# PyBindGen Documentation

*Release 0.17.0*

Gustavo J. A. M. Carneiro, Mathieu Lacage

February 16, 2014

# Contents

Contents:

CHAPTER 1

# PyBindGen Tutorial

## 1.1 What is PyBindGen ?

PyBindGen is a tool which can be used to generate python bindings for C or C++ APIs. It is similar in scope to tools such as boost::python, SWIG, and a few others but has a number of specific features which make it especially useful in a number of cases:

- PyBindGen is implemented in python and is used and controlled through python;
- PyBindGen error messages do not involve c++ template deciphering (as in boost::python);
- PyBindGen generates highly-readable C or C++ code so it is possible to step into and debug the bindings;
- In simple cases, PyBindGen is really easy to use. In more complicated cases, it does offer all the flexibility you need to wrap complex C or C++ APIs;
- PyBindGen also provides an optional tool to parse C and C++ headers and generate automatically bindings for them, potentially using extra inline or out-of-line annotations. This tool is based on gccxml and pygccxml: it can be used to generate the first version of the bindings and tweak them by hand later or as a fully automated tool to continuously generate bindings for changing C/C++ APIs.

This tutorial will show how to build bindings for a couple of common C and C++ API idioms and, then, will proceed to show how to use the automatic binding generator.

## 1.2 Supported Python versions

PyBindGen officially supports Python versions 2.6, 2.7, and >= 3.3[*] (tested in 3.3 and 3.4).

PyBindGen does not support Python versions 3.0, 3.1, and 3.2.

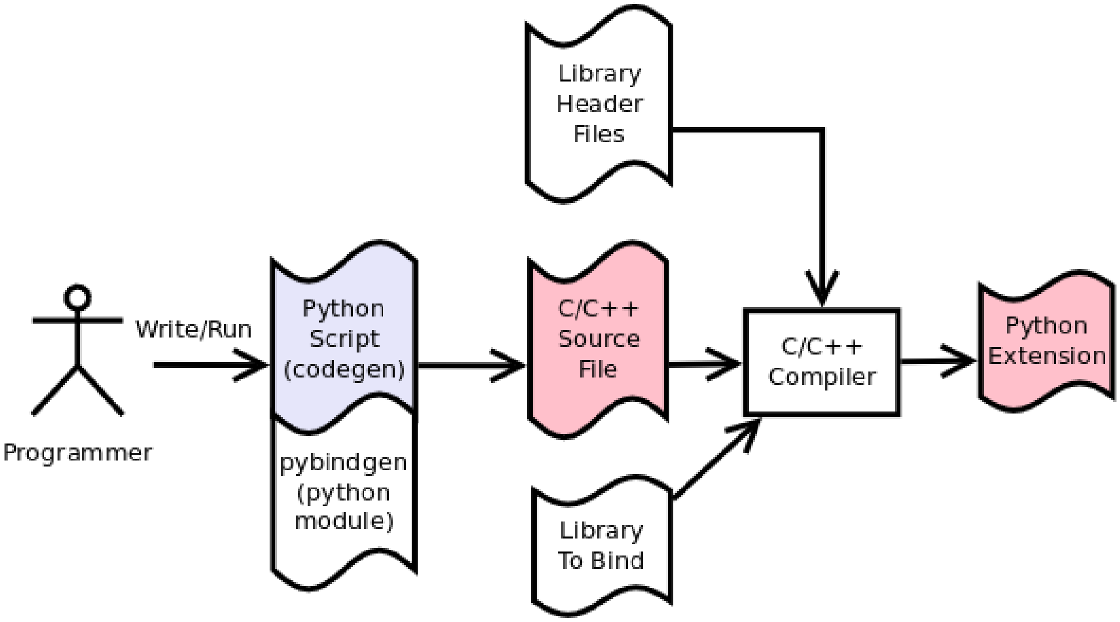
Note that C files generated by PyBindGen transparently support multiple Python versions. In particular, the generated code can build in either Python 2.x or Python 3.x.

[*] The automatic header file scanning feature of PyBindGen (submodule *pybindgen.gccxmlparser*) does not work in any Python 3.x version, since the pygccxml package has not been ported to Python 3.

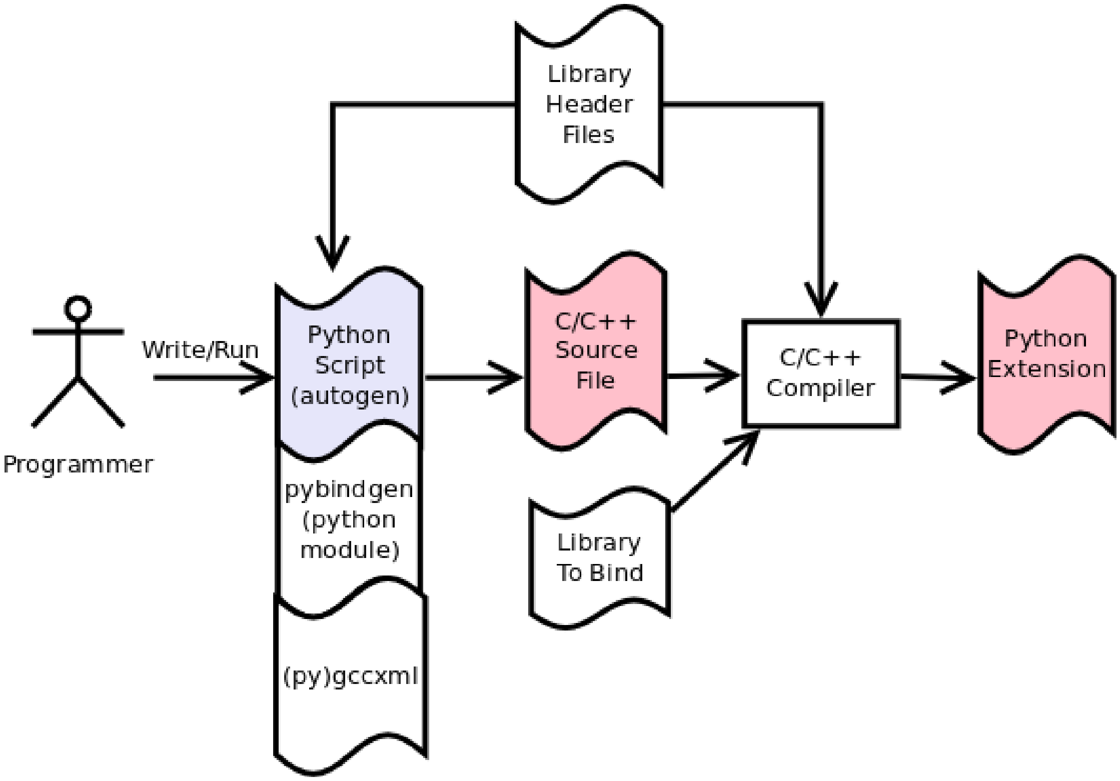
## 1.3 Work flows

PyBindGen is only a Python module. The programmer must write a Python script that uses the module in order to generate the bindings. There are several ways that PyBindGen can be used:

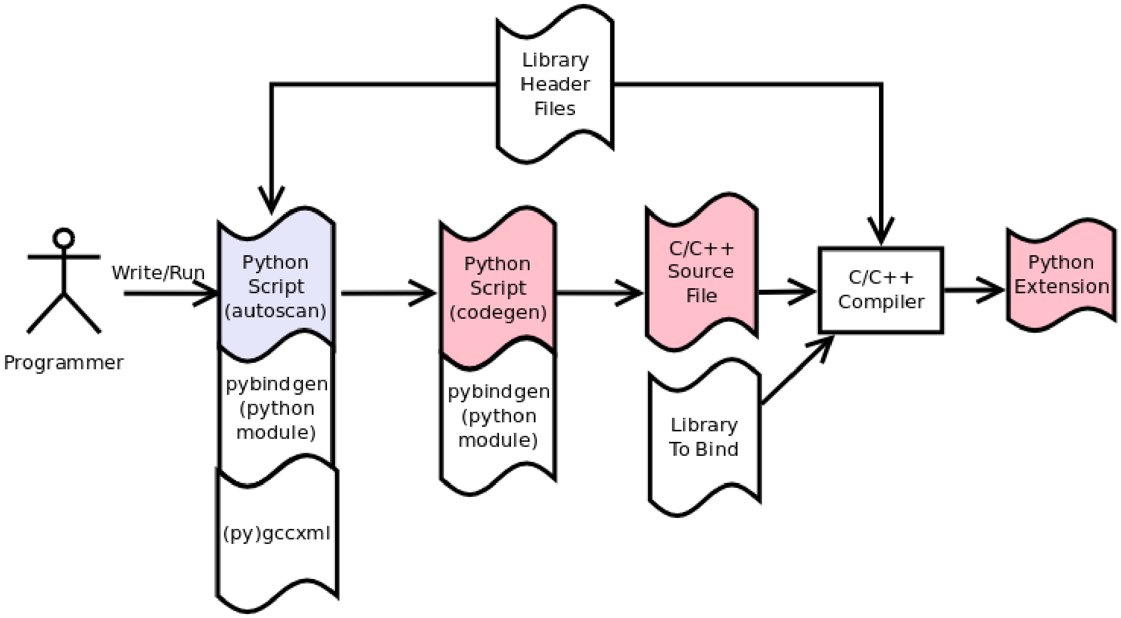
1. Basic mode, script that directly generates the bindings;



2. Automatically scan header files and generate the bindings directly (see *Header file scanning with (py)gccxml*);



3. Automatically scan header files to generate API defs file, that file can later be used to generate the bindings. See (see *Header file scanning with (py)gccxml: python intermediate file*);



## 1.4 A simple example

The best way to get a feel for what PyBindGen looks like is to go through a simple example.

### 1.4.1 Code generation script

Let's assume that we have a simple C API as shown below declared in a header my-module.h:

```
void MyModuleDoAction (void);
```

What we want to do is call this C function from python and be able to write python code such as:

```
import MyModule

MyModule.MyModuleDoAction ()
```

Getting there is, hopefully, not very complicated: we just need to write a small python program whose job is to generate the C code which will act as a bridge between our user's python program and the underlying C function. First, we import the pybindgen and the sys modules:

```
import pybindgen
import sys
```

Then, we create an object to represent the module we want to generate:

```
mod = pybindgen.Module('MyModule')
```

add our C header:

```
mod.add_include('"my-module.h"')
```

and register our function which returns no value (hence, the second argument 'None'), and, takes no arguments (hence, the third argument, the empty list '[]'):

```
mod.add_function('MyModuleDoAction', None, [])
```

Finally, we generate code for this binding directed to standard output:

```
mod.generate(sys.stdout)
```

The final program is pretty short:

```
import pybindgen
import sys

mod = pybindgen.Module('MyModule')
mod.add_include('"my-module.h"')
mod.add_function('MyModuleDoAction', None, [])
mod.generate(sys.stdout)
```

### 1.4.2 Building it using Python setup.py (distutils)

This very small example is located in the `first-example directory`, together with a small makefile which will build our extension module:

```
$ cd first-example/
$ python setup.py build
```

The *setup.py* is mostly a standard Python distutils driver script. Please refer to the Python documentation for more information.

