py3c Documentation

Release 0.6

Petr Viktorin

May 19, 2016

Contents

This is *py3c*, a library for easing porting C extensions to Python 3, providing macros for *single-source compatibility* between Python 2.6, 2.7, and 3.3+. It could be described as "the six for C extensions".

Pick the docs you wish to read:

- Porting guide

  A detailed **walkthrough** for porting to Python 3. Read if you wish to start porting a C extension to Python 3.

- Cheatsheet

  A **quick reference**, helpful if you're in the middle of porting. Also useful if you find yourself working on a project that someone else is porting, and don't understand what's going on.

  If you want something to print out a stick on your wall, use this – compared to the other docs, you'll save trees.

- Definition Summary

  A **table** summarizing how py3c's macros are defined. Convenient if you already know the differences between Python 2 and 3, or before a dive into py3c's internals.

  Also serves as a summary of where py3c provides the Python 3 API, and where it resorts to inventing its own macros.

- Reference

  Lists **every macro** py3c defines. The search will point you here when it finds a term.

- Index and Search

  Head here if you're looking for something specific.

CHAPTER 1

Project info

The py3c library is available under the MIT license. This documentation is available under the Creative Commons Attribution-ShareAlike 3.0 Unported license. May they serve you well.

The high-level history of py3c is chronicled in the Changelog.

If you'd like to contribute code, words, suggestions, bug reports, or anything else, do so at the Github page. For more info, see Contributing.

Oh, and you should pronounce "py3c" with a hard "c", if you can manage to do so.

1.1 Porting guide for Python C Extensions

This guide is written for authors of *C extensions* for Python, who want to make their extension compatible with Python 3. It provides comprehensive, step-by-step porting instructions.

Before you start adding Python 3 compatibility to your C extension, consider your options:

- If you are writing a wrapper for a C library, take a look at CFFI, a C Foreign Function Interface for Python. This lets you call C from Python 2.6+ and 3.3+, as well as PyPy. A C compiler is required for development, but not for installation.
- For more complex code, consider Cython, which compiles a Python-like language to C, has great support for interfacing with C libraries, and generates code that works on Python 2.6+ and 3.3+.

Using CFFI or Cython will make your code more maintainable in the long run, at the cost of rewriting the entire extension. If that's not an option, you will need to update the extension to use Python 3 APIs. This is where py3c can help.

This is an *opinionated* guide to porting. It does not enumerate your options, but rather provides one tried way of doing things.

This doesn't mean you can't do things your way – for example, you can cherry-pick the macros you need and put them directly in your files. However, dedicated headers for backwards compatibility will make them easier to find when the time comes to remove them.

If you want more details, consult the "Migrating C extensions" chapter from Lennart Regebro's book "Porting to Python 3", the C porting guide from Python documentation, and the py3c headers for macros to use.

The py3c library lives at Github. See the README for installation instructions.

1.1.1 Modernization

Before porting a C extension to Python 3, you'll need to make sure that you're not using features deprecated even in Python 2. Also, many of Python 3's improvements have been backported to Python 2.6, and using them will make the porting process easier.

For all changes you do, be sure add tests to ensure you do not break anything.

Comparisons

Python 2.1 introduced *rich comparisons* for custom objects, allowing separate behavior for the ==, !=, <, >, <=, >= operators, rather than calling one `__cmp__` function and interpreting its result according to the requested operation. (See PEP 207 for details.)

In Python 3, the original `__cmp__`-based object comparison is removed, so all code needs to switch to rich comparisons. Instead of a

```
static int cmp(PyObject *obj1, PyObject *obj2)
```

function in the `tp_compare` slot, there is now a

```
static PyObject* richcmp(PyObject *obj1, PyObject *obj2, int op)
```

in the `tp_richcompare` slot. The `op` argument specifies the comparison operation: Py_EQ (==), Py_GT (>), Py_LE (<=), etc.

Additionally, Python 3 brings a semantic change. Previously, objects of disparate types were ordered according to type, where the ordering of types was undefined (but consistent across, at least, a single invocation of Python). In Python 3, objects of different types are unorderable. It is usually possible to write a comparison function that works for both versions by returning NotImplemented to explicitly fall back to default behavior.

To help porting from `__cmp__` operations, py3c defines a convenience macro, PY3C_RICHCMP, which evaluates to the right PyObject * result based on two values orderable by C's comparison operators. A typical rich comparison function will look something like this:

```
static PyObject* mytype_richcmp(PyObject *obj1, PyObject *obj2, int op)
{
    if (mytype_Check(obj2)) {
        return PY3C_RICHCMP(get_data(obj1), get_data(obj2), op);
    }
    Py_RETURN_NOTIMPLEMENTED;
}
```

where `get_data` returns an orderable C value (e.g. a pointer or int), and mytype_Check checks if `get_data` is of the correct type (usually via PyObject_TypeCheck). Note that the first argument, obj1, is guaranteed to be of the type the function is defined for.

If a "cmp"-style function is provided by the C library, use `PY3C_RICHCMP(mytype_cmp(obj1, obj2), 0, op)`.

Also, py3c defines the Py_RETURN_NOTIMPLEMENTED macro if it's not provided by your Python version (3.3 and lower).

Note that if you use PY3C_RICHCMP, you will need to include the header `py3c/comparison.h` (or copy the macro to your code) even after your port to Python 3 is complete. The is also needed for Py_RETURN_NOTIMPLEMENTED until you drop support for Python 3.3.

Note: The `tp_richcompare` slot is inherited in subclasses together with `tp_hash` and (in Python 2) `tp_compare`: iff the subclass doesn't define any of them, all are inherited.

This means that if a class is modernized, its subclasses don't have to be, *unless* the subclass manipulates compare/hash slots after class creation (e.g. after the `PyType_Ready` call).

PyObject Structure Members

To conform to C's strict aliasing rules, PyObject_HEAD, which provides members such as `ob_type` and `ob_refcnt`, is a separate struct in Python 3. Access to these members is provided by macros, which have been ported to Python 2.6:

Instead of	use
obj->ob_type	Py_TYPE(obj)
obj->ob_refcnt	Py_REFCNT(obj)
obj->ob_size	Py_SIZE(obj)

And for initialization of type objects, the sequence

```
PyObject_HEAD_INIT(NULL)
0, /* ob_size */
```

must be replaced with

```
PyVarObject_HEAD_INIT(NULL, 0)
```

Adding module-level constants

Often, module initialization uses code like this:

```
PyModule_AddObject(m, "RDWR", PyInt_FromLong(O_RDWR));
PyModule_AddObject(m, "__version__", PyString_FromString("6.28"));
```

Python 2.6 introduced convenience functions, which are shorter to write:

```
PyModule_AddIntConstant(m, "RDWR", O_RDWR)
PyModule_AddStringConstant(m, "__version__", "6.28")
```

These will use native int and str types in both Python versions.

