Wrappyr Documentation

Release 0.1a1

Vincent den Boer

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Introduction

Wrappyr is a collection of tools aiming for tight integration both-way between Python and C/C++ using Ctypes. This means that when this goal is reached, you can write parts of your code in Python or C/C++ and switch between them without needing to change the rest of your program.

It is composed of three parts:

- A tool that imports the XML file produced by the Clang plugin and does two things:
  - Optional: Generate a C API for your C++ code.
  - Generate an XML file describing how the Python API should look, which C calls should be made where, etc.
- A tool that creates the Ctypes-based Python API from an XML file.
- A Clang plugin that dumps classes, functions, etc. from a header file to XML, which you can use when GCCXML produces incomplete output.

Requirements

General requirements:

- Recommended: GCCXML, which you can probably install through your package manager.

Python requirements:

- Python 2.6+
- lxml (installable through pip)
- argparse, available in standard library in Python >= 2.7 and through pip for earlier versions
- importlib, available in standard library in Python >= 2.7 and through pip for earlier versions

Getting started

To go from C/C++ to a working Python API, quite some things need to happen. First some data about the C/C++ code must be extracted. Then some code needs to be generated and compiled to make your code accessible through Ctypes. Also, an XML file needs to be generated which describes the final Python API and how it exactly it will use Ctypes wich will be used to generate the final Python code. You can do all these steps by hand using the commmands available in wrappyr/generate.py, but since you'll need to do these steps more than once, it's better to write a class (called a Package) that helps Wrappyr do most of these things by itself. We'll use the Box2D package shipped with Wrappyr for this example.

To generate all files needed for a working Python API, you can use the following commands (in the Wrappyr directory) to generate all needed code:

```
$ export PYTHONPATH="."
$ wrappyr/generate.py --package path.to.Package generate_from_package /tmp/ # Will␣
↪create Python packages in /tmp/
```

So to generate a Python API for Box2D (assuming it's installed on your system), use this command:

```
$ wrappyr/generate.py --package packages.Box2D.Box2DPackage generate_from_package /
↪tmp/ # Will generate /tmp/Box2D
```

After that you'll need to compile the C API to your C++ yourself. The code generated for Box2D is located in /tmp/ and you can compile it with the following commands:

```
$ cd /tmp/
$ g++ -shared -o libBox2DC.so box2d.cpp -lBox2D
```

Then after ensuring both the wrappyr and your generated packages are in your Python path, you're ready to use the generated Python API:

```
>>> from Box2D.common import b2Vec2
>>> v = b2Vec2(1.0, 3.1)
>>> v.Normalize()
3.2572994232177734
>>> v.x
0.30700278282165527
```

If you have any trouble setting getting started, feel free to contact me.

Where to go from here

Since the chances of Wrappyr generating a Python API that fully satisfies all your needs straight out of the box is very small, you'll want to write your own Package class to adapt the generation of the Python API to your needs. See *Creating Package classes* for information on how to do this.

CHAPTER 2

Creating Package classes

Create a Package class is the preferred way to customize to generation of a Python API. In a Package class you can:

- Tell Wrappyr some basic stuff about your project, like whether you're using C or C++, which header file read, etc.
- Tell Wrappyr to use your custom classes when generating the C API and the initial XML file describing the Python API.
- Preprocess the data exported from GCCXML or Clang.
- Preprocess the final Python API just before the code is generated.

To create a Package, subclass from *wrappyr.Package*. Below, I will describe each method you can override and how you can use it.

class wrappyr.**Package**
Base class for all Wrappyr packages.

Overriding the methods of this class is the preferred way to hook into the wrappyr/generate.py script. To use your Package subclass, pass the dotted path to it to the `--package` option of the wrappyr/generate.py script.

get_header_export()
Returns class used to generate the header of the C API.

You will almost always want to return a subclass of `wrappyr.code_data_conversion.HeaderExport`. The main reason to do this is to use a custom export filter controlling which classes and functions get exported to the Python API.

get_source_export()
Returns class used to generate the source of the C API.

You will almost always want to return a subclass of `wrappyr.code_data_conversion.SourceExport`. The main reasons to do this are to use a custom export filter controlling which classes and functions get exported to the Python API and to add custom includes to the source of the C API.

get_ctypes_export()
Returns class used to generate the XML describing the Python API.