The unusual part about this *setup.py* is that it imports *mymodulegen* and calls the generate function, with the *build/my-module-binding.c* as argument:

```
from mymodulegen import generate
module_fname = os.path.join("build", "my-module-binding.c")
with open(module_fname, "wt") as file_:
    print("Generating file {}".format(module_fname))
    generate(file_)
```

After *build/my-module-binding.c* having been generated, with the help of PyBindGen (as seen in the previous section), we can use it as one of the source files for our extension module:

```
mymodule = Extension('mymodule',
                     sources = [module_fname, 'my-module.c'],
                     include_dirs=['.'])
```

The rest is standard setup.py code.

### 1.4.3 Testing it

Once all of that code is built, we obviously want to run it. Setting up your system to make sure that the python module is found by the python runtime is outside the scope of this tutorial but, for most people, the following session should be self-explanatory:

```
$ cd first-example/
$ python setup.py buid
Generating file build/my-module-binding.c
running build
running build_ext
building 'mymodule' extension
```

```
creating build/temp.linux-x86_64-2.7
creating build/temp.linux-x86_64-2.7/build
[...]
$ export PYTHONPATH=build/lib.linux-x86_64-2.7/
$ python
Python 2.7.5+ (default, Sep 19 2013, 13:48:49)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import mymodule
>>> mymodule.MyModuleDoAction ()
You called MyModuleDoAction !
>>>
```

## 1.5 Wrapping types by value

### 1.5.1 Primitive types

The first example showed how to call a function which takes no arguments and returns no values which, obviously, is not especially interesting so, let's look at how we can give meaningfull arguments to our function:

```
int MyModuleDoAction (int v1, int v2);
```

and the corresponding bit from the code generation script: the second argument to add_function specifies that our function returns a value of type 'int' and the third argument specifies that our function takes as a single argument an 'int' of name 'value':

```
mod.add_function('MyModuleDoAction',
                  pybindgen.retval ('int'),
                 [pybindgen.param ('int', 'v1'),
                  pybindgen.param ('int', 'v2')])
```

The above then allows you to write:

```
>>> import MyModule
>>> v = MyModule.MyModuleDoAction (10, -1)
You called MyModuleDoAction: 10
>>> print v
10
>>> v = MyModule.MyModuleDoAction (v2=5, v1=-2)
You called MyModuleDoAction: -2
>>> print v
-2
```

Which shows how the argument name can be used to avoid using positional arguments.

Of course, the above example could be rewritten to the more compact and readable:

```
from pybindgen import *
mod.add_function('MyModuleDoAction', retval ('int'),
                 [param ('int', 'v1'),
                  param ('int', 'v2')])
```

In the following examples, this is what we will do to avoid extra typing.

### 1.5.2 Enum types

Enums are often used to define C and C++ constants as shown below:

```
enum MyEnum_e
{
  CONSTANT_A,
  CONSTANT_B,
  CONSTANT_C
};
void MyModuleDoAction (enum enum_e value);
```

And wrapping them is also pretty trivial:

```
from pybindgen import *
import sys

mod = Module(’MyModule’)
mod.add_include(’"my-module.h"’)
mod.add_enum(’MyEnum_e’, [’CONSTANT_A’, ’CONSTANT_B’, ’CONSTANT_C’])
mod.add_function(’MyModuleDoAction’, None, [param(’MyEnum_e’, ’value’)])
mod.generate(sys.stdout)
```

With the resulting python-visible API:

```
>>> import MyModule
>>> print MyModule.CONSTANT_A
0
>>> print MyModule.CONSTANT_B
1
>>> print MyModule.CONSTANT_C
2
>>> MyModule.MyModuleDoAction (MyModule.CONSTANT_B)
MyModuleDoAction: 1
```

### 1.5.3 Compound types

Passing a structure to and from C is not really more complicated than our previous example. The API below:

```
struct MyModuleStruct
{
  int a;
  int b;
};
struct MyModuleStruct MyModuleDoAction (struct MyModuleStruct value);
```

can be bound to python using the following script:

```
from pybindgen import *
import sys

mod = Module(’MyModule’)
mod.add_include(’"my-module.h"’)
struct = mod.add_struct(’MyModuleStruct’)
struct.add_instance_attribute(’a’, ’int’)
struct.add_instance_attribute(’b’, ’int’)
mod.add_function(’MyModuleDoAction’, retval (’MyModuleStruct’), [param (’MyModuleStruct’, ’value’)])
mod.generate(sys.stdout)
```

The most obvious change here is that we have to define the new structure type:

```
struct = mod.add_struct('MyModuleStruct')
```

and register the names and types of each of the members we want to make accessible from python:

```
struct.add_instance_attribute('a', 'int')
struct.add_instance_attribute('b', 'int')
```

The name of the method called here, ‘add_instance_attribute’ reflects the fact that PyBindGen can wrap both C and C++ APIs: in C++, there exist both instance and static members so, PyBindGen provides two methods: add_instance_attribute and add_static_attribute to register these two kinds of members.

**Our C API then becomes accessible from python::**

```
>>> import MyModule
>>> st = MyModule.MyModuleStruct ()
>>> st.a = 10
>>> st.b = -20
>>> st.c = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyModule.MyModuleStruct' object has no attribute 'c'
>>> v = MyModule.MyModuleDoAction (st)
You called MyModuleDoAction: 10
>>> print v
<MyModule.MyModuleStruct object at 0x2b5ef522b150>
>>> print v.a
10
>>> print v.b
-20
```

### 1.5.4 C++ classes

Wrapping C++ classes is very similar to wrapping a C struct with a few functions: we will thus start by extending our C API with a C++ class declaration:

```
class MyClass
{
public:
  void SetInt (int value);
  int GetInt (void) const;
};
```

We first need to declare a C++ class:

```
mod = Module('MyModule')
klass = mod.add_class('MyClass')
```

and, then, specify that it has a constructor:

```
klass.add_constructor([])
```

We can declare the setter method which is really a straightforward extension from the add_function function:

```
klass.add_method('SetInt', None, [param('int', 'value')])
```

The getter is also pretty straightforward except for the declaration of constness:

```
klass.add_method(’GetInt’, retval(’int’), [], is_const=True)
```

Using this API is also very similar to the struct example we went through in the previous section:

```
>>> my = MyModule.MyClass()
>>> my.SetInt(10)
>>> v = my.GetInt()
>>> print v
10
```

It is also possible to bind inner classes and enums such as these:

```
class Outer
{
public:
  void Do (void);
  // an inner enum
  enum inner_e
  {
    INNER_A,
    INNER_B,
    INNER_C
  };
  // an inner class
  class Inner
  {
  public:
    void Do (enum Outer::inner_e value);
  };
};
```

We just need to bind the outer class:

```
outer = mod.add_class(’Outer’)
outer.add_constructor([])
outer.add_method(’Do’, None, [])
```

Then, bind its inner enum:

```
mod.add_enum(’inner_e’, [’INNER_A’, ’INNER_B’, ’INNER_C’], outer_class=outer)
```

and, finally, bind its inner class:

```
mod.add_class(’Inner’, outer_class=outer)
inner.add_constructor([])
```

The only slightly tricky part is binding the Do method of the Inner class since it refers to the enum type defined in the Outer class: we simply need to carefully use the fully scoped name of the enum.:

```
inner.add_method(’Do’, None, [param(’Outer::inner_e’, value)])
```

The resulting python API reflects the underlying C++ API very closely:

```
>>> import MyModule
>>> print MyModule.Outer.INNER_A
0
>>> print MyModule.Outer.INNER_B
1
>>> outer = MyModule.Outer()
>>> outer.Do()
>>> inner = MyModule.Outer.Inner()
>>> inner.Do(MyModule.Outer.INNER_A)
```

### 1.5.5 C++ namespaces

Wrapping multiple nested namespaces is, of course, possible and represents no special challenge. Let's look at an example:

```
namespace Outer {
  void Do (void);
  class MyClass
  {};
  namespace Inner {
    void Do (void);
    class MyClass
    {};
  } // namespace Inner
} // namespace Outer
```

First, we need to define the Outer namespace:

```
mod = Module('MyModule')
outer = mod.add_cpp_namespace('Outer')
```

Then, register its classes and functions:

```
outer.add_class('MyClass')
outer.add_function('Do', None, [])
```

and, finally, define the Inner namespace and its associated functions and methods:

```
inner = outer.add_cpp_namespace('Inner')
inner.add_class('MyClass')
inner.add_function('Do', None, [])
```

The resulting API, again, sticks to the underlying C++ API by defining one python module for each C++ namespace and making sure that the hierarchy of python modules matches the hierarchy of C++ namespaces:

```
>>> import MyModule
>>> o = MyModule.Outer.MyClass()
>>> i = MyModule.Outer.Inner.MyClass()
>>> from MyModule.Outer.Inner import *
>>> i = MyClass()
```

## 1.6 Memory management for pointer types

Until then, we have shown how to pass back and forth data through C/C++ APIs only by value but, a large fraction of real-world APIs use raw pointers (and, in the case of C++, smart pointers) as arguments or return values of functions/methods.

Rather than try to explain the detail of every option offered by PyBindGen to deal with pointers, we will go through a couple of very classic memory management schemes and examples.

### 1.6.1 Function returns pointer

The API to bind:

```
class MyClass;
MyClass *DoSomethingAndReturnClass (void);
```

First, we declare the MyClass type:

```
mod.add_class('MyClass')
...
```

Then, if we assume that the function returns ownership of the pointer to the caller, we can write:

```
mod.add_function('DoSomethingAndReturnClass', retval('MyClass *', caller_owns_return=True), [])
```

The above will tell PyBindGen that the caller (the python runtime) becomes responsible for deleting the instance of MyClass returned by the function DoSomethingAndReturnClass when it is done with it.

Of course, it is possible to not give back ownership of the returned pointer to the caller:

```
mod.add_function('DoSomethingAndReturnClass', retval('MyClass *', caller_owns_return=False), [])
```

Which would make the python runtime assume that the lifetime of the returned pointer is longer than the associated python object.

### 1.6.2 Function takes pointer

The API to bind:

```
class MyClass;
void DoWithClass (MyClass *cls);
```

If we assume that the callee takes ownership of the input pointer, we can write:

```
mod.add_function('DoWithClass', None, [param('MyClass *', 'cls', transfer_ownership=True)])
```

Which will make python keep a handle on the MyClass instance but never destroy it himself and rely on the callee to destroy it at the right time. This kind of scheme is obviously a bit dangerous because python has no way of knowing when the underlying MyClass instance is really destroyed so, if you try to invoke methods on it _after_ it has been destroyed, bad things will obviously happen.

If, instead, we assume that the caller keeps ownership of the pointer, we can write the much safer version:

```
mod.add_function('DoWithClass', None, [param('MyClass *', 'cls', transfer_ownership=False)])
```

Which will allow python to delete the MyClass instance only when the associated python wrapper disappears.

