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.

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:

```c
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:

```c
#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:

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:

```c
#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:

```c
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.

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:

```c
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:

```c
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.

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.

“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:

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

    pA = PyLong_FromLong(a);        /* pA: New reference. */
    pB = PyLong_FromLong(b);        /* pB: New reference. */
    r = PyNumber_Subtract(pA, pB);  /*  r: New reference. */
    Py_DECREF(pA);                  /* My responsibility to decref. */
    Py_DECREF(pB);                  /* My responsibility to decref. */
    return r;                       /* Callers responsibility to decref. */
}
```

`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:

```c
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.

"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.

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. */
 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.

"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:

```c
static PyObject *pop_and_print_BAD(PyObject *pList) {
    PyObject *pLast;

    pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);
    fprintf(stdout, "Ref count was: %zd\n", pLast->ob_refcnt);
    do_something(pList);
    fprintf(stdout, "Ref count now: %zd\n", pLast->ob_refcnt);
    PyObject_Print(pLast, stdout, 0);
    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!

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

```python
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)
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:

```c
static PyObject *pop_and_print_BAD(PyObject *pList) {
    PyObject *pLast;

    pLast = PyList_GetItem(pList, PyList_Size(pList) - 1);
    Py_INCREF(pLast);       /* Prevent pLast being deallocated. */
    /* ... */
    do_something(pList);
    /* ... */
    Py_DECREF(pLast);       /* No longer interested in pLast, it might     */
    pLast = NULL;           /* get deallocated here but we shouldn't care. */
    /* ... */
    Py_RETURN_NONE;
}
```

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.

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.

Footnotes

[1] Huge, but pretty consistent once mastered.

[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.

[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:

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