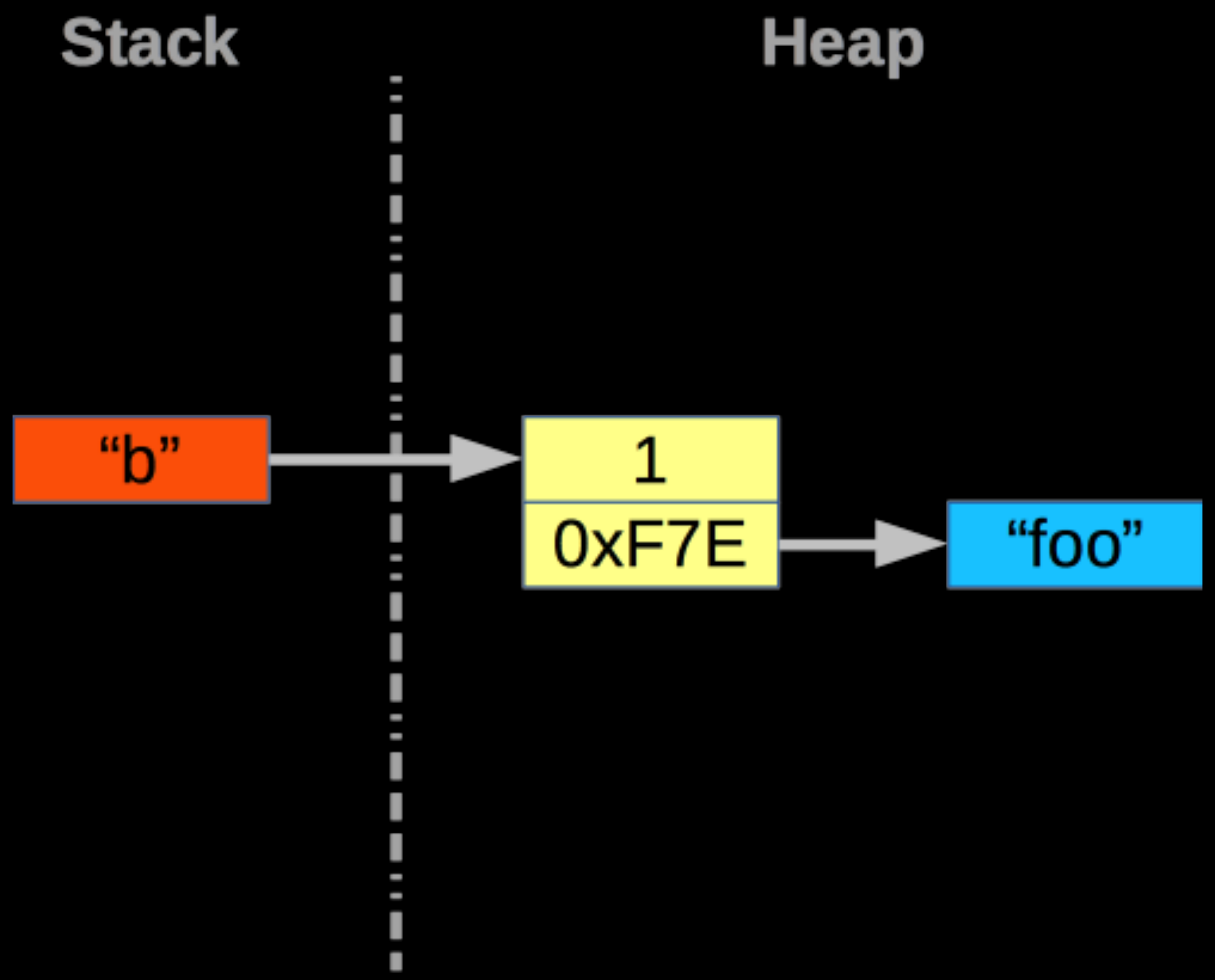
```
>>> a = 'foo'
>>> b = a
>>> a = 'bar'
>>> del a
```



© Paul Ross 2016

Reference Counting

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



© Paul Ross 2016

Garbage Collection

- The GC is just there to resolve cyclic references
  - Only works with containers
- It is 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

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 deallocation

C Memory Leaks

```
void leak() {
    char *p = malloc(1024);
    p[8] = 'A';
}
```

C Access After free()

["free" is **not** "make impossible to access"]

```c
void access_after_free() {
    char *p = malloc(1024);
    p[8] = 'A';

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

Py Memory Leaks

```c
#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 DecRef

```
#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 DecRef

Please don't do this

```
Py_DECREF(pObj);