New-Style Classes

The old-style classes (`PyClass_*` and `PyInstance_*`) will be removed in Python 3. Instead, use type objects, which have been available since Python 2.2.

PyCObject to PyCapsule

The PyCObject API has been removed in Python 3.3. You should instead use its replacement, PyCapsule, which is available in Python 2.7 and 3.1+. For the rationale behind Capsules, see CPython issue 5630.

If you need to support Python 2.6, you can use `capsulethunk.h`, which implements the PyCapsule API (with some limitations) in terms of PyCObject. For instructions, see the chapter PyCapsule API for Python 2.6.

The port to PyCapsule API should be straightforward:

- Instead of `PyCObject_FromVoidPtr(obj, destr)`, use `PyCapsule_New(obj, name, destr)`. If the capsule will be available as a module attribute, use `"<modulename>.<attrname>"` for *name*. Otherwise, use your best judgment, but try making the name unique.

- Instead of `PyCObject_FromVoidPtrAndDesc(obj, desc, destr)`, use `PyCapsule_New()` as above; then call `PyCapsule_SetContext(obj, desc)`.
- Instead of `PyCObject_AsVoidPtr(obj)`, use `PyCapsule_GetPointer(obj, name)`. You will need to provide a capsule name, which is checked at runtime as a form of type safety.
- Instead of `PyCObject_GetDesc()`, use `PyCapsule_GetContext()`.
- Instead of `PyCObject_SetVoidPtr()`, use `PyCapsule_SetPointer()`.
- Change all CObject destructors to `PyCapsule destructors`, which take the PyCapsule object as their only argument.

Done!

When your project is sufficiently modernized, and the tests still pass under Python 2, you're ready to start the actual Porting.

1.1.2 Porting – Adding Support for Python 3

After you modernize your C extension to use the latest features available in Python 2, it is time to address the differences between Python 2 and 3.

The recommended way to port is keeping single-source compatibility between Python 2 and 3, until support Python 2 can be safely dropped. For Python code, you can use libraries like six and future, and, failing that, `if sys.version_info >= (3, 0):` blocks for conditional code. For C, the py3c library provides common tools, and for special cases you can use conditional compilation with `#if IS_PY3`.

To start using py3c, `#include <py3c.h>`, and instruct your compiler to find the header.

The Bytes/Unicode split

The most painful change for extension authors is the bytes/unicode split: unlike Python 2's `str` or C's `char*`, Python 3 introduces a sharp divide between *human-readable strings* and *binary data*. You will need to decide, for each string value you use, which of these two types you want.

Make the division as sharp as possible: mixing the types tends to lead to utter chaos. Functions that take both Unicode strings and bytes (in a single Python version) should be rare, and should generally be convenience functions in your interface; not code deep in the internals.

However, you can use a concept of **native strings**: a type that corresponds to the `str` type in Python: PyBytes on Python 2, and PyUnicode in Python 3. This is the type that you will need to return from functions like __str__ and ‘‘__repr__.

Using the **native string** extensively is suitable for conservative projects: it affects the semantics under Python 2 as little as possible, while not requiring the resulting Python 3 API to feel contorted.

With py3c, functions for the native string type are PyStr_* (PyStr_FromString, PyStr_Type, PyStr_Check, etc.). They correspond to PyString on Python 2, and PyUnicode on Python 3. The supported API is the intersection of PyString_* and PyUnicode_*, except PyStr_Size (see below) and the deprecated PyUnicode_Encode; additionally PyStr_AsUTF8String is defined.

Keep in mind py3c expects that native strings are always encoded with `utf-8` under Python 2. If you use a different encoding, you will need to convert between bytes and text manually.

For binary data, use PyBytes_* (PyBytes_FromString, PyBytes_Type, PyBytes_Check, etc.). Python 3.x provides them under these names only; in Python 2.6+ they are aliases of PyString_*. (For even older Pythons, py3c also provides these aliases.) The supported API is the intersection of PyString_* and PyBytes_*,

Porting mostly consists of replacing "`PyString_`" to either "`PyStr_`" or "`PyBytes_`"; just see the caveat about size below.

To summarize the four different string type names:

String kind	py2	py3	Use
PyStr_*	PyString_*	PyUnicode_*	Human-readable text
PyBytes_*	PyString_*		Binary data
PyUnicode_*			Unicode strings
PyString_*		error	In unported code

String size

When dealing with Unicode strings, the concept of "size" is tricky, since the number of characters doesn't necessarily correspond to the number of bytes in the string's UTF-8 representation.

To prevent subtle errors, this library does *not* provide a PyStr_Size function.

Instead, use PyStr_AsUTF8AndSize. This functions like Python 3's PyUnicode_AsUTF8AndSize, except under Python 2, the string is not encoded (as it should already be in UTF-8), the size pointer must not be NULL, and the size may be stored even if an error occurs.

Ints

While string type is split in Python 3, the int is just the opposite: `int` and `long` were unified. `PyInt_*` is gone and only `PyLong_*` remains (and, to confuse things further, PyLong is named "int" in Python code). The py3c headers alias PyInt to PyLong, so if you're using them, there's no need to change at this point.

Floats

In Python 3, the function *`PyFloat_FromString`* lost its second, ignored argument.

The py3c headers redefine the function to take one argument even in Python 2. You will need to remove the excess argument from all calls.

Argument Parsing

The format codes for argument-parsing functions of the PyArg_Parse family have changed somewhat.

In Python 3, the `s`, `z`, `es`, `es#` and `U` (plus the new `C`) codes accept only Unicode strings, while `c` and `S` only accept bytes.

Formats accepting Unicode strings usually encode to char* using UTF-8. Specifically, these are `s`, `s*`, `s#`, `z`, `z*`, `z#`, and also `es`, `et`, `es#`, and `et#` when the encoding argument is set to NULL. In Python 2, the default encoding was used instead.

There is no variant of `z` for bytes, which means htere's no built-in way to accept "bytes or NULL" as a `char*`. If you need this, write an `O&` converter.

Python 2 lacks an `y` code, which, in Python 3, works on byte objects. The use cases needing `bytes` in Python 3 and `str` in Python 2 should be rare; if needed, use `#ifdef IS_PY3` to select a compatible PyArg_Parse call.

Compare the Python 2 and Python 3 docs for full details.

Defining Extension Types

If your module defines extension types, i.e. variables of type `PyTypeObject` (and related structures like `PyNumberMethods` and `PyBufferProcs`), you might need to make changes to these definitions. Please read the Extension types guide for details.