### 1.6.3 A reference-counted object

A nice way to avoid some of the ambiguities of the above-mentioned API bindings is to use reference-counted C or C++ objects which must provide a pair of functions or methods to increase or decrease the reference count of the object. For example, a classic C++ reference-counted class:

```
class MyClass
{
public:
  void Ref (void);
  void Unref (void);
  uint32_t PeekRef (void);
};
```

And the associated function which takes a pointer:

```
void DoSomething (MyClass *cls);
```

To wrap this class, we first need to declare our class:

```
from pybindgen import cppclass
[...]
mod.add_class('MyClass', memory_policy=cppclass.ReferenceCountingMethodsPolicy(
                  incref_method='Ref',
                  decref_method='Unref',
                  peekref_method='PeekRef'))
```

The above allows PyBindGen to maintain and track the reference count of the MyClass object while the code below shows how we can declare a function taking a pointer as input:

```
mod.add_function('DoSomething', None, [param('MyClass *', 'cls', transfer_ownership=False)]
```

Here, the meaning of transfer_ownership changes slightly. Whithout reference counting, transfer_ownership refers to the transfer of the object as a whole, i.e. either the caller or callee will own the object in the end, but not both. With reference counting, transfer_ownership refers to the transfer of a _reference_. In this example, transfer_ownership=False means that the caller will not "steal" our reference, i.e. it will either not keep a reference to our object for itself, or if it does it creates its own reference to the object by calling the incref method. If transfer_ownership=True it would mean that the caller would keep the passed in reference to itself, and if the caller wants to keep the reference it must call the incref method first.

A more interesting case is that of returning such a reference counted object from a function:

```
MyClass *DoSomething (void);
```

While classic reference counting rules require that the callee returns a reference to the caller (i.e., it calls Ref on behalf of the caller before returning the pointer), some APIs will undoubtedly return a pointer and expect the caller to acquire a reference to the returned object by calling Ref himself. PyBindGen hopefully can be made to support this case too:

```
mod.add_function('DoSomething', retval('MyClass *', caller_owns_return=False), [])
```

Which instructs PyBindGen that DoSomething is not to be trusted and that it should acquire ownership of the returned pointer if it needs to keep track of it.

### 1.6.4 A STL container

If you have a function that takes a STL container, you have to tell PyBindGen to wrap the container first:

```
void DoSomething (std::list<std::string> const &listOfStrings);
```

Is wrapped by:

```
module.add_container('std::list<std::string>', 'std::string', 'list') # declare a container only onc
[...]
mod.add_function('DoSomething', None, [param('std::list<std::string> const &', 'listOfStrings')])
```

## 1.7 Advanced usage

### 1.7.1 Basic interface with error handling

It is also possible to declare a error handler. The error handler will be invoked for API definitions that cannot be wrapped for some reason:

```
#! /usr/bin/env python

import sys

import pybindgen
from pybindgen import Module, FileCodeSink, retval, param

import pybindgen.settings
import warnings

class ErrorHandler(pybindgen.settings.ErrorHandler):
    def handle_error(self, wrapper, exception, traceback_):
        warnings.warn("exception %r in wrapper %s" % (exception, wrapper))
        return True
pybindgen.settings.error_handler = ErrorHandler()


def my_module_gen(out_file):
    pybindgen.write_preamble(FileCodeSink(out_file))

    mod = Module(’a’)
    mod.add_include(’"a.h"’)

    mod.add_function(’ADoA’, None, [])
    mod.add_function(’ADoB’, None, [param(’uint32_t’, ’b’)])
    mod.add_function(’ADoC’, retval(’uint32_t’), [])

    mod.generate(FileCodeSink(out_file) )

if __name__ == ’__main__’:
    my_module_gen(sys.stdout)
```

In this example, we register a error handler that allows PyBindGen to simply ignore API definitions with errors, and not wrap them, but move on.

The difference between is Parameter.new(...) and param(...), as well as between ReturnValue.new(...) and retval(...) is to be noted here. The main difference is not that param(...) and retval(...) are shorter, it is that they allow delayed error handling. For example, when you put Parameter.new(“type that does not exist”, “foo”) in your python script, a TypeLookupError exception is raised and it is not possible for the error handler to catch it. However, param(...) does not try to lookup the type handler immediately and instead lets Module.add_function() do that in a way that the error handler can be invoked and the function is simply not wrapped if the error handler says so.

### 1.7.2 Header file scanning with (py)gccxml

If you have gccxml and pygccxml installed, PyBindGen can use them to scan the API definitions directly from the header files:

```
#! /usr/bin/env python
```

```
import sys

import pybindgen
from pybindgen import FileCodeSink
from pybindgen.gccxmlparser import ModuleParser

def my_module_gen():
    module_parser = ModuleParser('a1', '::')
    module = module_parser.parse([sys.argv[1]])
    module.add_include('"a.h"')

    pybindgen.write_preamble(FileCodeSink(sys.stdout))
    module.generate(FileCodeSink(sys.stdout))

if __name__ == '__main__':
    my_module_gen()
```

The above script will generate the bindings for the module directly. It expects the input header file, a.h, as first command line argument.

### 1.7.3 Header file scanning with (py)gccxml: python intermediate file

The final code generation flow supported by PyBindGen is a hybrid of the previous ones. One script scans C/C++ header files, but instead of generating C/C++ binding code directly it instead generates a PyBindGen based Python script:

```
#! /usr/bin/env python

import sys

from pybindgen import FileCodeSink
from pybindgen.gccxmlparser import ModuleParser

def my_module_gen():
    module_parser = ModuleParser('a2', '::')
    module_parser.parse([sys.argv[1]], includes=['"a.h"'], pygen_sink=FileCodeSink(sys.stdout))

if __name__ == '__main__':
    my_module_gen()
```

The above script produces a Python program on stdout. Running the generated Python program will, in turn, generate the C++ code binding our interface.

CHAPTER 2

# PyBindGen API Reference

## 2.1 Higher layers

### 2.1.1 module: generate Python modules and submodules

Objects that represent – and generate code for – C/C++ Python extension modules.

#### Modules and Sub-modules

A L{Module} object takes care of generating the code for a Python module. The way a Python module is organized is as follows. There is one "root" L{Module} object. There can be any number of L{SubModule}s. Sub-modules themselves can have additional sub-modules. Calling L{Module.generate} on the root module will trigger code generation for the whole module, not only functions and types, but also all its sub-modules.

In Python, a sub-module will appear as a I{built-in} Python module that is available as an attribute of its parent module. For instance, a module I{foo} having a sub-module I{xpto} appears like this:

```
|>>> import foo
|>>> foo.xpto
|<module 'foo.xpto' (built-in)>
```

#### Modules and C++ namespaces

Modules can be associated with specific C++ namespaces. This means, for instance, that any C++ class wrapped inside that module must belong to that C++ namespace. Example:

```
|>>> from cppclass import *
|>>> mod = Module("foo", cpp_namespace="::foo")
|>>> mod.add_class("Bar")
|<pybindgen.CppClass 'foo::Bar'>
```

When we have a toplevel C++ namespace which contains another nested namespace, we want to wrap the nested namespace as a Python sub-module. The method L{ModuleBase.add_cpp_namespace} makes it easy to create sub-modules for wrapping nested namespaces. For instance:

```
|>>> from cppclass import *
|>>> mod = Module("foo", cpp_namespace="::foo")
|>>> submod = mod.add_cpp_namespace('xpto')
|>>> submod.add_class("Bar")
|<pybindgen.CppClass 'foo::xpto::Bar'>
```

**class** `pybindgen.module.`**`Module`**(*name*, *docstring=None*, *cpp_namespace=None*)

Bases: `pybindgen.module.ModuleBase`

**Parameters**

- **name** – module name
- **docstring** – docstring to use for this module
- **cpp_namespace** – C++ namespace prefix associated with this module

**`generate`**(*out*, *module_file_base_name=None*)

Generates the module

**Parameters** **module_file_base_name** – base name of the module file.

This is useful when we want to produce a _foo module that will be imported into a foo module, to avoid making all types docstrings contain _foo.Xpto instead of foo.Xpto.

**`generate_c_to_python_type_converter`**(*value_type*, *code_sink*)

Generates a c-to-python converter function for a given type and returns the name of the generated function. If called multiple times with the same name only the first time is the converter function generated.

Use: this method is to be considered pybindgen internal, used by code generation modules.

**Returns** name of the converter function

**`generate_python_to_c_type_converter`**(*value_type*, *code_sink*)

Generates a python-to-c converter function for a given type and returns the name of the generated function. If called multiple times with the same name only the first time is the converter function generated.

Use: this method is to be considered pybindgen internal, used by code generation modules.

**Returns** name of the converter function

**`get_c_to_python_type_converter_function_name`**(*value_type*)

Internal API, do not use.

**`get_python_to_c_type_converter_function_name`**(*value_type*)

Internal API, do not use.

**class** `pybindgen.module.`**`ModuleBase`**(*name*, *parent=None*, *docstring=None*, *cpp_namespace=None*)

Bases: `dict`

ModuleBase objects can be indexed dictionary style to access contained types. Example:

```
>>> from enum import Enum
>>> from cppclass import CppClass
>>> m = Module("foo", cpp_namespace="foo")
>>> subm = m.add_cpp_namespace("subm")
>>> c1 = m.add_class("Bar")
>>> c2 = subm.add_class("Zbr")
>>> e1 = m.add_enum("En1", ["XX"])
>>> e2 = subm.add_enum("En2", ["XX"])
>>> m["Bar"] is c1
True
>>> m["foo::Bar"] is c1
True
>>> m["En1"] is e1
True
>>> m["foo::En1"] is e1
True
>>> m["badname"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'badname'
>>> m["foo::subm::Zbr"] is c2
True
>>> m["foo::subm::En2"] is e2
True
```

Note: this is an abstract base class, see L{Module}

**Parameters**

- **name** – module name
- **parent** – parent L{module<Module>} (i.e. the one that contains this submodule) or None if this is a root module
- **docstring** – docstring to use for this module
- **cpp_namespace** – C++ namespace prefix associated with this module

**Returns** a new module object

**add_class** (**args*, ***kwargs*)
Add a class to the module. See the documentation for L{CppClass.__init__} for information on accepted parameters.

**add_container** (**args*, ***kwargs*)
Add a container to the module. See the documentation for L{Container.__init__} for information on accepted parameters.

**add_cpp_namespace** (*name*)
Add a nested module namespace corresponding to a C++ namespace. If the requested namespace was already added, the existing module is returned instead of creating a new one.

**Parameters** **name** – name of C++ namespace (just the last component,

not full scoped name); this also becomes the name of the submodule.

**Returns** a L{SubModule} object that maps to this namespace.

**add_custom_function_wrapper** (**args*, ***kwargs*)
Add a function, using custom wrapper code, to the module/namespace. See the documentation for `pybindgen.function.CustomFunctionWrapper` for information on accepted parameters.