/* Is ob_refcnt really for the same object? */
if (pObj->ob_refcnt > 0) {
    PyObject_Print(pObj, stdout, 0);
}
```

Summary

1 coding pattern to keep the dragons at bay

2 things to avoid

3 kinds of `PyObject*` used in CPython

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: appending a new value to 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 deallocate it
  - Or give it to someone who will
- **Stolen** `PyObject*` The 'thief' will deallocate it
  - Do not do so yourself
- **Borrowed** `PyObject*` The real owner can deallocate 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 ref */
    pB = PyLong_FromLong(b);        /* New ref */

    r = PyNumber_Subtract(pA, pB);  /* New ref */

    Py_DECREF(pA);        /* I must decref */
    Py_DECREF(pB);        /* I must decref */
    return r;             /* Caller must decref */
}
```

New References

Please Don't do this

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

Stolen References

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

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

    v = PyLong_FromLong(1L);    /* New ref */
    PyTuple_SetItem(r, 0, v);

    v = PyLong_FromLong(2L);    /* New ref */
    PyTuple_SetItem(r, 1, v);

    /* More common pattern */
    PyTuple_SetItem(r, 2, PyLong_FromLong(3L));
    return r;    /* Callers must decref */
}
```

Stolen References

Please don't do this

```c
PyObject *r, *v;

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

v = PyLong_FromLong(1L);    /* New ref */
PyTuple_SetItem(r, 0, v);   /* r 'steals' v */

Py_DECREF(v); /* NO! v 'belongs' to r */
```

'Borrowed' References

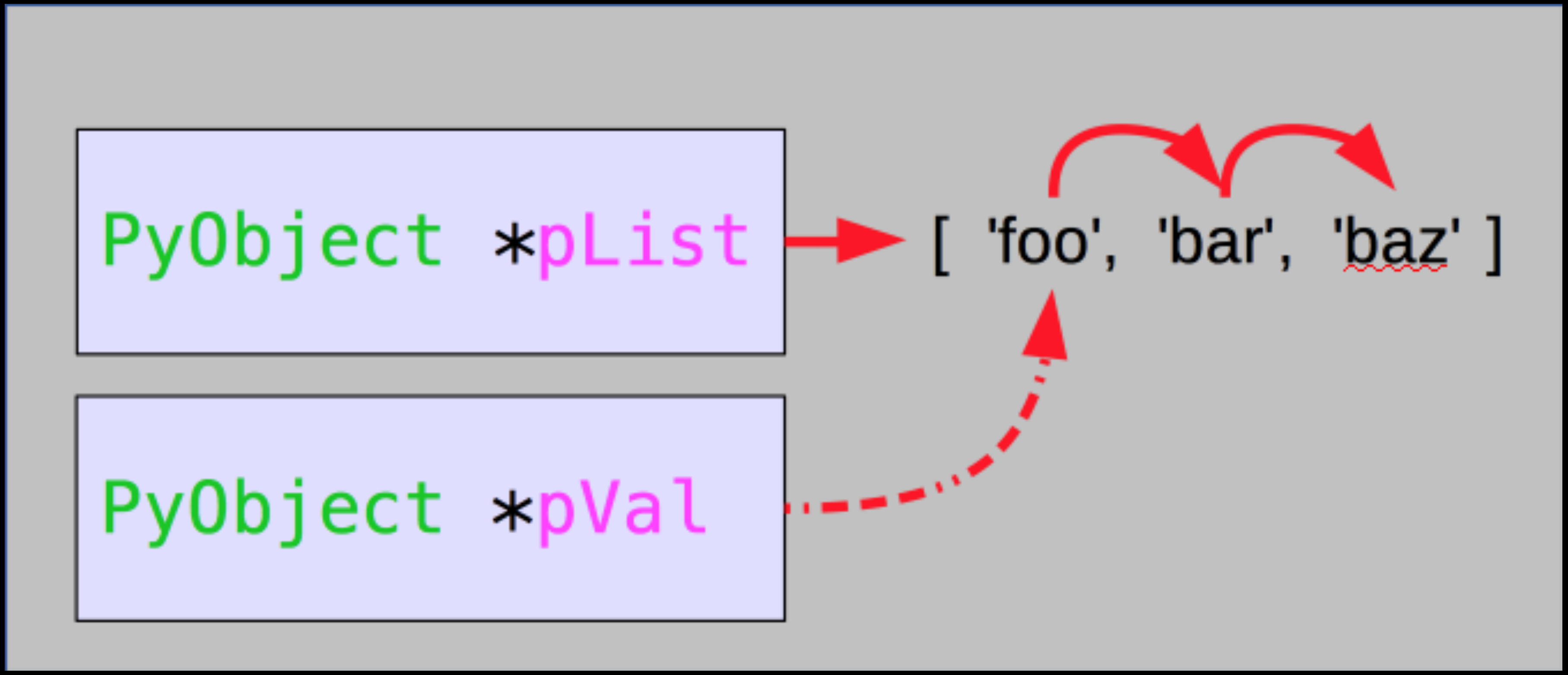
- These are generally 'getters'

