Python Extension Patterns Documentation

Release 0.1.0

Paul Ross

Mar 14, 2021

CONTENTS

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.

CHAPTER

ONE

INTRODUCTION

Writing Python C Extensions can be daunting; you have to cast aside the security and fluidity of Python and embrace C, not just C but Pythons C API, which is huge[1]. Not only do you have to worry about just your standard `malloc()` and `free()` cases but now you have to contend with how CPython's does its memory management which is by *reference counting*.

I describe some of the pitfalls you (I am thinking of you as a savvy C coder) can encounter and some of the coding patterns that you can use to avoid them.

First up: understanding reference counts and Python's terminology.

[1] Huge, but pretty consistent once mastered.

CHAPTER

TWO

PYOBJECTS AND REFERENCE COUNTING

A `PyObject` can represent any Python object. It is a fairly minimal C struct consisting of a reference count and a pointer to the object proper:

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

In Python C extensions you always create and deallocate these `PyObjects` *indirectly*. Creation is via Python's C API and destruction is done by decrementing the reference count. If this count hits zero then CPython will free all the resources used by the object.

Here is an example of a normal `PyObject` creation and deallocation:

```
#include "Python.h"

void print_hello_world(void) {
    PyObject *pObj = NULL;

    pObj = PyBytes_FromString("Hello world\n"); /* Object creation, ref count = 1. */
    PyObject_Print(pLast, stdout, 0);
    Py_DECREF(pObj);     /* ref count becomes 0, object deallocated.
                          * Miss this step and you have a memory leak. */
}
```

The twin challenges in Python extensions are:

- Avoiding undefined behaviour such as object access after an object's reference count is zero. This is analogous in C to access after `free()` or a using dangling pointer.
- Avoiding memory leaks where an object's reference count never reaches zero and there are no references to the object. This is analogous in C to a `malloc()` with no corresponding `free()`.

Here are some examples of where things can go wrong:

2.1 Access After Free

Taking the above example of a normal `PyObject` creation and deallocation then in the grand tradition of C memory management after the `Py_DECREF` the `pObj` is now referencing free'd memory:

```
#include "Python.h"

void print_hello_world(void) {
    PyObject *pObj = NULL:

    pObj = PyBytes_FromString("Hello world\n");    /* Object creation, ref count = 1. */
    PyObject_Print(pLast, stdout, 0);
    Py_DECREF(pObj);                               /* ref count = 0 so object deallocated. */
    /* Accidentally use pObj... */
}
```

Accessing `pObj` may or may not give you something that looks like the original object.

The corresponding issue is if you decrement the reference count without previously incrementing it then the caller might find *their* reference invalid:

```
static PyObject *bad_incref(PyObject *pObj) {
    /* Forgotten Py_INCREF(pObj); here... */

    /* Use pObj... */

    Py_DECREF(pObj); /* Might make reference count zero. */
    Py_RETURN_NONE;  /* On return caller might find their object free'd. */
}
```

After the function returns the caller *might* find the object they naively trusted you with but probably not. A classic access-after-free error.

2.2 Memory Leaks

Memory leaks occur with a `PyObject` if the reference count never reaches zero and there is no Python reference or C pointer to the object in scope. Here is where it can go wrong: in the middle of a great long function there is an early return on error. On that path this code has a memory leak:

```
static PyObject *bad_incref(PyObject *pObj) {
    Py_INCREF(pObj);
    /* ... a metric ton of code here ... */
    if (error) {
        /* No matching Py_DECREF, pObj is leaked. */
        return NULL;
    }
    /* ... more code here ... */
    Py_DECREF(pObj);
    Py_RETURN_NONE;
}
```

The problem is that the reference count was not decremented before the early return, if `pObj` was a 100 Mb string then that memory is lost. Here is some C code that demonstrates this:

```
static PyObject *bad_incref(PyObject *pModule, PyObject *pObj) {
    Py_INCREF(pObj);
    Py_RETURN_NONE;
}
```

And here is what happens to the memory if we use this function from Python (`cPyRefs.incref(...)` in Python calls `bad_incref()` in C):

```
>>> import cPyRefs           # Process uses about 1Mb
>>> s = ' ' * 100 * 1024**2 # Process uses about 101Mb
>>> del s                    # Process uses about 1Mb
>>> s = ' ' * 100 * 1024**2 # Process uses about 101Mb
>>> cPyRefs.incref(s)        # Now do an increment without decrement
>>> del s                    # Process still uses about 101Mb - leaked
>>> s                        # Officially 's' does not exist
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 's' is not defined
>>>                          # But process still uses about 101Mb - 's' is leaked
```

Warning: Do not be tempted to read the reference count itself to determine if the object is alive. The reason is that if `Py_DECREF` sees a refcount of one it can free and then reuse the address of the refcount field for a completely different object which makes it highly unlikely that that field will have a zero in it. There are some examples of this later on.

2.3 Python Terminology

The Python documentation uses the terminology "New", "Stolen" and "Borrowed" references throughout. These terms identify who is the *real owner* of the reference and whose job it is to clean it up when it is no longer needed:

- **New** references occur when a `PyObject` is constructed, for example when creating a new list.
- **Stolen** references occur when composing a `PyObject`, for example appending a value to a list. "Setters" in other words.
- **Borrowed** references occur when inspecting a `PyObject`, for example accessing a member of a list. "Getters" in other words. *Borrowed* does not mean that you have to return it, it just means you that don't own it. If *shared* references or pointer aliases mean more to you than *borrowed* references that is fine because that is exactly what they are.

This is about programming by contract and the following sections describe the contracts for each reference type.

First up **New** references.

2.3.1 "New" References

When you create a "New" `PyObject` from a Python C API then you own it and it is your job to either:

- Dispose of the object when it is no longer needed with `Py_DECREF`[2].
- Give it to someone else who will do that for you.

If neither of these things is done you have a memory leak in just like a `malloc()` without a corresponding `free()`.

Here is an example of a well behaved C function that take two C longs, converts them to Python integers and, subtracts one from the other and returns the Python result:

```
1  static PyObject *subtract_long(long a, long b) {
2      PyObject *pA, *pB, *r;
3
4      pA = PyLong_FromLong(a);         /* pA: New reference. */
5      pB = PyLong_FromLong(b);         /* pB: New reference. */
6      r = PyNumber_Subtract(pA, pB);   /*  r: New reference. */
7      Py_DECREF(pA);                   /* My responsibility to decref. */
8      Py_DECREF(pB);                   /* My responsibility to decref. */
9      return r;                        /* Callers responsibility to decref. */
10 }
```

`PyLong_FromLong()` returns a *new* reference which means we have to clean up ourselves by using `Py_DECREF`.

`PyNumber_Subtract()` also returns a *new* reference but we expect the caller to clean that up. If the caller doesn't then there is a memory leak.

So far, so good but what would be really bad is this:

```
r = PyNumber_Subtract(PyLong_FromLong(a), PyLong_FromLong(b));
```

You have passed in two *new* references to `PyNumber_Subtract()` and that function has no idea that they have to be decref'd once used so the two PyLong objects are leaked.

The contract with *new* references is: either you decref it or give it to someone who will. If neither happens then you have a memory leak.

2.3.2 "Stolen" References

This is also to do with object creation but where another object takes responsibility for decref'ing (possibly freeing) the object. Typical examples are when you create a `PyObject` that is then inserted into an existing container such as a tuple list, dict etc.

The analogy with C code is malloc'ing some memory, populating it and then passing that pointer to a linked list which then takes on the responsibility to free the memory if that item in the list is removed. If you were to free the memory you had malloc'd then you will get a double free when the linked list (eventually) frees its members.

Here is an example of creating a 3-tuple, the comments describe what is happening contractually:

```
1  static PyObject *make_tuple(void) {
2      PyObject *r;
3      PyObject *v;
4
5      r = PyTuple_New(3);          /* New reference. */
```

(continues on next page)

[2] To be picky we just need to decrement the use of *our* reference to it. Other code that has incremented the same reference is responsible for decrementing their use of the reference.

(continued from previous page)

```
6       v = PyLong_FromLong(1L);     /* New reference. */
7       /* PyTuple_SetItem "steals" the new reference v. */
8       PyTuple_SetItem(r, 0, v);
9       /* This is fine. */
10      v = PyLong_FromLong(2L);
11      PyTuple_SetItem(r, 1, v);
12      /* More common pattern. */
13      PyTuple_SetItem(r, 2, PyUnicode_FromString("three"));
14      return r; /* Callers responsibility to decref. */
15  }
```

Note line 10 where we are overwriting an existing pointer with a new value, this is fine as `r` has taken responsibility for the first pointer value. This pattern is somewhat alarming to dedicated C programmers so the more common pattern, without the assignment to `v` is shown in line 13.

What would be bad is this:

```
v = PyLong_FromLong(1L);     /* New reference. */
PyTuple_SetItem(r, 0, v);    /* r takes ownership of the reference. */
Py_DECREF(v);                /* Now we are interfering with r's internals. */
```

Once `v` has been passed to `PyTuple_SetItem` then your `v` becomes a *borrowed* reference with all of their problems which is the subject of the next section.

The contract with *stolen* references is: the thief will take care of things so you don't have to. If you try to the results are undefined.

2.3.3 "Borrowed" References

When you obtain a reference to an existing `PyObject` in a container using a 'getter' you are given a *borrowed* reference and this is where things can get tricky. The most subtle bugs in Python C Extensions are usually because of the misuse of borrowed references.

The analogy in C is having two pointers to the same memory location: so who is responsible for freeing the memory and what happens if the other pointer tries to access that free'd memory?

Here is an example where we are accessing the last member of a list with a "borrowed" reference. This is the sequence of operations:

- Get a *borrowed* reference to a member of the list.
- Do some operation on that list, in this case call `do_something()`.
- Access the *borrowed* reference to the member of the original list, in this case just print it out.

Here is a C function that *borrows* a reference to the last object in a list, prints out the object's reference count, calls another C function `do_something()` with that list, prints out the reference count of the object again and finally prints out the Python representation of the object:

```
1   static PyObject *pop_and_print_BAD(PyObject *pList) {
2       PyObject *pLast;
3
4       pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);
5       fprintf(stdout, "Ref count was: %zd\n", pLast->ob_refcnt);
6       do_something(pList);
7       fprintf(stdout, "Ref count now: %zd\n", pLast->ob_refcnt);
8       PyObject_Print(pLast, stdout, 0);
```

(continues on next page)

(continued from previous page)

```
    fprintf(stdout, "\n");
    Py_RETURN_NONE;
}
```

The problem is that if `do_something()` mutates the list it might invalidate the item that we have a pointer to.

Suppose `do_something()` 'removes' every item in the list[3]. Then whether reference `pLast` is still "valid" depends on what other references to it exist and you have no control over that. Here are some examples of what might go wrong in that case (C `pop_and_print_BAD` is mapped to the Python `cPyRefs.popBAD`):

```
>>> l = ["Hello", "World"]
>>> cPyRefs.popBAD(l)        # l will become empty
Ref count was: 1
Ref count now: 4302027608
'World'
```

The reference count is bogus, however the memory has not been *completely* overwritten so the object (the string "World") *appears* to be OK.

If we try a different string:

```
>>> l = ['abc' * 200]
>>> cPyRefs.popBAD(l)
Ref count was: 1
Ref count now: 2305843009213693952
Segmentation fault: 11
```

At least this will get your attention!

Incidentially from Python 3.3 onwards there is a module faulthandler that can give useful debugging information (file `FaultHandlerExample.py`):

```
import faulthandler
faulthandler.enable()
import cPyRefs
l = ['abc' * 200]
cPyRefs.popBAD(l)
```

And this is what you get:

```
$ python3 FaultHandlerExample.py
Ref count was: 1
Ref count now: 2305843009213693952
Fatal Python error: Segmentation fault

Current thread 0x00007fff73c88310:
  File "FaultHandlerExample.py", line 7 in <module>
Segmentation fault: 11
```

There is a more subtle issue; suppose that in your Python code there is a reference to the last item in the list, then the problem suddenly "goes away":

```
>>> l = ["Hello", "World"]
>>> a = l[-1]
>>> cPyRefs.popBAD(l)
```

(continues on next page)

[3] Of course we never *remove* items in a list we merely decrement their reference count (and if that hits zero then they are deleted). Such as:

(continued from previous page)

```
Ref count was: 2
Ref count now: 1
'World'
```

The reference count does not go to zero so the object is preserved. The problem is that the correct behaviour of your C function depends entirely on that caller code having a extra reference.

This can happen implicitly as well:

```
>>> l = list(range(8))
>>> cPyRefs.popBAD(l)
Ref count was: 20
Ref count now: 19
7
```

The reason for this is that (for efficiency) CPython maintains the integers -5 to 255 permanently so they never go out of scope. If you use different integers we are back to the same access-after-free problem:

```
>>> l = list(range(800,808))
>>> cPyRefs.popBAD(l)
Ref count was: 1
Ref count now: 4302021872
807
```

The problem with detecting these errors is that the bug is data dependent so your code might run fine for a while but some change in user data could cause it to fail. And it will fail in a manner that is not easily detectable.

Fortunately the solution is easy: with borrowed references you should increment the reference count whilst you have an interest in the object, then decrement it when you no longer want to do anything with it:

```
1  static PyObject *pop_and_print_BAD(PyObject *pList) {
2      PyObject *pLast;
3
4      pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);
5      Py_INCREF(pLast);        /* Prevent pLast being deallocated. */
6      /* ... */
7      do_something(pList);
8      /* ... */
9      Py_DECREF(pLast);        /* No longer interested in pLast, it might      */
10     pLast = NULL;            /* get deallocated here but we shouldn't care. */
11     /* ... */
12     Py_RETURN_NONE;
13 }
```

The `pLast = NULL;` line is not necessary but is good coding style as it will cause any subsequent acesses to `pLast` to fail.

An important takeaway here is that incrementing and decrementing reference counts is a cheap operation but the consequences of getting it wrong can be expensive. A precautionary approach in your code might be to *always* increment borrowed references when they are instantiated and then *always* decrement them before they go out of scope. That way you incur two cheap operations but eliminate a vastly more expensive one.

2.4 An Example Leak Problem

Here is an example that exhibits a leak. The object is to add the integers 400 to 404 to the end of a list. You might want to study it to see if you can spot the problem:

```
static PyObject *
list_append_one_to_four(PyObject *list) {
    for (int i = 400; i < 405; ++i) {
        PyList_Append(list, PyLong_FromLong(i));
    }
    Py_RETURN_NONE;
}
```

The problem is that `PyLong_FromLong` creates `PyObject` (an int) with a reference count of 1 **but** `PyList_Append` increments the reference count of the object passed to it by 1 to 2. This means when the list is destroyed the list element reference counts drop by one (to 1) but *no lower* as nothing else references them. Therefore they never get deallocated so there is a memory leak.

The append operation *must* behave this way, consider this Python code

```
l = []
a = 400
# The integer object '400' has a reference count of 1 as only
# one symbol references it: a.
l.append(a)
# The integer object '400' must now have a reference count of
# 2 as two symbols reference it: a and l, specifically l[-1].
```

The fix is to create a temporary item and then decref *that* once appended (error checking omitted):

```
1  static PyObject *
2  list_append_one_to_four(PyObject *list) {
3      PyObject *temporary_item = NULL;
4
5      for (int i = 400; i < 405; ++i) {
6          /* Create the object to append to the list. */
7          temporary_item = PyLong_FromLong(i);
8          /* temporary_item->ob_refcnt == 1 now */
9          /* Append it. This will increment the reference count to 2. */
10         PyList_Append(list, temporary_item);
11         /* Decrement our reference to it leaving the list having the only reference.␣
   ↪*/
12         Py_DECREF(temporary_item);
13         /* temporary_item->ob_refcnt == 1 now */
14         temporary_item =  NULL; /* Good practice... */
15     }
16     Py_RETURN_NONE;
17 }
```

2.5 Summary

The contracts you enter into with these three reference types are:

Type	Contract
New	Either you decref it or give it to someone who will, otherwise you have a memory leak.
Stolen	The thief will take of things so you don't have to. If you try to the results are undefined.
Borrowed	The lender can invalidate the reference at any time without telling you. Bad news. So increment a borrowed reference whilst you need it and decrement it when you are finished.

```
void do_something(PyObject *pList) {
    while (PyList_Size(pList) > 0) {
        PySequence_DelItem(pList, 0);
    }
}
```

CHAPTER

THREE

EXCEPTION RAISING

A brief interlude on how to communicate error conditions from C code to Python.

These CPython calls are the most useful:

- `PyErr_SetString(...)` - To set an exception type with a fixed string.
- `PyErr_Format(...)` - To set an exception type with a formatted string.
- `PyErr_Occurred()` - To check if an exception has already been set in the flow of control.
- `PyErr_Clear()` - Clearing any set exceptions, have good reason to do this!

Indicating an error condition is a two stage process; your code must register an exception and then indicate failure by returning `NULL`. Here is a C function doing just that:

```
static PyObject *_raise_error(PyObject *module) {

    PyErr_SetString(PyExc_ValueError, "Ooops.");
    return NULL;
}
```

You might want some dynamic information in the exception object, in that case `PyErr_Format` will do:

```
static PyObject *_raise_error_formatted(PyObject *module) {

    PyErr_Format(PyExc_ValueError,
                 "Can not read %d bytes when offset %d in byte length %d.", \
                 12, 25, 32
                 );
    return NULL;
}
```

If one of the two actions is missing then the exception will not be raised correctly. For example returning `NULL` without setting an exception type:

```
/* Illustrate returning NULL but not setting an exception. */
static PyObject *_raise_error_bad(PyObject *module) {
    return NULL;
}
```

Executing this from Python will produce a clear error message (the C function `_raise_error_bad()` is mapped to the Python function `cExcep.raiseErrBad()`

```
>>> cExcep.raiseErrBad()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
SystemError: error return without exception set
```

If the opposite error is made, that is setting an exception but not signalling then the function will succeed but leave a later runtime error:

```
static PyObject *_raise_error_mixup(PyObject *module) {
    PyErr_SetString(PyExc_ValueError, "ERROR: _raise_error_mixup()");
    Py_RETURN_NONE;
}
```

The confusion can arise is that if a subsequent function then tests to see if an exception is set, if so signal it. It will appear that the error is coming from the second function when actually it is from the first:

```
static PyObject *_raise_error_mixup_test(PyObject *module) {
    if (PyErr_Occurred()) {
        return NULL;
    }
    Py_RETURN_NONE;
}
```

The other thing to note is that if there are multiple calls to `PyErr_SetString` only the last one counts:

```
static PyObject *_raise_error_overwrite(PyObject *module) {
    PyErr_SetString(PyExc_RuntimeError, "FORGOTTEN.");
    PyErr_SetString(PyExc_ValueError, "ERROR: _raise_error_overwrite()");
    assert(PyErr_Occurred());
    return NULL;
}
```

3.1 Common Exception Patterns

Here are some common use cases for raising exceptions.

3.1.1 Type Checking

A common requirement is to check the types of the arguments and raise a `TypeError` if they are wrong. Here is an example where we require a `bytes` object:

```
1  static PyObject*
2  function(PyObject *self, PyObject *arg) {
3      /* ... */
4      if (! PyBytes_Check(arg)) {
5          PyErr_Format(PyExc_TypeError,
6                       "Argument \"value\" to %s must be a bytes object not a \"%s\"",
7                       __FUNCTION__, Py_TYPE(arg)->tp_name);
8          goto except;
9      }
10     /* ... */
11 }
```

Thats fine if you have a macro such as `PyBytes_Check` and for your own types you can create a couple of suitable macros:

```
#define PyMyType_CheckExact(op) (Py_TYPE(op) == &PyMyType_Type)
#define PyMyType_Check(op) PyObject_TypeCheck(op, &PyMyType_Type)
```

Incidentially `PyObject_TypeCheck` is defined as:

```
#define PyObject_TypeCheck(ob, tp) \
    (Py_TYPE(ob) == (tp) || PyType_IsSubtype(Py_TYPE(ob), (tp)))
```

3.2 Creating Specialised Excpetions

Often you need to create an Exception class that is specialised to a particular module. This can be done quite easily using either the `PyErr_NewException` or the `PyErr_NewExceptionWithDoc` functions. These create new exception classes that can be added to a module. For example:

```
/* Exception types as static to be initialised during module initialisation. */
static PyObject *ExceptionBase;
static PyObject *SpecialisedError;

/* Standard module initialisation: */
static PyModuleDef noddymodule = {
    PyModuleDef_HEAD_INIT,
    "noddy",
    "Example module that creates an extension type.",
    -1,
    NULL, NULL, NULL, NULL, NULL
};

PyMODINIT_FUNC
PyInit_noddy(void)
{
    PyObject* m;

    noddy_NoddyType.tp_new = PyType_GenericNew;
    if (PyType_Ready(&noddy_NoddyType) < 0)
        return NULL;

    m = PyModule_Create(&noddymodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&noddy_NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);

    /* Initialise exceptions here.
     *
     * Firstly a base class exception that inherits from the builtin Exception.
     * This is acheieved by passing NULL as the PyObject* as the third argument.
     *
     * PyErr_NewExceptionWithDoc returns a new reference.
     */
    ExceptionBase = PyErr_NewExceptionWithDoc(
        "noddy.ExceptionBase", /* char *name */
        "Base exception class for the noddy module.", /* char *doc */
        NULL, /* PyObject *base */
        NULL /* PyObject *dict */);
```

(continues on next page)

(continued from previous page)

```
    /* Error checking: this is oversimplified as it should decref
     * anything created above such as m.
     */
    if (! ExceptionBase) {
        return NULL;
    } else {
        PyModule_AddObject(m, "ExceptionBase", ExceptionBase);
    }
    /* Now a sub-class exception that inherits from the base exception above.
     * This is acheieved by passing non-NULL as the PyObject* as the third argument.
     *
     * PyErr_NewExceptionWithDoc returns a new reference.
     */
    SpecialisedError = PyErr_NewExceptionWithDoc(
        "noddy.SpecialsiedError", /* char *name */
        "Some specialised problem description here.", /* char *doc */
        ExceptionBase, /* PyObject *base */
        NULL /* PyObject *dict */);
    if (! SpecialisedError) {
        return NULL;
    } else {
        PyModule_AddObject(m, "SpecialisedError", SpecialisedError);
    }
    /* END: Initialise exceptions here. */

    return m;
}
```

To illustrate how you raise one of these exceptions suppose we have a function to test raising one of these exceptions:

```
static PyMethodDef Noddy_module_methods[] = {
    ...
    {"_test_raise", (PyCFunction)Noddy__test_raise, METH_NOARGS, "Raises a␣
↪SpecialisedError."},
    ...
    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

We can either access the exception type directly:

```
static PyObject *Noddy__test_raise(PyObject *_mod/* Unused */)
{
    if (SpecialisedError) {
        PyErr_Format(SpecialisedError, "One %d two %d three %d.", 1, 2, 3);
    } else {
        PyErr_SetString(PyExc_RuntimeError, "Can not raise exception, module not␣
↪initialised correctly");
    }
    return NULL;
}
```

Or fish it out of the module (this will be slower):

```
static PyObject *Noddy__test_raise(PyObject *mod)
{
    PyObject *err = PyDict_GetItemString(PyModule_GetDict(mod), "SpecialisedError");
    if (err) {
```

(continues on next page)

(continued from previous page)

```
5           PyErr_Format(err, "One %d two %d three %d.", 1, 2, 3);
6       } else {
7           PyErr_SetString(PyExc_RuntimeError, "Can not find exception in module");
8       }
9       return NULL;
10  }
```

CHAPTER

FOUR

A PYTHONIC CODING PATTERN FOR C FUNCTIONS

To avoid all the errors we have seen it is useful to have a C coding pattern for handling `PyObjects` that does the following:

- No early returns and a single place for clean up code.
- Borrowed references incref'd and decref'd correctly.
- No stale Exception from previous execution path.
- Exceptions set and tested.
- NULL is returned when an exception is set.
- Non-NULL is returned when no exception is set.

The basic pattern in C is similar to Python's try/except/finally pattern:

```
try:
    /* Do fabulous stuff here. */
except:
    /* Handle every abnormal condition and clean up. */
finally:
    /* Clean up under normal conditions and return an appropriate value. */
```

Firstly we set any local `PyObject` (s) and the return value to `NULL`:

```
static PyObject *function(PyObject *arg_1) {
    PyObject *obj_a     = NULL;
    PyObject *ret       = NULL;
```

Then we have a little bit of Pythonic C - this can be omitted:

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

Check that there are no lingering Exceptions:

```
assert(! PyErr_Occurred());
```

An alternative check for no lingering Exceptions:

```
if(PyErr_Occurred()) {
    goto except;
}
```

Now we assume that any argument is a "Borrowed" reference so we increment it (we need a matching `Py_DECREF` before function exit, see below). The first pattern assumes a non-NULL argument.