**add_enum** (**args*, ***kwargs*)
Add an enumeration to the module. See the documentation for L{Enum.__init__} for information on accepted parameters.

**add_exception** (**args*, ***kwargs*)
Add a C++ exception to the module. See the documentation for L{CppException.__init__} for information on accepted parameters.

**add_function** (**args*, ***kwargs*)
Add a function to the module/namespace. See the documentation for `Function.__init__()` for information on accepted parameters.

**add_include** (*include*)
Adds an additional include directive, needed to compile this python module

**Parameters** **include** – the name of the header file to include, including surrounding “” or <>.

**add_struct** (**args*, ***kwargs*)
Add a struct to the module.

In addition to the parameters accepted by L{CppClass.__init__}, this method accepts the following keyword parameters:

- no_constructor (bool): if True, the structure will not have a constructor by default (if omitted, it will be considered to have a trivial constructor).
- no_copy (bool): if True, the structure will not have a copy constructor by default (if omitted, it will be considered to have a simple copy constructor).

**add_typedef** (*wrapper*, *alias*)

**Declares an equivalent to a typedef in C::** typedef Foo Bar;

**Parameters**

- **wrapper** – the wrapper object to alias (Foo in the example)
- **alias** – name of the typedef alias

@note: only typedefs for CppClass objects have been implemented so far; others will be implemented in the future.

**begin_section** (*section_name*)

Declare that types and functions registered with the module in the future belong to the section given by that section_name parameter, until a matching end_section() is called.

**Note:** begin_section()/end_section() are silently ignored unless a MultiSectionFactory object is used as code generation output.

**current_section**

**declare_one_time_definition** (*definition_name*)

Internal helper method for code geneneration to coordinate generation of code that can only be defined once per compilation unit

(note: assuming here one-to-one mapping between 'module' and 'compilation unit').

**Parameters** **definition_name** – a string that uniquely identifies the code

definition that will be added. If the given definition was already declared KeyError is raised.

```
>>> module = Module('foo')
>>> module.declare_one_time_definition("zbr")
>>> module.declare_one_time_definition("zbr")
Traceback (most recent call last):
...
KeyError: 'zbr'
>>> module.declare_one_time_definition("bar")
```

**do_generate** (*out*, *module_file_base_name=None*)

(internal) Generates the module.

**end_section** (*section_name*)

Declare the end of a section, i.e. further types and functions will belong to the main module.

**Parameters** **section_name** – name of section; must match the one in the previous begin_section() call.

**generate_forward_declarations** (*code_sink*)

(internal) generate forward declarations for types

**get_current_section** ()

**get_module_path**()
Get the full [module, submodule, submodule,...] path

**get_name**()

**get_namespace_path**()
Get the full [root_namespace, namespace, namespace,...] path (C++)

**get_root**()

**Returns** the root `Module` (even if it is self)

**get_submodule**(*submodule_name*)
get a submodule by its name

**name**

**register_type**(*name*, *full_name*, *type_wrapper*)
Register a type wrapper with the module, for easy access in the future. Normally should not be called by the programmer, as it is meant for internal pybindgen use and called automatically.

**Parameters**

- **name** – type name without any C++ namespace prefix, or None
- **full_name** – type name with a C++ namespace prefix, or None
- **type_wrapper** – the wrapper object for the type (e.g. L{CppClass} or L{Enum})

**set_c_function_name_transformer**(*transformer*)
Sets the function to be used when transforming a C function name into the python function name; the given given function is called like this:

```
python_name = transformer(c_name)
```

**set_name**(*name*)

**set_strip_prefix**(*prefix*)
Sets the prefix string to be used when transforming a C function name into the python function name; the given prefix string is removed from the C function name.

**class** pybindgen.module.**MultiSectionFactory**
Bases: `object`

Abstract base class for objects providing support for multi-section code generation, i.e., splitting the generated C/C++ code into multiple files. The generated code will generally have the following structure:

1.For each section there is one source file specific to that section;

2. There is a I{main} source file, e.g. C{foomodule.cc}. Code that does not belong to any section will be included in this main file;

3. Finally, there is a common header file, (e.g. foomodule.h), which is included by the main file and section files alike. Typically this header file contains function prototypes and type definitions.

@see: L{Module.generate}

x.__init__(...) initializes x; see help(type(x)) for signature

**get_common_header_code_sink**()
Create and/or return a code sink for the common header.

**get_common_header_include**()
Return the argument for an #include directive to include the common header.

**Returns** a string with the header name, including surrounding

“” or <>. For example, “‘foomodule.h”’.

**get_main_code_sink**()

Create and/or return a code sink for the main file.

**get_section_code_sink**(*section_name*)

Create and/or return a code sink for a given section.

**Parameters section_name** – name of the section

**Returns** a L{CodeSink} object that will receive generated code belonging to the section C{section_name}

**class** pybindgen.module.**SubModule**(*name*, *parent*, *docstring=None*, *cpp_namespace=None*)

Bases: pybindgen.module.ModuleBase

**Parameters**

- **parent** – parent L{module<Module>} (i.e. the one that contains this submodule)
- **name** – name of the submodule
- **docstring** – docstring to use for this module
- **cpp_namespace** – C++ namespace component associated with this module

### 2.1.2 function: C/C++ function wrappers

C function wrapper

**class** pybindgen.function.**CustomFunctionWrapper**(*function_name*, *wrapper_name*, *wrapper_body=None*, *flags=('METH_VARARGS', 'METH_KEYWORDS')*)

Bases: pybindgen.function.Function

Adds a custom function wrapper. The custom wrapper must be prepared to support overloading, i.e. it must have an additional “PyObject **return_exception” parameter, and raised exceptions must be returned by this parameter.

**Parameters**

- **function_name** – name for function, Python side
- **wrapper_name** – name of the C wrapper function
- **wrapper_body** – if not None, the function wrapper is generated containing this parameter value as function body

**NEEDS_OVERLOADING_INTERFACE = True**

**generate**(*code_sink*, *dummy_wrapper_name=None*, *extra_wrapper_params=()*)

**generate_call**(**args*, ***kwargs*)

**class** pybindgen.function.**Function**(*function_name*, *return_value*, *parameters*, *docstring=None*, *unblock_threads=None*, *template_parameters=()*, *custom_name=None*, *deprecated=False*, *foreign_cpp_namespace=None*, *throw=()*)

Bases: pybindgen.typehandlers.base.ForwardWrapperBase

Class that generates a wrapper to a C function.

**Parameters**

- **function_name** – name of the C function
- **return_value** (*L{ReturnValue}*) – the function return value
- **parameters** (*list of L{Parameter}*) – the function parameters
- **custom_name** – an alternative name to give to this function at python-side; if omitted, the name of the function in the python module will be the same name as the function in C++ (minus namespace).
- **deprecated** – deprecation state for this API: - False: Not deprecated - True: Deprecated - "message": Deprecated, and deprecation warning contains the given message
- **foreign_cpp_namespace** – if set, the function is assumed to belong to the given C++ namespace, regardless of the C++ namespace of the python module it will be added to.
- **throw** (*list of L{CppException}*) – list of C++ exceptions that the function may throw

**add_custodian_and_ward**(*custodian*, *ward*, *postcall=None*)

Add a custodian/ward relationship to the function wrapper

A custodian/ward relationship is one where one object (custodian) keeps a references to another object (ward), thus keeping it alive. When the custodian is destroyed, the reference to the ward is released, allowing the ward to be freed if no other reference to it is being kept by the user code. Please note that custodian/ward manages the lifecycle of Python wrappers, not the C/C++ objects referenced by the wrappers. In most cases, the wrapper owns the C/C++ object, and so the lifecycle of the C/C++ object is also managed by this. However, there are cases when a Python wrapper does not own the underlying C/C++ object, only references it.

The custodian and ward objects are indicated by an integer with the following meaning:

- C{-1}: the return value of the function
- value > 0: the nth parameter of the function, starting at 1

**Parameters**

- **custodian** – number of the object that assumes the role of custodian
- **ward** – number of the object that assumes the role of ward
- **postcall** – if True, the relationship is added after the C function call, if False it is added before the call. If not given, the value False is assumed if the return value is not involved, else postcall=True is used.

**clone**()

Creates a semi-deep copy of this function wrapper. The returned function wrapper clone contains copies of all parameters, so they can be modified at will.

**generate**(*code_sink*, *wrapper_name=None*, *extra_wrapper_params=()*)

Generates the wrapper code

**Parameters**

- **code_sink** – a CodeSink instance that will receive the generated code
- **wrapper_name** – name of wrapper function

**generate_call**()

virtual method implementation; do not call

**generate_declaration**(*code_sink*, *extra_wrapper_parameters=()*)

**get_module**()
Get the Module object this function belongs to

**get_py_method_def**(*name*)
Returns an array element to use in a PyMethodDef table. Should only be called after code generation.

**Parameters** **name** – python function/method name

**module**
Get the Module object this function belongs to

**set_module**(*module*)
Set the Module object this function belongs to

**class** pybindgen.function.**OverloadedFunction**(*wrapper_name*)
Bases: pybindgen.overloading.OverloadedWrapper

Adds support for overloaded functions

wrapper_name – C/C++ name of the wrapper

**ERROR_RETURN = 'return NULL;'**

**RETURN_TYPE = 'PyObject *'**

### 2.1.3 enum: wrap enumrations

Wraps enumerations

**class** pybindgen.enum.**Enum**(*name*, *values*, *values_prefix=''*, *cpp_namespace=None*, *outer_class=None*, *import_from_module=None*)
Bases: object

Class that adds support for a C/C++ enum type

Creates a new enum wrapper, which should be added to a module with module.add_enum().

**Parameters**

- **name** – C name of the enum type
- **values** – a list of strings with all enumeration value names, or list of (name, C-value-expr) tuples.
- **values_prefix** – prefix to add to value names, or None
- **cpp_namespace** – optional C++ namespace identifier, or None. Note: this namespace is *in addition to* whatever namespace of the module the enum belongs to. Typically this parameter is to be used when wrapping enums declared inside C++ classes.
- **import_from_module** – if not None, the enum is defined in another module, this parameter gives the name of the module

**generate**(*unused_code_sink*)

**generate_declaration**(*sink*, *module*)

**get_module**()
Get the Module object this class belongs to

**module**
Get the Module object this class belongs to

**set_module**(*module*)
Set the Module object this class belongs to; can only be set once

### 2.1.4 cppclass: wrap C++ classes or C structures

**class** `pybindgen.cppclass.`**`BoostSharedPtr`**(*class_name*)

Bases: `pybindgen.cppclass.SmartPointerPolicy`

Create a memory policy for using boost::shared_ptr<> to manage instances of this object.