```
PyObject *pList = ...
PyObject *pVal  = PyList_GetItem(pList, 0);
```

'Borrowed' References

- These are generally 'getters'

```
PyObject *pList = ...
PyObject *pVal  = PyList_GetItem(pList, 0);
```



'Borrowed' References

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

'Borrowed' References

```
static PyObject *
borrow_BAD(PyObject *pList) {
    PyObject *pFirst;
    pFirst = PyList_GetItem(pList, 0);

    function(pList);    /* Dragons ahoy! */
    PyObject_Print(pFirst, stdout, 0);


    Py_RETURN_NONE;
}
```

Hmm… Suppose

```
static PyObject *
borrow_BAD(PyObject *pList) {
    PyObject *pFirst;
    pFirst = PyList_GetItem(pList, 0);

    function(pList);    /* Dragons ahoy! */
    PyObject_Print(pFirst, stdout, 0);


    Py_RETURN_NONE;
}
```

Hmm… Suppose

```c
static PyObject *
borrow_BAD(PyObject *pList) {
    PyObject *pFirst;
    pFirst = PyList_GetItem(pList, 0);

    function(pList);    /* Dragons ahoy! */
    PyObject_Print(pFirst, stdout, 0);


    Py_RETURN_NONE;
}
```

This removed the first 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[0]
>>> cPyRefs.borrow_bad(l) # Works fine!
```

Borrowed Ref Dragon 1

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

```
>>> import cPyRefs
>>> x ='foo'
>>> l = ['bar', 'baz']
>>> l.insert(0, x)
>>> 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

© Paul Ross 2016

The Problem

```
static PyObject *
borrow_BAD(PyObject *pList) {
    PyObject *pFirst;
    pFirst = PyList_GetItem(pList, 0);

    function(pList);    /* Dragons ahoy! */
    PyObject_Print(pFirst, stdout, 0);


    Py_RETURN_NONE;
}
```

The Fix

```
static PyObject *
borrow_BAD(PyObject *pList) {
    PyObject *pFirst;
    pFirst = PyList_GetItem(pList, 0);

    function(pList);    /* Dragons ahoy! */
    PyObject_Print(pFirst, stdout, 0);


    Py_RETURN_NONE;
}
```

The Fix

```
static PyObject *
borrow_BAD(PyObject *pList) {
    PyObject *pFirst;
    pFirst = PyList_GetItem(pLi
    Py_INCREF(pFirst);
    function(pList);    /* Dragons tamed. */
    PyObject_Print(pLast, stdout, 0);
    Py_DECREF(pFirst);
    pFirst = NULL;
    Py_RETURN_NONE;
}
```

Register your interest!

Let go

Summary

1 coding pattern to keep the dragons at bay

2 things to avoid

3 kinds of `PyObject*` used in CPython

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 function(obj):
    ret = None;

    try:
        # Do fabulous stuff here
        # On error, raise
    except ... as err:
        # Handle exceptions
    finally:
        # And we are out
    return ret;
```