You will almost always want to return a subclass of `wrappyr.code_data_conversion.CtypesExport`. Such a subclass can have a custom filter used to control which classes and functions get exported to the Python API. You can also use this to provide a basic namespace to package mapping.

get_source_language()
Returns which language your C/C++ is using as a string.

Return either lowercase `c` or `c+`.

get_source_header_path()
Returns the path of the header used to generate the Python API.

The default implementation searches /usr/local/include and /usr/include for the header file in the *source_header_name* attribute.

get_generated_header_path()
Returns the path where to write the header of the generated C API.

By default this generates a random temporary file ending in .h and stores it name in `generated_wrapper_prefix` so *get_generated_source_path* can use the same name ending in .cpp .

get_generated_source_path()
Returns the path where to write the source of the generated C API.

By default this generates a random temporary file ending in .cpp and stores it name in `generated_wrapper_prefix` so *get_generated_header_path* can use the same name ending in .h .

process_code_import(*importer*)
Preprocess the imported code data.

You can use this to store information about the C/C++ code. See the Box2D package in packages/Box2D.py which uses this to store the location of classes, which it uses in *process_ctypes_structure* to organize the final Python API into packages.

process_ctypes_structure(*structure*)
Preprocess the final Python API imported from XML.

This method is passed an argument named *structure*, which is an instance of the class *wrappyr.ctypes_api_builder.structure.CTypesStructure*.

Use this to do both changes that are cosmetic and changes that are necessary for the Python API to work. Cosmetic changes include converting method names to PEP8 convention and organizing classes into packages. Necessary changes include renaming arguments that are reserved Python keywords and removing ambiguous overloads.

See *Ctypes API generation* for the docs you'll need and some examples. Also, see the Box2D package in packages/Box2D.py for a real life example.

source_header_name
Property used by the default implementation of get_source_header_path to find the header file used as a starting point for generating everything necessary for the Python API. You can set this to something like `Box2D/Box2D.h`.

CHAPTER 3

Ctypes API generation

One of the parts of the Wrappyr project is the Ctypes API builder. It loads a description of a Ctypes based API (typically from XML) and generates Python code from that. The code that does this lives in `wrappyr.ctypes_api_builder`. Before Wrappyr generates the Python code, it gives you the chance to manipulate the API by calling the the *process_ctypes_structure* of your Package class. Here you can do things like:

- Grouping C functions into classes
- Renaming classes, methods, parameter names to match PEP8 naming conventions
- Organizing the API into packages
- And more...

This document describes all classes used to represent a Python API that you can manipulate and examples on how to use this API.

Reference

class `wrappyr.ctypes_api_builder.structure.`**Node**

The base class for all classes used to represent the API.

name

The name of this node, which will be used to look up Nodes by path.

parent

The parent node of this node. It will be set automatically if you you make this node a child of another node, like when you add a *Method* to a *Class*

get_path([*top*])

Get the dotted path from *top* to this node. If top is not given, the path from the furthest ancestor is returned.

```
>>> struct = CTypesStructure()
>>> Box2D = Package("Box2D")
>>> dyn = Package("dynamics")
>>> Box2D.add_package(dyn)
```

```
>>> dyn.get_path()
'dynamics'
>>> struct.add_package(Box2D)
>>> dyn.get_path()
'Box2D.dynamics'
>>> dyn.get_path(Box2D)
'dynamics'
```

get_closest_parent_of_type(*type*)
Get the closest ancestor of *type*

```
>>> p = Package("package")
>>> m = Module("module")
>>> c = Class("class")
>>> p.add_module(m)
>>> m.add_class(c)
>>> m == c.get_closest_parent_of_type(Module)
True
>>> p == c.get_closest_parent_of_type(Package)
True
```

get_closest_parent_module()
Shorthand for get_closest_parent_of_type(Module)

get_distance_to_parent(*parent*)
Returns the depth of this node calculated from *parent* or None if *parent* is not an ancestor of this node.

```
>>> p = Package("package")
>>> m = Module("module")
>>> c = Class("class")
>>> p.add_module(m)
>>> m.add_class(c)
>>> c.get_distance_to_parent(m)
1
>>> c.get_distance_to_parent(p)
2
```