A common incompatibility comes from type flags, like `Py_TPFLAGS_HAVE_WEAKREFS` and `Py_TPFLAGS_HAVE_ITER`, which are removed in Python 3 (where the functionality is always present). If you are only using these flags in type definitions, (and *not* for example in `PyType_HasFeature()`), you can include `<py3c/tpflags.h>` to define them to zero under Python 3. For more information, read the *Type flags* section.

Module initialization

The module creation process was overhauled in Python 3. py3c provides a compatibility wrapper so most of the Python 3 syntax can be used.

PyModuleDef and PyModule_Create

Module object creation with py3c is the same as in Python 3.

First, create a PyModuleDef structure:

```
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,  /* m_base */
    "spam",                 /* m_name */
    NULL,                   /* m_doc */
    -1,                     /* m_size */
    spam_methods            /* m_methods */
};
```

Then, where a Python 2 module would have

```
m = Py_InitModule3("spam", spam_methods, "Python wrapper ...");
```

use instead

```
m = PyModule_Create(&moduledef);
```

For `m_size`, use -1. (If you are sure the module supports multiple subinterpreters, you can use 0, but this is tricky to achieve portably.) Additional members of the PyModuleDef structure are not accepted under Python 2.

See Python documentation for details on PyModuleDef and PyModule_Create.

Module creation entrypoint

Instead of the `void init<name>` function in Python 2, or a Python3-style `PyObject *PyInit_<name>` function, use the MODULE_INIT_FUNC macro to define an initialization function, and return the created module from it:

```
MODULE_INIT_FUNC(name)
{
    ...
    m = PyModule_Create(&moduledef);
    ...
    if (error) {
```

```
        return NULL;
    }
    ...
    return m;
}
```

The File API

The `PyFile` API was severely reduced `in Python 3`. The new version is specifically intended for internal error reporting in Python.

Native Python file objects are officially no longer backed by `FILE*`.

Use the Python API from the `io` module instead of handling files in C. The Python API supports all kinds of file-like objects, not just built-in files – though, admittedly, it's cumbersome to use from plain C.

If you really need to access an API that deals with `FILE*` only (e.g. for debugging), see py3c's limited file API shim.

Other changes

If you find a case where py3c doesn't help, use `#if IS_PY3` to include code for only one or the other Python version. And if your think others might have the same problem, consider contributing a macro and docs to py3c!

Building

When building your extension, note that Python 3.2 introduced ABI version tags (PEP 3149), which can be added to shared library filenames to ensure that the library is loaded with the correct Python version. For example, instead of `foo.so`, the shared library for the extension module `foo` might be named `foo.cpython-33m.so`.

Your buildsystem might generate these for you already, but if you need to modify it, you can get the tags from `systonfig`:

```
>>> import sysconfig
>>> sysconfig.get_config_var('EXT_SUFFIX')
'.cpython-34m.so'
>>> sysconfig.get_config_var('SOABI')
'cpython-34m'
```

This is completely optional; the old filenames without ABI tags are still valid.

Done!

Do your tests now pass under both Python 2 and 3? (And do you have enough tests?) Then you're done porting!

Once you decide to drop compatibility with Python 2, you can move to the Cleanup section.

1.1.3 Cleanup – Dropping Support for Python 2

When users of your C extension are not using Python 2 any more, or you need to use one of Python 3's irresistible new features, you can convert the project to use Python 3 only. As mentioned earlier, it is usually not a good idea to do this until you have full support for both Pythons.

With py3c, dropping Python 2 basically amounts to expanding all its compat macros. In other words, remove the `py3c.h` header, and fix the compile errors.

- Convert `PyStr_*` to `PyUnicode_*`; `PyInt_*` to `PyLong_*`.
- Instead of `MODULE_INIT_FUNC(<name>)`, write:

```
PyMODINIT_FUNC PyInit_<name>(void);
PyMODINIT_FUNC PyInit_<name>(void)
```

- Remove `Py_TPFLAGS_HAVE_WEAKREFS` and `Py_TPFLAGS_HAVE_ITER` (py3c defines them as 0).
- Replace PY3C_RICHCMP by its expansion, unless you keep the `py3c/comparison.h` header.
- Replace Py_RETURN_NOTIMPLEMENTED by its expansion, unless you either support Python 3.3+ only or keep the `py3c/comparison.h` header.
- Drop `capsulethunk.h`, if you're using it.
- Remove any code in `#if !IS_PY3` blocks, and the ifs around `#if IS_PY3` ones.

You will want to check the code as you're doing this. For example, replacing PyLong can easily result in code like `if (PyInt_Check(o) || PyInt_Check(o))`.

Enjoy your Python 3-compatible extension!

Overview

Porting a C extension to Python 3 involves three phases:

1. Modernization, where the code is migrated to the latest Python 2 features, and tests are added to prevent bugs from creeping in later. After this phase, the project will support Python 2.6+.
2. Porting, where support for Python 3 is introduced, but Python 2 compatibility is kept. After this phase, the project will support Python 2.6+ and 3.3+.
3. Cleanup, where support for Python 2 is removed, and you can start using Python 3-only features. After this phase, the project will support Python 3.3+.

The first two phases can be done simultaneously; I separate them here because the porting might require involved discussions/decisions about longer-term strategy, while modernization can be done immediately (as soon as support for Python 2.5 is dropped). But do not let the last two stages overlap, unless the port is trivial enough to be done in a single patch. This way you will have working code at all time.

Generally, *libraries*, on which other projects depend, will support both Python 2 and 3 for a longer time, to allow dependent code to make the switch. For libraries, the start of phase 3 might be delayed for many years. On the other hand, *applications* can often switch at once, dropping Python 2 support as soon as the porting is done.

Ready? The Modernization section is waiting!

1.2 The py3c Cheatsheet

1.2.1 Strings

- PyStr_* – for human-readable strings
- PyBytes_* – for binary data
- PyUnicode_* – when you used `unicode` in Python 2
- PyString_* – when you don't care about Python 3 yet

Use PyStr_AsUTF8AndSize to get a char* and its length.

1.2.2 Ints

Use whatever you used in Python 2. For py3-only code, use `PyLong`.

1.2.3 Floats

Don't pass the useless second argument to *`PyFloat_FromString()`* as you needed to do in Python 2.

1.2.4 Comparisons

Use rich comparisons:

```
static PyObject* mytype_richcmp(PyObject *obj1, PyObject *obj2, int op)
{
    if (mytype_Check(obj2)) {
        return PY3C_RICHCMP(get_data(obj1), get_data(obj2), op);
    }
    Py_RETURN_NOTIMPLEMENTED;
}
```

```
.tp_richcompare = mytype_richcmp
```

1.2.5 Objects & Types

Instead of	use
obj->ob_type	Py_TYPE(obj)
obj->ob_refcnt	Py_REFCNT(obj)
obj->ob_size	Py_SIZE(obj)
PyVarObject_HEAD_INIT(NULL, 0)	PyObject_HEAD_INIT(NULL), 0

1.2.6 Module initialization

```
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    .m_doc = PyDoc_STR("Python wrapper for the spam submodule."),
    .m_size = -1,
    .m_methods = spam_methods,
};

MODULE_INIT_FUNC(name)
{
    ...
    m = PyModule_Create(&moduledef);
    ...
    if (error) {
        return NULL;
    }
    ...
    return m;
}
```

1.2.7 CObject

Use the PyCapsule API. If you need to support 2.6, see the chapter PyCapsule API for Python 2.6.

