py3c Documentation

Release 0.2

Petr Viktorin

August 25, 2015

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.

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.

PyCapsule is available in Python 2.7 and 3.1+. 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. It is explained in C Porting HOWTO from the Python documentation.

If you use py3c, you can include this header as `<py3c/capsulethunk.h>`. It is not included from `<py3c.h>`, and it comes under a different license.

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*`, there is 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. Function that takes both Unicode strings and bytes should be rare, and should generally be convenience functions in your interface; not code deep in the internals.

With py3c, the human-readable strings 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.

For binary data, use PyBytes_* (PyBytes_FromString, PyBytes_Type, PyBytes_Check, etc.). These correspond to PyString on Python 2, and Python 3 provides them directly. 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.

You might meet two more string types. One is PyUnicode_*, which is provided by both Python versions directly, and should be used wherever you used PyUnicode in Python 2 code already. The other is PyString_*, the Python 2 type used to store both kinds of stringy data. This type is not in Python 3, and must be replaced.

To summarize:

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.

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.

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_name = "spam",
    .m_doc = PyDoc_STR("Python wrapper for the spam submodule."),
    .m_size = -1,
    .m_methods = spam_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;
}
```

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 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
```

1.2.4 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

1.2.5 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;
}
```

1.2.6 CObject

Use the PyCapsule API. If you need to support 2.6, you'll currently need `#if PY_3` blocks.

1.3 Definitions in py3c

This table summarizes the various macros py3c defines.

Macro	py2	py3
IS_PY3	→ 0	→ 1
PyStr_*	→ PyString_*	→ PyUnicode_*
PyBytes_*	→ PyString_*	
PyUnicode_*		
PyString_*		*error*
PyStr_AsUTF8AndSize	see below	
PyInt_*		→ PyLong_*
PyLong_*		
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		
Py_TPFLAGS_HAVE_WEAKREFS		→ 0
Py_TPFLAGS_HAVE_ITER		→ 0
PyCapsule_*	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

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 and 3.0, the the Python documentation provides
a compatibility header, which is distributed with py3c for
convenience.

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

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)`

Types

Removed type flags are defined as 0 in Python 3.

Py_TPFLAGS_HAVE_WEAKREFS

Python 2: `(provided)`
Python 3: `0`

Py_TPFLAGS_HAVE_ITER

Python 2: `(provided)`
Python 3: `0`

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 Capsules

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

This header is copied from the Python documentation. The following text is adapted from its official docs:

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.

`capsulethunk.h` simulates Capsules using CObjects. However, `CObject` provides no place to store the capsule's "name". As a result 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` in the Python source distribution as Doc/includes/capsulethunk.h. We also include it as `py3c/capsulethunk.h` for your convenience:

1.5 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`.

1.5.1 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
```

1.5.2 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.

Symbols

A

B

C

I

L

M

O

P

S

T

U