**Parameters** **class_name** – the full name of the class, e.g. foo::Bar

**`get_delete_code`**(*cpp_class*)

**`get_instance_creation_function`**()

**`get_pointer_type`**(*class_full_name*)

**`get_pystruct_init_code`**(*cpp_class*, *obj*)

**class** `pybindgen.cppclass.`**`CppClass`**(*name*, *parent=None*, *incref_method=None*, *decref_method=None*, *automatic_type_narrowing=None*, *allow_subclassing=None*, *is_singleton=False*, *outer_class=None*, *peekref_method=None*, *template_parameters=()*, *custom_template_class_name=None*, *incomplete_type=False*, *free_function=None*, *incref_function=None*, *decref_function=None*, *python_name=None*, *memory_policy=None*, *foreign_cpp_namespace=None*, *docstring=None*, *custom_name=None*, *import_from_module=None*, *destructor_visibility='public'*)

Bases: `object`

A CppClass object takes care of generating the code for wrapping a C++ class

**Parameters**

- **name** – class name
- **parent** – optional parent class wrapper, or list of parents. Valid values are None, a CppClass instance, or a list of CppClass instances.
- **incref_method** – (deprecated in favour of memory_policy) if the class supports reference counting, the name of the method that increments the reference count (may be inherited from parent if not given)
- **decref_method** – (deprecated in favour of memory_policy) if the class supports reference counting, the name of the method that decrements the reference count (may be inherited from parent if not given)
- **automatic_type_narrowing** – if True, automatic return type narrowing will be done on objects of this class and its descendants when returned by pointer from a function or method.
- **allow_subclassing** – if True, generated class wrappers will allow subclassing in Python.
- **is_singleton** – if True, the class is considered a singleton, and so the python wrapper will never call the C++ class destructor to free the value.
- **peekref_method** – (deprecated in favour of memory_policy) if the class supports reference counting, the name of the method that returns the current reference count.
- **free_function** – (deprecated in favour of memory_policy) name of C function used to deallocate class instances
- **incref_function** – (deprecated in favour of memory_policy) same as incref_method, but as a function instead of method

- **decref_function** – (deprecated in favour of memory_policy) same as decref_method, but as a function instead of method
- **python_name** – name of the class as it will appear from Python side. This parameter is DEPRECATED in favour of custom_name.
- **memory_policy** (*L{MemoryPolicy}*) – memory management policy; if None, it inherits from the parent class. Only root classes can have a memory policy defined.
- **foreign_cpp_namespace** – if set, the class is assumed to belong to the given C++ namespace, regardless of the C++ namespace of the python module it will be added to. For instance, this can be useful to wrap std classes, like std::ofstream, without having to create an extra python submodule.
- **docstring** – None or a string containing the docstring that will be generated for the class
- **custom_name** – an alternative name to give to this class at python-side; if omitted, the name of the class in the python module will be the same name as the class in C++ (minus namespace).
- **import_from_module** – if not None, the type is imported from a foreign Python module with the given name.

**`add_binary_comparison_operator`**(*operator*)

Add support for a C++ binary comparison operator, such as == or <.

The binary operator is assumed to operate with both operands of the type of the class, either by reference or by value.

**Parameters** **operator** – string indicating the name of the operator to support, e.g. '=='

**`add_binary_numeric_operator`**(*operator*, *result_cppclass=None*, *left_cppclass=None*, *right=None*)

Add support for a C++ binary numeric operator, such as +, -, *, or /.

**Parameters**

- **operator** – string indicating the name of the operator to support, e.g. '=='
- **result_cppclass** – the CppClass object of the result type, assumed to be this class if omitted
- **left_cppclass** – the CppClass object of the left operand type, assumed to be this class if omitted
- **right** – the type of the right parameter. Can be a CppClass, Parameter, or param spec. Assumed to be this class if omitted

**`add_class`**(**args*, ***kwargs*)

Add a nested class. See L{CppClass} for information about accepted parameters.

**`add_constructor`**(**args*, ***kwargs*)

Add a constructor to the class. See the documentation for L{CppConstructor.__init__} for information on accepted parameters.

**`add_container_traits`**(**args*, ***kwargs*)

**`add_copy_constructor`**()

Utility method to add a 'copy constructor' method to this class.

**`add_custom_instance_attribute`**(*name*, *value_type*, *getter*, *is_const=False*, *setter=None*, *getter_template_parameters=*[ ], *setter_template_parameters=*[ ])

**Parameters**

- **value_type** – a ReturnValue object
- **name** – attribute name (i.e. the name of the class member variable)
- **is_const** – True if the attribute is const, i.e. cannot be modified
- **getter** – None, or name of a method of this class used to get the value
- **setter** – None, or name of a method of this class used to set the value
- **getter_template_parameters** – optional list of template parameters for getter function
- **setter_template_parameters** – optional list of template parameters for setter function

**add_custom_method_wrapper**(**args*, ***kwargs*)
Adds a custom method wrapper. See L{CustomCppMethodWrapper} for more information.

**add_enum**(**args*, ***kwargs*)
Add a nested enum. See L{Enum} for information about accepted parameters.

**add_function_as_constructor**(**args*, ***kwargs*)
Wrap a function that behaves as a constructor to the class. See the documentation for L{CppFunctionAsConstructor.__init__} for information on accepted parameters.

**add_function_as_method**(**args*, ***kwargs*)
Add a function as method of the class. See the documentation for L{Function.__init__} for information on accepted parameters. TODO: explain the implicit first function parameter

**add_helper_class_hook**(*hook*)
Add a hook function to be called just prior to a helper class being generated. The hook function applies to this class and all subclasses. The hook function is called like this:

```
hook_function(helper_class)
```

**add_inplace_numeric_operator**(*operator*, *right=None*)
Add support for a C++ inplace numeric operator, such as +=, -=, *=, or /=.

**Parameters**

- **operator** – string indicating the name of the operator to support, e.g. '+='
- **right** – the type of the right parameter. Can be a CppClass, Parameter, or param spec. Assumed to be this class if omitted

**add_instance_attribute**(*name*, *value_type*, *is_const=False*, *getter=None*, *setter=None*)

**Parameters**

- **value_type** – a ReturnValue object
- **name** – attribute name (i.e. the name of the class member variable)
- **is_const** – True if the attribute is const, i.e. cannot be modified
- **getter** – None, or name of a method of this class used to get the value
- **setter** – None, or name of a method of this class used to set the value

**add_method**(**args*, ***kwargs*)
Add a method to the class. See the documentation for L{CppMethod.__init__} for information on accepted parameters.

**add_output_stream_operator**()
Add str() support based on C++ output stream operator.

Calling this method enables wrapping of an assumed to be defined operator function:

```
std::ostream & operator << (std::ostream &, MyClass const &);
```

The wrapper will be registered as an str() python operator, and will call the C++ operator function to convert the value to a string.

**add_static_attribute**(*name*, *value_type*, *is_const=False*)

**Parameters**

- **value_type** – a ReturnValue object
- **name** – attribute name (i.e. the name of the class member variable)
- **is_const** – True if the attribute is const, i.e. cannot be modified

**add_unary_numeric_operator**(*operator*, *result_cppclass=None*, *left_cppclass=None*)
Add support for a C++ unary numeric operators, currently only -.

**Parameters**

- **operator** – string indicating the name of the operator to support, e.g. '-'
- **result_cppclass** – the CppClass object of the result type, assumed to be this class if omitted
- **left_cppclass** – the CppClass object of the left operand type, assumed to be this class if omitted

**generate**(*code_sink*, *module*)
Generates the class to a code sink

**generate_forward_declarations**(*code_sink*, *module*)
Generates forward declarations for the instance and type structures.

**generate_typedef**(*module*, *alias*)
Generates the appropriate Module code to register the class with a new name in that module (typedef alias).

**get_all_implicit_conversions**()
Gets a new list of all other classes whose value can be implicitly converted to a value of this class.

```
>>> Foo = CppClass("Foo")
>>> Bar = CppClass("Bar")
>>> Zbr = CppClass("Zbr")
>>> Bar.implicitly_converts_to(Foo)
>>> Zbr.implicitly_converts_to(Bar)
>>> l = Foo.get_all_implicit_conversions()
>>> l.sort(lambda cls1, cls2: cmp(cls1.name, cls2.name))
>>> [cls.name for cls in l]
['Bar']
```

**get_all_methods**()
Returns an iterator to iterate over all methods of the class

**get_construct_name**()
Get a name usable for new %s construction, or raise CodeGenerationError if none found

**get_have_pure_virtual_methods**()
Returns True if the class has pure virtual methods with no implementation (which would mean the type is not instantiable directly, only through a helper class).

**get_helper_class**()
gets the "helper class" for this class wrapper, creating it if necessary

**get_instance_creation_function**()

**get_module**()
Get the Module object this class belongs to

**get_mro**()
Get the method resolution order (MRO) of this class.

**Returns** an iterator that gives CppClass objects, from leaf to root class

**get_post_instance_creation_function**()

**get_pystruct**()

**get_python_name**()

**get_type_narrowing_root**()
Find the root CppClass along the subtree of all parent classes that have automatic_type_narrowing=True Note: multiple inheritance not implemented

**have_pure_virtual_methods**
Returns True if the class has pure virtual methods with no implementation (which would mean the type is not instantiable directly, only through a helper class).

**have_sequence_methods**()
Determine if this object has sequence methods registered.

**implicitly_converts_to**(*other*)
Declares that values of this class can be implicitly converted to another class; corresponds to a operator AnotherClass(); special method.

**inherit_default_constructors**()
inherit the default constructors from the parentclass according to C++ language rules

**is_subclass**(*other*)
Return True if this CppClass instance represents a class that is a subclass of another class represented by the CppClasss object 'other'.

**module**
Get the Module object this class belongs to

**pystruct**

**register_alias**(*alias*)
Re-register the class with another base name, in addition to any registrations that might have already been done.

**set_cannot_be_constructed**(*reason*)

**set_helper_class_disabled**(*flag=True*)

**set_instance_creation_function**(*instance_creation_function*)
Set a custom function to be called to create instances of this class and its subclasses.

**Parameters instance_creation_function** – instance creation function; see default_instance_creation_function() for signature and example.

**set_module**(*module*)
Set the Module object this class belongs to

**set_post_instance_creation_function**(*post_instance_creation_function*)
Set a custom function to be called to add code after an instance is created (usually by the "instance creation function") and registered with the Python runtime.

**Parameters post_instance_creation_function** – post instance creation function

**wrapper_registry**

**write_allocate_pystruct**(*code_block*, *lvalue*, *wrapper_type=None*)
Generates code to allocate a python wrapper structure, using PyObject_New or PyObject_GC_New, plus some additional strcture initialization that may be needed.