1.3 Definitions in py3c

This table summarizes the various macros py3c defines, or mentions in the Porting Guide.

Macro	py2	py3
IS_PY3	→ 0	→ 1
PyStr_*	→ PyString_*	→ PyUnicode_*
PyBytes_*	→ PyString_*	
PyUnicode_*		
PyString_*		*error*
PyStr_AsUTF8AndSize	see below	
PyInt_*		→ PyLong_*
PyLong_*		
PyFloat_FromString	see below	
PyModuleDef	see below	
PyModuleDef_HEAD_INIT	→ 0	
PyModule_Create	see below	
MODULE_INIT_FUNC	see below	see below
Rich comparisons		
PY3C_RICHCMP	see below	see below
Py_RETURN_NOTIMPLEMENTED	=	=
Py_TYPE		
Py_REFCNT		
Py_SIZE		
PyVarObject_HEAD_INIT		
PyCapsule_*	see below	
Py_TPFLAGS_*		see below

Legend:

– provided by Python
→ – defined in py3c as a simple alias for
= – provided by at least Python 3.4; py3c backports it to Python versions that don't define it

The following non-trivial macros are defined:

PyStr_AsUTF8AndSize() Python 2: defined in terms of PyString_Size and PyString_AsString. Differences from Python 3:

- no encoding (string is assumed to be UTF-8-encoded)
- size pointer must not be NULL
- size may be stored even if an error occurs

`PyFloat_FromSting()`

Python 2: Only takes one argument, as in Python 3.

PyModuleDef

Python 2: contains `m_name`, `m_doc`, `m_size`, `m_methods` fields from Python 3, and `m_base` to accomodate PyModuleDef_HEAD_INIT.

PyModule_Create()

Python 2: calls Py_InitModule3; semantics same as in Python 3

MODULE_INIT_FUNC(<mod>)

Python 3: declares PyInit_<mod> and provides function header for it

Python 2: declares & defines init<mod>; declares a static PyInit_<mod> and provides function header for it

PY3C_RICHCMP()

See *docs*. (Purely a convenience macro, same in both versions.)

*PyCapsule_**

Capsules are included in Python 2.7 and 3.1+.

For 2.6, see the chapter PyCapsule API for Python 2.6.

*Py_TPFLAGS_** Type flags that were removed in Python 3 are defined to `0` in `<py3c/tpflags.h>`. Read the documentation before including the file.

1.4 py3c reference

1.4.1 Compatibility Layer

```
#include <py3c/compat.h>  // (included in <py3c.h>)
```

IS_PY3
Defined as `1` when building for Python 3; `0` otherwise.

PyStr

These functions are the intersection of PyString in Python 2, and PyUnicode in Python 3, with a few helpers thrown it.

All follow the Python 3 API, except `PyStr` is substituted for `PyUnicode`.

PyStr_Type
A `PyTypeObject` instance representing a human-readable string. Exposed in Python as `str`.

Python 2: `PyString_Type`
Python 3: `(provided)`

int **PyStr_Check**(PyObject *o)
Check that *o* is an instance of *PyStr* or a subtype.

Python 2: `PyString_Check`
Python 3: `PyUnicode_Check`

int **PyStr_CheckExact**(PyObject *o)
Check that *o* is an instance of *PyStr*, but not a subtype.

Python 2: `PyString_CheckExact`
Python 3: `PyUnicode_CheckExact`

PyObject* **PyStr_FromString**(const char *u)
Create a *PyStr* from a UTF-8 encoded null-terminated character buffer.

Python 2: `PyString_FromString`
Python 3: `PyUnicode_FromString`

PyObject* **PyStr_FromStringAndSize**(const char *u, Py_ssize_t *len*)
Create a *PyStr* from a UTF-8 encoded character buffer, and corresponding size in bytes.

Note that human-readable strings should not contain null bytes; but if the size is known, this is more efficient than *`PyStr_FromString()`*.

Python 2: `PyString_FromStringAndSize`
Python 3: `PyUnicode_FromStringAndSize`

PyObject* **PyStr_FromFormat**(const char *format, ...)
Create a *PyStr* from a C printf-style format string and arguments.

Note that formatting directives that were added in Python 3 (`%li`, `%lli`, `zi`, `%A`, `%U`, `%V`, `%S`, `%R`) will not work in Python 2.

Python 2: `PyString_FromFormat`
Python 3: `PyUnicode_FromFormat`

PyObject* **PyStr_FromFormatV**(const char *format, va_list *vargs*)
As *`PyStr_FromFormat()`*, but takes a `va_list`.

Python 2: `PyString_FromFormatV`
Python 3: `PyUnicode_FromFormatV`

const char* **PyStr_AsString**(PyObject *s)
Return a null-terminated representation of the contents of *s*. The buffer is owned by *s* and must bot be modified, deallocated, or used after *s* is deallocated.

Uses the UTF-8 encoding on Python 3.

If given an Unicode string on Python 2, uses Python's default encoding.

Python 2: `PyString_AsString`
Python 3: `PyUnicode_AsUTF8` (!)

PyObject* **PyStr_Concat** (PyObject *left, PyObject *right)
Concatenates two strings giving a new string.

Python 2: implemented in terms of `PyString_Concat`
Python 3: `PyUnicode_Concat`

PyObject* **PyStr_Format** (PyObject *format, PyObject *args)
Format a string; analogous to the Python expression `format % args`. The *args* must be a tuple or dict.

Python 2: `PyString_Format`
Python 3: `PyUnicode_Format`

void **PyStr_InternInPlace** (PyObject **string)
Intern *string*, in place.

Python 2: `PyString_InternInPlace`
Python 3: `PyUnicode_InternInPlace`

PyObject* **PyStr_InternFromString** (const char *v)
Create an interned string from a buffer. Combines *PyStr_FromString()* and *PyStr_InternInPlace()*.

In Python 3, *v* must be UTF-8 encoded.

Python 2: `PyString_InternFromString`
Python 3: `PyUnicode_InternFromString`

PyObject* **PyStr_Decode** (const char *s, Py_ssize_t *size*, const char *encoding, const char *errors)
Create a new string by decoding *size* bytes from *s*, using the specified *encoding*.