```
assert(arg_1);
Py_INCREF(arg_1);
```

If you are willing to accept NULL arguments then this pattern would be more suitable:

```
if (arg_1) {
    Py_INCREF(arg_1);
}
```

Of course the same test must be used when calling `Py_DECREF`, or just use `Py_XDECREF`.

Now we create any local objects, if they are "Borrowed" references we need to incref them. With any abnormal behaviour we do a local jump straight to the cleanup code.

```
/* Local object creation. */
/* obj_a = ...; */
if (! obj_a) {
    PyErr_SetString(PyExc_ValueError, "Ooops.");
    goto except;
}
/* If obj_a is a borrowed reference rather than a new reference. */
Py_INCREF(obj_a);
```

Create the return value and deal with abnormal behaviour in the same way:

```
/* More of your code to do stuff with arg_1 and obj_a. */
/* Return object creation, ret should be a new reference otherwise you are in trouble.
↪ */
/* ret = ...; */
if (! ret) {
    PyErr_SetString(PyExc_ValueError, "Ooops again.");
    goto except;
}
```

You might want to check the contents of the return value here. On error jump to `except:` otherwise jump to `finally:`.

```
/* Any return value checking here. */

/* If success then check exception is clear,
 * then goto finally; with non-NULL return value. */
assert(! PyErr_Occurred());
assert(ret);
goto finally;
```

This is the except block where we cleanup any local objects and set the return value to NULL.

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

Notice the `except:` block falls through to the `finally:` block.

```
finally:
    /* All _local_ PyObjects declared at the entry point are Py_XDECREF'd here.
     * For new references this will free them. For borrowed references this
     * will return them to their previous refcount.
     */
    Py_XDECREF(obj_a);
    /* Decrement the ref count of externally supplied the arguments here.
     * If you allow arg_1 == NULL then Py_XDECREF(arg_1). */
    Py_DECREF(arg_1);
    /* And return...*/
    return ret;
}
```

Here is the complete code with minimal comments:

```
static PyObject *function(PyObject *arg_1) {
    PyObject *obj_a    = NULL;
    PyObject *ret      = NULL;

    goto try;
try:
    assert(! PyErr_Occurred());
    assert(arg_1);
    Py_INCREF(arg_1);

    /* obj_a = ...; */
    if (! obj_a) {
        PyErr_SetString(PyExc_ValueError, "Ooops.");
        goto except;
    }
    /* Only do this if obj_a is a borrowed reference. */
    Py_INCREF(obj_a);

    /* More of your code to do stuff with obj_a. */

    /* Return object creation, ret must be a new reference. */
    /* ret = ...; */
    if (! ret) {
        PyErr_SetString(PyExc_ValueError, "Ooops again.");
        goto except;
    }
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    Py_XDECREF(ret);
    assert(PyErr_Occurred());
    ret = NULL;
finally:
    /* Only do this if obj_a is a borrowed reference. */
    Py_XDECREF(obj_a);
    Py_DECREF(arg_1);
    return ret;
}
```

CHAPTER

FIVE

PARSING PYTHON ARGUMENTS

This section describes how you write functions that accept Python `*args` and `**kwargs` arguments and how to extract `PyObject` or C fundamental types from them.

5.1 No Arguments

The simplest from is a global function in a module that takes no arguments at all:

```
static PyObject *_parse_no_args(PyObject *module) {
    PyObject *ret = NULL;

    /* Your code here...*/

    Py_INCREF(Py_None);
    ret = Py_None;
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    Py_XDECREF(ret);
    ret = NULL;
finally:
    return ret;
}
```

This function is added to the module methods with the `METH_NOARGS` value. The Python interpreter will raise a `TypeError` in any arguments are offered.

```
static PyMethodDef cParseArgs_methods[] = {
    /* Other functions here... */
    {"argsNone", (PyCFunction)_parse_no_args, METH_NOARGS,
        "No arguments."
    },
    /* Other functions here... */
    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

5.2 One Argument

There is no parsing required here, a single `PyObject` is expected:

```
static PyObject *_parse_one_arg(PyObject *module,
                                PyObject *arg
                                ) {
    PyObject *ret = NULL;
    assert(arg);
    /* arg as a borrowed reference and the general rule is that you Py_INCREF them
     * whilst you have an interest in them. We do _not_ do that here for reasons
     * explained below.
     */
    // Py_INCREF(arg);

    /* Your code here...*/

    Py_INCREF(Py_None);
    ret = Py_None;
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    Py_XDECREF(ret);
    ret = NULL;
finally:
    /* If we were to treat arg as a borrowed reference and had Py_INCREF'd above we
     * should do this. See below. */
    // Py_DECREF(arg);
    return ret;
}
```

This function can be added to the module with the `METH_O` flag:

```
static PyMethodDef cParseArgs_methods[] = {
    /* Other functions here... */
    {"argsOne", (PyCFunction)_parse_one_arg, METH_O,
        "One argument."
    },
    /* Other functions here... */
    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

5.2.1 Arguments as Borrowed References

There is some subtlety here as indicated by the comments. `*arg` is not our reference, it is a borrowed reference so why don't we increment it at the beginning of this function and decrement it at the end? After all we are trying to protect against calling into some malicious/badly written code that could hurt us. For example:

```
static PyObject *foo(PyObject *module,
                     PyObject *arg
                     ) {
    /* arg has a minimum recount of 1. */
    call_malicious_code_that_decrefs_by_one_this_argument(arg);
    /* arg potentially could have had a ref count of 0 and been deallocated. */
```

(continues on next page)

(continued from previous page)

```
    /* ... */
    /* So now doing something with arg could be undefined. */
}
```

A solution would be, since `arg` is a 'borrowed' reference and borrowed references should always be incremented whilst in use and decremented when done with. This would suggest the following:

```
static PyObject *foo(PyObject *module,
                     PyObject *arg
                     ) {
    /* arg has a minimum recount of 1. */
    Py_INCREF(arg);
    /* arg now has a minimum recount of 2. */
    call_malicious_code_that_decrefs_by_one_this_argument(arg);
    /* arg can not have a ref count of 0 so is safe to use. */
    /* Use arg to your hearts content... */
    /* Do a matching decref. */
    Py_DECREF(arg);
    /* But now arg could have had a ref count of 0 so is unsafe to use by the caller.
 ↪*/
}
```

But now we have just pushed the burden onto our caller. They created `arg` and passed it to us in good faith and whilst we have protected ourselves have not protected the caller and they can fail unexpectedly. So it is best to fail fast, an near the error site, that dastardly `call_malicious_code_that_decrefs_by_one_this_argument()`.

Side note: Of course this does not protect you from malicious/badly written code that decrements by more than one :-)

5.3 Variable Number of Arguments

The function will be called with two arguments, the module and a `PyListObject` that contains a list of arguments. You can either parse this list yourself or use a helper method to parse it into Python and C types.

In the following code we are expecting a string, an integer and an optional integer whose default value is 8. In Python the equivalent function declaration would be:

```
def argsOnly(theString, theInt, theOptInt=8):
```

Here is the C code, note the string that describes the argument types passed to `PyArg_ParseTuple`, if these types are not present a `ValueError` will be set.

```
static PyObject *_parse_args(PyObject *module,
                             PyObject *args
                             ) {
    PyObject *ret = NULL;
    PyObject *pyStr = NULL;
    int arg1, arg2;

    arg2 = 8; /* Default value. */
    if (! PyArg_ParseTuple(args, "Si|i", &pyStr, &arg1, &arg2)) {
        goto except;
    }

    /* Your code here...*/
```

(continues on next page)

(continued from previous page)

```
14
15      Py_INCREF(Py_None);
16      ret = Py_None;
17      assert(! PyErr_Occurred());
18      assert(ret);
19      goto finally;
20  except:
21      Py_XDECREF(ret);
22      ret = NULL;
23  finally:
24      return ret;
25  }
```

This function can be added to the module with the `METH_VARARGS` flag:

```
static PyMethodDef cParseArgs_methods[] = {
    /* Other functions here... */
    {"argsOnly", (PyCFunction)_parse_args, METH_VARARGS,
        "Reads args only."
    },
    /* Other functions here... */
    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

5.4 Variable Number of Arguments and Keyword Arguments

The function will be called with two arguments, the module, a `PyListObject` that contains a list of arguments and a `PyDictObject` that contains a dictionary of keyword arguments. You can either parse these yourself or use a helper method to parse it into Python and C types.

In the following code we are expecting a string, an integer and an optional integer whose default value is 8. In Python the equivalent function declaration would be:

```
def argsKwargs(theString, theOptInt=8):
```

Here is the C code, note the string that describes the argument types passed to `PyArg_ParseTuple`, if these types are not present a `ValueError` will be set.

```
1  static PyObject *_parse_args_kwargs(PyObject *module,
2                                      PyObject *args,
3                                      PyObject *kwargs
4                                      ) {
5      PyObject *ret = NULL;
6      PyObject *pyStr = NULL;
7      int arg2;
8      static char *kwlist[] = {
9          "theString",
10         "theOptInt",
11         NULL
12     };
13
14     /* If you are interested this is a way that you can trace the input.
15     PyObject_Print(module, stdout, 0);
16     fprintf(stdout, "\n");
```

(continues on next page)

(continued from previous page)

```
    PyObject_Print(args, stdout, 0);
    fprintf(stdout, "\n");
    PyObject_Print(kwargs, stdout, 0);
    fprintf(stdout, "\n");
     * End trace */

    arg2 = 8; /* Default value. */
    if (! PyArg_ParseTupleAndKeywords(args, kwargs, "S|i",
                                      kwlist, &pyStr, &arg2)) {
        goto except;
    }

    /* Your code here...*/

    Py_INCREF(Py_None);
    ret = Py_None;
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    Py_XDECREF(ret);
    ret = NULL;
finally:
    return ret;
}
```

This function can be added to the module with the `METH_VARARGS` and `METH_KEYWORDS` flags:

```
static PyMethodDef cParseArgs_methods[] = {
    /* Other functions here... */
    {"argsKwargs", (PyCFunction)_parse_args_kwargs,
        METH_VARARGS | METH_KEYWORDS,
        _parse_args_kwargs_docstring
    },
    /* Other functions here... */
    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

All arguments are keyword arguments so this function can be called in a number of ways, all of the following are equivalent:

```
argsKwargs('foo')
argsKwargs('foo', 8)
argsKwargs(theString='foo')
argsKwargs(theOptInt=8, theString='foo')
argsKwargs(theString, theOptInt=8)
```

If you want the function signature to be `argsKwargs(theString, theOptInt=8)` with a single argument and a single optional keyword argument then put an empty string in the kwlist array:

```
/* ... */
static char *kwlist[] = {
    "",
    "theOptInt",
    NULL
};
/* ... */
```

Note: If you use `|` in the parser format string you have to set the default values for those optional arguments yourself in the C code. This is pretty straightforward if they are fundamental C types as `arg2 = 8` above. For Python values is a bit more tricky as described next.

5.4.1 Keyword Arguments and C++11

C++11 compilers warn when creating non-const `char*` from string literals as we have done with the keyword array above. The solution is to declare these `const char*` however `PyArg_ParseTupleAndKeywords` expects a `char **`. The solution is to cast away const in the call:

```
/* ... */
static const char *kwlist[] = { "foo", "bar", "baz", NULL };
if (! PyArg_ParseTupleAndKeywords(args, kwds, "OOO",
                                  const_cast<char**>(kwlist),
                                  &foo, &bar, &baz)) {
    return NULL;
}
/* ... */
```

5.5 Being Pythonic with Default Arguments

If the arguments default to some C fundamental type the code above is fine. However if the arguments default to Python objects then a little more work is needed. Here is a function that has a tuple and a dict as default arguments, in other words the Python signature:

```
def function(arg_0=(42, "this"), arg_1={}):
```

The first argument is immutable, the second is mutable and so we need to mimic the well known behaviour of Python with mutable arguments. Mutable default arguments are evaluated once only at function definition time and then becomes a (mutable) property of the function. For example:

```
>>> def f(l=[]):
...   l.append(9)
...   print(l)
...
>>> f()
[9]
>>> f()
[9, 9]
>>> f([])
[9]
>>> f()
[9, 9, 9]
```

In C we can get this behaviour by treating the mutable argument as `static`, the immutable argument does not need to be `static` but it will do no harm if it is (if non-`static` it will have to be initialised on every function call).

My advice: Always make all `PyObject*` references to default arguments `static`.

So first we declare a `static PyObject*` for each default argument:

```
static PyObject *_parse_args_with_python_defaults(PyObject *module, PyObject *args) {
    PyObject *ret = NULL;

    /* This first pointer need not be static as the argument is immutable
     * but if non-static must be NULL otherwise the following code will be undefined.
     */
    static PyObject *pyObjDefaultArg_0;
    static PyObject *pyObjDefaultArg_1; /* Must be static if mutable. */
```

Then we declare a `PyObject*` for each argument that will either reference the default or the passed in argument. It is important that these `pyObjArg_...` pointers are NULL so that we can subsequently detect if `PyArg_ParseTuple` has set them non-`NULL`.

```
/* These 'working' pointers are the ones we use in the body of the function
 * They either reference the supplied argument or the default (static) argument.
 * We must treat these as "borrowed" references and so must incref them
 * while they are in use then decref them when we exit the function.
 */
PyObject *pyObjArg_0 = NULL;
PyObject *pyObjArg_1 = NULL;
```

Then, if the default values have not been initialised, initialise them. In this case it is a bit tedious merely because of the nature of the arguments. So in practice this might be clearer if this was in separate function:

```
1  /* Initialise first argument to its default Python value. */
2  if (! pyObjDefaultArg_0) {
3      pyObjDefaultArg_0 = PyTuple_New(2);
4      if (! pyObjDefaultArg_0) {
5          PyErr_SetString(PyExc_RuntimeError, "Can not create tuple!");
6          goto except;
7      }
8      if(PyTuple_SetItem(pyObjDefaultArg_0, 0, PyLong_FromLong(42))) {
9          PyErr_SetString(PyExc_RuntimeError, "Can not set tuple[0]!");
10         goto except;
11     }
12     if(PyTuple_SetItem(pyObjDefaultArg_0, 1, PyUnicode_FromString("This"))) {
13         PyErr_SetString(PyExc_RuntimeError, "Can not set tuple[1]!");
14         goto except;
15     }
16 }
17 /* Now the second argument. */
18 if (! pyObjDefaultArg_1) {
19     pyObjDefaultArg_1 = PyDict_New();
20 }
```

Now parse the given arguments to see what, if anything, is there. `PyArg_ParseTuple` will set each working pointer non-`NULL` if the argument is present. As we set the working pointers `NULL` prior to this call we can now tell if any argument is present.

```
if (! PyArg_ParseTuple(args, "|OO", &pyObjArg_0, &pyObjArg_1)) {
    goto except;
}
```

Now switch our working pointers to the default argument if no argument is given. We also treat these as "borrowed" references regardless of whether they are default or supplied so increment the refcount (we must decrement the refcount when done).

```
/* First argument. */
if (! pyObjArg_0) {
    pyObjArg_0 = pyObjDefaultArg_0;
}
Py_INCREF(pyObjArg_0);

/* Second argument. */
if (! pyObjArg_1) {
    pyObjArg_1 = pyObjDefaultArg_1;
}
Py_INCREF(pyObjArg_1);
```

Now write the main body of your function and that must be followed by this clean up code:

```
/* Your code here using pyObjArg_0 and pyObjArg_1 ...*/

Py_INCREF(Py_None);
ret = Py_None;
assert(! PyErr_Occurred());
assert(ret);
goto finally;
```

Now the two blocks `except` and `finally`.

```
except:
    assert(PyErr_Occurred());
    Py_XDECREF(ret);
    ret = NULL;
finally:
    /* Decrement refcount to match the increment above. */
    Py_XDECREF(pyObjArg_0);
    Py_XDECREF(pyObjArg_1);
    return ret;
}
```

An important point here is the use of `Py_XDECREF` in the `finally:` block, we can get here through a number of paths, including through the `except:` block and in some cases the `pyObjArg_...` will be `NULL` (for example if `PyArg_ParseTuple` fails). So `Py_XDECREF` it must be.

Here is the complete C code:

```
static PyObject *_parse_args_with_python_defaults(PyObject *module, PyObject *args) {
    PyObject *ret = NULL;
    static PyObject *pyObjDefaultArg_0;
    static PyObject *pyObjDefaultArg_1;
    PyObject *pyObjArg_0 = NULL;
    PyObject *pyObjArg_1 = NULL;

    if (! pyObjDefaultArg_0) {
        pyObjDefaultArg_0 = PyTuple_New(2);
        if (! pyObjDefaultArg_0) {
            PyErr_SetString(PyExc_RuntimeError, "Can not create tuple!");
            goto except;
        }
        if(PyTuple_SetItem(pyObjDefaultArg_0, 0, PyLong_FromLong(42))) {
            PyErr_SetString(PyExc_RuntimeError, "Can not set tuple[0]!");
            goto except;
```

(continues on next page)

(continued from previous page)

```
        }
        if(PyTuple_SetItem(pyObjDefaultArg_0, 1, PyUnicode_FromString("This"))) {
            PyErr_SetString(PyExc_RuntimeError, "Can not set tuple[1]!");
            goto except;
        }
    }
    if (! pyObjDefaultArg_1) {
        pyObjDefaultArg_1 = PyDict_New();
    }

    if (! PyArg_ParseTuple(args, "|OO", &pyObjArg_0, &pyObjArg_1)) {
        goto except;
    }
    if (! pyObjArg_0) {
        pyObjArg_0 = pyObjDefaultArg_0;
    }
    Py_INCREF(pyObjArg_0);
    if (! pyObjArg_1) {
        pyObjArg_1 = pyObjDefaultArg_1;
    }
    Py_INCREF(pyObjArg_1);

    /* Your code here...*/

    Py_INCREF(Py_None);
    ret = Py_None;
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    assert(PyErr_Occurred());
    Py_XDECREF(ret);
    ret = NULL;
finally:
    Py_XDECREF(pyObjArg_0);
    Py_XDECREF(pyObjArg_1);
    return ret;
}
```

5.5.1 Simplifying Macros

For simple default values some macros may help. The first one declares and initialises the default value. It takes three arguments:

- The name of the argument variable, a static `PyObject` named `default_<name>` will also be created.
- The default value which should return a new reference.
- The value to return on failure to create a default value, usually -1 or `NULL`.

```
#define PY_DEFAULT_ARGUMENT_INIT(name, value, ret) \
    PyObject *name = NULL; \
    static PyObject *default_##name = NULL; \
    if (! default_##name) { \
        default_##name = value; \
        if (! default_##name) { \
```

(continues on next page)

(continued from previous page)

```
            PyErr_SetString(PyExc_RuntimeError, "Can not create default value for "
↪#name); \
            return ret; \
        } \
    }
```

The second one assigns the argument to the default if it is not initialised and increments the reference count. It just takes the name of the argument:

```
#define PY_DEFAULT_ARGUMENT_SET(name) if (! name) name = default_##name; \
    Py_INCREF(name)
```

And they can be used like this when implementing a Python function signature such as:

```
def do_something(self, encoding='utf-8', the_id=0, must_log=True):
    # ...
    return None
```

Here is that function implemented in C:

```
static PyObject*
do_something(something *self, PyObject *args, PyObject *kwds) {
    PyObject *ret = NULL;
    /* Initialise default arguments. Note: these might cause an early return. */
    PY_DEFAULT_ARGUMENT_INIT(encoding,  PyUnicode_FromString("utf-8"),  NULL);
    PY_DEFAULT_ARGUMENT_INIT(the_id,    PyLong_FromLong(0L),            NULL);
    PY_DEFAULT_ARGUMENT_INIT(must_log,  PyBool_FromLong(1L),            NULL);

    static const char *kwlist[] = { "encoding", "the_id", "must_log", NULL };
    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|Oip",
                                      const_cast<char**>(kwlist),
                                      &encoding, &the_id, &must_log)) {
        return NULL;
    }
    /*
     * Assign absent arguments to defaults and increment the reference count.
     * Don't forget to decrement the reference count before returning!
     */
    PY_DEFAULT_ARGUMENT_SET(encoding);
    PY_DEFAULT_ARGUMENT_SET(the_id);
    PY_DEFAULT_ARGUMENT_SET(must_log);

    /*
     * Use encoding, the_id, must_log from here on...
     */

    Py_INCREF(Py_None);
    ret = Py_None;
    assert(! PyErr_Occurred());
    assert(ret);
    goto finally;
except:
    assert(PyErr_Occurred());
    Py_XDECREF(ret);
    ret = NULL;
finally:
```

(continues on next page)

(continued from previous page)

```
    Py_DECREF(encoding);
    Py_DECREF(the_id);
    Py_DECREF(must_log);
    return ret;
}
```

5.5.2 Simplifying C++11 class

With C++ we can make this a bit smoother. We declare a class thus:

```
/** Class to simplify default arguments.
 *
 * Usage:
 *
 * static DefaultArg arg_0(PyLong_FromLong(1L));
 * static DefaultArg arg_1(PyUnicode_FromString("Default string."));
 * if (! arg_0 || ! arg_1) {
 *      return NULL;
 * }
 *
 * if (! PyArg_ParseTupleAndKeywords(args, kwargs, "...",
                                     const_cast<char**>(kwlist),
                                     &arg_0, &arg_1, ...)) {
        return NULL;
    }
 *
 * Then just use arg_0, arg_1 as if they were a PyObject* (possibly
 * might need to be cast to some specific PyObject*).
 *
 * WARN: This class is designed to be statically allocated. If allocated
 * on the heap or stack it will leak memory. That could be fixed by
 * implementing:
 *
 * ~DefaultArg() { Py_XDECREF(m_default); }
 *
 * But this will be highly dangerous when statically allocated as the
 * destructor will be invoked with the Python interpreter in an
 * uncertain state and will, most likely, segfault:
 * "Python(39158,0x7fff78b66310) malloc: *** error for object 0x100511300: pointer␣
↪being freed was not allocated"
 */
class DefaultArg {
public:
    DefaultArg(PyObject *new_ref) : m_arg { NULL }, m_default { new_ref } {}
    // Allow setting of the (optional) argument with
    // PyArg_ParseTupleAndKeywords
    PyObject **operator&() { m_arg = NULL; return &m_arg; }
    // Access the argument or the default if default.
    operator PyObject*() const {
        return m_arg ? m_arg : m_default;
    }
    // Test if constructed successfully from the new reference.
    explicit operator bool() { return m_default != NULL; }
protected:
    PyObject *m_arg;
```

(continues on next page)

(continued from previous page)

```
    PyObject *m_default;
};
```

And we can use `DefaultArg` like this:

```
static PyObject*
do_something(something *self, PyObject *args, PyObject *kwds) {
    PyObject *ret = NULL;
    /* Initialise default arguments. */
    static DefaultArg encoding { PyUnicode_FromString("utf-8") };
    static DefaultArg the_id { PyLong_FromLong(0L) };
    static DefaultArg must_log { PyBool_FromLong(1L) };

    /* Check that the defaults are non-NULL i.e. succesful. */
    if (!encoding || !the_id || !must_log) {
        return NULL;
    }

    static const char *kwlist[] = { "encoding", "the_id", "must_log", NULL };
    /* &encoding etc. accesses &m_arg in DefaultArg because of PyObject **operator&() */
    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|Oip",
                                      const_cast<char**>(kwlist),
                                      &encoding, &the_id, &must_log)) {
        return NULL;
    }
    /*
     * Use encoding, the_id, must_log from here on as PyObject* since we have
     * operator PyObject*() const ...
     *
     * So if we have a function:
     * set_encoding(PyObject *obj) { ... }
     */
     set_encoding(encoding);
    /* ... */
}
```

CHAPTER

SIX

CREATING NEW TYPES

The creation of new extension types (AKA 'classes') is pretty well described in the Python documentation tutorial and reference. This section just describes a rag bag of tricks and examples.

6.1 Properties

6.1.1 Referencing Existing Properties

If the property is part of the extension type then it is fairly easy to make it directly accessible as described here

For example the `Noddy` struct has a Python object (a string) and a C object (an int):

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    /* ... */
    int number;
} Noddy;
```

These can be exposed by identifying them as members with an array of `PyMemberDef` like this:

```
static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    /* ... */
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};
```

And the type struct must reference this array of `PyMemberDef` thus:

```
static PyTypeObject NoddyType = {
    /* ... */
    Noddy_members,              /* tp_members */
    /* ... */
};
```

Reference to PyMemberdef.