Writing Pythonic C

Writing Pythonic C

```c
static PyObject *function(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 *function(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:

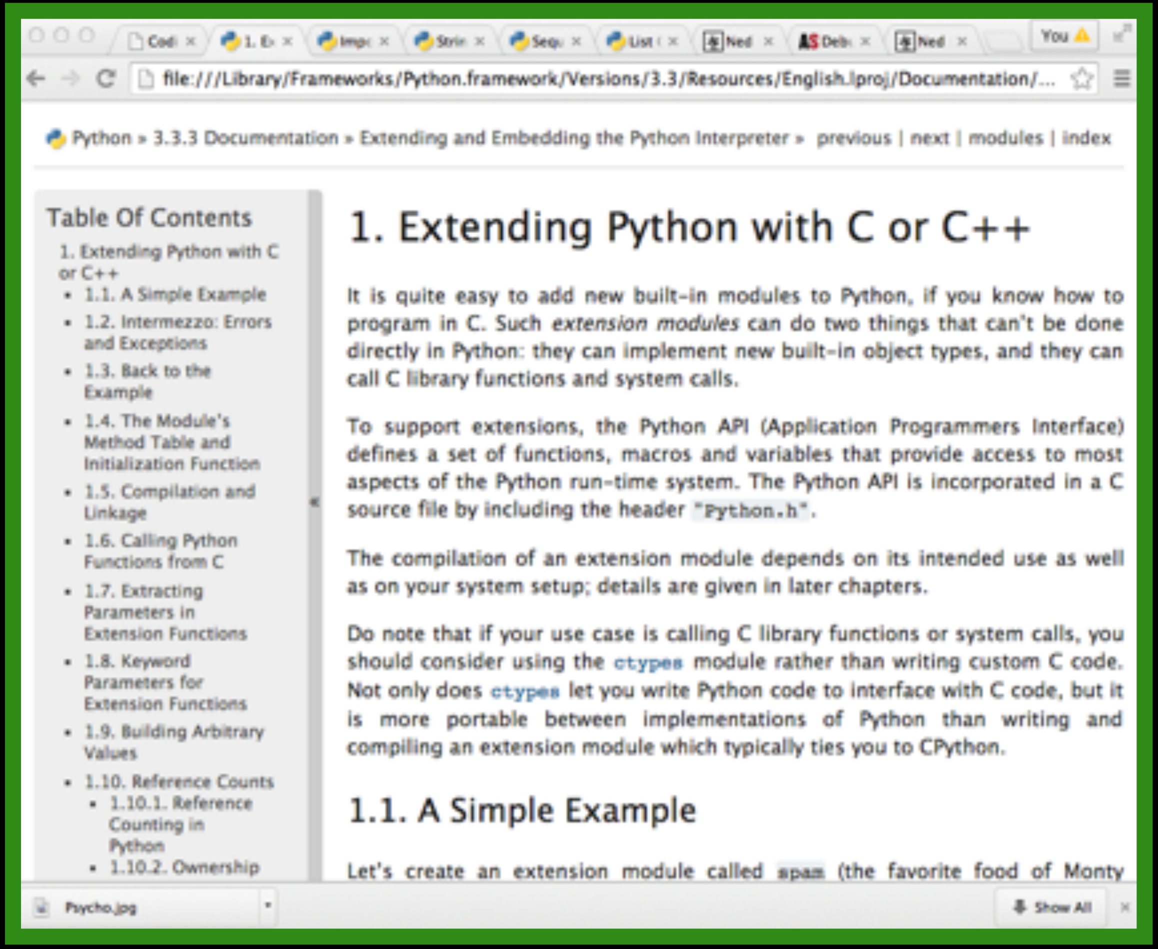
```
finally:
    /* All _local_ PyObjects are Py_XDECREF'd here.
     * For new references this will free them.
     * For borrowed references this
     * will return them to their previous state. */
    Py_XDECREF(obj_a);
    /* Decrement the ref count of given arguments
     * if they have been incremented.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

All this and more…

https://github.com/paulross

In “PythonExtensionPatterns”

The Documentation is Excellent - Use it!



Codi
1. Ex
Impc
Strin
Sequ
List (
Ned
Deb
Ned
You
file:///Library/Frameworks/Python.framework/Versions/3.3/Resources/English.lproj/Documentation/...
Psycho.jpg
Show All

Python » 3.3.3 Documentation » Extending and Embedding the Python Interpreter » previous | next | modules | index

Table Of Contents

1. Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such *extension modules* can do two things that can't be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header `"Python.h"`.