**write_create_instance**(*code_block*, *lvalue*, *parameters*, *construct_type_name=None*)

**write_post_instance_creation_code**(*code_block*, *lvalue*, *parameters*, *construct_type_name=None*)

**class** pybindgen.cppclass.**CppClassParameter**(*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*)
Bases: `pybindgen.cppclass.CppClassParameterBase`

Class parameter "by-value" handler

**Parameters**

- **ctype** – C type, normally 'MyClass*'
- **name** – parameter name

**CTYPES = []**

**DIRECTIONS = [1]**

**convert_c_to_python**(*wrapper*)
Write some code before calling the Python method.

**convert_python_to_c**(*wrapper*)
parses python args to get C++ value

**cpp_class = None**

**class** pybindgen.cppclass.**CppClassParameterBase**(*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*)
Bases: `pybindgen.typehandlers.base.Parameter`

Base class for all C++ Class parameter handlers

**Parameters**

- **ctype** – C type, normally 'MyClass*'
- **name** – parameter name

**CTYPES = []**

**DIRECTIONS = [1]**

**cpp_class = None**

**class** pybindgen.cppclass.**CppClassPtrParameter**(*ctype*, *name*, *direction=1*, *transfer_ownership=None*, *custodian=None*, *is_const=False*, *null_ok=False*, *default_value=None*)
Bases: `pybindgen.cppclass.CppClassParameterBase`

Class* handlers

Type handler for a pointer-to-class parameter (MyClass*)

**Parameters**

- **ctype** – C type, normally 'MyClass*'
- **name** – parameter name

- **transfer_ownership** – if True, the callee becomes responsible for freeing the object. If False, the caller remains responsible for the object. In either case, the original object pointer is passed, not a copy. In case transfer_ownership=True, it is invalid to perform operations on the object after the call (calling any method will cause a null pointer dereference and crash the program).
- **custodian** – if given, points to an object (custodian) that keeps the python wrapper for the parameter alive. Possible values are:
  - None: no object is custodian;
  - -1: the return value object;
  - **0: the instance of the method in which** the ReturnValue is being used will become the custodian;
  - **integer > 0: parameter number, starting at 1** (i.e. not counting the self/this parameter), whose object will be used as custodian.
- **is_const** – if true, the parameter has a const attached to the leftmost
- **null_ok** – if true, None is accepted and mapped into a C NULL pointer
- **default_value** – default parameter value (as C expression string); probably, the only default value that makes sense here is probably 'NULL'.

**Note:** Only arguments which are instances of C++ classes wrapped by PyBindGen can be used as custodians.

**`CTYPES`** = []

**`DIRECTIONS`** = [1, 2, 3]

**`SUPPORTS_TRANSFORMATIONS`** = True

**`convert_c_to_python`**(*wrapper*)
foo

**`convert_python_to_c`**(*wrapper*)
parses python args to get C++ value

**`cpp_class`** = None

**class** `pybindgen.cppclass.`**`CppClassPtrReturnValue`**(*ctype*, *caller_owns_return=None*, *custodian=None*, *is_const=False*, *reference_existing_object=None*, *return_internal_reference=None*)

Bases: `pybindgen.cppclass.CppClassReturnValueBase`

Class* return handler

**Parameters**

- **ctype** – C type, normally 'MyClass*'
- **caller_owns_return** – if true, ownership of the object pointer is transferred to the caller
- **custodian** – bind the life cycle of the python wrapper for the return value object (ward) to that of the object indicated by this parameter (custodian). Possible values are:
  - None: no object is custodian;
  - **0: the instance of the method in which** the ReturnValue is being used will become the custodian;

  - **integer > 0: parameter number, starting at 1** (i.e. not counting the self/this parameter), whose object will be used as custodian.
- **reference_existing_object** – if true, ownership of the pointed-to object remains to be the caller's, but we do not make a copy. The callee gets a reference to the existing object, but is not responsible for freeing it. Note that using this memory management style is dangerous, as it exposes the Python programmer to the possibility of keeping a reference to an object that may have been deallocated in the mean time. Calling methods on such an object would lead to a memory error.
- **return_internal_reference** – like reference_existing_object, but additionally adds custodian/ward to bind the lifetime of the 'self' object (instance the method is bound to) to the lifetime of the return value.

**Note:** Only arguments which are instances of C++ classes wrapped by PyBindGen can be used as custodians.

**`CTYPES`** = []

**`SUPPORTS_TRANSFORMATIONS`** = True

**`convert_c_to_python`**(*wrapper*)
See ReturnValue.convert_c_to_python

**`convert_python_to_c`**(*wrapper*)
See ReturnValue.convert_python_to_c

**`cpp_class`** = None

**`get_c_error_return`**()
See ReturnValue.get_c_error_return

**class** `pybindgen.cppclass.`**`CppClassRefParameter`**(*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*, *default_value_type=None*)

Bases: `pybindgen.cppclass.CppClassParameterBase`

Class& handlers

**Parameters**

- **ctype** – C type, normally 'MyClass*'
- **name** – parameter name

**`CTYPES`** = []

**`DIRECTIONS`** = [1, 2, 3]

**`convert_c_to_python`**(*wrapper*)
Write some code before calling the Python method.

**`convert_python_to_c`**(*wrapper*)
parses python args to get C++ value

**`cpp_class`** = None

**class** `pybindgen.cppclass.`**`CppClassRefReturnValue`**(*ctype*, *is_const=False*, *caller_owns_return=False*, *reference_existing_object=None*, *return_internal_reference=None*)

Bases: `pybindgen.cppclass.CppClassReturnValueBase`

Class return handlers

**CTYPES** = []

**REQUIRES_ASSIGNMENT_CONSTRUCTOR** = True

**convert_c_to_python**(*wrapper*)
see ReturnValue.convert_c_to_python

**convert_python_to_c**(*wrapper*)
see ReturnValue.convert_python_to_c

**cpp_class** = None

**get_c_error_return**()
See ReturnValue.get_c_error_return

**class** pybindgen.cppclass.**CppClassReturnValue**(*ctype*, *is_const=False*)
Bases: pybindgen.cppclass.CppClassReturnValueBase

Class return handlers

override to fix the ctype parameter with namespace information

**CTYPES** = []

**REQUIRES_ASSIGNMENT_CONSTRUCTOR** = True

**convert_c_to_python**(*wrapper*)
see ReturnValue.convert_c_to_python

**convert_python_to_c**(*wrapper*)
see ReturnValue.convert_python_to_c

**cpp_class** = None

**get_c_error_return**()
See ReturnValue.get_c_error_return

**class** pybindgen.cppclass.**CppClassReturnValueBase**(*ctype*, *is_const=False*)
Bases: pybindgen.typehandlers.base.ReturnValue

Class return handlers – base class

**CTYPES** = []

**cpp_class** = None

**class** pybindgen.cppclass.**CppClassSharedPtrParameter**(*ctype*, *name*, *direction=1*, *is_const=False*, *null_ok=False*, *default_value=None*)
Bases: pybindgen.cppclass.CppClassParameterBase

Class* handlers

Type handler for a pointer-to-class parameter (MyClass*)

**Parameters**

- **ctype** – C type, normally ‘MyClass*’
- **name** – parameter name
- **is_const** – if true, the parameter has a const attached to the leftmost
- **null_ok** – if true, None is accepted and mapped into a C NULL pointer
- **default_value** – default parameter value (as C expression string); probably, the only default value that makes sense here is probably ‘NULL’.

**Note:** Only arguments which are instances of C++ classes wrapped by PyBindGen can be used as custodians.

**CTYPES = []**

**DIRECTIONS = [1, 2, 3]**

**SUPPORTS_TRANSFORMATIONS = False**

**convert_c_to_python**(*wrapper*)
foo

**convert_python_to_c**(*wrapper*)
parses python args to get C++ value

**cpp_class = None**

**class** pybindgen.cppclass.**CppClassSharedPtrReturnValue**(*ctype*, *is_const=False*)
Bases: pybindgen.cppclass.CppClassReturnValueBase

Class* return handler

**Parameters ctype** – C type, normally 'MyClass*'

**CTYPES = []**

**SUPPORTS_TRANSFORMATIONS = True**

**convert_c_to_python**(*wrapper*)
See ReturnValue.convert_c_to_python

**convert_python_to_c**(*wrapper*)
See ReturnValue.convert_python_to_c

**cpp_class = None**

**get_c_error_return**()
See ReturnValue.get_c_error_return

**class** pybindgen.cppclass.**CppHelperClass**(*class_*)
Bases: object

Generates code for a C++ proxy subclass that takes care of forwarding virtual methods from C++ to Python.

**Parameters class** – original CppClass wrapper object

**add_custom_method**(*declaration*, *body=None*)
Add a custom method to the helper class, given by a declaration line and a body. The body can be None, in case the whole method definition is included in the declaration itself.

**add_post_generation_code**(*code*)
Add custom code to be included right after the helper class is generated.

**add_virtual_method**(*method*)

**add_virtual_parent_caller**(*parent_caller*)
Add a new CppVirtualMethodParentCaller object to this helper class

**add_virtual_proxy**(*virtual_proxy*)
Add a new CppVirtualMethodProxy object to this class

**generate**(*code_sink*)
Generate the proxy class (virtual method bodies only) to a given code sink. returns pymethodef list of parent callers

**generate_forward_declarations**(*code_sink_param*)
Generate the proxy class (declaration only) to a given code sink

**class** pybindgen.cppclass.**FreeFunctionPolicy**(*free_function*)
Bases: pybindgen.cppclass.MemoryPolicy

**get_delete_code**(*cpp_class*)

**class** pybindgen.cppclass.**MemoryPolicy**
Bases: object

memory management policy for a C++ class or C/C++ struct

**get_delete_code**(*cpp_class*)

**get_free_code**(*object_expression*)
Return a code statement to free an underlying C/C++ object.

**get_instance_creation_function**()

**get_pointer_type**(*class_full_name*)

**get_pystruct_init_code**(*cpp_class*, *obj*)

**class** pybindgen.cppclass.**ReferenceCountingFunctionsPolicy**(*incref_function*, *decref_function*, *peekref_function=None*)
Bases: pybindgen.cppclass.ReferenceCountingPolicy

**get_delete_code**(*cpp_class*)

**write_decref**(*code_block*, *obj_expr*)

**write_incref**(*code_block*, *obj_expr*)

**class** pybindgen.cppclass.**ReferenceCountingMethodsPolicy**(*incref_method*, *decref_method*, *peekref_method=None*)
Bases: pybindgen.cppclass.ReferenceCountingPolicy

**get_delete_code**(*cpp_class*)

**write_decref**(*code_block*, *obj_expr*)

**write_incref**(*code_block*, *obj_expr*)

**class** pybindgen.cppclass.**ReferenceCountingPolicy**
Bases: pybindgen.cppclass.MemoryPolicy

**write_decref**(*code_block*, *obj_expr*)
Write code to decrease the reference code of an object of this class (the real C++ class, not the wrapper). Should only be called if the class supports reference counting, as reported by the attribute *CppClass.has_reference_counting*.

**write_incref**(*code_block*, *obj_expr*)
Write code to increase the reference code of an object of this class (the real C++ class, not the wrapper). Should only be called if the class supports reference counting, as reported by the attribute *CppClass.has_reference_counting*.

