cppy Documentation

Release 1.1.0

Nucleic team

Jun 04, 2021

Contents

Cppy is a small C++ header library which makes it easier to write Python extension modules. The primary feature is a PyObject smart pointer which automatically handles reference counting and provides convenience methods for performing common object operations.

CHAPTER 1

Installation and use with setuptools

Since Cppy is nothing else than a collection of header that are only compiled when used, installing it is extremely straightforward using pip:

```
$ pip install cppy
```

If you want to run the development version, you can install directly from GitHub:

```
$ pip install https://github.com/nucleic/cppy
```

1.1 Using Cppy in an extensions

To use Cppy in your extension (written in C++), you simply need to include it.

```
#include <cppy/cppy.h>
```

Cppy includes Python.h so when including cppy.h you do not need to also include Python.h.

Every functions, classes exposed by Cppy are stored in the *cppy* namespace.

```
cppy::ptr obj_ptr( PyUnicode_FromString("test") )
```

1.2 Use with setuptools

Cppy is only needed during the installation step of the projects using it. The following example setup.py script illustrates how to use Cppy without requiring it to be installed before *setup.py* is run.

```
from setuptools import setup, Extension
from setuptools.command.build_ext import build_ext

```

(continues on next page)

(continued from previous page)

```
ext_modules = [
    Extension(
        'project',
        ['module.cpp],
        include_dirs=['.'],
        language='c++',
    ),
]

class BuildExt(build_ext):

    def build_extensions(self):

        # Delayed import of cppy to let setup_requires install it if
        # necessary
        import cppy

        ct = self.compiler.compiler_type
        for ext in self.extensions:
            # cppy.get_include() collect the path of the header files
            ext.include_dirs.insert(0, cppy.get_include())
        build_ext.build_extensions(self)

setup(
    name='project',
    python_requires='>=3.5',
    setup_requires=['cppy'],
    ext_modules=ext_modules,
    cmdclass={'build_ext': BuildExt},
)
```

CHAPTER 2

Cppy smart pointer

CPython relies on reference counting to manage object lifetime. A large pitfall when writing C-extension is to properly handle increfing and decrefing the reference count. Cppy aims at simplifying this process by providing a smart pointer class. Before diving into the details of it helps lets start a CPython reference counting crash-course.

2.1 CPython reference counting crash course

Each object allocated by Python has a reference count, indicating how many times this object is 'used'. When the reference count of an object goes to zero, it is de-allocated. Outside of C extension, one does not need to manage the reference count manually.

When a function part of Python C-API returns a Python object, it returns a pointer to it. At the time at which the function returns, the referenced object is live and its reference count is above zero. Depending of the function, you do not have the same responsibility with respect to that object reference count:

- Owned references: Most functions return a *new* reference which means that you are responsible for decrefing the object reference count when you are done with it (basically the function increfed the object reference count before returning). In this situation you own a reference.
- Borrowed reference: Some functions (*PyList_GetItem*, *PyTuple_GetItem*, *PyDict_GetItem*, ...) do not incref the object count before returning. In that case, you have only a borrowed reference, you are not responsible for decrefing the object reference count.

Borrowed references allow to avoid the cost of increfing/decrefing which is nice. However since you do not own the reference, if the object referenced is removed from its owner (list, tuple for the above two mentioned functions) it may just disappear and your reference becomes invalid. This can cause issues. If the object should outlive the container, or the time it will spend in the container you have to incref it manually. Lets now discuss the convention when calling a function.

When calling a function, the caller is expected to own a reference to each of the arguments passed to the callee. The callee does not own the references, it only borrows them. As a consequence, it should not decref the reference and if it needs to store the object, in for example a C structure, it should incref it. Note that this does not apply in general to Python container since those are manipulated using functions that take care of it. There are however some exceptions

that steals a reference, meaning that you are not the owner of the reference after the call. *PyList_SetItem*, for example, steal references.

An easy way to get reference count wrong is forgetting to decref some intermediate object before leaving a function. This is particularly true if the function has some early exit point because an exception should be raised. A good practice is to have a single exit point, however it is not always possible/practical and even like this it is possible to miss references, this is typically where cppy can help.

This is a very brief introduction to reference counting. You can read a bit more in the official Python documentation and in the Python API documentation.