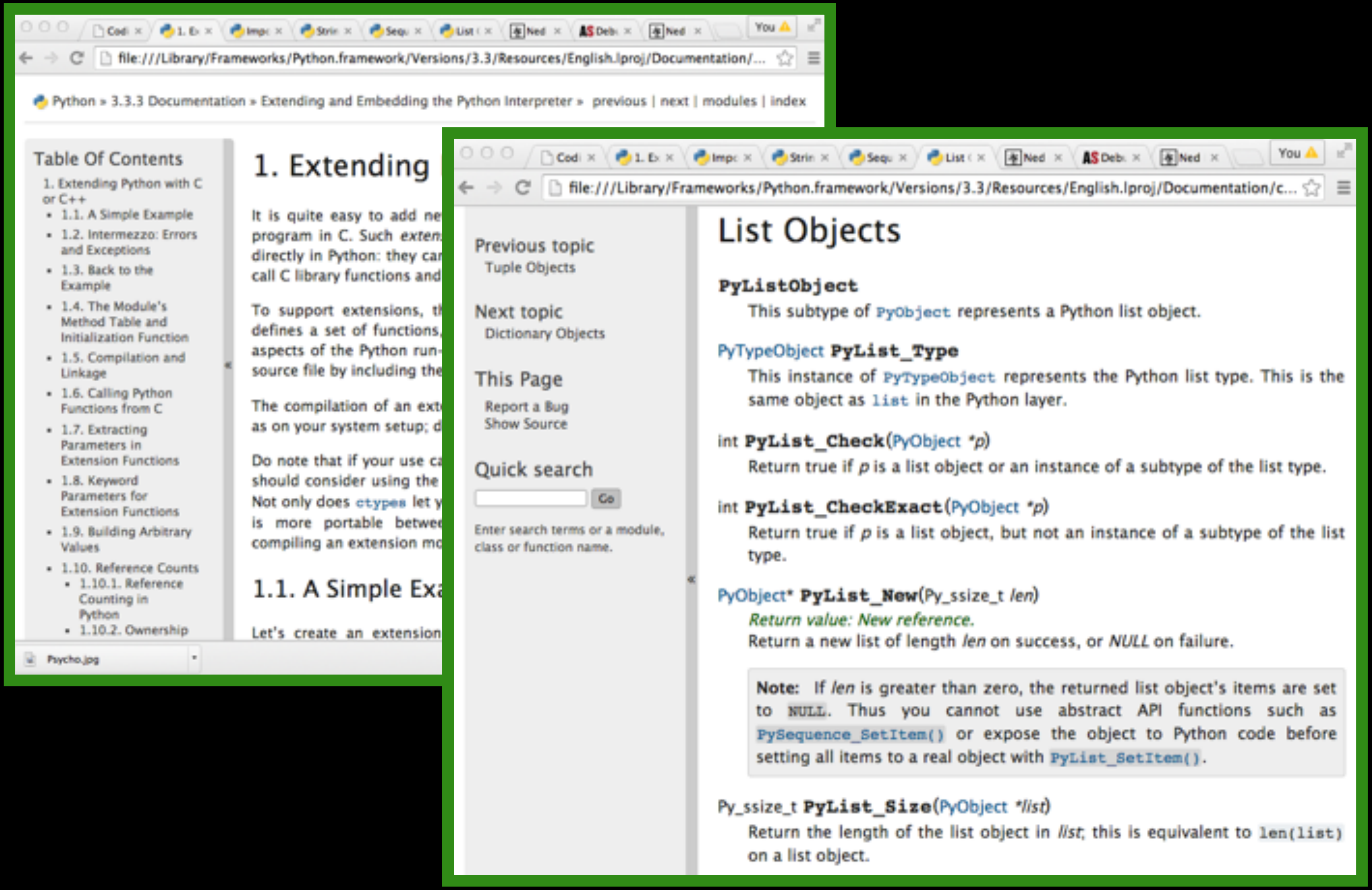
The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

Do note that if your use case is calling C library functions or system calls, you should consider using the `ctypes` module rather than writing custom C code. Not only does `ctypes` let you write Python code to interface with C code, but it is more portable between implementations of Python than writing and compiling an extension module which typically ties you to CPython.

1.1. A Simple Example

Let's create an extension module called `spam` (the favorite food of Monty

The Documentation is Excellent - Use it!



file:///Library/Frameworks/Python.framework/Versions/3.3/Resources/English.lproj/Documentation/...

Python » 3.3.3 Documentation » Extending and Embedding the Python Interpreter » previous | next | modules | index

Table Of Contents

1. Extending Python with C or C++
- 1.1. A Simple Example
- 1.2. Intermezzo: Errors and Exceptions
- 1.3. Back to the Example
- 1.4. The Module's Method Table and Initialization Function
- 1.5. Compilation and Linkage
- 1.6. Calling Python Functions from C
- 1.7. Extracting Parameters in Extension Functions
- 1.8. Keyword Parameters for Extension Functions
- 1.9. Building Arbitrary Values
- 1.10. Reference Counts
  - 1.10.1. Reference Counting in Python
  - 1.10.2. Ownership

1. Extending

It is quite easy to add ne program in C. Such exten directly in Python: they ca call C library functions and

To support extensions, t defines a set of functions, aspects of the Python run- source file by including the

The compilation of an ext as on your system setup; d

Do note that if your use ca should consider using the Not only does `ctypes` let y is more portable betwee compiling an extension mo

1.1. A Simple Exa

Let's create an extension

Psycho.jpg

file:///Library/Frameworks/Python.framework/Versions/3.3/Resources/English.lproj/Documentation/c...

Previous topic
Tuple Objects

Next topic
Dictionary Objects

This Page
Report a Bug
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

List Objects

`PyListObject`
This subtype of `PyObject` represents a Python list object.

PyTypeObject **`PyList_Type`**
This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

int **`PyList_Check`**(PyObject **p*)
Return true if *p* is a list object or an instance of a subtype of the list type.