**class** pybindgen.cppclass.**SmartPointerPolicy**
Bases: pybindgen.cppclass.MemoryPolicy

**pointer_name = None**

`pybindgen.cppclass.`**`boost_shared_ptr_instance_creation_function`**(*cpp_class*, *code_block*, *lvalue*, *parameters*, *construct_type_name*)

boost::shared_ptr "instance creation function"; it is called whenever a new C++ class instance needs to be created

**Parameters**

- **cpp_class** – the CppClass object whose instance is to be created
- **code_block** – CodeBlock object on which the instance creation code should be generated
- **lvalue** – lvalue expression that should hold the result in the end
- **parameters** – stringified list of parameters
- **construct_type_name** – actual name of type to be constructed (it is not always the class name, sometimes it's the python helper class)

`pybindgen.cppclass.`**`common_shared_object_return`**(*value*, *py_name*, *cpp_class*, *code_block*, *type_traits*, *caller_owns_return*, *reference_existing_object*, *type_is_pointer*)

`pybindgen.cppclass.`**`default_instance_creation_function`**(*cpp_class*, *code_block*, *lvalue*, *parameters*, *construct_type_name*)

Default "instance creation function"; it is called whenever a new C++ class instance needs to be created; this default implementation uses a standard C++ new allocator.

**Parameters**

- **cpp_class** – the CppClass object whose instance is to be created
- **code_block** – CodeBlock object on which the instance creation code should be generated
- **lvalue** – lvalue expression that should hold the result in the end
- **parameters** – stringified list of parameters
- **construct_type_name** – actual name of type to be constructed (it is not always the class name, sometimes it's the python helper class)

`pybindgen.cppclass.`**`get_c_to_python_converter`**(*value*, *root_module*, *code_sink*)

`pybindgen.cppclass.`**`get_python_to_c_converter`**(*value*, *root_module*, *code_sink*)

`pybindgen.cppclass.`**`implement_parameter_custodians_postcall`**(*wrapper*)

`pybindgen.cppclass.`**`implement_parameter_custodians_precall`**(*wrapper*)

`pybindgen.cppclass.`**`scan_custodians_and_wards`**(*wrapper*)

Scans the return value and parameters for custodian/ward options, converts them to add_custodian_and_ward API calls. Wrappers that implement custodian_and_ward are: CppMethod, Function, and CppConstructor.

### 2.1.5 cppmethod: wrap class methods and constructors

Wrap C++ class methods and constructods.

**class** `pybindgen.cppmethod.`**`CppConstructor`**(*parameters*, *unblock_threads=None*, *visibility='public'*, *deprecated=False*, *throw=()*)

Bases: `pybindgen.typehandlers.base.ForwardWrapperBase`

Class that generates a wrapper to a C++ class constructor. Such wrapper is used as the python class __init__ method.

**Parameters**

- **parameters** – the constructor parameters
- **deprecated** – deprecation state for this API: False=Not deprecated; True=Deprecated; "message"=Deprecated, and deprecation warning contains the given message
- **throw** (list of `pybindgen.cppexception.CppException`) – list of C++ exceptions that the constructor may throw

**add_custodian_and_ward**(*custodian*, *ward*, *postcall=None*)

Add a custodian/ward relationship to the constructor wrapper

A custodian/ward relationship is one where one object (custodian) keeps a references to another object (ward), thus keeping it alive. When the custodian is destroyed, the reference to the ward is released, allowing the ward to be freed if no other reference to it is being kept by the user code. Please note that custodian/ward manages the lifecycle of Python wrappers, not the C/C++ objects referenced by the wrappers. In most cases, the wrapper owns the C/C++ object, and so the lifecycle of the C/C++ object is also managed by this. However, there are cases when a Python wrapper does not own the underlying C/C++ object, only references it.

The custodian and ward objects are indicated by an integer with the following meaning:

- C{0}: the object being constructed (self)
- value > 0: the nth parameter of the function, starting at 1

**Parameters**

- **custodian** – number of the object that assumes the role of custodian
- **ward** – number of the object that assumes the role of ward
- **postcall** – if True, the relationship is added after the C function call, if False it is added before the call. If not given, the value False is assumed if the return value is not involved, else postcall=True is used.

**class_**

Get the class wrapper object (CppClass)

**clone**()

Creates a semi-deep copy of this constructor wrapper. The returned constructor wrapper clone contains copies of all parameters, so they can be modified at will.

**generate**(*code_sink*, *wrapper_name=None*, *extra_wrapper_params=()*)

Generates the wrapper code :param code_sink: a CodeSink instance that will receive the generated code :returns: the wrapper function name.

**generate_call**(*class_=None*)

virtual method implementation; do not call

**get_class**()

Get the class wrapper object (CppClass)

**set_class**(*class_*)

Set the class wrapper object (CppClass)

**class** pybindgen.cppmethod.**CppDummyMethod**(*method_name*, *return_value*, *parameters*, **args*, ***kwargs*)

Bases: `pybindgen.cppmethod.CppMethod`

A 'dummy' method; cannot be generated due to incomple or incorrect parameters, but is added to the class to model the missing method.

**class** `pybindgen.cppmethod.`**`CppFunctionAsConstructor`**(*c_function_name*, *return_value*, *parameters*, *unblock_threads=None*)

Bases: `pybindgen.cppmethod.CppConstructor`

Class that generates a wrapper to a C/C++ function that appears as a contructor.

**Parameters**

- **c_function_name** – name of the C/C++ function; FIXME: for now it is implied that this function returns a pointer to the a class instance with caller_owns_return=True semantics.
- **return_value** (*L{ReturnValue}*) – function return value type
- **parameters** (*list of L{Parameter}*) – the function/constructor parameters

**`generate_call`**(*class_=None*)

virtual method implementation; do not call

**class** `pybindgen.cppmethod.`**`CppMethod`**(*method_name*, *return_value*, *parameters*, *is_static=False*, *template_parameters=()*, *is_virtual=None*, *is_const=False*, *unblock_threads=None*, *is_pure_virtual=False*, *custom_template_method_name=None*, *visibility='public'*, *custom_name=None*, *deprecated=False*, *docstring=None*, *throw=()*)

Bases: `pybindgen.typehandlers.base.ForwardWrapperBase`

Class that generates a wrapper to a C++ class method

Create an object the generates code to wrap a C++ class method.

**Parameters**

- **return_value** (*L{ReturnValue}*) – the method return value
- **method_name** – name of the method
- **parameters** (list of `pybindgen.typehandlers.base.Parameter`) – the method parameters
- **is_static** – whether it is a static method
- **template_parameters** (*list of strings, each element a template parameter expression*) – optional list of template parameters needed to invoke the method
- **is_virtual** – whether the method is virtual (pure or not)
- **is_const** – whether the method has a const modifier on it
- **unblock_threads** – whether to release the Python GIL around the method call or not. If None or omitted, use global settings. Releasing the GIL has a small performance penalty, but is recommended if the method is expected to take considerable time to complete, because otherwise no other Python thread is allowed to run until the method completes.
- **is_pure_virtual** – whether the method is defined as "pure virtual", i.e. virtual method with no default implementation in the class being wrapped.
- **custom_name** – alternate name to give to the method, in python side.
- **custom_template_method_name** – (deprecated) same as parameter 'custom_name'.
- **visibility** (*a string (allowed values are 'public', 'protected', 'private')*) – visibility of the method within the C++ class

- **deprecated** – deprecation state for this API: - False: Not deprecated - True: Deprecated - "message": Deprecated, and deprecation warning contains the given message
- **throw** (*list of L{CppException}*) – list of C++ exceptions that the function may throw

**add_custodian_and_ward**(*custodian*, *ward*, *postcall=None*)

Add a custodian/ward relationship to the method wrapper

A custodian/ward relationship is one where one object (custodian) keeps a references to another object (ward), thus keeping it alive. When the custodian is destroyed, the reference to the ward is released, allowing the ward to be freed if no other reference to it is being kept by the user code. Please note that custodian/ward manages the lifecycle of Python wrappers, not the C/C++ objects referenced by the wrappers. In most cases, the wrapper owns the C/C++ object, and so the lifecycle of the C/C++ object is also managed by this. However, there are cases when a Python wrapper does not own the underlying C/C++ object, only references it.

The custodian and ward objects are indicated by an integer with the following meaning:

- C{-1}: the return value of the function
- C{0}: the instance of the method (self)
- value > 0: the nth parameter of the function, starting at 1

**Parameters**

- **custodian** – number of the object that assumes the role of custodian
- **ward** – number of the object that assumes the role of ward
- **postcall** – if True, the relationship is added after the C function call, if False it is added before the call. If not given, the value False is assumed if the return value is not involved, else postcall=True is used.

**class_**

get the class object this method belongs to

**clone**()

Creates a semi-deep copy of this method wrapper. The returned method wrapper clone contains copies of all parameters, so they can be modified at will.

**custom_name**

**generate**(*code_sink*, *wrapper_name=None*, *extra_wrapper_params=()*)

Generates the wrapper code code_sink – a CodeSink instance that will receive the generated code method_name – actual name the method will get extra_wrapper_params – extra parameters the wrapper function should receive

Returns the corresponding PyMethodDef entry string.

**generate_call**(*class_=None*)

virtual method implementation; do not call

**get_class**()

get the class object this method belongs to

**get_helper_class**()

Get the C++ helper class, which is used for overriding virtual methods

**get_py_method_def**(*method_name*)

Get the PyMethodDef entry suitable for this method

**get_py_method_def_flags**()
Get the PyMethodDef flags suitable for this method

**get_wrapper_signature**(*wrapper_name*, *extra_wrapper_params=()*)

**helper_class**
Get the C++ helper class, which is used for overriding virtual methods

**matches_signature**(*other*)

**set_class**(*class_*)
set the class object this method belongs to

**set_custom_name**(*custom_name*)

**set_helper_class**(*helper_class*)
Set the C++ helper class, which is used for overriding virtual methods

**class** pybindgen.cppmethod.**CppNoConstructor**(*reason*)
Bases: pybindgen.typehandlers.base.ForwardWrapperBase

Class that generates a constructor that raises an exception saying that the class has no constructor.

**Parameters** **reason** – string indicating reason why the class cannot be constructed.

**generate**(*code_sink*, *class_*)
Generates the wrapper code

**Parameters**

- **code_sink** – a CodeSink instance that will receive the generated code
- **class** – the c++ class wrapper the method belongs to

Returns the wrapper function name.

**generate_call**()
dummy method, not really called

**class** pybindgen.cppmethod.**CppOverloadedConstructor**(*wrapper_name*)
Bases: pybindgen.overloading.OverloadedWrapper

Support class for overloaded constructors

wrapper_name – C/C++ name of the wrapper

**ERROR_RETURN = 'return -1;'**

**RETURN_TYPE = 'int'**

**class** pybindgen.cppmethod.**CppOverloadedMethod**(*wrapper_name*)
Bases: pybindgen.overloading.OverloadedWrapper

Support class for overloaded methods

wrapper_name – C/C++ name of the wrapper

**ERROR_RETURN = 'return NULL;'**

**RETURN_TYPE = 'PyObject *'**

**class** pybindgen.cppmethod.**CppVirtualMethodParentCaller**(*method*, *unblock_threads=None*)
Bases: pybindgen.cppmethod.CppMethod

Class that generates a wrapper that calls a virtual method default implementation in a parent base class.