Python 2: `PyString_Decode`
Python 3: `PyUnicode_Decode`

char* **PyStr_AsUTF8** (PyObject *str)
Encode a string using UTF-8 and return the result as a char*. Under Python 3, the result is UTF-8 encoded.

Python 2: `PyString_AsString`
Python 3: `PyUnicode_AsUTF8`

PyObject* **PyStr_AsUTF8String** (PyObject *str)
Encode a string using UTF-8 and return the result as PyBytes.

In Python 2, (where PyStr is bytes in UTF-8 encoding already), this is a no-op.

Python 2: identity
Python 3: `PyUnicode_AsUTF8String`

char ***PyStr_AsUTF8AndSize**(PyObject **str*, Py_ssize_t **size*)
Return the UTF-8-encoded representation of the string, and set *size* to the number of bytes in this representation. The *size* may not be NULL.

In Python 2, the string is assumed to be UTF8-encoded.

On error, *size* may or may not be set.

Python 2: (*size = `PyString_Size(str)`, `PyString_AsString(str)`)
Python 3: `PyUnicode_AsUTF8AndSize`

PyBytes

These functions are the intersection of PyString in Python 2, and PyBytes in Python 3.

All follow the Python 3 API.

PyBytes_Type
A `PyTypeObject` instance representing a string of binary data. Exposed in Python 2 as `str`, and in Python 3 as `bytes`.

Python 2: `PyString_Type`
Python 3: `(provided)`

int **PyBytes_Check**(PyObject **o*)
Check that *o* is an instance of *PyBytes* or a subtype.

Python 2: `PyString_Check`
Python 3: `(provided)`

int **PyBytes_CheckExact**(PyObject **o*)
Check that *o* is an instance of *PyBytes*, but not a subtype.

Python 2: `PyString_CheckExact`
Python 3: `(provided)`

PyObject* **PyBytes_FromString**(const char **v*)
Create a *PyBytes* from a NULL-terminated C buffer.

Note that binary data might contain null bytes; consider using `PyBytes_FromStringAndSize()` instead.

Python 2: `PyString_FromString`
Python 3: `(provided)`

PyObject* **PPyBytes_FromStringAndSize**(const char *v, Py_ssize_t *len*)
Create a *PyBytes* from a C buffer and size.

Python 2: `PyString_FromStringAndSize`
Python 3: `(provided)`

PyObject* **PyBytes_FromFormat**(const char *format, ...)
Create a *PyBytes* from a C printf-style format string and arguments.

Python 2: `PyString_FromFormat`
Python 3: `(provided)`

PyObject* **PyBytes_FromFormatV**(const char *format, va_list *args*)
As *`PyBytes_FromFormat()`*, but takes a `va_list`.

Python 2: `PyString_FromFormatV`
Python 3: `(provided)`

Py_ssize_t **PyBytes_Size**(PyObject *o)
Return the number of bytes in a *PyBytes* object.

Python 2: `PyString_Size`
Python 3: `(provided)`

Py_ssize_t **PyBytes_GET_SIZE**(PyObject *o)
As *`PyBytes_Size()`* but without error checking.

Python 2: `PyString_GET_SIZE`
Python 3: `(provided)`

char ***PyBytes_AsString**(PyObject *o)
Return the buffer in a *PyBytes* object. The data must not be modifiet or deallocated, or used after a reference to *o* is no longer held.

Python 2: `PyString_AsString`

Python 3: `(provided)`

char ***PyBytes_AS_STRING**(PyObject *o)*
As *`PyBytes_AsString()`* but without error checking.

Python 2: `PyString_AS_STRING`
Python 3: `(provided)`

int **PyBytes_AsStringAndSize**(PyObject *obj, char **buffer, Py_ssize_t *length)
Get the buffer and size stored in a *PyBytes* object.

Python 2: `PyString_AsStringAndSize`
Python 3: `(provided)`

void **PyBytes_Concat**(PyObject **bytes, PyObject *newpart)
Concatenate *newpart* to *bytes*, returning a new object in *bytes*, and discarding the old.

Python 2: `PyString_Concat`
Python 3: `(provided)`

void **PyBytes_ConcatAndDel**(PyObject **bytes, PyObject *newpart)
As *`PyBytes_AsString()`* but decreases reference count of *newpart*.

Python 2: `PyString_ConcatAndDel`
Python 3: `(provided)`

int **_PyBytes_Resize**(PyObject **string, Py_ssize_t *newsize*)
Used for efficiently build bytes objects; see the Python docs.

Python 2: `_PyString_Resize`
Python 3: `(provided)`

PyInt

These functions allow extensions to make the diistinction between ints and longs on Python 2.

All follow the Python 2 API.

PyInt_Type
A `PyTypeObject` instance representing an integer that fits in a C long.

Python 2: `(provided)`
Python 3: `PyLong_Type`

int **PyInt_Check**(PyObject *o)
Check that *o* is an instance of *PyInt* or a subtype.

Python 2: `(provided)`
Python 3: `PyLong_Check`

int **PyInt_CheckExact**(PyObject *o)
Check that *o* is an instance of *PyInt*, but not a subtype.

Python 2: `(provided)`
Python 3: `PyLong_CheckExact`

PyObject* **PyInt_FromString**(char *str, char **pend, int base)
Convert a string to *PyInt*. See the Python docs.

Python 2: `(provided)`
Python 3: `PyLong_FromString`

PyObject* **PyInt_FromLong**(long i)
Convert a C long int to *PyInt*.

Python 2: `(provided)`
Python 3: `PyLong_FromLong`

PyObject* **PyInt_FromSsize_t**(Py_ssize_t i)
Convert a Py_ssize_t int to *PyInt*.

Python 2: `(provided)`
Python 3: `PyLong_FromSsize_t`

PyObject* **PyInt_FromSize_t**(Py_size_t i)
Convert a Py_size_t int to *PyInt*.

Python 2: `(provided)`
Python 3: `PyLong_FromSize_t`

long **PyInt_AsLong**(PyObject *o)
Convert a *PyInt* to a C long.

Python 2: `(provided)`
Python 3: `PyLong_AsLong`

long **PyInt_AS_LONG**(PyObject *o)
As *`PyInt_AsLong()`*, but with no error checking.

Python 2: `(provided)`
Python 3: `PyLong_AS_LONG`

unsigned long **PyInt_AsUnsignedLongLongMask**(PyObject *o)
Convert a Python object to int, and return its value as an unsigned long.

Python 2: `(provided)`
Python 3: `PyLong_AsUnsignedLongLongMask`

Py_ssize_t **PyInt_AsSsize_t**(PyObject *o)
Convert a Python object to int, and return its value as a Py_ssize_t.

Python 2: `(provided)`
Python 3: `PyLong_AsSsize_t`

PyFloat

PyObject* **PyFloat_FromString**(PyObject *str)
Create a `PyFloatObject` object. The signature follows the Python 3 API, even on Python 2.