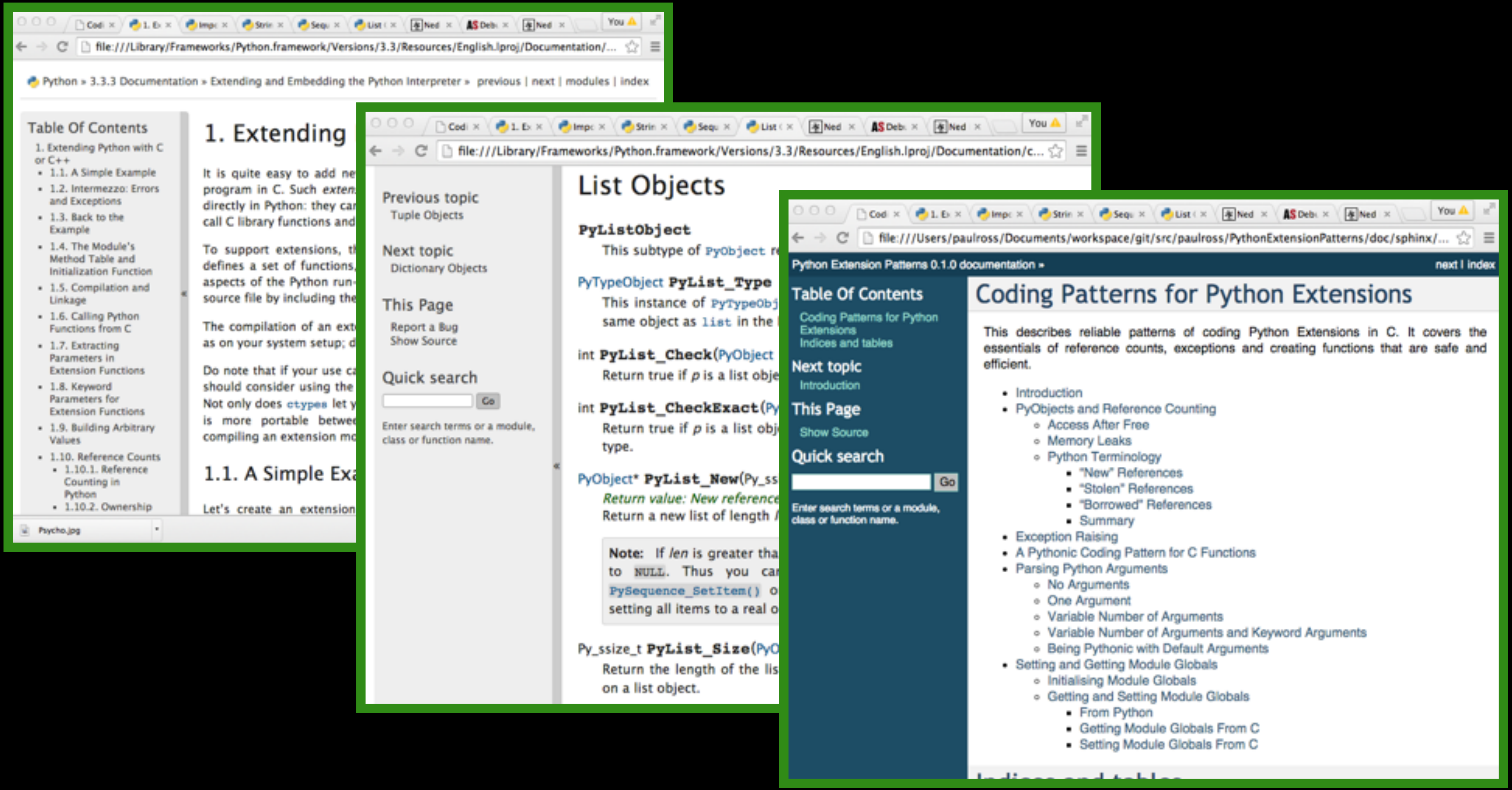
int **`PyList_CheckExact`**(PyObject **p*)
Return true if *p* is a list object, but not an instance of a subtype of the list type.

PyObject* **`PyList_New`**(Py_ssize_t *len*)
Return value: New reference.
Return a new list of length *len* on success, or *NULL* on failure.

> **Note:** If *len* is greater than zero, the returned list object's items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()`.

Py_ssize_t **`PyList_Size`**(PyObject **list*)
Return the length of the list object in *list*; this is equivalent to `len(list)` on a list object.

The Documentation is Excellent - Use it!



file:///Library/Frameworks/Python.framework/Versions/3.3/Resources/English.lproj/Documentation/...

Python » 3.3.3 Documentation » Extending and Embedding the Python Interpreter » previous | next | modules | index

Table Of Contents

1. Extending Python with C or C++
- 1.1. A Simple Example
- 1.2. Intermezzo: Errors and Exceptions
- 1.3. Back to the Example
- 1.4. The Module's Method Table and Initialization Function
- 1.5. Compilation and Linkage
- 1.6. Calling Python Functions from C
- 1.7. Extracting Parameters in Extension Functions
- 1.8. Keyword Parameters for Extension Functions
- 1.9. Building Arbitrary Values
- 1.10. Reference Counts
  - 1.10.1. Reference Counting in Python
  - 1.10.2. Ownership

1. Extending

It is quite easy to add ne
program in C. Such *exten*
directly in Python: they car
call C library functions and

To support extensions, th
defines a set of functions,
aspects of the Python run-
source file by including the

The compilation of an ext
as on your system setup; d

Do note that if your use ca
should consider using the
Not only does `ctypes` let y
is more portable betwee
compiling an extension mo

1.1. A Simple Exa

Let's create an extension

Psycho.jpg

file:///Library/Frameworks/Python.framework/Versions/3.3/Resources/English.lproj/Documentation/c...