6.1.2 Created Properties

If the properties are not directly accessible, for example they might need to be created, then an array of `PyGetSetDef` structures is used in the PyTypeObject.tp_getset slot.

```
static PyObject*
Foo_property_getter(Foo* self, void * /* closure */) {
    return /* ... */;
}

int
Foo_property_setter(Foo* self, PyObject *value) {
    return /* 0 on success, -1 on failure with error set. */;
}

static PyGetSetDef Foo_properties[] = {
    {"id", (getter) Foo_property_getter, (setter) Foo_property_setter,
        "The property documentation.", NULL },
    {NULL}  /* Sentinel */
};
```

And the type struct must reference this array of `PyMemberDef` thus:

```
static PyTypeObject FooType = {
    /* ... */
    Foo_properties,             /* tp_getset */
    /* ... */
};
```

Reference to PyGetSetDef.

CHAPTER

SEVEN

SETTING AND GETTING MODULE GLOBALS

This section describes how you create and access module globals from Python C Extensions.

In this module, written as a Python extension in C, we are going to have a string, int, list, tuple and dict in global scope. In the C code we firstly define names for them:

```
const char *NAME_INT = "INT";
const char *NAME_STR = "STR";
const char *NAME_LST = "LST";
const char *NAME_TUP = "TUP";
const char *NAME_MAP = "MAP";
```

These are the names of the objects that will appear in the Python module:

```
>>> import cModuleGlobals
>>> dir(cModuleGlobals)
['INT', 'LST', 'MAP', 'STR', 'TUP', '__doc__', '__file__', '__loader__', '__name__',
↪'__package__', 'print']
```

7.1 Initialising Module Globals

This is the module declaration, it will be called `cModuleGlobals` and has just one function; `print()` that will access the module globals from C:

```
1  static PyMethodDef cModuleGlobals_methods[] = {
2      {"print", (PyCFunction)_print_globals, METH_NOARGS,
3          "Access and print out th globals."
4      },
5      {NULL, NULL, 0, NULL}  /* Sentinel */
6  };
7
8
9  static PyModuleDef cModuleGlobals_module = {
10     PyModuleDef_HEAD_INIT,
11     "cModuleGlobals",
12     "Examples of global values in a module.",
13     -1,
14     cModuleGlobals_methods, /* cModuleGlobals_methods */
15     NULL, /* inquiry m_reload */
16     NULL, /* traverseproc m_traverse */
17     NULL, /* inquiry m_clear */
18     NULL, /* freefunc m_free */
19 };
```

The module initialisation code is next, this uses the Python C API to create the various global objects:

```
PyMODINIT_FUNC
PyInit_cModuleGlobals(void)
{
    PyObject *m = NULL;

    m = PyModule_Create(&cModuleGlobals_module);

    if (m == NULL) {
        goto except;
    }
    /* Adding module globals */
    if (PyModule_AddIntConstant(m, NAME_INT, 42)) {
        goto except;
    }
    if (PyModule_AddStringConstant(m, NAME_STR, "String value")) {
        goto except;
    }
    if (PyModule_AddObject(m, NAME_TUP, Py_BuildValue("iii", 66, 68, 73))) {
        goto except;
    }
    if (PyModule_AddObject(m, NAME_LST, Py_BuildValue("[iii]", 66, 68, 73))) {
        goto except;
    }
    /* An invented convenience function for this dict. */
    if (_add_map_to_module(m)) {
        goto except;
    }
    goto finally;
except:
    Py_XDECREF(m);
    m = NULL;
finally:
    return m;
}
```

The dict is added in a separate C function merely for readability:

```
/* Add a dict of {str : int, ...}.
 * Returns 0 on success, 1 on failure.
 */
int _add_map_to_module(PyObject *module) {
    int ret = 0;
    PyObject *pMap = NULL;

    pMap = PyDict_New();
    if (! pMap) {
        goto except;
    }
    /* Load map. */
    if (PyDict_SetItem(pMap, PyBytes_FromString("66"), PyLong_FromLong(66))) {
        goto except;
    }
    if (PyDict_SetItem(pMap, PyBytes_FromString("123"), PyLong_FromLong(123))) {
        goto except;
    }
    /* Add map to module. */
```

(continues on next page)

(continued from previous page)

```
    if (PyModule_AddObject(module, NAME_MAP, pMap)) {
        goto except;
    }
    ret = 0;
    goto finally;
except:
    Py_XDECREF(pMap);
    ret = 1;
finally:
    return ret;
}
```

7.2 Getting and Setting Module Globals

7.2.1 From Python

Once the module is built we can access the globals from Python as usual:

```
>>> import cModuleGlobals
>>> dir(cModuleGlobals)
['INT', 'LST', 'MAP', 'STR', 'TUP', '__doc__', '__file__', '__loader__', '__name__',
↪'__package__', 'print']
>>> cModuleGlobals.STR
'String value'
>>> cModuleGlobals.STR = 'F'
>>> cModuleGlobals.STR
'F'
>>> cModuleGlobals.MAP
{b'123': 123, b'66': 66}
>>> cModuleGlobals.MAP[b'asd'] = 9
>>> cModuleGlobals.MAP
{b'123': 123, b'asd': 9, b'66': 66}
```

7.2.2 Getting Module Globals From C

Accessing Python module globals from C is a little bit more tedious as we are getting borrowed references from the modules `__dict__` and we should increment and decrement them appropriately. Here we print out the global `INT` as both a Python object and a 'C' `long`:

```
static PyObject *_print_global_INT(PyObject *pMod) {
    PyObject *ret = NULL;
    PyObject *pItem = NULL;
    long val;

    /* Sanity check. */
    assert(pMod);
    assert(PyModule_CheckExact(pMod));
    assert(! PyErr_Occurred());

    /* NOTE: PyModule_GetDict(pMod); never fails and returns a borrowed
     * reference. pItem is NULL or a borrowed reference.
     */
```

(continues on next page)

(continued from previous page)

```
    pItem = PyDict_GetItemString(PyModule_GetDict(pMod), NAME_INT);
    if (! pItem) {
        PyErr_Format(PyExc_AttributeError,
                     "Module '%s' has no attibute '%s'.", \
                     PyModule_GetName(pMod), NAME_INT
                     );
        goto except;
    }
    Py_INCREF(pItem);
    fprintf(stdout, "Integer: \"%s\" ", NAME_INT);
    PyObject_Print(pItem, stdout, 0);
    val = PyLong_AsLong(pItem);
    fprintf(stdout, " C long: %ld ", val);
    fprintf(stdout, "\n");

    assert(! PyErr_Occurred());
    Py_INCREF(Py_None);
    ret = Py_None;
    goto finally;
except:
    assert(PyErr_Occurred());
    Py_XDECREF(ret);
    ret = NULL;
finally:
    Py_DECREF(pItem);
    return ret;
}
```

From Python we would see this (C's `_print_global_INT()` is mapped to Python's `cModuleGlobals.printINT()`):

```
>>> import cModuleGlobals
>>> cModuleGlobals.printINT()
Module:
<module 'cModuleGlobals' from './cModuleGlobals.so'>
Integer: "INT" 42 C long: 42
```

7.2.3 Setting Module Globals From C

This is similar to the get code above but using `int PyDict_SetItemString(PyObject *p, const char *key, PyObject *val)` where val will be a *stolen* reference:

```
static PyObject *some_set_function(PyObject *pMod) {
    PyObject *ret = NULL;
    long val = ...; /* Some computed value. */

    if (PyDict_SetItemString(PyModule_GetDict(pMod), NAME_INT, PyLong_FromLong(val)))
↪{
        PyErr_Format(PyExc_AttributeError,
                     "Can not set Module '%s' attibute '%s'.", \
                     PyModule_GetName(pMod), NAME_INT
                     );
        goto except;
    }

```

(continues on next page)

(continued from previous page)

```
    assert(! PyErr_Occurred());
    Py_INCREF(Py_None);
    ret = Py_None;
    goto finally;
except:
    assert(PyErr_Occurred());
    Py_XDECREF(ret);
    ret = NULL;
finally:
    return ret;
}
```

CHAPTER

EIGHT

CALLING `SUPER()` FROM C

I needed to call super() from a C extension and I couldn't find a good description of how to do this online so I am including this here.

TODO: This code is specific to Python 3, add Python 2 support.

Suppose we wanted to subclass a list and record how many times `append()` was called. This is simple enough in pure Python:

```
class SubList(list):
    def __init__(self, *args, **kwargs):
        self.appends = 0
        super().__init__(*args, **kwargs)

    def append(self, v):
        self.appends += 1
        return super().append(v)
```

To do it in C is a bit trickier. Taking as our starting point the example of sub-classing a list in the Python documentation, amended a little bit for our example.

Our type contains an integer count of the number of appends. That is set to zero on construction and can be accesssed like a normal member.

```
typedef struct {
    PyListObject list;
    int appends;
} Shoddy;


static int
Shoddy_init(Shoddy *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *)self, args, kwds) < 0) {
        return -1;
    }
    self->appends = 0;
    return 0;
}

static PyMemberDef Shoddy_members[] = {
    ...
    {"appends", T_INT, offsetof(Shoddy, appends), 0,
        "Number of append operations."},
    ...
```

(continues on next page)

(continued from previous page)

```
    {NULL, 0, 0, 0, NULL}  /* Sentinel */
};
```

We now need to create the `append()` function, this function will call the superclass `append()` and increment the `appends` counter:

```
static PyMethodDef Shoddy_methods[] = {
    ...
    {"append", (PyCFunction)Shoddy_append, METH_VARARGS,
        PyDoc_STR("Append to the list")},
    ...
    {NULL,  NULL, 0, NULL},
};
```

This is where it gets tricky, how do we implement `Shoddy_append`?

8.1 The Obvious Way is Wrong

A first attempt might do something like a method call on the `PyListObject`:

```
typedef struct {
    PyListObject list;
    int appends;
} Shoddy;

/* Other stuff here. */

static PyObject *
Shoddy_append(Shoddy *self, PyObject *args) {
    PyObject *result = PyObject_CallMethod((PyObject *)&self->list, "append", "O",␣
↪args);
    if (result) {
        self->appends++;
    }
    return result;
}
```

This leads to infinite recursion as the address of the first element of a C struct (`list`) is the address of the struct so `self` is the same as `&self->list`. This function is recursive with no base case.

8.2 Doing it Right

Our append method needs to use *super* to search our super-classes for the "append" method and call that.

Here are a couple of ways of calling `super()` correctly:

- Construct a `super` object directly and call that.
- Extract the `super` object from the builtins module and call that.

8.2.1 Construct a `super` object directly

The plan is to do this:

- Create the arguments to initialise an instance of the class `super`.
- Call `super.__new__` with those arguments.
- Call `super.__init__` with those arguments.
- With that `super` object then search for the method we want to call. This is `append` in our case. This calls the `super_getattro` method that performs the search and returns the Python function.
- Call that Python function and return the result.

Our function is defined thus, for simplicity there is no error checking here. For the full function see below:

```
PyObject *
call_super_pyname(PyObject *self, PyObject *func_name, PyObject *args, PyObject␣
↪*kwargs) {
    PyObject *super      = NULL;
    PyObject *super_args = NULL;
    PyObject *func       = NULL;
    PyObject *result     = NULL;

    // Create the arguments for super()
    super_args = PyTuple_New(2);
    Py_INCREF(self->ob_type); // Py_INCREF(&ShoddyType); in our specific case
    PyTuple_SetItem(super_args, 0, (PyObject*)self->ob_type)); // PyTuple_
↪SetItem(super_args, 0, (PyObject*)&ShoddyType) in our specific case
    Py_INCREF(self);
    PyTuple_SetItem(super_args, 1, self));
    // Creat the class super()
    super = PyType_GenericNew(&PySuper_Type, super_args, NULL);
    // Instantiate it with the tuple as first arg, no kwargs passed to super() so NULL
    super->ob_type->tp_init(super, super_args, NULL);
    // Use super to find the 'append' method
    func = PyObject_GetAttr(super, func_name);
    // Call that method
    result = PyObject_Call(func, args, kwargs);
    Py_XDECREF(super);
    Py_XDECREF(super_args);
    Py_XDECREF(func);
    return result;
}
```

We can make this function quite general to be used in the CPython type system. For convenience we can create two functions, one calls the super function by a C NTS, the other by a PyObject string. The following code is essentially the same as above but with error checking.

The header file might be py_call_super.h which just declares our two functions:

```
#ifndef __PythonSubclassList__py_call_super__
#define __PythonSubclassList__py_call_super__

#include <Python.h>

extern PyObject *
call_super_pyname(PyObject *self, PyObject *func_name,
                  PyObject *args, PyObject *kwargs);
```

(continues on next page)

(continued from previous page)

```
extern PyObject *
call_super_name(PyObject *self, const char *func_cname,
                PyObject *args, PyObject *kwargs);

#endif /* defined(__PythonSubclassList__py_call_super__) */
```

And the implementation file would be py_call_super.c, this is the code above with full error checking:

```
PyObject *
call_super_pyname(PyObject *self, PyObject *func_name,
                  PyObject *args, PyObject *kwargs) {
    PyObject *super      = NULL;
    PyObject *super_args = NULL;
    PyObject *func       = NULL;
    PyObject *result     = NULL;

    if (! PyUnicode_Check(func_name)) {
        PyErr_Format(PyExc_TypeError,
                     "super() must be called with unicode attribute not %s",
                     func_name->ob_type->tp_name);
    }

    super_args = PyTuple_New(2);
    //    Py_INCREF(&ShoddyType);
    Py_INCREF(self->ob_type);
    //    if (PyTuple_SetItem(super_args, 0, (PyObject*)&ShoddyType)) {
    if (PyTuple_SetItem(super_args, 0, (PyObject*)self->ob_type)) {
        assert(PyErr_Occurred());
        goto except;
    }
    Py_INCREF(self);
    if (PyTuple_SetItem(super_args, 1, self)) {
        assert(PyErr_Occurred());
        goto except;
    }

    super = PyType_GenericNew(&PySuper_Type, super_args, NULL);
    if (! super) {
        PyErr_SetString(PyExc_RuntimeError, "Could not create super().");
        goto except;
    }
    // Make tuple as first arg, second arg (i.e. kwargs) should be NULL
    super->ob_type->tp_init(super, super_args, NULL);
    if (PyErr_Occurred()) {
        goto except;
    }
    func = PyObject_GetAttr(super, func_name);
    if (! func) {
        assert(PyErr_Occurred());
        goto except;
    }
    if (! PyCallable_Check(func)) {
        PyErr_Format(PyExc_AttributeError,
                     "super() attribute \"%S\" is not callable.", func_name);
        goto except;
    }
    result = PyObject_Call(func, args, kwargs);
```

(continues on next page)

(continued from previous page)

```
    assert(! PyErr_Occurred());
    goto finally;
except:
    assert(PyErr_Occurred());
    Py_XDECREF(result);
    result = NULL;
finally:
    Py_XDECREF(super);
    Py_XDECREF(super_args);
    Py_XDECREF(func);
    return result;
}
```

8.2.2 Extract the `super` object from the builtins

Another way to do this is to fish out the *super* class from the builtins module and use that. Incidentially this is how Cython does it.

The steps are:

1. Get the *builtins* module.
2. Get the *super* class from the *builtins* module.
3. Create a tuple of the arguments to pass to the super class.
4. Create the *super* object with the arguments.
5. Use this *super* object to call the function with the appropriate function arguments.

Again this code has no error checking for simplicity:

```
extern PyObject *
call_super_pyname_lookup(PyObject *self, PyObject *func_name,
                         PyObject *args, PyObject *kwargs) {
    PyObject *builtins = PyImport_AddModule("builtins");
    // Borrowed reference
    Py_INCREF(builtins);
    PyObject *super_type = PyObject_GetAttrString(builtins, "super");
    PyObject *super_args = PyTuple_New(2);
    Py_INCREF(self->ob_type);
    PyTuple_SetItem(super_args, 0, (PyObject*)self->ob_type);
    Py_INCREF(self);
    PyTuple_SetItem(super_args, 1, self);
    PyObject *super = PyObject_Call(super_type, super_args, NULL);
    PyObject *func = PyObject_GetAttr(super, func_name);
    PyObject *result = PyObject_Call(func, args, kwargs);
    Py_XDECREF(builtins);
    Py_XDECREF(super_args);
    Py_XDECREF(super_type);
    Py_XDECREF(super);
    Py_XDECREF(func);
    return result;
}
```

Here is the function with full error checking:

```
extern PyObject *
call_super_pyname_lookup(PyObject *self, PyObject *func_name,
                         PyObject *args, PyObject *kwargs) {
    PyObject *result        = NULL;
    PyObject *builtins      = NULL;
    PyObject *super_type    = NULL;
    PyObject *super         = NULL;
    PyObject *super_args    = NULL;
    PyObject *func          = NULL;

    builtins = PyImport_AddModule("builtins");
    if (! builtins) {
        assert(PyErr_Occurred());
        goto except;
    }
    // Borrowed reference
    Py_INCREF(builtins);
    super_type = PyObject_GetAttrString(builtins, "super");
    if (! super_type) {
        assert(PyErr_Occurred());
        goto except;
    }
    super_args = PyTuple_New(2);
    Py_INCREF(self->ob_type);
    if (PyTuple_SetItem(super_args, 0, (PyObject*)self->ob_type)) {
        assert(PyErr_Occurred());
        goto except;
    }
    Py_INCREF(self);
    if (PyTuple_SetItem(super_args, 1, self)) {
        assert(PyErr_Occurred());
        goto except;
    }
    super = PyObject_Call(super_type, super_args, NULL);
    if (! super) {
        assert(PyErr_Occurred());
        goto except;
    }
    func = PyObject_GetAttr(super, func_name);
    if (! func) {
        assert(PyErr_Occurred());
        goto except;
    }
    if (! PyCallable_Check(func)) {
        PyErr_Format(PyExc_AttributeError,
                     "super() attribute \"%S\" is not callable.", func_name);
        goto except;
    }
    result = PyObject_Call(func, args, kwargs);
    assert(! PyErr_Occurred());
    goto finally;
except:
    assert(PyErr_Occurred());
    Py_XDECREF(result);
    result = NULL;
finally:
    Py_XDECREF(builtins);
```

(continues on next page)

(continued from previous page)

```
58      Py_XDECREF(super_args);
59      Py_XDECREF(super_type);
60      Py_XDECREF(super);
61      Py_XDECREF(func);
62      return result;
63  }
```

CHAPTER

NINE

SETTING COMPILER FLAGS

It is sometimes difficult to decide what flags to set for the compiler and the best advice is to use the same flags that the version of Python you are using was compiled with. Here are a couple of ways to do that.