find_lowest_common_parent(*other*)
Returns the first ancestor that this node shares with *other*

```
>>> dynamics = Package("dynamics")
>>> b2Body = Class("b2Body")
>>> joints = Package("joints")
>>> b2WeldJoint = Class("b2WeldJoint")
>>> dynamics.add_class(b2Body)
>>> dynamics.add_package(joints)
>>> joints.add_class(b2WeldJoint)
>>> dynamics == b2WeldJoint.find_lowest_common_parent(b2Body)
True
```

parents()
Returns a generator that iterates over all ancestors starting with the closest one.

```
>>> dynamics = Package("dynamics")
>>> joints = Package("joints")
>>> b2WeldJoint = Class("b2WeldJoint")
>>> dynamics.add_package(joints)
>>> joints.add_class(b2WeldJoint)
```

```
>>> tuple(b2WeldJoint.parents()) == (joints, dynamics)
True
```

class wrappyr.ctypes_api_builder.structure.**Module**(*Node*)

Class used to represent a Python module. A module can contain:

- *Library* instances
- *Function* instances
- *Class* instances
- PointerType instances

You can create a *Module* from XML by using the <module> tag:

```
<module name="test">
    ...
</module>
```

find_library([*name*])

Find *Library* with specified *name* in this *Module* or one of it's parents. If *name* is not given, it will search for a library specified as default.

add_library(*library*)

Add *Library* instance *library* to this module and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

remove_library(*library*)

Remove *Library* instance *library* from this module and set its parent to None.

every_library()

Returns all *Library* instances that this module contains.

add_class(*class*)

Add *Class* instance *class* to this module and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

remove_class(*class*)

Remove *Class* instance *class* from this module and set its parent to None.

every_class()

Returns all *Class* instances that this module contains.

add_function(*function*)

Add *Function* instance *function* to this module and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

remove_function(*function*)

Remove *Function* instance *function* from this module and set its parent to None.

every_function()

Returns all *Function* instances that this module contains.

add_pointer(*pointer*)

Add PointerType instance *pointer* to this module and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

Since a PointerType does not need to be a child of a Node to be used, this is probably only useful for loading from XML.

remove_pointer(*pointer*)
Remove `PointerType` instance *pointer* from this module and set its parent to `None`.

every_pointer()
Returns all `PointerType` instances that this module contains.

exception LibraryNotFound(*Exception*)
Exception thrown by *Module.find_library()* when it cannot find the requested *Library*.

class wrappyr.ctypes_api_builder.structure.**Package**(*Module*)
Class used to represent a Python package. In addition to everything a *Module* can contain, a package can contain:

- *Package* instances
- *Module* instances

You can create a *Package* from XML by using the <package> tag:

```
<package name="test">
    ...
</module>
```

add_package(*package*)
Add *Package* instance *package* to this package and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

remove_package(*package*)
Remove *Package* instance *package* from this package and set its parent to `None`.

every_package()
Returns all *Package* instances that this package contains.

add_module(*module*)
Add *Module* instance *module* to this package and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

remove_module(*module*)
Remove *Module* instance *module* from this package and set its parent to `None`.

every_module()
Returns all *Module* instances that this package contains.

class wrappyr.ctypes_api_builder.structure.**Library**(*Node*)
Class used to represent a C library. A library will be available for the *Node* it's placed in (either a *Package* or a *Module*) and all of its descendents.

You can create a *Library* from XML by using the <library> tag:

```
<package name="test">
    <library name="libA" path="libA.so" default="true" />
    <library name="libB" path="libB.so" />

    <function name="func_a">
        <call symbol="funcA">
            <!--
            Since libA is the default for this Package,
            the symbol funcA will be retrieved from libA.so
            -->
            ...
        </call>
    </function>
```

```
    <function name="func_b">
        <call symbol="funcB" library="libB">
            <!--
            We've explicitly chosen libB, so the symbol funcB
            will be retrieved from libB.so
            -->
            ...
        </call>
    </function>
</module>
```

class wrappyr.ctypes_api_builder.structure.**Class**(*Node*)
Class used to represent a Python class.