Python 2: `PyFloat_FromString(str, NULL)`
Python 3: `PyFloat_FromString(str)`

Module Initialization

MODULE_INIT_FUNC(<name>)
Use this macro as the header for the module initialization function.

Python 2:

```
static PyObject *PyInit_<name>(void);
void init<name>(void);
void init<name>(void) { PyInit_<name>(); }
static PyObject *PyInit_<name>(void)
```

Python 3:

```
PyMODINIT_FUNC PyInit_<name>(void);
PyMODINIT_FUNC PyInit_<name>(void)
```

PyModuleDef

Python 2:

int **m_base**
: Always set this to *PyModuleDef_HEAD_INIT*

char ***m_name**

char ***m_doc**

Py_ssize_t **m_size**
: Set this to -1. (Or if your module supports `subinterpreters`, use 0)

PyMethodDef **m_methods**

Python 3: `(provided)`

PyModuleDef_HEAD_INIT

Python 2: `0`

Python 3: `(provided)`

PyObject* **PyModule_Create**(*PyModuleDef def*)

Python 2: `Py_InitModule3`(def->m_name, def->m_methods, def->m_doc)

Python 3: `(provided)`

1.4.2 Comparison Helpers

```
#include <py3c/comparison.h>  // (included in <py3c.h>)
```

Py_RETURN_NOTIMPLEMENTED

Backported from `Python 3.4` for older versions.

PyObject* **PY3C_RICHCMP**(val1, val2, int *op*)

Compares two arguments orderable by C comparison operators (such as C ints or floats). The third argument specifies the requested operation, as for a `rich comparison function`. Evaluates to a new reference to `Py_True` or `Py_False`, depending on the result of the comparison.

```
((op) == Py_EQ) ? PyBool_FromLong((val1) == (val2)) : \
((op) == Py_NE) ? PyBool_FromLong((val1) != (val2)) : \
((op) == Py_LT) ? PyBool_FromLong((val1) < (val2)) : \
((op) == Py_GT) ? PyBool_FromLong((val1) > (val2)) : \
((op) == Py_LE) ? PyBool_FromLong((val1) <= (val2)) : \
((op) == Py_GE) ? PyBool_FromLong((val1) >= (val2)) : \
(Py_INCREF(Py_NotImplemented), Py_NotImplemented)
```

1.4.3 Types

```
#include <py3c/tpflags.h>  /* (*NOT* included in <py3c.h>) */
```

Removed type flags are defined as `0` in Python 3, which is *only* useful in type definitions.

In particular, these macros are *not* suitable for `PyType_HasFeature()` in Python 3.

Py_TPFLAGS_HAVE_GETCHARBUFFER

Py_TPFLAGS_HAVE_SEQUENCE_IN

Py_TPFLAGS_HAVE_INPLACEOPS

Py_TPFLAGS_CHECKTYPES

Py_TPFLAGS_HAVE_RICHCOMPARE

Py_TPFLAGS_HAVE_WEAKREFS

Py_TPFLAGS_HAVE_ITER

Py_TPFLAGS_HAVE_CLASS

Py_TPFLAGS_HAVE_INDEX

Py_TPFLAGS_HAVE_NEWBUFFER

Python 2: (provided), e.g. `Py_TPFLAGS_HAVE_WEAKREFS`
Python 3: `0`

1.4.4 Capsules

```
#include <py3c/capsulethunk.h>  // (*NOT* included in <py3c.h>)
```

This file provides a PyCapsule API compatibility layer for Python 2.6.

Capsules are simulated in terms of PyCObject. The PyCapsule API for Python 2.6 chapter lists the limitations of this solution.

PyCapsule_Type

Python 2.6: `PyCObject_Type`
2.7 and 3.x: `(provided)`

PyCapsule_CheckExact(PyObject *p)

Python 2.6: `PyCObject_Check`
2.7 and 3.x: `(provided)`

PyCapsule_IsValid(PyObject *capsule, const char *name)

Python 2.6: `PyCObject_Check(capsule)`
2.7 and 3.x: `(provided)`

PyCapsule_New(void *pointer, const char *name, PyCapsule_Destructor destructor)

Python 2.6: `PyCObject_FromVoidPtr(pointer, destructor)`
2.7 and 3.x: `(provided)`

PyCapsule_GetPointer(PyObject *capsule, const char *name)

Python 2.6: `PyCObject_AsVoidPtr(capsule)` – name is not checked!
2.7 and 3.x: `(provided)`

PyCapsule_SetPointer(PyObject **capsule*, void **pointer*)

Python 2.6: uses CPython internals; effect similar to `PyCObject_SetVoidPtr()`
2.7 and 3.x: `(provided)`

PyCapsule_GetDestructor(PyObject **capsule*)

Python 2.6: uses CPython internals to get the a CObject's destructor
2.7 and 3.x: `(provided)`

PyCapsule_SetDestructor(PyObject **capsule*, PyCapsule_Destructor *destructor*)

Python 2.6: uses CPython internals to replace a CObject's destructor
2.7 and 3.x: `(provided)`

PyCapsule_GetName(PyObject **capsule*)

Python 2.6: NULL
2.7 and 3.x: `(provided)`

PyCapsule_SetName(PyObject **capsule*)

Python 2.6: Always raises `NotImplementedError`
2.7 and 3.x: `(provided)`

PyCapsule_GetContext(PyObject **capsule*)

Python 2.6: uses CPython internals to get the CObject "desc" field
2.7 and 3.x: `(provided)`

PyCapsule_SetContext(PyObject **capsule*, PyCapsule_Destructor *destructor*)

Python 2.6: uses CPython internals to replace CObject "desc" field
2.7 and 3.x: `(provided)`

PyCapsule_Import(const char **name*, int *no_block*)

Python 2.6: backported
2.7 and 3.x: `(provided)`

1.4.5 Files

```
#include <py3c/fileshim.h>  // (*NOT* included in <py3c.h>)
```

py3c_PyFile_AsFileWithMode(PyObject **py_file*, const char **mode*)
Quick-and-dirty substitute for the removed `PyFile_AsFile()`. Read the file shim chapter before using.

1.5 Special Porting Guides

Some porting notes are mentioned outside the main guide, because they either, affect fewer project, or need more extensive discussion. They are linked from the main porting guide.

Here is a list:

- **Extension Types:** Porting definitions of extension types (`PyTypeObject` and related objects like `PyNumberMethods`)
- **PyCapsule API for Python 2.6:** Porting PyCObject while retaining support for Python 2.6
- **PyFile shim:** Quick-and-dirty helpers for porting away from the removed PyFile API

1.5.1 Porting extension types

The extension type structure, PyTypeObject, has seen some changes in Python 3. You might wish to refresh your memory with the Python documentation on this (Python 2, Python 3); here we concentrate only on the differences.