9.1 From the Command Line

In the Python install directory there is a *pythonX.Y-config* executable that can be used to extract the compiler flags where X is the major version and Y the minor version. For example (output is wrapped here for clarity):

```
$ which python
/usr/bin/python
$ python -V
Python 2.7.5
$ /usr/bin/python2.7-config --cflags
-I/System/Library/Frameworks/Python.framework/Versions/2.7/include/python2.7
-I/System/Library/Frameworks/Python.framework/Versions/2.7/include/python2.7
-fno-strict-aliasing -fno-common -dynamic -arch x86_64 -arch i386 -g -Os -pipe
-fno-common -fno-strict-aliasing -fwrapv -DENABLE_DTRACE -DMACOSX -DNDEBUG -Wall
-Wstrict-prototypes -Wshorten-64-to-32 -DNDEBUG -g -fwrapv -Os -Wall
-Wstrict-prototypes -DENABLE_DTRACE
```

9.2 Programatically from Within a Python Process

The `sysconfig` module contains information about the build environment for the particular version of Python:

```
>>> import sysconfig
>>> sysconfig.get_config_var('CFLAGS')
'-fno-strict-aliasing -fno-common -dynamic -arch x86_64 -arch i386 -g -Os -pipe -fno-
↪common -fno-strict-aliasing -fwrapv -DENABLE_DTRACE -DMACOSX -DNDEBUG -Wall -
↪Wstrict-prototypes -Wshorten-64-to-32 -DNDEBUG -g -fwrapv -Os -Wall -Wstrict-
↪prototypes -DENABLE_DTRACE'
>>> import pprint
>>> pprint.pprint(sysconfig.get_paths())
{'data': '/System/Library/Frameworks/Python.framework/Versions/2.7',
 'include': '/System/Library/Frameworks/Python.framework/Versions/2.7/include/python2.
↪7',
 'platinclude': '/System/Library/Frameworks/Python.framework/Versions/2.7/include/
↪python2.7',
 'platlib': '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
↪site-packages',
```

(continues on next page)

(continued from previous page)

```
 'platstdlib': '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7
↪',
 'purelib': '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/
↪site-packages',
 'scripts': '/System/Library/Frameworks/Python.framework/Versions/2.7/bin',
 'stdlib': '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7'}
>>> sysconfig.get_paths()['include']
'/System/Library/Frameworks/Python.framework/Versions/2.7/include/python2.7'
```

9.3 From the Command Line using `sysconfig`

This very verbose output will give you a complete picture of your environment:

```
$ python3 -m sysconfig
Platform: "macosx-10.6-intel"
Python version: "3.4"
Current installation scheme: "posix_prefix"

Paths:
    data = "/Library/Frameworks/Python.framework/Versions/3.4"
    include = "/Library/Frameworks/Python.framework/Versions/3.4/include/python3.4m"
    platinclude = "/Library/Frameworks/Python.framework/Versions/3.4/include/python3.
↪4m"
    platlib = "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-
↪packages"
    platstdlib = "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4"
    purelib = "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-
↪packages"
    scripts = "/Library/Frameworks/Python.framework/Versions/3.4/bin"
    stdlib = "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4"

Variables:
    ABIFLAGS = "m"
    AC_APPLE_UNIVERSAL_BUILD = "1"
    AIX_GENUINE_CPLUSPLUS = "0"
    AR = "ar"
    ARFLAGS = "rc"
    ASDLGEN = "python /Users/sysadmin/build/v3.4.4/Parser/asdl_c.py"
    ASDLGEN_FILES = "/Users/sysadmin/build/v3.4.4/Parser/asdl.py /Users/sysadmin/
↪build/v3.4.4/Parser/asdl_c.py"
    AST_ASDL = "/Users/sysadmin/build/v3.4.4/Parser/Python.asdl"
    AST_C = "Python/Python-ast.c"
    AST_C_DIR = "Python"
    AST_H = "Include/Python-ast.h"
    AST_H_DIR = "Include"
    BASECFLAGS = "-fno-strict-aliasing -fno-common -dynamic"
    BASECPPFLAGS = ""
    BASEMODLIBS = ""
    BINDIR = "/Library/Frameworks/Python.framework/Versions/3.4/bin"
    BINLIBDEST = "/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4"
...
```

9.4 Setting Flags Automatically in `setup.py`

The sysconfig module allows you to create a generic `setup.py` script for Python C extensions (see highlighted line):

```
from distutils.core import setup, Extension
import os
import sysconfig

_DEBUG = False
# Generally I write code so that if DEBUG is defined as 0 then all optimisations
# are off and asserts are enabled. Typically run times of these builds are x2 to x10
# release builds.
# If DEBUG > 0 then extra code paths are introduced such as checking the integrity of
# internal data structures. In this case the performance is by no means comparable
# with release builds.
_DEBUG_LEVEL = 0

# Common flags for both release and debug builds.
extra_compile_args = sysconfig.get_config_var('CFLAGS').split()
extra_compile_args += ["-std=c++11", "-Wall", "-Wextra"]
if _DEBUG:
    extra_compile_args += ["-g3", "-O0", "-DDEBUG=%s" % _DEBUG_LEVEL, "-UNDEBUG"]
else:
    extra_compile_args += ["-DNDEBUG", "-O3"]

setup(
    name                = '...',
    version             = '...',
    author              = '...',
    author_email        = '...',
    maintainer          = '...',
    maintainer_email    = '...',
    description         = '...',
    long_description    = """...
""",
    platforms           = ['Mac OSX', 'POSIX',],
    classifiers         = [
        '...',
    ],
    license             = 'GNU Lesser General Public License v2 or later (LGPLv2+)',
    ext_modules=[
        Extension("MyExtension",
            sources=[
                    '...',
            ],
            include_dirs=[
                '.',
                '...',
                os.path.join(os.getcwd(), 'include'),
            ],
            library_dirs = [os.getcwd(),],  # path to .a or .so file(s)
            extra_compile_args=extra_compile_args,
            language='c++11',
        ),
    ]
)
```

CHAPTER

TEN

DEBUGGING

This is very much work in progress. I will add to it/correct it as I develop new techniques.

10.1 Debugging Tools

First create your toolbox, in this one we have:

- Debug version of Python - great for finding out more detail of your Python code as it executes.
- Valgrind - the goto tool for memory leaks. It is a little tricky to get working but should be in every developers toolbox.
- OS memory monitioring - this is a quick and simple way of identifying whether memory leaks are happening or not. An example is given below: *A Simple Memory Monitor*

10.1.1 Build a Debug Version of Python

There are a large combination of debug builds of Python that you can create and each one will give you extra information when you either:

- Invoke Python with a command line option.
  - Example: a `Py_DEBUG` build invoking Python with `python -X showrefcount`
- Set an environment variable.
  - Example: a `Py_DEBUG` build invoking Python with `PYTHONMALLOCSTATS=1 python`
- Additional functions that are added to the `sys` module that can give useful information.
  - Example: a `Py_DEBUG` build an calling `sys.getobjects(...)`.

See here *Building and Using a Debug Version of Python* for instructions on how to do this.

10.1.2 Valgrind

See here *Building Python for Valgrind* for instructions on how to build Valgrind.

See here *Using Valgrind* for instructions on how to use Valgrind.

Here *Finding Where the Leak is With Valgrind* is an example of finding a leak with Valgrind.

10.1.3 A Simple Memory Monitor

Here is a simple process memory monitor using the `psutil` library:

```
import sys
import time

import psutil

def memMon(pid, freq=1.0):
    proc = psutil.Process(pid)
    print(proc.memory_info_ex())
    prev_mem = None
    while True:
        try:
            mem = proc.memory_info().rss / 1e6
            if prev_mem is None:
                print('{:10.3f} [Mb]'.format(mem))
            else:
                print('{:10.3f} [Mb] {:+10.3f} [Mb]'.format(mem, mem - prev_mem))
            prev_mem = mem
            time.sleep(freq)
        except KeyboardInterrupt:
            try:
                input(' Pausing memMon, <cr> to continue, ^C to end...')
            except KeyboardInterrupt:
                print('\n')
                return

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print('Usage: python pidmon.py <PID>')
        sys.exit(1)
    pid = int(sys.argv[1])
    memMon(pid)
    sys.exit(0)
```

Lets test it. In one shell fire up Python and find its PID:

```
>>> import os
>>> os.getpid()
13360
```

In a second shell fire up pidmon.py with this PID:

```
$ python3 pidmon.py 13360
pextmem(rss=7364608, vms=2526482432, pfaults=9793536, pageins=24576)
     7.365 [Mb]
     7.365 [Mb]     +0.000 [Mb]
```

(continues on next page)

(continued from previous page)

```
     7.365 [Mb]     +0.000 [Mb]
...
```

Pause pidmon.py with Ctrl-C:

```
^C Pausing memMon, <cr> to continue, ^C to end...
```

Go back to the first shell and create a large string (1Gb):

```
>>> s = ' ' * 1024**3
```

In the second shell continue pidmon.py with <cr> and we see the memory usage:

```
1077.932 [Mb]  +1070.567 [Mb]
1077.932 [Mb]     +0.000 [Mb]
1077.932 [Mb]     +0.000 [Mb]
...
```

Go back to the first shell and delete the string:

```
>>> del s
```

In the second shell we see the memory usage drop:

```
1077.953 [Mb]     +0.020 [Mb]
1077.953 [Mb]     +0.000 [Mb]
 272.679 [Mb]   -805.274 [Mb]
   4.243 [Mb]   -268.435 [Mb]
   4.243 [Mb]     +0.000 [Mb]
...
```

In the second shell halt pidmon with two Ctrl-C commands:

```
^C Pausing memMon, <cr> to continue, ^C to end...^C
```

So we can observe the total memory usage of another process simply and cheaply. This is often the first test to do when examining processes for memory leaks.

10.2 Building and Using a Debug Version of Python

There is a spectrum of debug builds of Python that you can create. This chapter describes how to create them.

10.2.1 Building a Standard Debug Version of Python

Download and unpack the Python source. Then in the source directory create a debug directory for the debug build:

```
mkdir debug
cd debug
../configure --with-pydebug
make
make test
```

10.2.2 Specifying Macros

They can be specified at the configure stage, this works:

```
../configure CFLAGS='-DPy_DEBUG -DPy_TRACE_REFS' --with-pydebug
make
```

However the python documentation suggests the alternative way of specifying them when invoking make:

```
../configure --with-pydebug
make EXTRA_CFLAGS="-DPy_REF_DEBUG"
```

I don't know why one way would be regarded as better than the other.

10.2.3 The Debug Builds

The builds are controlled by the following macros:

Macro	Description	Must Rebuild Extensions?
`Py_DEBUG`	A standard debug build. `Py_DEBUG` sets `LLTRACE`, `Py_REF_DEBUG`, `Py_TRACE_REFS`, and `PYMALLOC_DEBUG` (if `WITH_PYMALLOC` is enabled).	Yes
`Py_REF_DEBUG`	Turns on aggregate reference counting which will be displayed in the interactive interpreter when invoked with `-X showrefcount` on the command line. If you are not keeping references to objects and the count is increasing there is probably a leak. Also adds `sys.gettotalrefcount()` to the `sys` module and this returns the total number of references.	No
`Py_TRACE_REFS`	Turns on reference tracing. Sets `Py_REF_DEBUG`.	Yes
`COUNT_ALLOCS`	Keeps track of the number of objects of each type have been allocated and how many freed. See: *Python Debug build with COUNT_ALLOCS*	Yes
`WITH_PYMALLOC`	Enables Pythons small memory allocator. For Valgrind this must be disabled, if using Pythons malloc debugger (using `PYMALLOC_DEBUG`) this must be enabled. See: *Python's Memory Allocator*	No
`PYMALLOC_DEBUG`	Enables Python's malloc debugger that annotates memory blocks. Requires `WITH_PYMALLOC`. See: *Python's Memory Allocator*	No

Here is the description of other debug macros that are set by one of the macros above:

Macro	Description
`LLTRACE`	Low level tracing. See `Python/ceval.c`.

In the source directory:

```
mkdir debug
cd debug
../configure --with-pydebug
make
make test
```

10.2.4 Python's Memory Allocator

A normal build of Python gives CPython a special memory allocator 'PyMalloc'. When enabled this mallocs largish chunks of memory from the OS and then uses this pool for the actual PyObjects. With PyMalloc active Valgrind can not see all allocations and deallocations.

There are two Python builds of interest to help solve memory problems:

- Disable PyMalloc so that Valgrind can analyse the memory usage.
- Enable PyMalloc in debug mode, this creates memory blocks with special bit patterns and adds debugging information on each end of any dynamically allocated memory. This pattern is checked on every alloc/free and if found to be corrupt a diagnostic is printed and the process terminated.

To make a version of Python with its memory allocator suitable for use with Valgrind:

```
../configure --with-pydebug --without-pymalloc
make
```

See *Using Valgrind* for using Valgrind.

To make a version of Python with its memory allocator using Python's malloc debugger either:

```
../configure CFLAGS='-DPYMALLOC_DEBUG' --with-pydebug
make
```

Or:

```
../configure --with-pydebug
make EXTRA_CFLAGS="-DPYMALLOC_DEBUG"
```

This builds Python with the `WITH_PYMALLOC` and `PYMALLOC_DEBUG` macros defined.

Finding Access after Free With `PYMALLOC_DEBUG`

Python built with `PYMALLOC_DEBUG` is the most effective way of detecting access after free. For example if we have this CPython code:

```
static PyObject *access_after_free(PyObject *pModule) {
    PyObject *pA = PyLong_FromLong(1024L);
    Py_DECREF(pA);
    PyObject_Print(pA, stdout, 0);
    Py_RETURN_NONE;
}
```

And we call this from the interpreter we get a diagnostic:

```
Python 3.4.3 (default, Sep 16 2015, 16:56:10)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import cPyRefs
>>> cPyRefs.afterFree()
<refcnt -2604246222170760229 at 0x10a474130>
>>>
```

Getting Statistics on PyMalloc

If the environment variable `PYTHONMALLOCSTATS` exists when running Python built with `WITH_PYMALLOC``+``PYMALLOC_DEBUG` then a (detailed) report of pymalloc activity is output on stderr whenever a new 'arena' is allocated.

```
PYTHONMALLOCSTATS=1 python.exe
```

I have no special knowledge about the output you see when running Python this way which looks like this:

```
>>> cPyRefs.leakNewRefs(1000, 10000)
loose_new_reference: value=1000 count=10000
Small block threshold = 512, in 64 size classes.

class   size   num pools   blocks in use  avail blocks
-----   ----   ---------   -------------  ------------
    4     40           2             139            63
    5     48           1               2            82
  ...
   62    504           3              21             3
   63    512           3              18             3

# times object malloc called       =            2,042,125
# arenas allocated total           =                  636
# arenas reclaimed                 =                    1
# arenas highwater mark            =                  635
# arenas allocated current         =                  635
635 arenas * 262144 bytes/arena    =          166,461,440

# bytes in allocated blocks        =          162,432,624
# bytes in available blocks        =              116,824
0 unused pools * 4096 bytes        =                    0
# bytes lost to pool headers       =            1,950,720
# bytes lost to quantization       =            1,961,272
# bytes lost to arena alignment    =                    0
Total                              =          166,461,440
Small block threshold = 512, in 64 size classes.

class   size   num pools   blocks in use  avail blocks
-----   ----   ---------   -------------  ------------
    4     40           2             139            63
    5     48           1               2            82
  ...
   62    504           3              21             3
   63    512           3              18             3

# times object malloc called       =            2,045,325
# arenas allocated total           =                  637
# arenas reclaimed                 =                    1
# arenas highwater mark            =                  636
# arenas allocated current         =                  636
636 arenas * 262144 bytes/arena    =          166,723,584

# bytes in allocated blocks        =          162,688,624
# bytes in available blocks        =              116,824
0 unused pools * 4096 bytes        =                    0
# bytes lost to pool headers       =            1,953,792
# bytes lost to quantization       =            1,964,344
```

(continues on next page)

(continued from previous page)

```
# bytes lost to arena alignment    =                   0
Total                              =         166,723,584
Small block threshold = 512, in 64 size classes.

class   size   num pools   blocks in use  avail blocks
-----   ----   ---------   -------------  ------------
    4     40           2             139            63
    5     48           1               2            82
  ...
   62    504           3              21             3
   63    512           3              18             3

# times object malloc called       =           2,048,525
# arenas allocated total           =                 638
# arenas reclaimed                 =                   1
# arenas highwater mark            =                 637
# arenas allocated current         =                 637
637 arenas * 262144 bytes/arena    =         166,985,728

# bytes in allocated blocks        =         162,944,624
# bytes in available blocks        =             116,824
0 unused pools * 4096 bytes        =                   0
# bytes lost to pool headers       =           1,956,864
# bytes lost to quantization       =           1,967,416
# bytes lost to arena alignment    =                   0
Total                              =         166,985,728
loose_new_reference: DONE
```

10.2.5 Python Debug build with `COUNT_ALLOCS`

A Python debug build with `COUNT_ALLOCS` give some additional information about each object *type* (not the individual objects themselves). A `PyObject` grows some extra fields that track the reference counts for that type. The fields are:

Field	Description
`tp_allocs`	The number of times an object of this type was allocated.
`tp_frees`	The number of times an object of this type was freed.
`tp_maxalloc`	The maximum seen value of `tp_allocs - tp_frees` so this is the maximum count of this type allocated at the same time.

The `sys` module also gets an extra function `sys.getcounts()` that returns a list of tuples: `[(tp_typename, tp_allocs, tp_frees, tp_maxalloc), ...]`.

Building the Python Executable with COUNT_ALLOCS

Either:

```
../configure CFLAGS='-DCOUNT_ALLOCS' --with-pydebug
make
```

Or:

```
../configure --with-pydebug
make EXTRA_CFLAGS="-DCOUNT_ALLOCS"
```

> **Warning:** When using `COUNT_ALLOCS` any Python extensions now need to be rebuilt with this Python executable as it fundementally changes the structure of a `PyObject`.

Using the Python Executable with COUNT_ALLOCS

An example of using this build is here: *Observing the Reference Counts for a Particular Type*

10.2.6 Identifying the Python Build Configuration from the Runtime

The module `sysconfig` allows you access to the configuration of the Python runtime. At its simplest, and most verbose, this can be used thus:

```
$ ./python.exe -m sysconfig
Platform: "macosx-10.9-x86_64"
Python version: "3.4"
Current installation scheme: "posix_prefix"

Paths:
    data = "/usr/local"
    ...
    stdlib = "/usr/local/lib/python3.4"

Variables:
    ABIFLAGS = "dm"
    AC_APPLE_UNIVERSAL_BUILD = "0"
    AIX_GENUINE_CPLUSPLUS = "0"
    AR = "ar"
    ...
    py_version = "3.4.3"
    py_version_nodot = "34"
    py_version_short = "3.4"
```

Importing `sysconfig` into an interpreter session gives two useful functions are `get_config_var(...)` which gets the setting for a particular macro and `get_config_vars()` which gets a dict of `{macro : value, ...}`. For example:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_DEBUG')
1
```

For advanced usage you can parse any `pyconfig.h` into a dict by opening that file and passing it to `sysconfig.parse_config_h(f)` as a file object. `sysconfig.get_config_h_filename()` will give you the configuration file for the runtime (assuming it still exists). So:

```
>>> with open(sysconfig.get_config_h_filename()) as f:
      cfg = sysconfig.parse_config_h(f)
```

10.3 Valgrind

This is about how to build Valgrind, a Valgrind friendly version of Python and finally how to use and interpret Valgrind.

Note: These instructions have been tested on Mac OS X 10.9 (Mavericks). They may or may not work on other OS's

10.3.1 Building Valgrind

This should be fairly straightforward:

```
svn co svn://svn.valgrind.org/valgrind
cd valgrind
./autogen.sh
./configure
make
make install
```

10.3.2 Building Python for Valgrind

Prepare the source by uncommenting `Py_USING_MEMORY_DEBUGGER` in Objects/obmalloc.c around line 1082 or so.

Configuring

`configure` takes the following aguments:

Argument	
`--enable-framework`	Installs it in /Library/Frameworks/Python.framework/Versions/
`--with-pydebug`	Debug build of Python. See Misc/SpecialBuilds.txt
`--without-pymalloc`	With Valgrind support Misc/README.valgrind

To make a framework install:

```
./configure --enable-framework --with-pydebug --without-pymalloc --with-valgrind
sudo make frameworkinstall
```

To make a local version cd to the source tree and we will build a Valgrind version of Python in the `valgrind/` directory:

```
mkdir valgrind
cd valgrind
../configure --with-pydebug --without-pymalloc --with-valgrind
make
```

Check debug build

```
$ python3 -X showrefcount
```

```
Python 3.4.3 (default, May 26 2015, 19:54:01)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 23
23
[54793 refs, 0 blocks]
>>> import sys
[54795 refs, 0 blocks]
>>> sys.gettotalrefcount()
54817
[54795 refs, 0 blocks]
>>> import sysconfig
>>> sysconfig.get_config_var('Py_DEBUG')
1
```

10.3.3 Using Valgrind

In the `<Python source>/Misc` directory there is a `valgrind-python.supp` file that supresses some Valgrind spurious warnings. I find that this needs editing so:

```
cp <Python source>/Misc/valgrind-python.supp ~/valgrind-python.supp
vi ~/valgrind-python.supp
```

Uncomment `PyObject_Free` and `PyObject_Realloc` in the valgrind suppression file.

Invoking the Python interpreter with Valgrind:

```
valgrind --tool=memcheck --dsymutil=yes --track-origins=yes --show-leak-kinds=all --
↪trace-children=yes --suppressions=$HOME/.valgrind-python.supp <Python source>/
↪valgrind/python.exe -X showrefcount
```

An example of using Valgrind to detect leaks is here: *Finding Where the Leak is With Valgrind*.

10.4 Leaked New References

This shows what happens if create new Python objects and leak them and how we might detect that.

10.4.1 A Leak Example

Here is an example function that deliberately creates leaks with new references. It takes an integer value to be created (and leaked) and a count of the number of times that this should happen. We can then call this from the Python interpreter and observe what happens.

```
static PyObject *leak_new_reference(PyObject *pModule,
                                    PyObject *args, PyObject *kwargs) {
    PyObject *ret = NULL;
    int value, count;
    static char *kwlist[] = {"value", "count", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "ii", kwlist, &value, &count)) {
        goto except;
    }
    fprintf(stdout, "loose_new_reference: value=%d count=%d\n", value, count);
    for (int i = 0; i < count; ++i) {
        PyLong_FromLong(value);     /* New reference, leaked. */
    }

    Py_INCREF(Py_None);
    ret = Py_None;
    goto finally;
except:
    Py_XDECREF(ret);
    ret = NULL;
finally:
    fprintf(stdout, "loose_new_reference: DONE\n");
    return ret;
}
```

And we add this to the `cPyRefs` module function table as the python function `leakNewRefs`:

```
static PyMethodDef cPyRefs_methods[] = {
    /* Other functions here
     * ...
     */
    {"leakNewRefs", (PyCFunction)leak_new_reference,
        METH_VARARGS | METH_KEYWORDS, "Leaks new references to longs."},

    {NULL, NULL, 0, NULL}  /* Sentinel */
};
```

In Python first we check what the size of a long is then we call the leaky function with the value 1000 (not the values -5 to 255 which are interned) one million times and there should be a leak of one million times the size of a long:

```
>>> import sys
>>> sys.getsizeof(1000)
44
>>> import cPyRefs
>>> cPyRefs.leakNewRefs(1000, 1000000)
loose_new_reference: value=1000 count=1000000
loose_new_reference: DONE
>>>
```

This should generate a leak of 44Mb or thereabouts.

10.4.2 Recognising Leaked New References

Leaked references can lay unnoticed for a long time, especially if they are small. The ways that you might detect that they are happening are:

- Noticing an unanticipated, ever increasing memory usage at the OS level (by using `top` for example).
- If you run a debug build of Python with `Py_REF_DEBUG` defined you might notice a very high level of total reference counts by either invoking Python with `-X showrefcount` or calling `sys.gettotalrefcount()`.
- Other types of debug build can be useful too.
- Examining Valgrind results.

Observing the Memory Usage

In the debug tools section there is a *A Simple Memory Monitor* that can examine the memory of another running process.

Lets test it. In one shell fire up Python and find its PID:

```
>>> import os
>>> os.getpid()
14488
```

In a second shell fire up pidmon.py with this PID:

```
$ python3 pidmon.py 14488
pextmem(rss=7659520, vms=2475937792, pfaults=8380416, pageins=2617344)
     7.660 [Mb]
     7.660 [Mb]     +0.000 [Mb]
     7.660 [Mb]     +0.000 [Mb]
     7.660 [Mb]     +0.000 [Mb]
```

Go back to the first shell and import `cPyRefs`:

```
>>> import cPyRefs
>>> cPyRefs.leakNewRefs(1000, 1000000)
loose_new_reference: value=1000 count=1000000
loose_new_reference: DONE
>>>
```

In the second shell pidmon.py shows the sudden jump in memory usage:

```
$ python3 pidmon.py 14488
pextmem(rss=7659520, vms=2475937792, pfaults=8380416, pageins=2617344)
     7.660 [Mb]
     7.660 [Mb]     +0.000 [Mb]
     7.660 [Mb]     +0.000 [Mb]
     7.660 [Mb]     +0.000 [Mb]
     ...
     7.684 [Mb]     +0.000 [Mb]
     7.684 [Mb]     +0.000 [Mb]
    56.443 [Mb]    +48.759 [Mb]
    56.443 [Mb]     +0.000 [Mb]
    56.443 [Mb]     +0.000 [Mb]
    56.443 [Mb]     +0.000 [Mb]
```

(continues on next page)

(continued from previous page)