add_method(*method*)
Add *Method* instance *method* to this class and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

remove_method(*method*)
Remove *Method* instance *method* from this class and set its parent to None.

every_module()
Returns all *Method* instances that this class contains.

add_member(*member*)
Add *Member* instance *member* to this class and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

remove_member(*member*)
Remove *Member* instance *member* from this class and set its parent to None.

every_module()
Returns all *Member* instances that this class contains.

add_pointer(*pointer*)
Add PointerType instance *pointer* to this class and set its parent to this node. This does not remove it from its current parent, so you must remove it from its parent first (if it has one of course).

Since a PointerType does not need to be a child of a Node to be used, this is probably only useful for loading from XML.

remove_pointer(*pointer*)
Remove PointerType instance *pointer* from this class and set its parent to None.

every_pointer()
Returns all PointerType instances that this class contains.

class wrappyr.ctypes_api_builder.structure.**Function**(*Node*)
Class used to represent a Python function.

You can create a *Function* from XML by using the <function> tag:

```
<function name="func_a">
    <call symbol="funcA">
        <!-- A function makes calls to C functions. -->
    </call>
</function>
```

class wrappyr.ctypes_api_builder.structure.**Method**(*Function*)
Class used to represent a method of a Python class.

You can create a *Method* from XML by using the <method> tag:

```
<class name="Test">
    <method name="spam">
        <!-- Python likes spam and eggs, C like foo and bar ;) -->
        <call symbol="foo" />
    </method>
</class>
```

class wrappyr.ctypes_api_builder.structure.**Member**(*Node*)
Class used to represent a member Python of a class.

You can create a *Member* from XML by using the <member> tag and specify its getter and setter using the <getter> and <setter> tags respectively:

```
<class name="Vector">
    <member name="x">
        <getter>
            <call symbol="Vector_GetX">
                <returns type="ctypes.c_float" />
            </call>
        </getter>
        <setter>
            <call symbol="Vector_SetX">
                <argument type="ctypes.c_float" />
            </call>
        </setter>
    </method>
</class>
```

class wrappyr.ctypes_api_builder.structure.**Call**(*Node*)
Class used to represent a call to a C function.

You can create a *Call* from XML by using the <call> tag:

```
<function name="take_over_the_world">
    <!--
    Obviously much more effecient in C, altough things
    will probably crash before anything useful happens.
    -->
    <call symbol="conquerWorld">
        <argument name="timeout" type="ctypes.c_uint" />
        <returns type="ctypes.c_bool">
    </call>
</function>
```

class wrappyr.ctypes_api_builder.structure.**Argument**(*Node*)
Class used to represent an argument to a C function.

You can create an *Argument* instance from XML by using the <argument> tag:

```
<module name="module">
    <class name="Class" />

    <function name="func">
        <call symbol="func">
            <!-- You can reference existing classes by using full dotted paths -->
            <argument name="a" type="module.Class" />
```

```
            <!-- You can also use ctypes.* types -->
            <argument name="b" type="ctypes.c_int" />
        </call>
    </function>
</module>
```

steals
Boolean value indicating whether passing an object as this argument to a call will steal the ownership over the object. Defaults to `False`. See *Memory management* for more information.

invalidates
Boolean value indicating whether passing an object as this argument to a call will invalidate the object. Defaults to `False`. See *Memory management* for more information.

class `wrappyr.ctypes_api_builder.structure.`**ReturnValue**(*Node*)
Class used to represent the return value of a C function.

You can create a *ReturnValue* instance from XML by using the <returns> tag:

```
<function name="func">
    <call symbol="func">
        <returns type="ctypes.c_float" />
    </call>
</function>
```

ownership
Boolean value indicating whether we have ownership over objects returned by this call. See *Memory management* for more information.

class `wrappyr.ctypes_api_builder.structure.`**CTypesStructure**(*Node*)
The root of the structure that describes the Python API. This will contain the root *Package* s and *Module* s of the Python API.