Previous topic
Tuple Objects

Next topic
Dictionary Objects

This Page
Report a Bug
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

List Objects

PyListObject
This subtype of `PyObject` r

PyTypeObject **PyList_Type**
This instance of `PyTypeObj`
same object as `list` in the

int **PyList_Check**(PyObject
Return true if *p* is a list obje

int **PyList_CheckExact**(Py
Return true if *p* is a list obj
type.

PyObject* **PyList_New**(Py_ss
Return value: New reference
Return a new list of length *l*

Note: If *len* is greater tha
to `NULL`. Thus you car
`PySequence_SetItem()` o
setting all items to a real o

Py_ssize_t **PyList_Size**(PyO
Return the length of the lis
on a list object.

file:///Users/paulross/Documents/workspace/git/src/paulross/PythonExtensionPatterns/doc/sphinx/...

Python Extension Patterns 0.1.0 documentation » next | index

Table Of Contents

Coding Patterns for Python Extensions
Indices and tables

Next topic
Introduction

This Page
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Coding Patterns for Python Extensions

This describes reliable patterns of coding Python Extensions in C. It covers the essentials of reference counts, exceptions and creating functions that are safe and efficient.

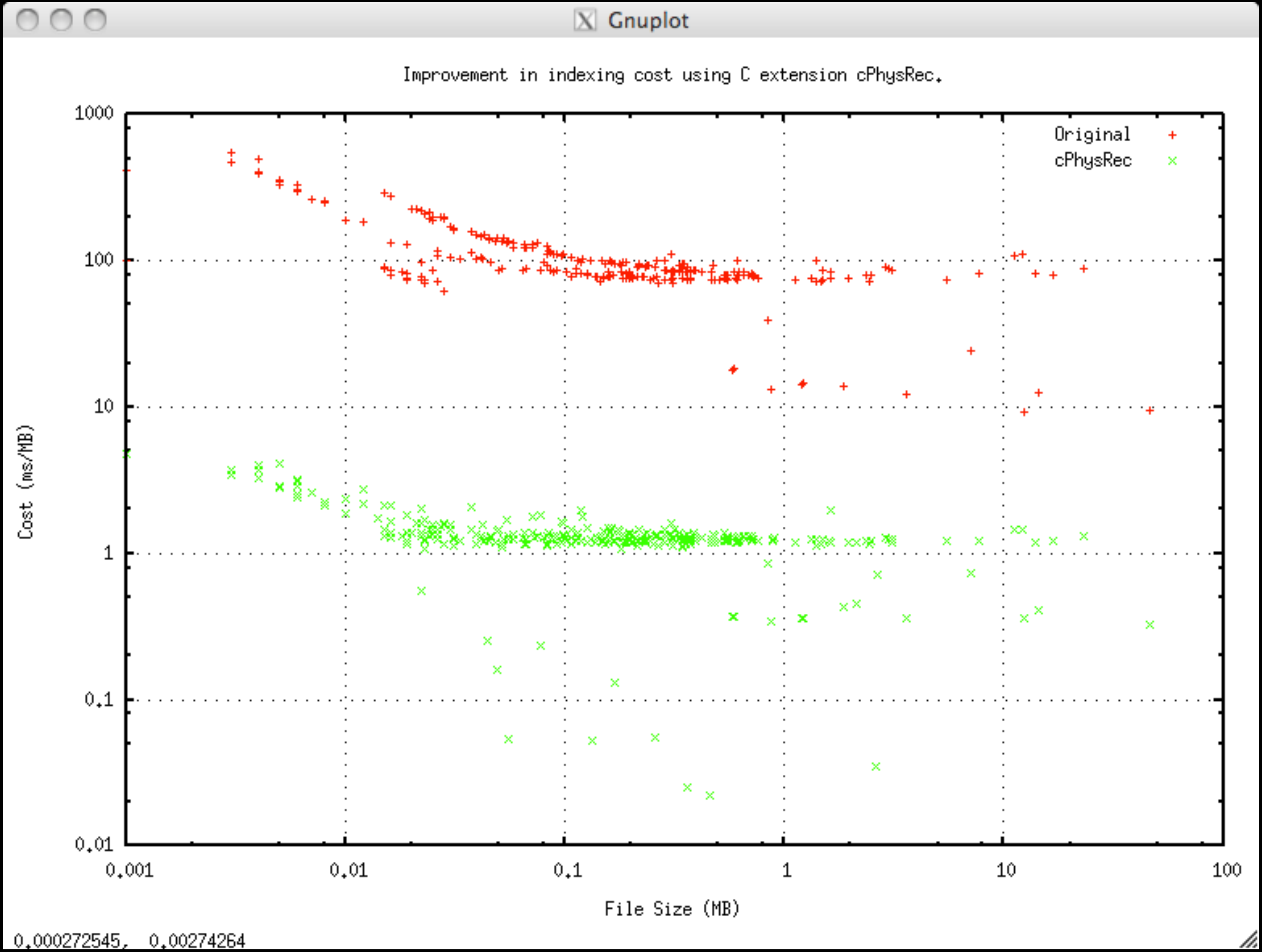
- Introduction
- PyObjects and Reference Counting
  - Access After Free
  - Memory Leaks
  - Python Terminology
    - "New" References
    - "Stolen" References
    - "Borrowed" References
    - Summary
- Exception Raising
- A Pythonic Coding Pattern for C Functions
- Parsing Python Arguments
  - No Arguments
  - One Argument
  - Variable Number of Arguments
  - Variable Number of Arguments and Keyword Arguments
  - Being Pythonic with Default Arguments
- Setting and Getting Module Globals
  - Initialising Module Globals
  - Getting and Setting Module Globals
    - From Python
    - Getting Module Globals From C
    - Setting Module Globals From C