```
    56.443 [Mb]     +0.000 [Mb]
     ...
```

Observing the Total Reference Counts

If you have a debug build of Python with `Py_REF_DEBUG` defined you might notice a very high level of total reference counts by either invoking Python with `-X showrefcount` or calling `sys.gettotalrefcount()`.

For example:

```
>>> import sys
>>> import cPyRefs
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
↪'cPyRefs', 'sys']
>>> sys.gettotalrefcount()
55019
>>> cPyRefs.leakNewRefs(1000, 1000000)
loose_new_reference: value=1000 count=1000000
loose_new_reference: DONE
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
↪'cPyRefs', 'sys']
>>> sys.gettotalrefcount()
1055019
```

Notice that `cPyRefs.leakNewRefs(1000, 1000000)` does not add anything to the result of `dir()` but adds 1m reference counts somewhere.

And those references are not collectable:

```
>>> import gc
>>> gc.collect()
0
>>> sys.gettotalrefcount()
1055519
```

Observing the Reference Counts for a Particular Type

If you have a debug build with `COUNT_ALLOCS` [See: *Python Debug build with COUNT_ALLOCS*] defined you can see the references counts for each type. This build will have a new function `sys.getcounts()` which returns a list of tuples `(tp_name, tp_allocs, tp_frees, tp_maxalloc)` where `tp_maxalloc` is the maximum ever seen value of the reference `tp_allocs - tp_frees`. The list is ordered by time of first object allocation:

```
>>> import pprint
>>> import sys
>>> pprint.pprint(sys.getcounts())
[('Repr', 1, 0, 1),
 ('symtable entry', 3, 3, 1),
 ('OSError', 1, 1, 1),
 ...
 ('int', 3342, 2630, 712),
 ...
```

(continues on next page)

(continued from previous page)

```
 ('dict', 1421, 714, 714),
 ('tuple', 13379, 9633, 3746)]
```

We can try our leaky code:

```
>>> import cPyRefs
>>> cPyRefs.leakNewRefs(1000, 1000000)
loose_new_reference: value=1000 count=1000000
loose_new_reference: DONE
>>> pprint.pprint(sys.getcounts())
[('memoryview', 103, 103, 1),
 ...
 ('int', 1004362, 3650, 1000712),
 ...
 ('dict', 1564, 853, 718),
 ('tuple', 22986, 19236, 3750)]
```

There is a big jump in `tp_maxalloc` for ints that is worth investigating.

When the Python process finishes you get a dump of this list as the interpreter is broken down:

```
memoryview alloc'd: 210, freed: 210, max in use: 1
managedbuffer alloc'd: 210, freed: 210, max in use: 1
PrettyPrinter alloc'd: 2, freed: 2, max in use: 1
...
int alloc'd: 1005400, freed: 4887, max in use: 1000737
...
str alloc'd: 21920, freed: 19019, max in use: 7768
dict alloc'd: 1675, freed: 1300, max in use: 718
tuple alloc'd: 32731, freed: 31347, max in use: 3754
fast tuple allocs: 28810, empty: 2101
fast int allocs: pos: 7182, neg: 20
null strings: 69, 1-strings: 5
```

10.4.3 Finding Where the Leak is With Valgrind

Now that we have noticed that there is a problem, then where, exactly, is the problem?

Lets run our debug version of Python with Valgrind and see if we can spot the leak (assumes *.valgrind-python.supp* is in your $HOME directory and `./Python.3.4.3D` is the debug executable):

```
valgrind --tool=memcheck --trace-children=yes --dsymutil=yes --leak-check=full --show-
↪leak-kinds=all --suppressions=~/.valgrind-python.supp ./Python.3.4.3D
```

Note: You need to use the option `--show-leak-kinds=all` for the next bit to work otherwise you see in the summary that memory has been leaked but not where from.

Then run this code:

```
>>> import cPyRefs
>>> cPyRefs.leakNewRefs(1000, 1000000)
loose_new_reference: value=1000 count=1000000
loose_new_reference: DONE
>>>^D
```

In the Valgrind output you should see this in the summary at the end:

```
==13042== LEAK SUMMARY:
==13042==    definitely lost: 6,925 bytes in 26 blocks
==13042==    indirectly lost: 532 bytes in 5 blocks
==13042==      possibly lost: 329,194 bytes in 709 blocks
==13042==    still reachable: 44,844,200 bytes in 1,005,419 blocks
```

The "still reachable" value is a clue that something has gone awry.

In the body of the Valgrind output you should find something like this - the important lines are highlighted:

```
==13042== 44,000,000 bytes in 1,000,000 blocks are still reachable in loss record 2,
↪325 of 2,325
==13042==    at 0x47E1: malloc (vg_replace_malloc.c:300)
==13042==    by 0x41927E: _PyMem_RawMalloc (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x418D80: PyObject_Malloc (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x3DFBA8: _PyLong_New (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x3DFF8D: PyLong_FromLong (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x11BFA40: leak_new_reference (cPyRefs.c:149)
==13042==    by 0x40D81C: PyCFunction_Call (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x555C15: call_function (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x54E02B: PyEval_EvalFrameEx (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x53A7B4: PyEval_EvalCodeEx (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x539414: PyEval_EvalCode (in /Library/Frameworks/Python.framework/
↪Versions/3.4/Python)
==13042==    by 0x5A605E: run_mod (in /Library/Frameworks/Python.framework/Versions/3.
↪4/Python)
==13042==
```

Which is exactly what we are looking for.

10.5 Debugging Tactics

So what is the problem that you are trying to solve?

10.5.1 Access After Free

Problem

You suspect that you are accessing a Python object after it has been free'd. Code such as this:

```
static PyObject *access_after_free(PyObject *pModule) {
    PyObject *pA = PyLong_FromLong(1024L);
    Py_DECREF(pA);
    PyObject_Print(pA, stdout, 0);
```

(continues on next page)

(continued from previous page)

```
    Py_RETURN_NONE;
}
```

pA has been freed before PyObject_Print is called.

Solution

```
Python 3.4.3 (default, Sep 16 2015, 16:56:10)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import cPyRefs
>>> cPyRefs.afterFree()
<refcnt -2604246222170760229 at 0x10a474130>
>>>
```

10.5.2 Py_INCREF called too often

Summary: If Py_INCREF is called once or more too often then memory will be held onto despite those objects not being visible in globals() or locals().

Symptoms: You run a test that creates objects and uses them. As the test exits all the objects should go out of scope and be de-alloc'd however you observe that memory is being permanently lost.

Problem

We can create a simulation of this by creating two classes, with one we will create a leak caused by an excessive Py_INCREF (we do this by calling cPyRefs.incref()). The other class instance will not be leaked.

Here is the code for incrementing the reference count in the cPyRefs module:

```
/* Just increfs a PyObject. */
static PyObject *incref(PyObject *pModule, PyObject *pObj) {
    fprintf(stdout, "incref(): Ref count was: %zd\n", pObj->ob_refcnt);
    Py_INCREF(pObj);
    fprintf(stdout, "incref(): Ref count now: %zd\n", pObj->ob_refcnt);
    Py_RETURN_NONE;
}
```

And the Python interpreter session we create two instances and excessively incref one of them:

```
>>> import cPyRefs # So we can create a leak
>>> class Foo : pass # Foo objects will be leaked
...
>>> class Bar : pass # Bar objects will not be leaked
...
>>> def test_foo_bar():
...     f = Foo()
...     b = Bar()
...     # Now increment the reference count of f, but not b
...     # This simulates what might happen in a leaky extension
...     cPyRefs.incref(f)
...
>>> # Call the test, the output comes from cPyRefs.incref()
```

(continues on next page)

(continued from previous page)

```
>>> test_foo_bar()
incref(): Ref count was: 2
incref(): Ref count now: 3
```

Solution

Use a debug version of Python with `COUNT_ALLOCS` defined. This creates `sys.getcounts()` which lists, for each type, how many allocs and de-allocs have been made. Notice the difference between `Foo` and `Bar`:

```
>>> import sys
>>> sys.getcounts()
[
    ('Bar', 1, 1, 1),
    ('Foo', 1, 0, 1),
    ...
]
```

This should focus your attention on the leaky type `Foo`.

You can find the count of all live objects by doing this:

```
>>> still_live = [(v[0], v[1] - v[2]) for v in sys.getcounts() if v[1] > v[2]]
>>> still_live
[('Foo', 1), ... ('str', 7783), ('dict', 714), ('tuple', 3875)]
```

10.6 Using gcov for C/C++ Code Coverage

This is about how to run your C/C++ code using gcov to gather coverage metrics.

Note: These instructions have been tested on Mac OS X 10.9 (Mavericks). They may or may not work on other OS's

10.6.1 gcov under Mac OS X

Configuring Xcode

This document tells you how to configure Xcode for gcov (actually llvm-cov).

Running and Analysing Code Coverage

In Xcode select Product->Run then once that has completed note the GCov_build directory in Xcode by selecting Products-><project> in the left hand pane. In the right hand pane under 'Identity and Type' you should see 'Full Path' of something like:

```
/Users/$USER/Library/Developer/Xcode/DerivedData/<project name and hash>/Build/
↪Products/GCov_Build/<project>
```

In the Terminal navigate to the 'Build' directory:

```
$ cd /Users/$USER/Library/Developer/Xcode/DerivedData/<project name and hash>/Build/
```

Now navigate to the Intermediates/ directory and in there you should find the code coverage data in a path such as this:

```
cd Intermediates/<project>.build/GCov_Build/<project>.build/Objects-normal/x86_64
```

In there the interesting file has a .gcno extension. To convert this into something readable we need to run gcov on it. We can use xcrun to find gcov and this gives an overall code coverage number for each file (and its includes):

```
$ xcrun gcov TimeBDT.gcno
...
File 'TimeBDT.cpp'
Lines executed:87.18% of 78
TimeBDT.cpp:creating 'TimeBDT.cpp.gcov'
...
```

This has now generated a detailed file TimeBDT.cpp.gcov that contains line by line coverage:

```
  $ ls -l TimeBDT.*
  -rw-r--r--  1 paulross  staff   14516  6 Oct 18:48 TimeBDT.cpp.gcov
  -rw-r--r--  1 paulross  staff     248  6 Oct 18:38 TimeBDT.d
  -rw-r--r--  1 paulross  staff    1120  6 Oct 18:38 TimeBDT.dia
  -rw-r--r--  1 paulross  staff   32252  6 Oct 18:45 TimeBDT.gcda
  -rw-r--r--  1 paulross  staff  121444  6 Oct 18:38 TimeBDT.gcno
  -rw-r--r--  1 paulross  staff    3671  6 Oct 18:48 TimeBDT.h.gcov
  -rw-r--r--  1 paulross  staff  310496  6 Oct 18:38 TimeBDT.o
  $ cat TimeBDT.cpp.gcov | less
  ...
      -:  208:/* Returns the total number of seconds in this particular year. */
      -:  209:time_t TimeBDTMapEntry::secsInThisYear() const {
6000000:  210:    return _isLeap ? SECONDS_IN_YEAR[1] : SECONDS_IN_YEAR[0];
      -:  211:}
  ...
      -:  289:        } else {
  #####:  290:            if (pTimeBDT->ttYearStart() > tt) {
  #####:  291:                --year;
  #####:  292:            } else {
  #####:  293:                ++year;
      -:  294:            }
      -:  295:        }
```

The first snippet shows that line 210 is executed 6000000 times. In the second snippet the ##### shows that lines 290-293 have not executed at all.

10.6.2 Using gcov on CPython Extensions

Whilst it is common to track code coverage in Python test code it gets a bit more tricky with Cpython extensions as Python code coverage tools can not track C/C++ extension code. The solution is to use `gcov` and run the tests in a C/C++ process by embedding the Python interpreter.

```
#include <Python.h>

int test_cpython_module_foo_functions(PyObject *pModule) {
    assert(pModule);
    int fail = 0;
```

(continues on next page)

(continued from previous page)

```
try:
    /* foo.function(...) */
    pFunc = PyObject_GetAttrString(pModule, "function");
    if (pFunc && PyCallable_Check(pFunc)) {

        pValue = PyObject_CallObject(pFunc, pArgs);
    }
    /*
     * Test other Foo functions here
     */
except:
    assert(fail != 0);
finally:
    return fail;
}

int test_cpython_module_foo() {
    int fail = 0;
    PyObject *pName = NULL;
try:
    pName = PyUnicode_FromString("foo");
    if (! pName) {
        fail = 1;
        goto except;
    }
    pModule = PyImport_Import(pName);
    if (! pModule) {
        fail = 2;
        goto except;
    }
    /* foo.function(...) */
    pFunc = PyObject_GetAttrString(pModule, "function");
    if (pFunc && PyCallable_Check(pFunc)) {

        pValue = PyObject_CallObject(pFunc, pArgs);
    }
    /*
     * Test other Foo functions here
     */
except:
    assert(fail != 0);
finally:
    Py_XDECREF(pName);
    Py_XDECREF(pModule);
    return fail;
}

void test_cpython_code() {
    Py_SetProgramName("TestCPythonExtensions");  /* optional, recommended */
    Py_Initialize();
    test_cpython_module_foo();
    /*
     * Test other modules here
     */
    Py_Finalize();
}

```

(continues on next page)

(continued from previous page)

```
int main(int argc, const char * argv[]) {
    std::cout << "Testing starting" << std::endl;
    test_cpython_code();
    /*
     * Non-CPython code tests here...
     */
    std::cout << "Testing DONE" << std::endl;
    return 0;
}
```

10.7 Debuging Python C Extensions in an IDE

`gdb` and `lldb` work well with Python extensions but if you want to step through your C extension in an IDE here is one way to do it.

The basic idea is to compile/link your C extension in your IDE and get `main()` to call a function `int import_call_execute(int argc, const char *argv[])` that embeds the Python interpreter which then imports a Python module, say a unit test, that exercises your C extension code.

This `import_call_execute()` entry point is fairly generic and takes the standard arguments to `main()` so it can be in its own .h/.c file.

10.7.1 Creating a Python Unit Test to Execute

Suppose you have a Python extension `ScList` that sub-classes a list and counts the number of times `.append(...)` was called making this count available as a `.appends` property. You have a unit test called `test_sclist.py` that looks like this with a single function `test()`:

```
import ScList

def test():
    s = ScList.ScList()
    assert s.appends == 0
    s.append(8)
    assert s.appends == 1
```

10.7.2 Writing a C Function to call any Python Unit Test

We create the `import_call_execute()` function that takes that same arguments as `main()` which can forward its arguments. `import_call_execute()` expects 4 arguments:

- `argc[0]` - Name of the executable.
- `argc[1]` - Path to the directory that the Python module is in.
- `argc[2]` - Name of the Python module to be imported. This could be a unit test module for example.
- `argc[3]` - Name of the Python function in the Python module (no arguments will be supplied, the return value is ignored). This could be a particular unit test.

The `import_call_execute()` function does this:

1. Check the arguments and initialises the Python interpreter

2. Add the path to the `argc[1]` to `sys.paths`.
3. Import `argc[2]`.
4. Find the function `argc[3]` in module `argc[2]` and call it.
5. Clean up.

Code Walk Through

`import_call_execute()` is quite generic and could be implemented in, say, `py_import_call_execute.c`.

Here is a walk through of the implementation code:

Step 1: Check the arguments and initialise the Python interpreter

```
/** This imports a Python module and calls a specific function in it.
 * It's arguments are similar to main():
 * argc - Number of strings in argv
 * argv - Expected to be 4 strings:
 *      - Name of the executable.
 *      - Path to the directory that the Python module is in.
 *      - Name of the Python module.
 *      - Name of the function in the module.
 *
 * The Python interpreter will be initialised and the path to the Python module
 * will be added to sys.paths then the module will be imported.
 * The function will be called with no arguments and its return value will be
 * ignored.
 *
 * This returns 0 on success, non-zero on failure.
 */
int import_call_execute(int argc, const char *argv[]) {
    int return_value = 0;
    PyObject *pModule   = NULL;
    PyObject *pFunc     = NULL;
    PyObject *pResult   = NULL;

    if (argc != 4) {
        fprintf(stderr,
                "Wrong arguments!"
                " Usage: %s package_path module function\n", argv[0]);
        return_value = -1;
        goto except;
    }
    Py_SetProgramName((wchar_t*)argv[0]);
    Py_Initialize();
```

Step 2: Add the path to the `test_sclist.py` to `sys.paths`. For convenience we have a separate function `add_path_to_sys_module()` to do this.

```
if (add_path_to_sys_module(argv[1])) {
    return_value = -2;
    goto except;
}
```

Here is the implementation of `add_path_to_sys_module()`:

```
/** Takes a path and adds it to sys.paths by calling PyRun_SimpleString.
 * This does rather laborious C string concatenation so that it will work in
 * a primitive C environment.
 *
 * Returns 0 on success, non-zero on failure.
 */
int add_path_to_sys_module(const char *path) {
    int ret = 0;
    const char *prefix = "import sys\nsys.path.append(\"";
    const char *suffix = "\")\n";
    char *command = (char*)malloc(strlen(prefix)
                                  + strlen(path)
                                  + strlen(suffix)
                                  + 1);
    if (! command) {
        return -1;
    }
    strcpy(command, prefix);
    strcat(command, path);
    strcat(command, suffix);
    ret = PyRun_SimpleString(command);
#ifdef DEBUG
    printf("Calling PyRun_SimpleString() with:\n");
    printf("%s", command);
    printf("PyRun_SimpleString() returned: %d\n", ret);
    fflush(stdout);
#endif
    free(command);
    return ret;
}
```

Step 3: Import `test_sclist`.

```
pModule = PyImport_ImportModule(argv[2]);
if (! pModule) {
    fprintf(stderr,
            EXECUTABLE_NAME ": Failed to load module \"%s\"\n", argv[2]);
    return_value = -3;
    goto except;
}
```

Step 4: Find the function `test()` in `test_sclist` and call it.

```
    pFunc = PyObject_GetAttrString(pModule, argv[3]);
    if (! pFunc) {
        fprintf(stderr,
                EXECUTABLE_NAME ": Can not find function \"%s\"\n", argv[3]);
        return_value = -4;
        goto except;
    }
    if (! PyCallable_Check(pFunc)) {
        fprintf(stderr,
                EXECUTABLE_NAME ": Function \"%s\" is not callable\n", argv[3]);
        return_value = -5;
        goto except;
    }
    pResult = PyObject_CallObject(pFunc, NULL);
```

(continues on next page)

(continued from previous page)

```
    if (! pResult) {
        fprintf(stderr, EXECUTABLE_NAME ": Function call failed\n");
        return_value = -6;
        goto except;
    }
#ifdef DEBUG
    printf(EXECUTABLE_NAME ": PyObject_CallObject() succeeded\n");
#endif
    assert(! PyErr_Occurred());
    goto finally;
```

Step 5: Clean up.

```
except:
    assert(PyErr_Occurred());
    PyErr_Print();
finally:
    Py_XDECREF(pFunc);
    Py_XDECREF(pModule);
    Py_XDECREF(pResult);
    Py_Finalize();
    return return_value;
}
```

And then we need `main()` to call this, thus:

```
#include "py_import_call_execute.h"

int main(int argc, const char *argv[]) {
    return import_call_execute(argc, argv);
}
```

Complete Code

The complete code for `py_import_call_execute.c` is here:

```
#include "py_import_call_execute.h"

#include <Python.h>

#ifdef __cplusplus
extern "C" {
#endif

/** Takes a path and adds it to sys.paths by calling PyRun_SimpleString.
 * This does rather laborious C string concatenation so that it will work in
 * a primitive C environment.
 *
 * Returns 0 on success, non-zero on failure.
 */
int add_path_to_sys_module(const char *path) {
    int ret = 0;
    const char *prefix = "import sys\nsys.path.append(\"";
    const char *suffix = "\")\n";
    char *command = (char*)malloc(strlen(prefix)
```

(continues on next page)

(continued from previous page)

```
                                    + strlen(path)
                                    + strlen(suffix)
                                    + 1);
    if (! command) {
        return -1;
    }
    strcpy(command, prefix);
    strcat(command, path);
    strcat(command, suffix);
    ret = PyRun_SimpleString(command);
#ifdef DEBUG
    printf("Calling PyRun_SimpleString() with:\n");
    printf("%s", command);
    printf("PyRun_SimpleString() returned: %d\n", ret);
    fflush(stdout);
#endif
    free(command);
    return ret;
}

/** This imports a Python module and calls a specific function in it.
 * It's arguments are similar to main():
 * argc - Number of strings in argv
 * argv - Expected to be 4 strings:
 *      - Name of the executable.
 *      - Path to the directory that the Python module is in.
 *      - Name of the Python module.
 *      - Name of the function in the module.
 *
 * The Python interpreter will be initialised and the path to the Python module
 * will be added to sys.paths then the module will be imported.
 * The function will be called with no arguments and its return value will be
 * ignored.
 *
 * This returns 0 on success, non-zero on failure.
 */
int import_call_execute(int argc, const char *argv[]) {
    int return_value = 0;
    PyObject *pModule   = NULL;
    PyObject *pFunc     = NULL;
    PyObject *pResult   = NULL;

    if (argc != 4) {
        fprintf(stderr,
                "Wrong arguments!"
                " Usage: %s package_path module function\n", argv[0]);
        return_value = -1;
        goto except;
    }
    Py_SetProgramName((wchar_t*)argv[0]);
    Py_Initialize();
    if (add_path_to_sys_module(argv[1])) {
        return_value = -2;
        goto except;
    }
    pModule = PyImport_ImportModule(argv[2]);
    if (! pModule) {
```

(continues on next page)

(continued from previous page)

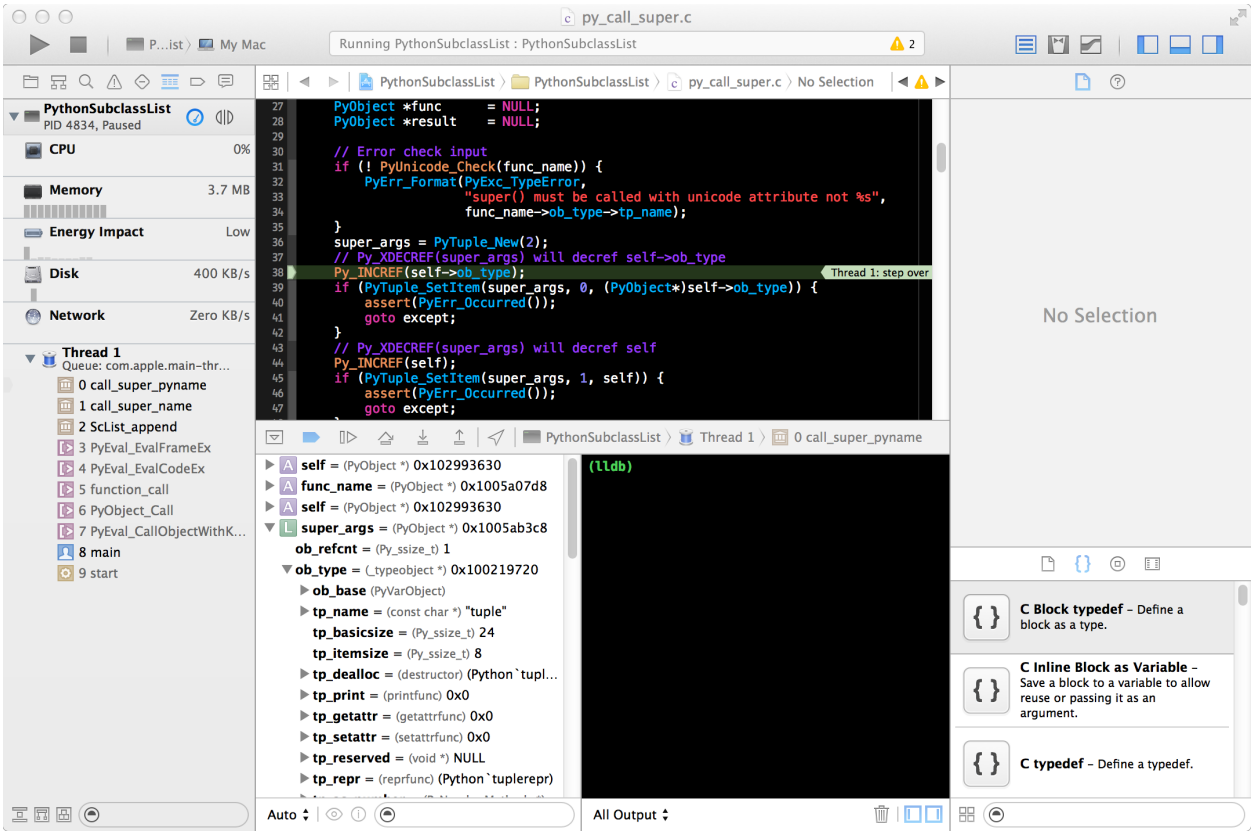
```
        fprintf(stderr,
                "%s: Failed to load module \"%s\"\n", argv[0], argv[2]);
        return_value = -3;
        goto except;
    }
    pFunc = PyObject_GetAttrString(pModule, argv[3]);
    if (! pFunc) {
        fprintf(stderr,
                "%s: Can not find function \"%s\"\n", argv[0], argv[3]);
        return_value = -4;
        goto except;
    }
    if (! PyCallable_Check(pFunc)) {
        fprintf(stderr,
                "%s: Function \"%s\" is not callable\n", argv[0], argv[3]);
        return_value = -5;
        goto except;
    }
    pResult = PyObject_CallObject(pFunc, NULL);
    if (! pResult) {
        fprintf(stderr, "%s: Function call failed\n", argv[0]);
        return_value = -6;
        goto except;
    }
#ifdef DEBUG
    printf("%s: PyObject_CallObject() succeeded\n", argv[0]);
#endif
    assert(! PyErr_Occurred());
    goto finally;
except:
    assert(PyErr_Occurred());
    PyErr_Print();
finally:
    Py_XDECREF(pFunc);
    Py_XDECREF(pModule);
    Py_XDECREF(pResult);
    Py_Finalize();
    return return_value;
}