This class is represented by the XML tag <ctypes>. Since this is the root of the structure, this must also be the root node of the XML document:

```
<?xml version="1.0"?>
<ctypes>
    <package name="MyPackage">
        ...
    </package>
</ctypes>
```

get_by_path(*path*)
Return the *Node* found under the dotted path *path*:

```
>>> structure = CTypesStructure()
>>> mod = Module("mod")
>>> cls = Class("Class")
>>> structure.add_module(module)
>>> mod.add_class(cls)
>>> cls == structure.get_by_path("mod.Class")
True
```

class `wrappyr.ctypes_api_builder.structure.`**CTypesStructureVisitor**
A convenience class that takes a *CTypesStructure* and calls visit_<class name> on itself for every node it finds. So as an example, it will call visit_Method(method) for every method it finds.

Example:

```
class ClassPrinter(CTypesStructureVisitor):
    def visit_Class(cls):
        print "Found class: " + cls.name
```

process(*node*)
Will start at *node* and call the corresponding visit_* method for *node* and all of its descendents.

Examples

The recommended way to use this API is to process a *CTypesStructure* from within a *wrappyr.Package*. See *Creating Package classes* for an introduction on Packages. This section gives a few exmples of how you might preprocess a Python API.

Reorganize an API that doesn't use namespaces into packages:

```
import wrappyr
from wrappyr.ctypes_api_builder.structure import Package

class Box2DPackage(wrappyr.Package):
    CLASS_TO_PACKAGE = {
        'b2Vec2': 'Box2D.common',
        'b2Vec3': 'Box2D.common',
        'b2Shape': 'Box2D.collision',
        'b2PolygonShape': 'Box2D.collision',
        'b2World': 'Box2D.dynamics',
        'b2Body': 'Box2D.dynamics',
        'b2Joint': 'Box2D.dynamic.joints',
        'b2WeldJoint': 'Box2D.dynamic.joints',
    }

    def process_ctypes_structure(structure):
        # Get all unique package paths
        package_paths = set(self.CLASS_TO_PACKAGE.values())
        # Sort by depth in tree
        package_paths = sorted(packages, lambda name: name.count("."))

        # Dictionary to hold all packages to be looked up by path
        packages = {}
        for path in package_paths:
            last_dot = path.rfind(".")

            # The name of the package is the part after the last dot
            name = path[last_dot + 1 :]
            package = Package(name)
            packages[path] = package

            # Add the package to its parent in the tree.
            # We assume that the Box2D package already exists.
            parent_path = path[: last_dot]
            parent = structure.get_by_path(parent_path)
            parent.add_package(package)

        # Now we remove classes from their current parent and
        # add them to their new package.
        for class_name, package_path in self.CLASS_TO_PACKAGE.items():
            cls = structure.get_by_path("Box2D." + class_name)
```

```
            cls.parent.remove_class(cls)
            packages[package_path].add_class(cls)
```

Convert camelCase method names into lowercase_with_underscores:

```
import re
import wrappyr
from wrappyr.ctypes_api_builder.structure import CTypesStructureVisitor

class CamelCaseTerminator(CTypesStructureVisitor):
    def __init__(self):
        self.to_rename = []

    def visit_Method(self, method):
        self.to_rename.append(method)

    def terminate(self):
        regex = re.compile(r'([a-z])([A-Z])')
        to_underscore = lambda match: "%s_%s" % (match.group(1), match.group(2).
↪lower())
        for method in self.to_rename:
            parent = method.parent
            parent.remove_method(method)
            method.name = regex.sub(to_underscore, method.name)
            parent.add_method(method)

class MyPackage(wrappyr.Package):
    def process_ctypes_structure(structure):
        terminator = CamelCaseTerminator()
        terminator.process(structure)
        terminator.terminate()
```

CHAPTER 4

Memory management

Warning: This is not actually implemented yet, but I'm writing this in the spirit of Documentation Driven Development. Memory management is currently broken in Wrappyr and will crash your program or leak memory if you're lucky. Yes, I'm working on it :).

Memory management is quite an important part of programming in C/C++. When working in Python however, we expect this to work automatically. For this to happen, we need to provide some hints to Wrappyr how it can safely clean up after the Python programmer.