Type Flags

The most common incompatibility in type definition involves feature flags like `Py_TPFLAGS_HAVE_WEAKREFS` and `Py_TPFLAGS_HAVE_ITER` (see *Type flags reference* for a full list).

These flags indicate capabilities that are always present in Python 3, so the macros are only available in Python 2. Most projects can simply define these to 0 in Python 3.

However, another use of the macros is feature checking, as in `PyType_HasFeature(cls, Py_TPFLAGS_HAVE_ITER)`. Defining the flags to `0` would cause that test to fail under Python 3, where it should instead always succeed! So, in these cases, the check should be done as `(IS_PY3 || PyType_HasFeature(cls, Py_TPFLAGS_HAVE_ITER))`.

If your project does not use `PyType_HasFeature`, or bypasses the check under Python 3 as above, you can include `<py3c/tpflags.h>` to define missing type flags as 0.

PyTypeObject

The differences in PyTypeObject itself are fairly minor. The `tp_compare` field became `void *tp_reserved`, and is ignored. If you use `tp_richcompare`, this field is ignored in Python 2. It is best set to NULL.

The change can case trouble if you use explicity types during definition for type safety, as in:

```
...
(destructor)Example_dealloc,            /* tp_dealloc */
(printfunc)0,                           /* tp_print */
(getattrfunc)0,                         /* tp_getattr */
(setattrfunc)0,                         /* tp_setattr */
(cmpfunc)0,                             /* tp_compare */
...
```

In this case, make an exception for tp_compare, and use just `NULL`.

Python 3 also adds new fields at the end of PyTypeObject – but that should not affect initialization.

PyNumberMethods

The `PyNumberMethods` structure, used to implement number-like behavior and operators, was changed. (Docs: `py2`, `py3`)

Specifically, several members were removed:

- `nb_divide` (Python3 calls `nb_floor_divide` or `nb_true_divide`)
- `nb_coerce`
- `nb_oct`
- `nb_hex`
- `nb_inplace_divide` (see nb_divide)

one was renamed:

- `nb_nonzero` became `nb_bool`

and one was blanked:

- `unaryfunc nb_long` became `void *nb_reserved` and must be NULL.

The mix of removal strategies on the CPython side makes the port somewhat annoying.

As of yet, the py3c library does not provide helpers for porting PyNumberMethods. More investigation is needed to be sure all projects' needs are addressed.

What you need to do depends on your initialization style:

CPython style

This style, used in CPython, works in both old C and C++:

```
static PyNumberMethods long_as_number = {
    (binaryfunc)long_add,       /*nb_add*/
    (binaryfunc)long_sub,       /*nb_subtract*/
    (binaryfunc)long_mul,       /*nb_multiply*/
    (binaryfunc)long_div,       /*nb_divide*/
    long_mod,                   /*nb_remainder*/
    long_divmod,                /*nb_divmod*/
    long_pow,                   /*nb_power*/
    (unaryfunc)long_neg,        /*nb_negative*/
    ...
```

When using this, wrap the removed elements in `#if !IS IS_PY3`.

If you use nb_long in Python 2, conditionally set it to NULL in Python 3. Make sure `nb_int` is set.

C99 style

If you don't support both C89 and C++ (!) compilers, you may use the named member initialization feature of C99:

```
static PyNumberMethods long_as_number = {
    .tp_add = long_add,
    .tp_div = long_div,
    ...
```

If this is the case, lump the non-NULL Python2-only members and `nb_long` together in a single `#if !IS IS_PY3` block. You will need another `#if`/`#else` block to handle to handle both names of `nb_nonzero`, if using that.

PyBufferProcs

The buffer protocol changed significantly in Python 3. Kindly read the documentation, and implement the new buffer protocol for Python 3.

If you find an easier way to port buffer-aware objects, which other projects could benefit from, remember that py3c welcomes contributions.

1.5.2 PyCapsule API for Python 2.6

The `capsulethunk.h` header implements the PyCapsule API (with some limitations) in terms of PYCObject. It is only necessary for compatibility with Python 2.6 (or 3.0).

Note: The `capsulethunk.h` header and this documentation was written by Larry Hastings for the Python documentation. [1] It is now maintained as part of the py3c project. [2]

CObject replaced with Capsule

The `Capsule` object was introduced in Python 3.1 and 2.7 to replace `CObject`. CObjects were useful, but the `CObject` API was problematic: it didn't permit distinguishing between valid CObjects, which allowed mismatched CObjects to crash the interpreter, and some of its APIs relied on undefined behavior in C. (For further reading on the rationale behind Capsules, please see CPython issue 5630.)

If you're currently using CObjects, and you want to migrate to Python 3, you'll need to switch to Capsules. See the *PyCObject section* in the porting guide for instructions.

`CObject` was deprecated in 3.1 and 2.7 and completely removed in Python 3.2. So, if you need to support versions of Python earlier than 2.7, or Python 3.0, you'll have to support both CObjects and Capsules.

The following example header file `capsulethunk.h` may solve the problem for you. Simply write your code against the `Capsule` API and include this header file after `Python.h`. Your code will automatically use Capsules in versions of Python with Capsules, and switch to CObjects when Capsules are unavailable.

If you're using py3c, you will need to explicitly `#include <py3c/capsulethunk.h>`. The file is not included from `py3c.h`.

Since `CObject` provides no place to store the capsule's "name", the simulated `Capsule` objects created by `capsulethunk.h` behave slightly differently from real Capsules. Specifically:

- The name parameter passed in to *`PyCapsule_New()`* is ignored.
- The name parameter passed in to *`PyCapsule_IsValid()`* and *`PyCapsule_GetPointer()`* is ignored, and no error checking of the name is performed.
- *`PyCapsule_GetName()`* always returns NULL.
- *`PyCapsule_SetName()`* always raises an exception and returns failure. (Since there's no way to store a name in a CObject, noisy failure of *`PyCapsule_SetName()`* was deemed preferable to silent failure here.) If this is inconvenient, feel free to modify your local copy as you see fit.