#ifdef __cplusplus
//    extern "C" {
#endif
```

10.7.3 Debugging Python C Extensions in Xcode

Build this code in Xcode, set break points in your extension, and run with the command line arguments:

```
<path to test_sclist.py> test_sclist test
```

And you should get something like this:



The full code for this is in *src/debugging/XcodeExample/PythonSubclassList/*.

10.7.4 Using a Debug Version of Python C with Xcode

To get Xcode to use a debug version of Python first build Python from source assumed here to be `<SOURCE_DIR>` with, as a minimum, `--with-pydebug`. This example is using Python 3.6:

```
cd <SOURCE_DIR>
mkdir debug-framework
cd debug-framework/
../configure --with-pydebug --without-pymalloc --with-valgrind --enable-framework
make
```

Then in Xcode select the project and "Add files to . . . " and add:

- `<SOURCE_DIR>/debug-framework/Python.framework/Versions/3.6/Python`
- `<SOURCE_DIR>/debug-framework/libpython3.6d.a`

In "Build Settings":

- add `/Library/Frameworks/Python.framework/Versions/3.6/include/python3.6m/` to "Header Search Paths". Alternatively add both `<SOURCE_DIR>/Include` *and* `<SOURCE_DIR>/debug-framework` to "Header Search Paths", the latter is needed for `pyconfig.h`.
- add `<SOURCE_DIR>/debug-framework` to "Library Search Paths".

Now you should be able to step into the CPython code.

10.7.5 Debugging Python C Extensions in Eclipse

The same `main()` can be used.

TODO: The intern can do this.

10.8 Instrumenting the Python Process for Your Structures

Some debugging problems can be solved by instrumenting your C extensions for the duration of the Python process and reporting what happened when the process terminates. The data could be: the number of times classes were instantiated, functions called, memory allocations/deallocations or anything else that you wish.

To take a simple case, suppose we have a class that implements a up/down counter and we want to count how often each `inc()` and `dec()` function is called during the entirety of the Python process. We will create a C extension that has a class that has a single member (an interger) and two functions that increment or decrement that number. If it was in Python it would look like this:

```
class Counter:
    def __init__(self, count=0):
        self.count = count

    def inc(self):
        self.count += 1

    def dec(self):
        self.count -= 1
```

What we would like to do is to count how many times `inc()` and `dec()` are called on *all* instances of these objects and summarise them when the Python process exits[1].

There is an interpreter hook `Py_AtExit()` that allows you to register C functions that will be executed as the Python interpreter exits. This allows you to dump information that you have gathered about your code execution.

10.8.1 An Implementation of a Counter

First here is the module `pyatexit` with the class `pyatexit.Counter` with no intrumentation (it is equivelent to the Python code above). We will add the instrumentation later:

```
1 #include <Python.h>
2 #include "structmember.h"
3
4 #include <stdio.h>
5
6 typedef struct {
```

(continues on next page)

[1] The `atexit` module in Python can be used to similar effect however registered functions are called at a different stage of interpreted teardown than `Py_AtExit`.

(continued from previous page)

```
    PyObject_HEAD int number;
} Py_Counter;

static void Py_Counter_dealloc(Py_Counter* self) {
    Py_TYPE(self)->tp_free((PyObject*)self);
}

static PyObject* Py_Counter_new(PyTypeObject* type, PyObject* args,
                                PyObject* kwds) {
    Py_Counter* self;
    self = (Py_Counter*)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->number = 0;
    }
    return (PyObject*)self;
}

static int Py_Counter_init(Py_Counter* self, PyObject* args, PyObject* kwds) {
    static char* kwlist[] = { "number", NULL };
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|i", kwlist, &self->number)) {
        return -1;
    }
    return 0;
}

static PyMemberDef Py_Counter_members[] = {
    { "count", T_INT, offsetof(Py_Counter, number), 0, "count value" },
    { NULL, 0, 0, 0, NULL } /* Sentinel */
};

static PyObject* Py_Counter_inc(Py_Counter* self) {
    self->number++;
    Py_RETURN_NONE;
}

static PyObject* Py_Counter_dec(Py_Counter* self) {
    self->number--;
    Py_RETURN_NONE;
}

static PyMethodDef Py_Counter_methods[] = {
    { "inc", (PyCFunction)Py_Counter_inc, METH_NOARGS, "Increments the counter" },
    { "dec", (PyCFunction)Py_Counter_dec, METH_NOARGS, "Decrements the counter" },
    { NULL, NULL, 0, NULL } /* Sentinel */
};

static PyTypeObject Py_CounterType = {
    PyVarObject_HEAD_INIT(NULL, 0) "pyatexit.Counter", /* tp_name */
    sizeof(Py_Counter), /* tp_basicsize */
    0, /* tp_itemsize */
    (destructor)Py_Counter_dealloc, /* tp_dealloc */
    0, /* tp_print */
    0, /* tp_getattr */
    0, /* tp_setattr */
    0, /* tp_reserved */
    0, /* tp_repr */
    0, /* tp_as_number */
```

(continues on next page)

(continued from previous page)

```
    0, /* tp_as_sequence */
    0, /* tp_as_mapping */
    0, /* tp_hash  */
    0, /* tp_call */
    0, /* tp_str */
    0, /* tp_getattro */
    0, /* tp_setattro */
    0, /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags */
    "Py_Counter objects", /* tp_doc */
    0, /* tp_traverse */
    0, /* tp_clear */
    0, /* tp_richcompare */
    0, /* tp_weaklistoffset */
    0, /* tp_iter */
    0, /* tp_iternext */
    Py_Counter_methods, /* tp_methods */
    Py_Counter_members, /* tp_members */
    0, /* tp_getset */
    0, /* tp_base */
    0, /* tp_dict */
    0, /* tp_descr_get */
    0, /* tp_descr_set */
    0, /* tp_dictoffset */
    (initproc)Py_Counter_init, /* tp_init */
    0, /* tp_alloc */
    Py_Counter_new, /* tp_new */
    0, /* tp_free */
};

static PyModuleDef pyexitmodule = {
    PyModuleDef_HEAD_INIT, "pyatexit",
    "Extension that demonstrates the use of Py_AtExit().",
    -1, NULL, NULL, NULL, NULL,
    NULL
};

PyMODINIT_FUNC PyInit_pyatexit(void) {
    PyObject* m;

    if (PyType_Ready(&Py_CounterType) < 0) {
        return NULL;
    }
    m = PyModule_Create(&pyexitmodule);
    if (m == NULL) {
        return NULL;
    }
    Py_INCREF(&Py_CounterType);
    PyModule_AddObject(m, "Counter", (PyObject*)&Py_CounterType);
    return m;
}
```

If this was a file `Py_AtExitDemo.c` then a Python `setup.py` file might look like this:

```
from distutils.core import setup, Extension
setup(
    ext_modules=[
```

(continues on next page)

(continued from previous page)

```
        Extension("pyatexit", sources=['Py_AtExitDemo.c']),
    ]
)
```

Building this with `python3 setup.py build_ext --inplace` we can check everything works as expected:

```
>>> import pyatexit
>>> c = pyatexit.Counter(8)
>>> c.inc()
>>> c.inc()
>>> c.dec()
>>> c.count
9
>>> d = pyatexit.Counter()
>>> d.dec()
>>> d.dec()
>>> d.count
-2
>>> ^D
```

10.8.2 Instrumenting the Counter

To add the instrumentation we will declare a macro `COUNT_ALL_DEC_INC` to control whether the compilation includes instrumentation.

```
#define COUNT_ALL_DEC_INC
```

In the global area of the file declare some global counters and a function to write them out on exit. This must be a `void` function taking no arguments:

```
#ifdef COUNT_ALL_DEC_INC
/* Counters for operations and a function to dump them at Python process end. */
static size_t count_inc = 0;
static size_t count_dec = 0;

static void dump_inc_dec_count(void) {
    fprintf(stdout, "==== dump_inc_dec_count() ====\n");
    fprintf(stdout, "Increments: %" PY_FORMAT_SIZE_T "d\n", count_inc);
    fprintf(stdout, "Decrements: %" PY_FORMAT_SIZE_T "d\n", count_dec);
    fprintf(stdout, "== dump_inc_dec_count() END ==\n");
}
#endif
```

In the `Py_Counter_new` function we add some code to register this function. This must be only done once so we use the static `has_registered_exit_function` to guard this:

```
static PyObject* Py_Counter_new(PyTypeObject* type, PyObject* args,
                                PyObject* kwds) {
    Py_Counter* self;
#ifdef COUNT_ALL_DEC_INC
    static int has_registered_exit_function = 0;
    if (! has_registered_exit_function) {
        if (Py_AtExit(dump_inc_dec_count)) {
            return NULL;
```

(continues on next page)

(continued from previous page)

```
        }
        has_registered_exit_function = 1;
    }
#endif
    self = (Py_Counter*)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->number = 0;
    }
    return (PyObject*)self;
}
```

Note: `Py_AtExit` can take, at most, 32 functions. If the function can not be registered then `Py_AtExit` will return -1.

Warning: Since Python's internal finalization will have completed before the cleanup function, no Python APIs should be called by any registered function.

Now we modify the `inc()` and `dec()` functions thus:

```
static PyObject* Py_Counter_inc(Py_Counter* self) {
    self->number++;
#ifdef COUNT_ALL_DEC_INC
    count_inc++;
#endif
    Py_RETURN_NONE;
}

static PyObject* Py_Counter_dec(Py_Counter* self) {
    self->number--;
#ifdef COUNT_ALL_DEC_INC
    count_dec++;
#endif
    Py_RETURN_NONE;
}
```

Now when we build this extension and run it we see the following:

```
>>> import pyatexit
>>> c = pyatexit.Counter(8)
>>> c.inc()
>>> c.inc()
>>> c.dec()
>>> c.count
9
>>> d = pyatexit.Counter()
>>> d.dec()
>>> d.dec()
>>> d.count
-2
>>> ^D
==== dump_inc_dec_count() ====
Increments: 2
```

(continues on next page)

(continued from previous page)

```
Decrements: 3
== dump_inc_dec_count() END ==
```

CHAPTER

ELEVEN

MEMORY LEAKS

11.1 Tools for Detecting Memory Leaks

11.1.1 `pymemtrace`

The `pymemtrace` package contains a number of tools (written by me) that help detect memory usage and leaks. The documentation contains advice on handling memory leaks.

- On PyPi: https://pypi.org/project/pymemtrace/
- Project: https://github.com/paulross/pymemtrace
- Documentation: https://pymemtrace.readthedocs.io/en/latest/index.html

Note: Some note.

CHAPTER

TWELVE

THREAD SAFETY

If your Extension is likely to be exposed to a multi-threaded environment then you need to think about thread safety. I had this problem in a separate project which was a C++ SkipList which could contain an ordered list of arbitrary Python objects. The problem in a multi-threaded environment was that the following sequence of events could happen:

- Thread A tries to insert a Python object into the SkipList. The C++ code searches for a place to insert it preserving the existing order. To do so it must call back into Python code for the user defined comparison function (using `functools.total_ordering` for example).
- At this point the Python interpreter is free to make a context switch allowing thread B to, say, remove an element from the SkipList. This removal may well invalidate C++ pointers held by thread A.
- When the interpreter switches back to thread A it accesses an invalid pointer and a segfault happens.

The solution, of course, is to use a lock to prevent a context switch until A has completed its insertion, but how? I found the existing Python documentation misleading and I couldn't get it to work reliably, if at all. It was only when I stumbled upon the source code for the bz module that I realised there was a whole other, low level way of doing this, largely undocumented.

Note: Your Python may have been compiled without thread support in which case we don't have to concern ourselves with thread locking. We can discover this from the presence of the macro `WITH_THREAD` so all our thread support code is conditional on the definition of this macro.

12.1 Coding up the Lock

First we need to include pythread.h as well as the usual includes:

```
#include <Python.h>
#include "structmember.h"

#ifdef WITH_THREAD
#include "pythread.h"
#endif
```

12.1.1 Adding a `PyThread_type_lock` to our object

Then we add a `PyThread_type_lock` (an opaque pointer) to the Python structure we are intending to protect. I'll use the example of the SkipList source code. Here is a fragment with the important lines highlighted:

```
typedef struct {
    PyObject_HEAD
    /* Other stuff here... */
#ifdef WITH_THREAD
    PyThread_type_lock lock;
#endif
} SkipList;
```

12.1.2 Creating a class to Acquire and Release the Lock

Now we add some code to acquire and release the lock. We can do this in a RAII fashion in C++ where the constructor blocks until the lock is acquired and the destructor releases the lock. The important lines are highlighted:

```
#ifdef WITH_THREAD
/* A RAII wrapper around the PyThread_type_lock. */
class AcquireLock {
public:
    AcquireLock(SkipList *pSL) : _pSL(pSL) {
        assert(_pSL);
        assert(_pSL->lock);
        if (! PyThread_acquire_lock(_pSL->lock, NOWAIT_LOCK)) {
            Py_BEGIN_ALLOW_THREADS
            PyThread_acquire_lock(_pSL->lock, WAIT_LOCK);
            Py_END_ALLOW_THREADS
        }
    }
    ~AcquireLock() {
        assert(_pSL);
        assert(_pSL->lock);
        PyThread_release_lock(_pSL->lock);
    }
private:
    SkipList *_pSL;
};
#else
/* Make the class a NOP which should get optimised out. */
class AcquireLock {
public:
    AcquireLock(SkipList *) {}
};
#endif
```

The code that acquires the lock is slightly clearer if the Py_BEGIN_ALLOW_THREADS and Py_END_ALLOW_THREADS macros are fully expanded[1]:

```
if (! PyThread_acquire_lock(_pSL->lock, NOWAIT_LOCK)) {
    {
        PyThreadState *_save;
        _save = PyEval_SaveThread();
```

(continues on next page)

[1] I won't pretend to understand all that is going on here, it does work however.

(continued from previous page)

```
        PyThread_acquire_lock(_pSL->lock, WAIT_LOCK);
        PyEval_RestoreThread(_save);
    }
}
```

12.1.3 Initialising and Deallocating the Lock

Now we need to set the lock pointer to `NULL` in the `_new` function:

```
static PyObject *
SkipList_new(PyTypeObject *type, PyObject * /* args */, PyObject * /* kwargs */) {
    SkipList *self = NULL;

    self = (SkipList *)type->tp_alloc(type, 0);
    if (self != NULL) {
        /*
         * Initialise other struct SkipList fields...
         */
#ifdef WITH_THREAD
        self->lock = NULL;
#endif
    }
    return (PyObject *)self;
}
```

In the `__init__` method we allocate the lock by calling `PyThread_allocate_lock()`[2] A lot of this code is specific to the SkipList but the lock allocation code is highlighted:

```
static int
SkipList_init(SkipList *self, PyObject *args, PyObject *kwargs) {
    int ret_val = -1;
    PyObject *value_type    = NULL;
    PyObject *cmp_func      = NULL;
    static char *kwlist[] = {
        (char *)"value_type",
        (char *)"cmp_func",
        NULL
    };
    assert(self);
#ifdef WITH_THREAD
    self->lock = PyThread_allocate_lock();
    if (self->lock == NULL) {
        PyErr_SetString(PyExc_MemoryError, "Unable to allocate thread lock.");
        goto except;
    }
#endif
    /*
     * Much more stuff here...
     */
    assert(! PyErr_Occurred());
    assert(self);
```

(continues on next page)

[2] What I don't understand is why putting this code in the `SkipList_new` function does not work, the lock does not get initialised and segfaults typically in `_pthread_mutex_check_init`. The order has to be: set the lock pointer NULL in `_new`, allocate it in `_init`, free it in `_dealloc`.

(continued from previous page)

```
    assert(self->pSl_void);
    ret_val = 0;
    goto finally;
except:
    assert(PyErr_Occurred());
    Py_XDECREF(self);
    ret_val = -1;
finally:
    return ret_val;
}
```

When deallocating the object we should free the lock pointer with `PyThread_free_lock`[3]:

```
static void
SkipList_dealloc(SkipList *self) {
    /*
     * Deallocate other fields here...
     */
#ifdef WITH_THREAD
        if (self->lock) {
            PyThread_free_lock(self->lock);
            self->lock = NULL;
        }
#endif
        Py_TYPE(self)->tp_free((PyObject*)self);
    }
}
```

12.1.4 Using the Lock

Before any critical code we create an `AcquireLock` object which blocks until we have the lock. Once the lock is obtained we can make any calls, including calls into the Python interpreter without preemption. The lock is automatically freed when we exit the code block:

```
static PyObject *
SkipList_insert(SkipList *self, PyObject *arg) {
    assert(self && self->pSl_void);
    /* Lots of stuff here...
     */
    {
        AcquireLock _lock(self);
        /* We can make calls here, including calls back into the Python
         * interpreter knowing that the interpreter will not preempt us.
         */
        try {
            self->pSl_object->insert(arg);
        } catch (std::invalid_argument &err) {
            // Thrown if PyObject_RichCompareBool returns -1
            // A TypeError should be set
            if (! PyErr_Occurred()) {
```

(continues on next page)

[3] A potiential weakness of this code is that we might be deallocating the lock *whilst the lock is acquired* which could lead to deadlock. This is very much implementation defined in `pythreads` and may vary from platform to platform. There is no obvious API in `pythreads` that allows us to determine if a lock is held so we can release it before deallocation. I notice that in the Python threading module (*Modules/_threadmodule.c*) there is an additional `char` field that acts as a flag to say when the lock is held so that the `lock_dealloc()` function in that module can release the lock before freeing the lock.

(continued from previous page)

```
                PyErr_SetString(PyExc_TypeError, err.what());
            }
            return NULL;
        }
        /* Lock automatically released here. */
    }
    /* More stuff here...
     */
     Py_RETURN_NONE;
}
```

And that is pretty much it.

CHAPTER

THIRTEEN

SOURCE CODE LAYOUT

I find it useful to physically separate out the source code into different categories:

Category	Language	`#include <Python.h>`?	Testable?	Where?	Description
Pure Python	Python	No	Yes	`py/`	Regular Python code tested by pytest or similar.
CPython interface	Mostly C	Yes	No	`cpy/`	C code that defines Python modules and classes. Functions that are exposed directly to Python.
CPython utilities	C, C++	Yes	Yes	`cpy/`	Utility C/C++ code that works with Python objects but these functions that are *not* exposed directly to Python. This code can be tested in a C/C++ environment with a specialised test framework. See *C++ RAII Wrappers Around PyObject** for some examples.
C/C++ core	C, C++	No	Yes	`cpp/`	C/C++ code that knows nothing about Python. This code can be tested in a C/C++ environment with a standard C/C++ test framework.

13.1 Testing CPython Utility Code

When making Python C API calls from a C/C++ environment it is important to initialise the Python interpreter. For example, this small program segfaults:

```
#include <Python.h>

int main(int /* argc */, const char *[] /* argv[] */) {
    /* Forgot this:
    Py_Initialize();
    */
    PyErr_Format(PyExc_TypeError, "Stuff",);
    return 0;
}
```

The reason is that `PyErr_Format` calls `PyThreadState *thread_state = PyThreadState_Get();` theen `thread_state` will be NULL unless the Python interpreter is initialised.

So you need to call `Py_Initialize()` to set up statically allocated interpreter data. Alternativley put `if (! Py_IsInitialized()) Py_Initialize();` in every test. See: https://docs.python.org/3/c-api/init.html

Here are a couple of useful C++ functions that assert all is well that can be used at the begining of any function:

```c
/* Returns non zero if Python is initialised and there is no Python error set.
 * The second version also checks that the given pointer is non-NULL
 * Use this thus, it will do nothing if NDEBUG is defined:
 *
 * assert(cpython_asserts());
 * assert(cpython_asserts(p));
 */
int cpython_asserts() {
    return Py_IsInitialized() && PyErr_Occurred() == NULL;
}

int cpython_asserts(PyObject *pobj) {
    return cpython_asserts() && pobj != NULL;
}
```

CHAPTER

FOURTEEN

USING C++ WITH CPYTHON CODE

Using C++ can take a lot of the pain out of interfacing CPython code, here are some examples.

14.1 C++ RAII Wrappers Around `PyObject*`

It is sometimes useful to wrap up a `PyObject*` in a class that will manage the reference count. Here is a base class that shows the general idea, it takes a `PyObject *` and provides:

- Construction with a `PyObject *` and access this with `operator PyObject*() const`.
- `PyObject **operator&()` to reset the underlying pointer, for example when using it with `PyArg_ParseTupleAndKeywords`.
- Decrementing the reference count on destruction (potientially freeing the object).