By default Wrappyr is extremely careful not to free memory it does not manage and will leak a lot of memory as a result. This is because there are different models of memory management. Let's say for example that you have a Window class with a method createWindow that returns a pointer to to newly created Window. Does Wrappyr need to destroy the Window after we're done with it, or will the parent Window destroy it? And when you pass a Window pointer to a its parent's removeWindow method, can we continue to work with that Window? Wrappyr relies on you for answers to the questions, because only you as a human fully know what is going on.

When Wrappyr runs, it ask the following questions to manage memory:

- I have a pointer; does it still point to a *valid* object so I can safely apply operations to it?
- I have a pointer; do I have *ownership* and do I need to deallocate the object when I'm done with it?
- I'm passing an object as an argument to a function; will the call *invalidate* my object?
- I'm passing an object as an argument to a function; will the call *steal* my *ownership* over the object?
- A function returned an object; do I now have the *ownership* over the object?

By default Wrappyr assumes the following to answer these questions:

- When a funtion returns an object that Wrappyr copied to the heap, Wrappyr has *ownership* over that copy. This applies to functions like:

```
Vector2 Cross(Vector2 a, Vector2 b);
```

- When a function returns a pointer or reference to an object, Wrappyr does **not** have *ownership* over the object, so you must deallocate it manually if necessary.
- Passing an object to a function **not** change whether the object is *valid* or wheter Wrappyr has *ownership* over it.

Providing memory management hints

Wrappyr uses memory management hints when generating the Python API, so to provide memory management hints to Wrappyr, you modify the Python API representation (which you typically do in a *wrappyr.Package*, specifically the *wrappyr.Package.process_ctypes_structure()* method).

Now I'll provide a list of hints you can give. Before you read this, make sure you have read and understood the concepts of *valid*, *ownership* and *stealing* is explained in the previous section.

- To indicate that passing an object as argument to a call will *invalidate* the object, set the *Argument.invalidates* attribute to `True`. If you then pass an object to this call, the object will be marked as invalid and its internal pointer to None. If you then try to pass the object to another call, an exception will be raised.
- To indicate that the *ownership* of an object will be *stolen* when passing it as an argument to a function, set the *Argument.steals* attribute to `True`. After this, the object won't be deallocated when it's not used in Python anymore. When you then try to pass the object to another call that will try to *steal* the ownership, an exception will be raised. To mark that it's safe to pass the object that is not owned to the stealing call, set the *Argument.steals* attribute to the string `safe`.
- To indicate a call gives you *ownership* over the object it returns, set the *ReturnValue.ownership* attribute to true.

Examples

Provide hints about memory management for some calls in the API:

```
import wrappyr

class hint(object):
    def __init__(name, argument = None, steals = None, invalidates = None, ownership␣
↪= None):
        self.name = name
        self.argument = argument
        self.steals = steals
        self.invalidates = invalidates
        self.ownership = ownership

    def apply(self, call):
        for prop in ("steals", "invalidates"):
            value = getattr(self, prop)
            if value is not None:
                for arg in call.args:
                    if arg.name == self.argument:
                        setattr(arg, prop, value)
        if self.ownership is not None and call.returns:
            call.returns.ownership = self.ownership

class MyGuiPackage(wrappyr.Package):
    HINTS = [
        hint("MyGui.Window.add_child", "child", ownership = True),
```

```
        hint("MyGui.Window.destroy_child", "child", invalidates = True),
        hint("MyGui.Window.detach_child", ownership = True),
    ]

    def process_ctypes_structure(self, structure):
        for hint in self.hints:
            f = structure.get_by_path(hint.name)
            for call in f.ops:
                hint.apply(call)
```

Assume that all methods that begin with `destroy_` invalidate the arguments passed to them:

```
import wrappyr
from wrappyr.ctypes_api_builder.structure import CTypesStructureVisitor

class FixDestroyMethods(CTypesStructureVisitor):
    def visit_Method(self, method):
        if method.raw:
            return # Method only conains Python code, so skip it.
        if not method.name.startswith("destroy_"):
            return # Method does not start with destroy_

        for call in method.ops:
            for arg in call.args:
                arg.invalidates = True

class MyPackage(wrappyr.Package):
    def process_ctypes_structure(self, structure):
        FixDestroyMethods().process(structure)
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

w

Index