You can find `capsulethunk.h` at include/py3c/capsulethunk.h. We also include it here for your convenience:

[1] CPython issue 13053: Add Capsule migration documentation to "cporting"
[2] CPython issue 24937: Multiple problems in getters & setters in capsulethunk.h

```
/* Copyright (c) 2011, Larry Hastings
 * Copyright (c) 2015, py3c contributors
 * Licensed under the MIT license; see py3c.h
 *
 * (Note: Relicensed from PSF: http://bugs.python.org/issue24937#msg250191 )
 */

#ifndef __CAPSULETHUNK_H
#define __CAPSULETHUNK_H

#if (    (PY_VERSION_HEX <  0x02070000) \
     || ((PY_VERSION_HEX >= 0x03000000) \
      && (PY_VERSION_HEX <  0x03010000)) )

#define __PyCapsule_GetField(capsule, field, error_value) \
    ( PyCapsule_CheckExact(capsule) \
        ? (((PyCObject *)capsule)->field) \
        : (PyErr_SetString(PyExc_TypeError, "CObject required"), (error_value)) \
    ) \

#define __PyCapsule_SetField(capsule, field, value) \
    ( PyCapsule_CheckExact(capsule) \
        ? (((PyCObject *)capsule)->field = value), 0 \
        : (PyErr_SetString(PyExc_TypeError, "CObject required"), 1) \
    ) \


#define PyCapsule_Type PyCObject_Type

#define PyCapsule_CheckExact(capsule) (PyCObject_Check(capsule))
#define PyCapsule_IsValid(capsule, name) (PyCObject_Check(capsule))


#define PyCapsule_New(pointer, name, destructor) \
    (PyCObject_FromVoidPtr(pointer, (void (*)(void*)) (destructor)))


#define PyCapsule_GetPointer(capsule, name) \
    (PyCObject_AsVoidPtr(capsule))

/* Don't call PyCObject_SetPointer here, it fails if there's a destructor */
#define PyCapsule_SetPointer(capsule, pointer) \
    __PyCapsule_SetField(capsule, cobject, pointer)


#define PyCapsule_GetDestructor(capsule) \
    __PyCapsule_GetField(capsule, destructor, (void (*)(void*)) NULL)

#define PyCapsule_SetDestructor(capsule, dtor) \
    __PyCapsule_SetField(capsule, destructor, (void (*)(void*)) dtor)


/*
 * Sorry, there's simply no place
 * to store a Capsule "name" in a CObject.
 */
#define PyCapsule_GetName(capsule) NULL
```

```
static int
PyCapsule_SetName(PyObject *capsule, const char *unused)
{
    unused = unused;
    PyErr_SetString(PyExc_NotImplementedError,
        "can't use PyCapsule_SetName with CObjects");
    return 1;
}



#define PyCapsule_GetContext(capsule) \
    __PyCapsule_GetField(capsule, desc, (void*) NULL)

#define PyCapsule_SetContext(capsule, context) \
    __PyCapsule_SetField(capsule, desc, context)


static void *
PyCapsule_Import(const char *name, int no_block)
{
    PyObject *object = NULL;
    void *return_value = NULL;
    char *trace;
    size_t name_length = (strlen(name) + 1) * sizeof(char);
    char *name_dup = (char *)PyMem_MALLOC(name_length);

    if (!name_dup) {
        return NULL;
    }

    memcpy(name_dup, name, name_length);

    trace = name_dup;
    while (trace) {
        char *dot = strchr(trace, '.');
        if (dot) {
            *dot++ = '\0';
        }

        if (object == NULL) {
            if (no_block) {
                object = PyImport_ImportModuleNoBlock(trace);
            } else {
                object = PyImport_ImportModule(trace);
                if (!object) {
                    PyErr_Format(PyExc_ImportError,
                        "PyCapsule_Import could not "
                        "import module \"%s\"", trace);
                }
            }
        } else {
            PyObject *object2 = PyObject_GetAttrString(object, trace);
            Py_DECREF(object);
            object = object2;
        }
        if (!object) {
            goto EXIT;
```

```
        }

        trace = dot;
    }

    if (PyCObject_Check(object)) {
        PyCObject *cobject = (PyCObject *)object;
        return_value = cobject->cobject;
    } else {
        PyErr_Format(PyExc_AttributeError,
            "PyCapsule_Import \"%s\" is not valid",
            name);
    }

EXIT:
    Py_XDECREF(object);
    if (name_dup) {
        PyMem_FREE(name_dup);
    }
    return return_value;
}

#endif /* #if PY_VERSION_HEX < 0x02070000 */

#endif /* __CAPSULETHUNK_H */
```

1.5.3 The PyFile API

In Python 3, the PyFile API was reduced to a few functions, and is now meant for internal interpreter use.

Python files (and file-like objects) should be manipulated with the API defined by the `io` module.

But, in the real world, some C libraries only provide debugging output to `FILE*`. For cases like this, py3c provides a quick-and-dirty replacement for `PyFile_AsFile()`:

FILE* **py3c_PyFile_AsFileWithMode**(PyObject **py_file*, const char **mode*)

Open a (file-backed) Python file object as `FILE*`.

Parameters

- **py_file** – The file object, which must have a working `fileno()` method
- **mode** – A mode appropriate for `fdopen`, such as `'r'` or `'w'`

This function presents several caveats:

- Only works on file-like objects backed by an actual file
- All C-level writes should be done before additional Python-level writes are allowed (e.g. by running Python code).
- Though the function tries to flush, due to different layers of buffering there is no guarantee that reads and writes will be ordered correctly.

1.6 Contributing to py3c

If you would like to contribute to py3c, be it code, documentation, suggestions, or anything else, please file an issue or send a pull request at the project's Github page.

If you are not familiar with Github, or prefer not to use it, you can e-mail contributions to `encukou` at `gmail` dot `com`.

1.6.1 Testing

Automatic testing is set up at Travis CI:

To test the code locally, you can run (using GNU make):

```
$ make test
```

This will test py3c against `python2` and `python3`. To test under a different interpreter, run for example:

```
$ make test-python35
```

1.6.2 Packaging

To install system-wide (for example, if you are a distro packager), use `make install`. There are no configure/compile steps for this header-only library.

The install target honors GNU standard environment variables to specify installation directories.

1.6.3 Building the Docs

To build the docs, you need Sphinx. If it's not in your system's package manager, it can be installed with:

```
$ pip install --user sphinx
```

To build the HTML documentation, do:

```
$ make doc
```

For more docs options, run *make* in the *doc* directory.

1.7 py3c Changes

1.7.1 Version History

v0.6 (2016-05-19)

Packaging:

- Fix file permissions when doing `make instal`

v0.5 (2016-05-13)

Packaging:

- Fix the pkgconfig file

v0.4 (2016-05-13)

Fixes:

- Fix unterminated #if in tpflags.h (thanks to MURAOKA Yusuke)

Additions:

- Support for C++ (with initial help from MURAOKA Yusuke)
- Support PyFloat_FromString (thanks to Christoph Zwerschke)
- Test suite is much more comprehensive

v0.3 (2015-09-09)

Breaking changes:

- Type flags moved to "tpflags.h", which is not included by default. See the warning in the file, or in documentation.

Other changes:

- Integrated capsulethunk.h
- Added PyFile_AsFile shim
- Discuss porting type definitions in the guide

v0.2 (2015-08-25)

- First tagged public release

1.7.2 More Details

For all changes, see the commit history on Github.

Symbols

A

B

C

I

L

M

O

P

S

T

U