```
/** General wrapper around a PyObject*.
 * This decrements the reference count on destruction.
 */
class DecRefDtor {
public:
    DecRefDtor(PyObject *ref) : m_ref { ref } {}
    Py_ssize_t ref_count() const { return m_ref ? Py_REFCNT(m_ref) : 0; }
    // Allow setting of the (optional) argument with PyArg_ParseTupleAndKeywords
    PyObject **operator&() {
        Py_XDECREF(m_ref);
        m_ref = NULL;
        return &m_ref;
    }
    // Access the argument
    operator PyObject*() const { return m_ref; }
    // Test if constructed successfully from the new reference.
    explicit operator bool() { return m_ref != NULL; }
    ~DecRefDtor() { Py_XDECREF(m_ref); }
protected:
    PyObject *m_ref;
};
```

14.1.1 C++ RAII Wrapper for a Borrowed `PyObject*`

There are two useful sub-classes, one for borrowed references, one for new references which are intended to be temporary. Using borrowed references:

```
/** Wrapper around a PyObject* that is a borrowed reference.
 * This increments the reference count on construction and
 * decrements the reference count on destruction.
 */
class BorrowedRef : public DecRefDtor {
public:
    BorrowedRef(PyObject *borrowed_ref) : DecRefDtor(borrowed_ref) {
        Py_XINCREF(m_ref);
    }
};
```

This can be used with borrowed references as follows:

```
void function(PyObject *obj) {
    BorrowedRef(obj); // Increment reference here.
    // ...
} // Decrement reference here.
```

14.1.2 C++ RAII Wrapper for a New `PyObject*`

Here is a sub-class that wraps a new reference to a `PyObject *` and ensures it is free'd when the wrapper goes out of scope:

```
/** Wrapper around a PyObject* that is a new reference.
 * This owns the reference so does not increment it on construction but
 * does decrement it on destruction.
 */
class NewRef : public DecRefDtor {
public:
    NewRef(PyObject *new_ref) : DecRefDtor(new_ref) {}
};
```

This new reference wrapper can be used as follows:

```
void function() {
    NewRef(PyLongFromLong(9)); // New reference here.
    // Use static_cast<PyObject*>(NewRef) ...
} // Decrement the new reference here.
```

14.2 Handling Default Arguments

Handling default, possibly mutable, arguments in a pythonic way is described here: *Being Pythonic with Default Arguments*. It is quite complicated to get it right but C++ can ease the pain with a generic class to simplify handling default arguments in CPython functions:

```
class DefaultArg {
public:
    DefaultArg(PyObject *new_ref) : m_arg { NULL }, m_default { new_ref } {}
```

(continues on next page)

(continued from previous page)

```
    // Allow setting of the (optional) argument with PyArg_ParseTupleAndKeywords
    PyObject **operator&() { m_arg = NULL; return &m_arg; }
    // Access the argument or the default if default.
    operator PyObject*() const { return m_arg ? m_arg : m_default; }
    // Test if constructed successfully from the new reference.
    explicit operator bool() { return m_default != NULL; }
protected:
    PyObject *m_arg;
    PyObject *m_default;
};
```

Suppose we have the Python function signature of `def function(encoding='utf8', cache={}):` then in C/C++ we can do this:

```
PyObject *
function(PyObject * /* module */, PyObject *args, PyObject *kwargs) {
    /* ... */
    static DefaultArg encoding(PyUnicode_FromString("utf8"));
    static DefaultArg cache(PyDict_New());
    /* Check constructed OK. */
    if (! encoding || ! cache) {
        return NULL;
    }
    static const char *kwlist[] = { "encoding", "cache", NULL };
    if (! PyArg_ParseTupleAndKeywords(args, kwargs, "|OO", const_cast<char**>(kwlist),
↪ &encoding, &cache)) {
        return NULL;
    }
    /* Then just use encoding, cache as if they were a PyObject* (possibly
     * might need to be cast to some specific PyObject*). */

    /* ... */
}
```

14.3 Homogeneous Python Containers and C++

Here are some useful generic functions that can convert homogeneous Python containers to and from their C++ STL equivalents. They use templates to identify the C++ type and function pointers to convert from Python to C++ objects and back. These functions must have a these characteristics on error:

- Converting from C++ to Python, on error set a Python error (e.g with `PyErr_SetString` or `PyErr_Format`) and return NULL.
- Converting from Python to C++ set a Python error and return a default C++ object (for example an empty `std::string`).

For illustration here are a couple of such functions that convert `PyBytesObject*` to and from `std::string`:

```
std::string py_bytes_to_std_string(PyObject *py_str) {
    std::string r;
    if (PyBytes_Check(py_str)) {
        r = std::string(PyBytes_AS_STRING(py_str));
    } else {
        PyErr_Format(PyExc_TypeError,
                     "Argument %s must be bytes not \"%s\"",
```

(continues on next page)

(continued from previous page)

```
                        __FUNCTION__, Py_TYPE(py_str)->tp_name);
    }
    return r;
}

PyObject *std_string_to_py_bytes(const std::string &str) {
    return PyBytes_FromStringAndSize(str.c_str(), str.size());
}
```

We can use this for a variety of containers, first Python lists of `bytes`.

14.3.1 Python Lists and C++ `std::vector<T>`

Python `list` to C++ `std::vector<T>`

This converts a python list to a `std::vector<T>`. `ConvertToT` is a function pointer to a function that takes a `PyObject*` and returns an instance of a `T` type. On failure to to convert a `PyObject*` this function should set a Python error (making `PyErr_Occurred()` non-NULL) and return a default `T`. On failure this function sets `PyErr_Occurred()` and the return value will be an empty vector.

```
template <typename T>
std::vector<T>
py_list_to_std_vector(PyObject *py_list, T (*ConvertToT)(PyObject *)) {
    assert(cpython_asserts(py_list));
    std::vector<T> cpp_vector;

    if (PyList_Check(py_list)) {
        cpp_vector.reserve(PyList_GET_SIZE(py_list));
        for (Py_ssize_t i = 0; i < PyList_GET_SIZE(py_list); ++i) {
            cpp_vector.emplace(cpp_vector.end(),
                               (*ConvertToT)(PyList_GetItem(py_list, i)));
            if (PyErr_Occurred()) {
               cpp_vector.clear();
               break;
            }
        }
    } else {
        PyErr_Format(PyExc_TypeError,
                     "Argument \"py_list\" to %s must be list not \"%s\"",
                     __FUNCTION__, Py_TYPE(py_list)->tp_name);
    }
    return cpp_vector;
}
```

If we have a function `std::string py_bytes_to_std_string(PyObject *py_str);` (above) we can use this thus, we have to specify the C++ template specialisation:

```
std::vector<std::string> result = py_list_to_std_vector<std::string>(py_list, &py_
↪bytes_to_std_string);
if (PyErr_Occurred()) {
    // Handle error condition.
} else {
    // All good.
}
```

C++ `std::vector<T>` to Python `list`

And the inverse that takes a C++ `std::vector<T>` and makes a Python `list`. `ConvertToPy` is a pointer to a function that takes an instance of a `T` type and returns a `PyObject*`, this should return NULL on failure and set `PyErr_Occurred()`. On failure this function sets `PyErr_Occurred()` and returns NULL.

```
template <typename T>
PyObject*
std_vector_to_py_list(const std::vector<T> &cpp_vec,
                      PyObject *(*ConvertToPy)(const T&)
                      ) {
    assert(cpython_asserts());
    PyObject *r = PyList_New(cpp_vec.size());
    if (! r) {
        goto except;
    }
    for (Py_ssize_t i = 0; i < cpp_vec.size(); ++i) {
        PyObject *item = (*ConvertToPy)(cpp_vec[i]);
        if (! item || PyErr_Occurred() || PyList_SetItem(r, i, item)) {
            goto except;
        }
    }
    assert(! PyErr_Occurred());
    assert(r);
    goto finally;
except:
    assert(PyErr_Occurred());
    // Clean up list
    if (r) {
        // No PyList_Clear().
        for (Py_ssize_t i = 0; i < PyList_GET_SIZE(r); ++i) {
            Py_XDECREF(PyList_GET_ITEM(r, i));
        }
        Py_DECREF(r);
        r = NULL;
    }
finally:
    return r;
}
```

If we have a function `PyObject *std_string_to_py_bytes(const std::string &str);` (above) we can use this thus:

```
std::vector<std::string> cpp_vector;
// Initialise cpp_vector...
PyObject *py_list = std_vector_to_py_list(cpp_vector, &std_string_to_py_bytes);
if (! py_list) {
    // Handle error condition.
} else {
    // All good.
}
```

14.3.2 Python Sets, Frozensets and C++ `std::unordered_set<T>`

Python `set` to C++ `std::unordered_set<T>`

Convert a Python `set` or `frozenset` to a `std::unordered_set<T>`. `ConvertToT` is a function pointer to a function that takes a `PyObject*` and returns an instance of a `T` type. This function should make `PyErr_Occurred()` true on failure to convert a `PyObject*` and return a default `T`. On failure this sets `PyErr_Occurred()` and the return value will be an empty container.

```
template <typename T>
std::unordered_set<T>
py_set_to_std_unordered_set(PyObject *py_set, T (*ConvertToT)(PyObject *)) {
    assert(cpython_asserts(py_set));
    std::unordered_set<T> cpp_set;

    if (PySet_Check(py_set) || PyFrozenSet_Check(py_set)) {
        // The C API does not allow direct access to an item in a set so we
        // make a copy and pop from that.
        PyObject *set_copy = PySet_New(py_set);
        if (set_copy) {
            while (PySet_GET_SIZE(set_copy)) {
                PyObject *item = PySet_Pop(set_copy);
                if (! item || PyErr_Occurred()) {
                    PySet_Clear(set_copy);
                    cpp_set.clear();
                    break;
                }
                cpp_set.emplace((*ConvertToT)(item));
                Py_DECREF(item);
            }
            Py_DECREF(set_copy);
        } else {
            assert(PyErr_Occurred());
        }
    } else {
        PyErr_Format(PyExc_TypeError,
             "Argument \"py_set\" to %s must be set or frozenset not \"%s\"",
             __FUNCTION__, Py_TYPE(py_set)->tp_name);
    }
    return cpp_set;
}
```

C++ `std::unordered_set<T>` to Python `set` or `frozenset`

Convert a `std::unordered_set<T>` to a new Python `set` or `frozenset`. `ConvertToPy` is a pointer to a function that takes an instance of a `T` type and returns a `PyObject*`, this function should return NULL on failure. On failure this function sets `PyErr_Occurred()` and returns NULL.

```
template <typename T>
PyObject*
std_unordered_set_to_py_set(const std::unordered_set<T> &cpp_set,
                            PyObject *(*ConvertToPy)(const T&),
                            bool is_frozen=false) {
    assert(cpython_asserts());
    PyObject *r = NULL;
```

(continues on next page)

(continued from previous page)

```
    if (is_frozen) {
        r = PyFrozenSet_New(NULL);
    } else {
        r = PySet_New(NULL);
    }
    if (! r) {
        goto except;
    }
    for (auto &iter: cpp_set) {
        PyObject *item = (*ConvertToPy)(iter);
        if (! item || PyErr_Occurred() || PySet_Add(r, item)) {
            goto except;
        }
    }
    assert(! PyErr_Occurred());
    assert(r);
    goto finally;
except:
    assert(PyErr_Occurred());
    // Clean up set
    if (r) {
        PySet_Clear(r);
        Py_DECREF(r);
        r = NULL;
    }
finally:
    return r;
}
```

14.3.3 Python Dicts and C++ `std::unordered_map<K, V>`

Python `dict` to C++ `std::unordered_map<K, V>`

Convert a Python `dict` to a `std::unordered_map<K, V>`. `PyKeyConvertToK` and `PyKeyConvertToK` are function pointers to functions that takes a `PyObject*` and returns an instance of a `K` or `V` type. On failure to convert a `PyObject*` this function should make `PyErr_Occurred()` true and return a default value.

On failure this function will make `PyErr_Occurred()` non-NULL and return an empty map.

```
template <typename K, typename V>
std::unordered_map<K, V>
py_dict_to_std_unordered_map(PyObject *dict,
                             K (*PyKeyConvertToK)(PyObject *),
                             V (*PyValConvertToV)(PyObject *)
                             ) {
    Py_ssize_t pos = 0;
    PyObject *key = NULL;
    PyObject *val = NULL;
    std::unordered_map<K, V> cpp_map;

    if (! PyDict_Check(dict)) {
        PyErr_Format(PyExc_TypeError,
                     "Argument \"dict\" to %s must be dict not \"%s\"",
                     __FUNCTION__, Py_TYPE(dict)->tp_name);
        return cpp_map;
```

(continues on next page)

(continued from previous page)

```
    }
    while (PyDict_Next(dict, &pos, &key, &val)) {
        K cpp_key = (*PyKeyConvertToK)(key);
        if (PyErr_Occurred()) {
            cpp_map.clear();
            break;
        }
        V cpp_val = (*PyValConvertToV)(val);
        if (PyErr_Occurred()) {
            cpp_map.clear();
            break;
        }
        cpp_map.emplace(cpp_key, cpp_val);
    }
    return cpp_map;
}
```

The following expects a Python dict of `{bytes : bytes}` and will convert it to a `std::unordered_map<std::string, std::string>`:

```
std::unordered_map<std::string, std::string> result;
result = py_dict_to_std_unordered_map(py_dict,
                                      &py_bytes_to_std_string,
                                      &py_bytes_to_std_string);
if (PyErr_Occurred()) {
    // Handle failure...
} else {
    // Success...
}
```

C++ `std::unordered_map<K, V>` to Python `dict`

This generic function converts a `std::unordered_map<K, V>` to a new Python `dict`. `KeyConvertToPy`, `ValConvertToPy` are pointers to functions that takes an instance of a `K` or `V` type and returns a `PyObject*`. These should return a new reference on success, NULL on failure.

```
template <typename K, typename V>
PyObject*
std_unordered_map_to_py_dict(const std::unordered_map<K, V> &cpp_map,
                             PyObject *(*KeyConvertToPy)(const K&),
                             PyObject *(*ValConvertToPy)(const V&)
                             ) {
    PyObject *key = NULL;
    PyObject *val = NULL;
    PyObject *r = PyDict_New();

    if (!r) {
        goto except;
    }
    for (auto &iter: cpp_map) {
        key = (*KeyConvertToPy)(iter.first);
        if (! key || PyErr_Occurred()) {
            goto except;
        }
        val = (*ValConvertToPy)(iter.second);
```

(continues on next page)

(continued from previous page)

```
        if (! val || PyErr_Occurred()) {
            goto except;
        }
        if (PyDict_SetItem(r, key, val)) {
            goto except;
        }
    }
    assert(! PyErr_Occurred());
    assert(r);
    goto finally;
except:
    assert(PyErr_Occurred());
    // Clean up dict
    if (r) {
        PyDict_Clear(r);
        Py_DECREF(r);
    }
    r = NULL;
finally:
    return r;
}
```

The following will convert a `std::unordered_map<std::string, std::string>` to a Python dict `{bytes : bytes}`:

```
std::unordered_map<std::string, std::string> cpp_map {
    {"Foo", "Bar"}
};
PyObject *py_dict = std_unordered_map_to_py_dict<std::string, std::string>(
        cpp_map,
        &std_string_to_py_bytes,
        &std_string_to_py_bytes
);
if (! py_dict) {
    // Handle failure...
} else {
    // All good...
}
```

14.4 Python Unicode Strings and C++

Yes Unicode is a pain but it here to stay, particularly with Python 3. This section looks at how you can bridge between Python and C++ unicode in Python extensions. This section is only about Python 3+ and C++11 or more.

Whilst Python is Unicode aware C++ is not, well C++11 added `std::basic_string` specialisations for 2 and 4 byte 'Unicode' characters but these are just containers, they have no real awareness of what they contain.

14.4.1 Basic Handling of Unicode

The task here is to:

1. Take any Python Unicode string as an argument.
2. Convert it into an appropriate C++ container.
3. Dump that C++ container out to `std::cout`.
4. Create and new Python Unicode string from that C++ container and return it.

This is just show that we can round-trip between the internal representations of the two languages.

Here is the despatch function that takes a single Unicode argument (note the `"U"` specification) and calls the appropriate handling function:

```
/* Handler functions, defined later. */
PyObject *unicode_1_to_string_and_back(PyObject *py_str);
PyObject *unicode_2_to_string_and_back(PyObject *py_str);
PyObject *unicode_4_to_string_and_back(PyObject *py_str);

PyObject*
unicode_to_string_and_back(PyObject * /* module */, PyObject *args) {
    PyObject *py_str = NULL;
    PyObject *ret_val = NULL;
    if (PyArg_ParseTuple(args, "U", &py_str)) {
        switch (PyUnicode_KIND(py_str)) {
            case PyUnicode_1BYTE_KIND:
                ret_val = unicode_1_to_string_and_back(py_str);
                break;
            case PyUnicode_2BYTE_KIND:
                ret_val = unicode_2_to_string_and_back(py_str);
                break;
            case PyUnicode_4BYTE_KIND:
                ret_val = unicode_4_to_string_and_back(py_str);
                break;
            default:
                PyErr_Format(PyExc_ValueError,
                             "In %s argument is not recognised as a Unicode 1, 2, 4␣
↪byte string",
                             __FUNCTION__);
                break;
        }
    }
    return ret_val;
}
```

The three handler functions are here, they use `std::string`, `std::u16string` and `std::u32string` as appropriate:

```
PyObject*
unicode_1_to_string_and_back(PyObject *py_str) {
    assert(PyUnicode_KIND(py_str) == PyUnicode_1BYTE_KIND);
    std::string result = std::string((char*)PyUnicode_1BYTE_DATA(py_str));
    dump_string(result);
    return PyUnicode_FromKindAndData(PyUnicode_1BYTE_KIND,
                                     result.c_str(),
                                     result.size());
}
```

(continues on next page)

(continued from previous page)

```
PyObject*
unicode_2_to_string_and_back(PyObject *py_str) {
    assert(PyUnicode_KIND(py_str) == PyUnicode_2BYTE_KIND);
    // std::u16string is a std::basic_string<char16_t>
    std::u16string result = std::u16string((char16_t*)PyUnicode_2BYTE_DATA(py_str));
    dump_string(result);
    return PyUnicode_FromKindAndData(PyUnicode_2BYTE_KIND,
                                     result.c_str(),
                                     result.size());
}

PyObject*
unicode_4_to_string_and_back(PyObject *py_str) {
    assert(PyUnicode_KIND(py_str) == PyUnicode_4BYTE_KIND);
    // std::u32string is a std::basic_string<char32_t>
    std::u32string result = std::u32string((char32_t*)PyUnicode_4BYTE_DATA(py_str));
    dump_string(result);
    return PyUnicode_FromKindAndData(PyUnicode_4BYTE_KIND,
                                     result.c_str(),
                                     result.size());
}
```

Each of these calls `dump_string` which is a template function:

```
template <typename T>
void dump_string(const std::basic_string<T> &str) {
    std::cout << "String size: " << str.size();
    std::cout << " word size: " << sizeof(T) << std::endl;
    for (size_t i = 0; i < str.size(); ++i) {
        std::cout << std::setfill('0');
        std::cout << "0x" << std::hex;
        std::cout << std::setw(2 * sizeof(T)) << static_cast<int>(str[i]);
        std::cout << " " << std::dec << std::setw(8) << static_cast<int>(str[i]);
        std::cout << std::setfill(' ');
        std::cout << " \"" << str[i] << "\""<< std::endl;
    }
}
```

For completeness here is the module code that creates a `cUnicode` module with a single `show()` function:

```
static PyMethodDef cUnicode_Methods[] = {
    {"show", (PyCFunction)unicode_to_string_and_back, METH_VARARGS,
        "Convert a Python unicode string to std::string and back."},
    {NULL, NULL, 0, NULL}        /* Sentinel */
};

static PyModuleDef cUnicodemodule = {
    PyModuleDef_HEAD_INIT,
    "cUnicode",
    "cUnicode works with unicode strings.",
    -1,
    cUnicode_Methods,
    NULL, NULL, NULL, NULL
};

PyMODINIT_FUNC
```

(continues on next page)

(continued from previous page)

```
PyInit_cUnicode(void)
{
    PyObject* m;

    m = PyModule_Create(&cUnicodemodule);
    if (m == NULL)
        return NULL;
    return m;
}
```

Here is an example of using this module:

```
>>> import cUnicode
>>> cUnicode.show('Hello')
String size: 5 word size: 1
0x00000048       72 "H"
0x00000065      101 "e"
0x0000006c      108 "l"
0x0000006c      108 "l"
0x0000006f      111 "o"
'Hello'
>>> s = "a\xac\u1234\u20ac\U00008000"
>>> r = cUnicode.show(s)
String size: 5 word size: 2
0x00000061       97 "97"
0x000000ac      172 "172"
0x00001234     4660 "4660"
0x000020ac     8364 "8364"
0x00008000    32768 "32768"
>>> r == s
True
>>> s = "a\xac\u1234\u20ac\U00018000"
>>> r = cUnicode.show(s)
String size: 5 word size: 4
0x00000061       97 "97"
0x000000ac      172 "172"
0x00001234     4660 "4660"
0x000020ac     8364 "8364"
0x00018000    98304 "98304"
>>> r == s
True
```

14.4.2 Working with `bytes`, `bytearray` and UTF-8 Unicode Arguments

It is fairly common to want to convert an argumennt that is `bytes`, `bytearray` or UTF-8 to a `std::string`. This function willl do just that:

```
/* Convert a PyObject to a std::string and return 0 if succesful.
 * If py_str is Unicode than treat it as UTF-8.
 * This works with Python 2.7 and Python 3.4 onwards.
 */
int py_string_to_std_string(const PyObject *py_str,
                            std::string &result,
                            bool utf8_only=true) {
    result.clear();
```

(continues on next page)

(continued from previous page)

```
    if (PyBytes_Check(py_str)) {
        result = std::string(PyBytes_AS_STRING(py_str));
        return 0;
    }
    if (PyByteArray_Check(py_str)) {
        result = std::string(PyByteArray_AS_STRING(py_str));
        return 0;
    }
    // Must be unicode then.
    if (! PyUnicode_Check(py_str)) {
        PyErr_Format(PyExc_ValueError,
                     "In %s \"py_str\" failed PyUnicode_Check()",
                     __FUNCTION__);
        return -1;
    }
    if (PyUnicode_READY(py_str)) {
        PyErr_Format(PyExc_ValueError,
                     "In %s \"py_str\" failed PyUnicode_READY()",
                     __FUNCTION__);
        return -2;
    }
    if (utf8_only && PyUnicode_KIND(py_str) != PyUnicode_1BYTE_KIND) {
        PyErr_Format(PyExc_ValueError,
                     "In %s \"py_str\" not utf-8",
                     __FUNCTION__);
        return -3;
    }
    // Python 3 and its minor versions (they vary)
    //    const Py_UCS1 *pChrs = PyUnicode_1BYTE_DATA(pyStr);
    //    result = std::string(reinterpret_cast<const char*>(pChrs));
#if PY_MAJOR_VERSION >= 3
    result = std::string((char*)PyUnicode_1BYTE_DATA(py_str));
#else
    // Nasty cast away constness because PyString_AsString takes non-const in Py2
    result = std::string((char*)PyString_AsString(const_cast<PyObject *>(py_str)));
#endif
    return 0;
}
```

And these three do the reverse:

```
PyObject*
std_string_to_py_bytes(const std::string &str) {
    return PyBytes_FromStringAndSize(str.c_str(), str.size());
}

PyObject*
std_string_to_py_bytearray(const std::string &str) {
    return PyByteArray_FromStringAndSize(str.c_str(), str.size());
}

PyObject*
std_string_to_py_utf8(const std::string &str) {
    // Equivelent to:
    // PyUnicode_FromKindAndData(PyUnicode_1BYTE_KIND, str.c_str(), str.size());
    return PyUnicode_FromStringAndSize(str.c_str(), str.size());
}
```

14.5 C++ and the Numpy C API

Numpy is a powerful arrary based data structure with fast vector and array operations. It has a fully featured C API. This section describes some aspects of using Numpy with C++.

14.5.1 Initialising Numpy

The Numpy C API must be setup so that a number of static data structures are initialised correctly. The way to do this is to call `import_array()` which makes a number of Python import statements so the Python interpreter must be initialised first. This is described in detail in the Numpy documentation so this document just presents a cookbook approach.

14.5.2 Verifying Numpy is Initialised

`import_array()` always returns `NUMPY_IMPORT_ARRAY_RETVAL` regardless of success instead we have to check the Python error status:

```
#include <Python.h>
#include "numpy/arrayobject.h" // Include any other Numpy headers, UFuncs for example.