Indices and tables

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.
Original +
cPhysRec ×
Cost (ms/MB)
1000
100
10
1
0.1
0.01
0.001
0.01
0.1
1
10
100
File Size (MB)
0.000272545, 0.00274264

Summary

1 coding pattern to keep the dragons at bay

2 things to avoid

- Allocation with no deallocation
- Access after deallocation

3 kinds of references to `PyObject*`

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

`github.com/paulross`

twitter.com/manahltech

Opinions expressed are those of the author and may not be shared by all personnel of Man Group plc ('Man'). These opinions are subject to change without notice, are for information purposes only and do not constitute an offer or invitation to make an investment in any financial instrument or in any product to which the Company and/ or its affiliates provides investment advisory or any other financial services. Any organizations, financial instrument or products described in this material are mentioned for reference purposes only which should not be considered a recommendation for their purchase or sale. Neither the Company nor the authors shall be liable to any person for any action taken on the basis of the information provided. Some statements contained in this material concerning goals, strategies, outlook or other non-historical matters may be forward-looking statements and are based on current indicators and expectations. These forward-looking statements speak only as of the date on which they are made, and the Company undertakes no obligation to update or revise any forward-looking statements. These forward-looking statements are subject to risks and uncertainties that may cause actual results to differ materially from those contained in the statements. The Company and/or its affiliates may or may not have a position in any financial instrument mentioned and may or may not be actively trading in any such securities. This material is proprietary information of the Company and its affiliates and may not be reproduced or otherwise disseminated in whole or in part without prior written consent from the Company. The Company believes the content to be accurate. However, accuracy is not warranted or guaranteed. The Company does not assume any liability in the case of incorrectly reported or incomplete information. Unless stated otherwise all information is provided by the Company.

Questions ???

github.com/paulross

twitter.com/manahltech

Opinions expressed are those of the author and may not be shared by all personnel of Man Group plc (‘Man’). These opinions are subject to change without notice, are for information purposes only and do not constitute an offer or invitation to make an investment in any financial instrument or in any product to which the Company and/ or its affiliates provides investment advisory or any other financial services. Any organizations, financial instrument or products described in this material are mentioned for reference purposes only which should not be considered a recommendation for their purchase or sale. Neither the Company nor the authors shall be liable to any person for any action taken on the basis of the information provided. Some statements contained in this material concerning goals, strategies, outlook or other non-historical matters may be forward-looking statements and are based on current indicators and expectations. These forward-looking statements speak only as of the date on which they are made, and the Company undertakes no obligation to update or revise any forward-looking statements. These forward-looking statements are subject to risks and uncertainties that may cause actual results to differ materially from those contained in the statements. The Company and/or its affiliates may or may not have a position in any financial instrument mentioned and may or may not be actively trading in any such securities. This material is proprietary information of the Company and its affiliates and may not be reproduced or otherwise disseminated in whole or in part without prior written consent from the Company. The Company believes the content to be accurate. However, accuracy is not warranted or guaranteed. The Company does not assume any liability in the case of incorrectly reported or incomplete information. Unless stated otherwise all information is provided by the Company.