2.2 Cppy smart pointer class

Cppy smart pointer (*cppy::ptr*) can be initialized with a pointer to a Python object as follows:

```
cppy::ptr obj_ptr( PyUnicode_FromString("test") )
```

When created, the class assume that you own the reference, if it is not the case you should incref it first:

```
PyObject* function( PyObject* obj )
{
    cppy::ptr obj_ptr( cppy::incref( obj ) );
    cppy::ptr obj_ptr2( obj, true );
}
```

Note: Cppy provides convenient inline function for common reference manipulation: - *cppy::incref*, *cppy::xincref*, *cppy::decref*, *cppy::xdecref* use the the similarly named Python macros and return the input value. - *cppy::clear*, *cppy::replace* are similar but return void.

You can also initialize a *cppy::ptr* from another *cppy::ptr* in which case the reference count will always be incremented.

The main advantage provided by *cppy::ptr* is that it implements a destructor that will be invoked automatically by the c++ runtime when the *cppy::ptr* goes out of scope. The destructor will decref the reference for you. As a consequence you can be sure that your reference you always be decremented when you leave the function.

Sometimes, however, that is not what you want, because you want to return the reference the *cppy::ptr* manage. You can request the *cppy::ptr* to give back the reference using its *release* method. Lets illustrate on a tiny example:

```
PyObject* function( PyObject* obj )
{
    cppy::ptr repr_ptr( PyObject_Repr( obj ) );
    return repr_ptr.release();
}
```

Function which are part of Python C-API are not aware of of *cppy::ptr* and when calling them you need to provide the original *PyObject**. To access, you simply need to call the *get* method of the *cppy::ptr* object.

```
PyObject* function( PyObject* obj )
{
    cppy::ptr l_ptr( PyList_New() );
    if( PyList_Append( l_ptr.get(), obj ) != 0 )
        return 0;
    return l_ptr.release();
}
```

Here we see that because we use *cppy::ptr* to manage the list, we do not have to worry about decrefing the reference if an exception occurs, the runtime will do it for us. If no exception occurs, we stop managing the reference and we are good.

Using cppy does not eliminate all the pitfalls of writing C-extensions. For example if you release too early (for example when passing the object to a function that may fail), you can still leak references. However it does alleviate some of the complexity.

2.3 Cppy::ptr methods

All methods that takes a *PyObject** can also accept a *cppy::ptr*. Most names should be self-explanatory, and apart from the is_ methods most of them rely on the PyObject_ functions similarly named:

```
bool is_none() const
bool is_true() const
bool is_false() const
bool is_bool() const
bool is_int() const
bool is_float() const
bool is_list() const
bool is_dict() const
bool is_set() const
bool is_bytes() const
bool is_str() const
bool is_unicode() const
bool is_callable() const
bool is_iter() const
bool is_type( PyTypeObject* cls ) const
int is_truthy() const
int is_instance( PyObject* cls ) const
int is_subclass( PyObject* cls ) const
PyObject* iter() const
PyObject* next() const
PyObject* repr() const
PyObject* str() const
PyObject* bytes() const
PyObject* unicode() const
Py_ssize_t length() const
PyTypeObject* type() const
int richcmp( PyObject* other, int opid ) const
long hash() const
bool hasattr( PyObject* attr ) const
bool hasattr( const char* attr ) const
bool hasattr( const std::string& attr ) const
PyObject* getattr( PyObject* attr ) const
PyObject* getattr( const char* attr ) const
PyObject* getattr( const std::string& attr ) const
bool setattr( PyObject* attr, PyObject* value ) const
bool setattr( const char* attr, PyObject* value ) const
bool setattr( const std::string& attr, PyObject* value ) const
bool delattr( PyObject* attr ) const
bool delattr( const char* attr ) const
bool delattr( const std::string& attr ) const
PyObject* getitem( PyObject* key ) const
bool setitem( PyObject* key, PyObject* value ) const
```

(continues on next page)

(continued from previous page)

```
bool delitem( PyObject* key )
PyObject* call( PyObject* args, PyObject* kwargs = 0 ) const
```

CHAPTER 3

Error reporting

In addition to *cppy::ptr*, cppy provides a set of convenience functions for reporting errors which all return a NULL pointer allowing them to be used as follows:

```
PyObject* function( PyObject* obj )
{
    cppy::ptr obj_ptr( cppy::incref( obj ) );
    if( !obj_ptr.is_bool() )
        return type_error( obj_ptr.get(), 'bool' )
    return obj_ptr.get()
}
```

3.1 Functions

Functions taking two arguments provide sensible pre-formatted error messages.

```
inline PyObject* system_error( const char* message )

inline PyObject* type_error( const char* message )

inline PyObject* type_error( PyObject* ob, const char* expected )

inline PyObject* value_error( const char* message )

inline PyObject* runtime_error( const char* message )

inline PyObject* attribute_error( const char* message )

inline PyObject* attribute_error( PyObject* ob, const char* attr )
```