// Initialise Numpy
import_array();
if (PyErr_Occurred()) {
    std::cerr << "Failed to import numpy Python module(s)." << std::endl;
    return NULL; // Or some suitable return value to indicate failure.
}
```

In other running code where Numpy is expected to be initialised then `PyArray_API` should be non-NULL and this can be asserted:

```
assert(PyArray_API);
```

14.5.3 Numpy Initialisation Techniques

Initialising Numpy in a CPython Module

Taking the simple example of a module from the Python documentation we can add Numpy access just by including the correct Numpy header file and calling `import_numpy()` in the module initialisation code:

```
#include <Python.h>

#include "numpy/arrayobject.h" // Include any other Numpy headers, UFuncs for example.

static PyMethodDef SpamMethods[] = {
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};

static struct PyModuleDef spammodule = {
   PyModuleDef_HEAD_INIT,
   "spam",   /* name of module */
   spam_doc, /* module documentation, may be NULL */
```

(continues on next page)

(continued from previous page)

```
   -1,       /* size of per-interpreter state of the module,
                or -1 if the module keeps state in global variables. */
   SpamMethods
};

PyMODINIT_FUNC
PyInit_spam(void) {
    ...
    assert(! PyErr_Occurred());
    import_numpy(); // Initialise Numpy
    if (PyErr_Occurred()) {
        return NULL;
    }
    ...
    return PyModule_Create(&spammodule);
}
```

That is fine for a singular translation unit but you have multiple translation units then each has to initialise the Numpy API which is a bit extravagant. The following sections describe how to manage this with multiple translation units.

Initialising Numpy in Pure C++ Code

This is mainly for development and testing of C++ code that uses Numpy. Your code layout might look something like this where `main.cpp` has a `main()` entry point and `class.h` has your class declarations and `class.cpp` has their implementations, like this:

```
.
└── src
    └── cpp
        ├── class.cpp
        ├── class.h
        └── main.cpp
```

The way of managing Numpy initialisation and access is as follows. In `class.h` choose a unique name such as `awesome_project` then include:

```
#define PY_ARRAY_UNIQUE_SYMBOL awesome_project_ARRAY_API
#include "numpy/arrayobject.h"
```

In the implementation file `class.cpp` we do not want to import Numpy as that is going to be handled by `main()` in `main.cpp` so we put this at the top:

```
#define NO_IMPORT_ARRAY
#include "class.h"
```

Finally in `main.cpp` we initialise Numpy:

```
#include "Python.h"
#include "class.h"

int main(int argc, const char * argv[]) {
    // ...
    // Initialise the Python interpreter
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
```

(continues on next page)

(continued from previous page)

```
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program);  /* optional but recommended */
    Py_Initialize();
    // Initialise Numpy
    import_array();
    if (PyErr_Occurred()) {
        std::cerr << "Failed to import numpy Python module(s)." << std::endl;
        return -1;
    }
    assert(PyArray_API);
    // ...
}
```

If you have multiple .h, .cpp files then it might be worth having a single .h file, say `numpy_init.h` with just this in:

```
#define PY_ARRAY_UNIQUE_SYMBOL awesome_project_ARRAY_API
#include "numpy/arrayobject.h"
```

Then each implementation .cpp file has:

```
#define NO_IMPORT_ARRAY
#include "numpy_init.h"
#include "class.h" // Class declarations
```

And `main.cpp` has:

```
#include "numpy_init.h"
#include "class_1.h"
#include "class_2.h"
#include "class_3.h"

int main(int argc, const char * argv[]) {
    // ...
    import_array();
    if (PyErr_Occurred()) {
        std::cerr << "Failed to import numpy Python module(s)." << std::endl;
        return -1;
    }
    assert(PyArray_API);
    // ...
}
```

Initialising Numpy in a CPython Module using C++ Code

Supposing you have laid out your source code in the following fashion:

```
.
└── src
    ├── cpp
    │   ├── class.cpp
    │   └── class.h
    └── cpython
        └── module.c
```

This is a hybrid of the above and typical for CPython C++ extensions where `module.c` contains the CPython code that allows Python to access the pure C++ code.

The code in `class.h` and `class.cpp` is unchanged and the code in `module.c` is essentially the same as that of a CPython module as described above where `import_array()` is called from within the `PyInit_<module>` function.

How These Macros Work Together

The two macros `PY_ARRAY_UNIQUE_SYMBOL` and `NO_IMPORT_ARRAY` work together as follows:

	`PY_ARRAY_UNIQUE_SYMBOL` NOT defined	**`PY_ARRAY_UNIQUE_SYMBOL`** defined as <NAME>
`NO_IMPORT_ARRAY` not defined	C API is declared as: `static void **PyArray_API` Which makes it only available to that translation unit.	C API is declared as: `void **<NAME>` so can be seen by other translation units.
`NO_IMPORT_ARRAY` defined	C API is declared as: `extern void **PyArray_API` so is available from another translation unit.	C API is declared as: `extern void **<NAME>` so is available from another translation unit.

Adding a Search Path to a Virtual Environment

If you are linking to the system Python this may not have numpy installed, here is a way to cope with that. Create a virtual environment from the system python and install numpy:

```
python -m venv <PATH_TO_VIRTUAL_ENVIRONMENT>
source <PATH_TO_VIRTUAL_ENVIRONMENT>/bin/activate
pip install numpy
```

Then in your C++ entry point add this function that manipulates `sys.path`:

```
/** Takes a path and adds it to sys.paths by calling PyRun_SimpleString.
 * This does rather laborious C string concatenation so that it will work in
 * a primitive C environment.
 *
 * Returns 0 on success, non-zero on failure.
 */
int add_path_to_sys_module(const char *path) {
    int ret = 0;
    const char *prefix = "import sys\nsys.path.append(\"";
    const char *suffix = "\")\n";
    char *command = (char*)malloc(strlen(prefix)
                                  + strlen(path)
                                  + strlen(suffix)
                                  + 1);
    if (! command) {
        return -1;
    }
    strcpy(command, prefix);
```

(continues on next page)

(continued from previous page)

```
    strcat(command, path);
    strcat(command, suffix);
    ret = PyRun_SimpleString(command);
#ifdef DEBUG
    printf("Calling PyRun_SimpleString() with:\n");
    printf("%s", command);
    printf("PyRun_SimpleString() returned: %d\n", ret);
    fflush(stdout);
#endif
    free(command);
    return ret;
}
```

`main()` now calls this with the path to the virtual environment `site-packages`:

```
int main(int argc, const char * argv[]) {
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    // Initialise the interpreter.
    Py_SetProgramName(program);  /* optional but recommended */
    Py_Initialize();
    const char *multiarray_path = "<PATH_TO_VIRTUAL_ENVIRONMENT_SITE_PACKAGES>";
    add_path_to_sys_module(multiarray_path);
    import_array();
    if (PyErr_Occurred()) {
        std::cerr << "Failed to import numpy Python module(s)." << std::endl;
        return -1;
    }
    assert(PyArray_API);
    // Your code here...
}
```

CHAPTER

FIFTEEN

PICKLING C EXTENSION TYPES

If you need to provide support for pickling your specialised types from your C extension then you need to implement some special functions.

This example shows you how to provided pickle support for for the `custom2.Custom` type described in the C extension tutorial in the Python documentation.

15.1 Pickle Version Control

Since the whole point of `pickle` is persistence then pickled objects can hang around in databases, file systems, data from the shelve module and whatnot for a long time. It is entirely possible that when un-pickled, sometime in the future, that your C extension has moved on and then things become awkward.

It is *strongly* recommended that you add some form of version control to your pickled objects. In this example I just have a single integer version number which I write to the pickled object. If the number does not match on unpickling then I raise an exception. When I change the type API I would, judiciously, change this version number.

Clearly more sophisticated strategies are possible by supporting older versions of the pickled object in some way but this will do for now.

We add some simple pickle version information to the C extension:

```
static const char* PICKLE_VERSION_KEY = "_pickle_version";
static int PICKLE_VERSION = 1;
```

Now we can implement `__getstate__` and `__setstate__`, think of these as symmetric operations. First `__getstate__`.

15.2 Implementing `__getstate__`

`__getstate__` pickles the object. `__getstate__` is expected to return a dictionary of the internal state of the `Custom` object. Note that a `Custom` object has two Python objects (`first` and `last`) and a C integer (`number`) that need to be converted to a Python object. We also need to add the version information.

Here is the C implementation:

```
1 /* Pickle the object */
2 static PyObject *
3 Custom___getstate__(CustomObject *self, PyObject *Py_UNUSED(ignored)) {
4     PyObject *ret = Py_BuildValue("{sOsOsisi}",
5                                   "first", self->first,
```

(continues on next page)

(continued from previous page)

```
                                    "last", self->last,
                                    "number", self->number,
                                    PICKLE_VERSION_KEY, PICKLE_VERSION);
    return ret;
}
```

15.3 Implementing `__setstate__`

The implementation of `__setstate__` un-pickles the object. This is a little more complicated as there is quite a lot of error checking going on. We are being passed an arbitrary Python object and need to check:

- It is a Python dictionary.
- It has a version key and the version value is one that we can deal with.
- It has the required keys and values to populate our `Custom` object.

Note that our `__new__` method (`Custom_new()`) has already been called on `self`. Before setting any member value we need to de-allocate the existing value set by `Custom_new()` otherwise we will have a memory leak.

15.3.1 Error Checking

```
/* Un-pickle the object */
static PyObject *
Custom___setstate__(CustomObject *self, PyObject *state) {
    /* Error check. */
    if (!PyDict_CheckExact(state)) {
        PyErr_SetString(PyExc_ValueError, "Pickled object is not a dict.");
        return NULL;
    }
    /* Version check. */
    /* Borrowed reference but no need to increment as we create a C long
     * from it. */
    PyObject *temp = PyDict_GetItemString(state, PICKLE_VERSION_KEY);
    if (temp == NULL) {
        /* PyDict_GetItemString does not set any error state so we have to. */
        PyErr_Format(PyExc_KeyError, "No \"%s\" in pickled dict.",
                     PICKLE_VERSION_KEY);
        return NULL;
    }
    int pickle_version = (int) PyLong_AsLong(temp);
    if (pickle_version != PICKLE_VERSION) {
        PyErr_Format(PyExc_ValueError,
                     "Pickle version mismatch. Got version %d but expected version %d.
↪",
                     pickle_version, PICKLE_VERSION);
        return NULL;
    }
```

15.3.2 Set the `first` Member

```
/* NOTE: Custom_new() will have been invoked so self->first and self->last
 * will have been allocated so we have to de-allocate them. */
Py_DECREF(self->first);
self->first = PyDict_GetItemString(state, "first"); /* Borrowed reference. */
if (self->first == NULL) {
    /* PyDict_GetItemString does not set any error state so we have to. */
    PyErr_SetString(PyExc_KeyError, "No \"first\" in pickled dict.");
    return NULL;
}
/* Increment the borrowed reference for our instance of it. */
Py_INCREF(self->first);
```

15.3.3 Set the `last` Member

```
/* Similar to self->first above. */
Py_DECREF(self->last);
self->last = PyDict_GetItemString(state, "last"); /* Borrowed reference. */
if (self->last == NULL) {
    /* PyDict_GetItemString does not set any error state so we have to. */
    PyErr_SetString(PyExc_KeyError, "No \"last\" in pickled dict.");
    return NULL;
}
Py_INCREF(self->last);
```

15.3.4 Set the `number` Member

This is a C fundamental type so the code is slightly different:

```
/* Borrowed reference but no need to incref as we create a C long from it. */
PyObject *number = PyDict_GetItemString(state, "number");
if (number == NULL) {
    /* PyDict_GetItemString does not set any error state so we have to. */
    PyErr_SetString(PyExc_KeyError, "No \"number\" in pickled dict.");
    return NULL;
}
self->number = (int) PyLong_AsLong(number);
```

And we are done.

```
    Py_RETURN_NONE;
}
```

15.3.5 `__setstate__` in Full

```
/* Un-pickle the object */
static PyObject *
Custom___setstate__(CustomObject *self, PyObject *state) {
    /* Error check. */
    if (!PyDict_CheckExact(state)) {
        PyErr_SetString(PyExc_ValueError, "Pickled object is not a dict.");
        return NULL;
    }
    /* Version check. */
    /* Borrowed reference but no need to increment as we create a C long
     * from it. */
    PyObject *temp = PyDict_GetItemString(state, PICKLE_VERSION_KEY);
    if (temp == NULL) {
        /* PyDict_GetItemString does not set any error state so we have to. */
        PyErr_Format(PyExc_KeyError, "No \"%s\" in pickled dict.",
                     PICKLE_VERSION_KEY);
        return NULL;
    }
    int pickle_version = (int) PyLong_AsLong(temp);
    if (pickle_version != PICKLE_VERSION) {
        PyErr_Format(PyExc_ValueError,
                     "Pickle version mismatch. Got version %d but expected version %d.
↪",
                     pickle_version, PICKLE_VERSION);
        return NULL;
    }

    /* NOTE: Custom_new() will have been invoked so self->first and self->last
     * will have been allocated so we have to de-allocate them. */
    Py_DECREF(self->first);
    self->first = PyDict_GetItemString(state, "first"); /* Borrowed reference. */
    if (self->first == NULL) {
        /* PyDict_GetItemString does not set any error state so we have to. */
        PyErr_SetString(PyExc_KeyError, "No \"first\" in pickled dict.");
        return NULL;
    }
    /* Increment the borrowed reference for our instance of it. */
    Py_INCREF(self->first);

    /* Similar to self->first above. */
    Py_DECREF(self->last);
    self->last = PyDict_GetItemString(state, "last"); /* Borrowed reference. */
    if (self->last == NULL) {
        /* PyDict_GetItemString does not set any error state so we have to. */
        PyErr_SetString(PyExc_KeyError, "No \"last\" in pickled dict.");
        return NULL;
    }
    Py_INCREF(self->last);

    /* Borrowed reference but no need to incref as we create a C long from it. */
    PyObject *number = PyDict_GetItemString(state, "number");
    if (number == NULL) {
        /* PyDict_GetItemString does not set any error state so we have to. */
        PyErr_SetString(PyExc_KeyError, "No \"number\" in pickled dict.");
        return NULL;
```

(continues on next page)

(continued from previous page)

```
    }
    self->number = (int) PyLong_AsLong(number);

    Py_RETURN_NONE;
}
```

15.4 Add the Special Methods

Now we need to add these two special methods to the methods table which now looks like this:

```
static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
             "Return the name, combining the first and last name"
    },
    {"__getstate__", (PyCFunction) Custom___getstate__, METH_NOARGS,
             "Pickle the Custom object"
    },
    {"__setstate__", (PyCFunction) Custom___setstate__, METH_O,
             "Un-pickle the Custom object"
    },
    {NULL}  /* Sentinel */
};
```

15.5 Pickling a `custom2.Custom` Object

We can test this with code like this that pickles one `custom2.Custom` object then creates another `custom2.Custom` object from that pickle. Here is some Python code that exercises our module:

```
import pickle

import custom2

original = custom2.Custom('FIRST', 'LAST', 11)
print(f'original is {original} @ 0x{id(original):x}')
print(f'original first: {original.first} last: {original.last} number: {original.
↪number} name: {original.name()}')
pickled_value = pickle.dumps(original)
print(f'Pickled original is {pickled_value}')
result = pickle.loads(pickled_value)
print(f'result is {result} @ 0x{id(result):x}')
print(f'result first: {result.first} last: {result.last} number: {result.number}␣
↪name: {result.name()}')
```

```
$ python main.py
original is <custom2.Custom object at 0x1049e6810> @ 0x1049e6810
original first: FIRST last: LAST number: 11 name: FIRST LAST
Pickled original is b'\x80\x04\x95[\x00\x00\x00\x00\x00\x00\x00\x8c\x07custom2\x94\
↪x8c\x06Custom\x94\x93\x94)\x81\x94}\x94(\x8c\x05first\x94\x8c\x05FIRST\x94\x8c\
↪x04last\x94\x8c\x04LAST\x94\x8c\x06number\x94K\x0b\x8c\x0f_pickle_version\x94K\
↪x01ub.'
result is <custom2.Custom object at 0x1049252d0> @ 0x1049252d0
```

(continues on next page)

(continued from previous page)

```
result first: FIRST last: LAST number: 11 name: FIRST LAST
```

So we have pickled one object and recreated a different, but equivalent, instance from the pickle of the original object which is what we set out to do.

15.6 The Pickled Object in Detail

If you are curious about the contents of the pickled object the the Python standard library provides the pickletools module. This allows you to inspect the pickled object. So if we run this code:

```
import pickle
import pickletools

import custom2

original = custom2.Custom('FIRST', 'LAST', 11)
pickled_value = pickle.dumps(original)
print(f'Pickled original is {pickled_value}')
# NOTE: Here we are adding annotations.
pickletools.dis(pickled_value, annotate=1)
```

The output will be something like this:

```
Pickled original is b'\x80\x04\x95[\x00\x00\x00\x00\x00\x00\x00\x8c\x07custom2\x94\
↪x8c\x06Custom\x94\x93\x94)\x81\x94}\x94(\x8c\x05first\x94\x8c\x05FIRST\x94\x8c\
↪x04last\x94\x8c\x04LAST\x94\x8c\x06number\x94K\x0b\x8c\x0f_pickle_version\x94K\
↪x01ub.'
    0: \x80 PROTO      4 Protocol version indicator.
    2: \x95 FRAME      91 Indicate the beginning of a new frame.
   11: \x8c SHORT_BINUNICODE 'custom2' Push a Python Unicode string object.
   20: \x94 MEMOIZE    (as 0)          Store the stack top into the memo.  The stack␣
↪is not popped.
   21: \x8c SHORT_BINUNICODE 'Custom'  Push a Python Unicode string object.
   29: \x94 MEMOIZE    (as 1)          Store the stack top into the memo.  The stack␣
↪is not popped.
   30: \x93 STACK_GLOBAL               Push a global object (module.attr) on the␣
↪stack.
   31: \x94 MEMOIZE    (as 2)          Store the stack top into the memo.  The stack␣
↪is not popped.
   32: )    EMPTY_TUPLE                Push an empty tuple.
   33: \x81 NEWOBJ                     Build an object instance.
   34: \x94 MEMOIZE    (as 3)          Store the stack top into the memo.  The stack␣
↪is not popped.
   35: }    EMPTY_DICT                 Push an empty dict.
   36: \x94 MEMOIZE    (as 4)          Store the stack top into the memo.  The stack␣
↪is not popped.
   37: (    MARK                       Push markobject onto the stack.
   38: \x8c     SHORT_BINUNICODE 'first' Push a Python Unicode string object.
   45: \x94     MEMOIZE    (as 5)        Store the stack top into the memo.  The␣
↪stack is not popped.
   46: \x8c     SHORT_BINUNICODE 'FIRST' Push a Python Unicode string object.
   53: \x94     MEMOIZE    (as 6)        Store the stack top into the memo.  The␣
↪stack is not popped.
   54: \x8c     SHORT_BINUNICODE 'last'  Push a Python Unicode string object.
```

(continues on next page)

(continued from previous page)

```
   60: \x94     MEMOIZE    (as 7)        Store the stack top into the memo.  The␣
↪stack is not popped.
   61: \x8c     SHORT_BINUNICODE 'LAST'  Push a Python Unicode string object.
   67: \x94     MEMOIZE    (as 8)        Store the stack top into the memo.  The␣
↪stack is not popped.
   68: \x8c     SHORT_BINUNICODE 'number' Push a Python Unicode string object.
   76: \x94     MEMOIZE    (as 9)         Store the stack top into the memo.  The␣
↪stack is not popped.
   77: K        BININT1    11             Push a one-byte unsigned integer.
   79: \x8c     SHORT_BINUNICODE '_pickle_version' Push a Python Unicode string␣
↪object.
   96: \x94     MEMOIZE    (as 10)                Store the stack top into the memo.␣
↪ The stack is not popped.
   97: K        BININT1    1                      Push a one-byte unsigned integer.
   99: u        SETITEMS   (MARK at 37)           Add an arbitrary number of␣
↪key+value pairs to an existing dict.
  100: b    BUILD                                 Finish building an object, via __
↪setstate__ or dict update.
  101: .    STOP                                  Stop the unpickling machine.
highest protocol among opcodes = 4
```

15.7 Pickling Objects with External State

This is just a simple example, if your object relies on external state such as open files, databases and the like you need to be careful, and knowledgeable about your state management. There is some useful information here: Handling Stateful Objects

15.8 References

- Python API documentation for __setstate__
- Python API documentation for __getstate__
- Useful documentation for Handling Stateful Objects
- Python pickle module
- Python shelve module

CHAPTER

SIXTEEN

MISCELLANEOUS

16.1 No `PyInit_...` Function Found

This is probably Mac OS X and Clang specific but when you import your extension and you get an error like:

```
ImportError: dynamic module does not define module export function
(PyInit_Foo)
```

Have a look at the binary.

```
$ nm -m Foo.cpython-36m-darwin.so | grep Init
00000000000010d0 (__TEXT,__text) non-external (was a private external) _PyInit_Foo
```

Sometimes (why?) clang does not make the symbol external. I have found that adding `__attribute__((visibility("default")))` to the module initialisation function can fix this:

```
__attribute__((visibility("default")))
PyMODINIT_FUNC
PyInit_Foo(void) {
    /* ... */
}
```

And the binary now looks like this:

```
$ nm -m Foo.cpython-36m-darwin.so | grep Init
00000000000010d0 (__TEXT,__text) external _PyInit_Foo
```

CHAPTER

SEVENTEEN

FURTHER READING

17.1 Useful Links

17.1.1 C Extensions

- python.org Tutorial: https://docs.python.org/3/extending/index.html
- python.org C/C++ reference: https://docs.python.org/3/c-api/index.html
- **Joe Jevnik's "How to Write and Debug C Extension Modules":**
  - Documentation: https://lllllllllll.github.io/c-extension-tutorial/index.html
  - Code: https://github.com/lllllllllll/c-extension-tutorial

17.1.2 Porting to Python 3

- General, and comprehensive: http://python3porting.com/
- Porting Python 2 code to Python 3: https://docs.python.org/3/howto/pyporting.html
- Porting C Extensions to Python 3: https://docs.python.org/3/howto/cporting.html
- **`py3c`: Python 2/3 compatibility layer for C extensions:**
  - Documentation: https://py3c.readthedocs.io/en/latest/
  - Project: https://github.com/encukou/py3c

CHAPTER

EIGHTEEN

INDICES AND TABLES

- search