**class_**

**clone**()
Creates a semi-deep copy of this method wrapper. The returned method wrapper clone contains copies of all parameters, so they can be modified at will.

**generate_call**(*class_=None*)
virtual method implementation; do not call

**generate_class_declaration**(*code_sink*, *extra_wrapper_parameters=()*)

**generate_declaration**(*code_sink*, *extra_wrapper_parameters=()*)

**generate_parent_caller_method**(*code_sink*)

**get_class**()

**get_py_method_def**(*method_name=None*)
Get the PyMethodDef entry suitable for this method

**class** pybindgen.cppmethod.**CppVirtualMethodProxy**(*method*)
Bases: pybindgen.typehandlers.base.ReverseWrapperBase

Class that generates a proxy virtual method that calls a similarly named python method.

**class_**
Get the class wrapper object (CppClass)

**generate**(*code_sink*)
generates the proxy virtual method

**generate_declaration**(*code_sink*)

**generate_python_call**()
code to call the python method

**get_class**()
Get the class wrapper object (CppClass)

**get_helper_class**()
Get the C++ helper class, which is used for overriding virtual methods

**helper_class**
Get the C++ helper class, which is used for overriding virtual methods

**set_helper_class**(*helper_class*)
Set the C++ helper class, which is used for overriding virtual methods

**class** pybindgen.cppmethod.**CustomCppConstructorWrapper**(*wrapper_name*, *wrapper_body*)
Bases: pybindgen.cppmethod.CppConstructor

Adds a custom constructor wrapper. The custom wrapper must be prepared to support overloading, i.e. it must have an additional "PyObject **return_exception" parameter, and raised exceptions must be returned by this parameter.

**NEEDS_OVERLOADING_INTERFACE = True**

**generate**(*code_sink*, *dummy_wrapper_name=None*, *extra_wrapper_params=()*)

**generate_call**(**args*, ***kwargs*)

**class** pybindgen.cppmethod.**CustomCppMethodWrapper**(*method_name*, *wrapper_name*, *wrapper_body=None*, *flags=('METH_VARARGS', 'METH_KEYWORDS')*)
Bases: pybindgen.cppmethod.CppMethod

Adds a custom method wrapper. The custom wrapper must be prepared to support overloading, i.e. it must have an additional "PyObject **return_exception" parameter, and raised exceptions must be returned by this parameter.

**NEEDS_OVERLOADING_INTERFACE = True**

**generate** (*code_sink*, *dummy_wrapper_name=None*, *extra_wrapper_params=()*)

**generate_call** (**args*, ***kwargs*)

**generate_declaration** (*code_sink*, *extra_wrapper_parameters=()*)

**class** pybindgen.cppmethod.**DummyParameter** (*arg*)

Bases: pybindgen.typehandlers.base.Parameter

Accepts either a Parameter object or a tuple as sole parameter. In case it's a tuple, it is assumed to be a retval spec (*args, **kwargs).

**CTYPES = []**

**DIRECTIONS = [1, 2, 3]**

A 'dummy' parameter object used for modelling methods that have incomplete or incorrect parameters or return values.

**convert_c_to_python** (*wrapper*)

**convert_python_to_c** (*wrapper*)

**class** pybindgen.cppmethod.**DummyReturnValue** (*arg*)

Bases: pybindgen.typehandlers.base.ReturnValue

Accepts either a ReturnValue object or a tuple as sole parameter. In case it's a tuple, it is assumed to be a retval spec (*args, **kwargs).

**CTYPES = []**

A 'dummy' return value object used for modelling methods that have incomplete or incorrect parameters or return values.

**convert_c_to_python** (*wrapper*)

**convert_python_to_c** (*wrapper*)

**get_c_error_return** ()

### 2.1.6 cppattribute: wrap class/instance attributes

Wraps C++ class instance/static attributes.

**class** pybindgen.cppattribute.**CppInstanceAttributeGetter** (*value_type*, *class_*, *attribute_name*, *getter=None*)

Bases: pybindgen.cppattribute.PyGetter

A getter for a C++ instance attribute.

**Parameters**

- **value_type** – a ReturnValue object handling the value type;
- **class** – the class (CppClass object)
- **attribute_name** – name of attribute
- **getter** – None, or name of a method of the class used to get the value

**generate** (*code_sink*)

**Parameters** **code_sink** – a CodeSink instance that will receive the generated code

**generate_call**()
virtual method implementation; do not call

**class** pybindgen.cppattribute.**CppInstanceAttributeSetter**(*value_type*, *class_*, *attribute_name*, *setter=None*)
Bases: pybindgen.cppattribute.PySetter

A setter for a C++ instance attribute.

**Parameters**

- **value_type** – a ReturnValue object handling the value type;
- **class** – the class (CppClass object)
- **attribute_name** – name of attribute
- **setter** – None, or name of a method of the class used to set the value

**generate**(*code_sink*)

**Parameters** **code_sink** – a CodeSink instance that will receive the generated code

**class** pybindgen.cppattribute.**CppStaticAttributeGetter**(*value_type*, *class_*, *attribute_name*)
Bases: pybindgen.cppattribute.PyGetter

A getter for a C++ class static attribute.

**Parameters**

- **value_type** – a ReturnValue object handling the value type;
- **c_value_expression** – C value expression

**generate**(*code_sink*)

**Parameters** **code_sink** – a CodeSink instance that will receive the generated code

**generate_call**()
virtual method implementation; do not call

**class** pybindgen.cppattribute.**CppStaticAttributeSetter**(*value_type*, *class_*, *attribute_name*)
Bases: pybindgen.cppattribute.PySetter

A setter for a C++ class static attribute.

**Parameters**

- **value_type** – a ReturnValue object handling the value type;
- **class** – the class (CppClass object)
- **attribute_name** – name of attribute

**generate**(*code_sink*)

**Parameters** **code_sink** – a CodeSink instance that will receive the generated code

**class** pybindgen.cppattribute.**PyGetSetDef**(*cname*)
Bases: object

Class that generates a PyGetSet table

**Parameters** **cname** – C name of the getset table

**add_attribute**(*name*, *getter*, *setter*)
Add a new attribute :param name: attribute name :param getter: a PyGetter object, or None :param setter: a PySetter object, or None

**empty**()

**generate**(*code_sink*)
Generate the getset table, return the table C name or '0' if the table is empty

**class** pybindgen.cppattribute.**PyGetter**(*return_value*, *parameters*, *parse_error_return*, *error_return*, *force_parse=None*, *no_c_retval=False*, *unblock_threads=False*)
Bases: pybindgen.typehandlers.base.ForwardWrapperBase

generates a getter, for use in a PyGetSetDef table

Base constructor

**Parameters**

- **return_value** – type handler for the return value
- **parameters** – a list of type handlers for the parameters
- **parse_error_return** – statement to return an error during parameter parsing
- **error_return** – statement to return an error after parameter parsing
- **force_parse** – force generation of code to parse parameters even if there are none
- **no_c_retval** – force the wrapper to not have a C return value
- **unblock_threads** – generate code to unblock python threads during the C function call

**generate**(*code_sink*)
Generate the code of the getter to the given code sink

**generate_call**()
(not actually called)

**class** pybindgen.cppattribute.**PyMetaclass**(*name*, *parent_metaclass_expr*, *getsets=None*)
Bases: object

Class that generates a Python metaclass

**Parameters**

- **name** – name of the metaclass (should normally end with Meta)
- **parent_metaclass_expr** – C expression that should give a pointer to the parent metaclass (should have a C type of PyTypeObject*)
- **getsets** – name of a PyGetSetDef C array variable, or None

**generate**(*code_sink*, *module*)
Generate the metaclass to code_sink and register it in the module.

**class** pybindgen.cppattribute.**PySetter**(*return_value*, *parameters*, *error_return=None*)
Bases: pybindgen.typehandlers.base.ReverseWrapperBase

generates a setter, for use in a PyGetSetDef table

Base constructor

**Parameters**

- **return_value** – type handler for the return value

- **parameters** – a list of type handlers for the parameters

**NO_GIL_LOCKING = True**

**generate** (*code_sink*)
Generate the code of the setter to the given code sink

**generate_python_call** ()
(not actually called)

### 2.1.7 cppexception: translate C++ exceptions into Python

**class** pybindgen.cppexception.**CppException** (*name*, *parent=None*, *outer_class=None*, *custom_name=None*, *foreign_cpp_namespace=None*, *message_rvalue=None*)

Bases: object

**Parameters**

- **name** – exception class name
- **parent** – optional parent class wrapper
- **custom_name** – an alternative name to give to this exception class at python-side; if omitted, the name of the class in the python module will be the same name as the class in C++ (minus namespace).
- **foreign_cpp_namespace** – if set, the class is assumed to belong to the given C++ namespace, regardless of the C++ namespace of the python module it will be added to. For instance, this can be useful to wrap std classes, like std::ofstream, without having to create an extra python submodule.
- **message_rvalue** – if not None, this parameter is a string that contains an rvalue C expression that evaluates to the exception message. The Python % operator will be used to substitute %(EXC)s for the caught exception variable name. The rvalue expression must return a string of type "char const*", a pointer owned by the exception instance.

**generate** (*code_sink*, *module*, *docstring=None*)
Generates the class to a code sink

**generate_forward_declarations** (*code_sink*, *dummy_module*)

**get_module** ()
Get the Module object this type belongs to

**module**
Get the Module object this type belongs to

**python_full_name**

**python_name**

**set_module** (*module*)
Set the Module object this type belongs to

**write_convert_to_python** (*code_block*, *variable_name*)

### 2.1.8 container: wrap STL containers

Wrap C++ STL containers

**class** `pybindgen.container.`**`Container`**(*name*, *value_type*, *container_type*, *outer_class=None*, *custom_name=None*)

Bases: `object`

**Parameters**

- **name** – C++ type name of the container, e.g. std::vector<int> or MyIntList
- **value_type** – a ReturnValue of the element type: note, for mapping containers, value_type is a tuple with two ReturnValue's: (key, element).
- **container_type** – a string with the type of container, one of 'list', 'deque', 'queue', 'priority_queue', 'vector', 'stack', 'set', 'multiset', 'hash_set', 'hash_multiset', 'map'
- **outer_class** (*None or L{CppClass}*) – if the type is defined inside a class, must be a reference to the outer class
- **custom_name** – alternative name to register with in the Python module

**`generate`**(*code_sink*, *module*, *docstring=None*)

Generates the class to a code sink

**`generate_forward_declarations`**(*code_sink*, *module*)

Generates forward declarations for the instance and type structures.

**`get_iter_pystruct`**()

**`get_module`**()

Get the Module object this type belongs to

**`get_pystruct`**()

**`iter_pystruct`**

**`module`**

Get the Module object this type belongs to

**`pystruct`**

**`python_full_name`**

**`python_name`**

**`register_alias`**(*alias*)

Re-register the class with another base name, in addition to any registrations that might have already been done.

**`set_module`**(*module*)

Set the Module object this type belongs to

**class** `pybindgen.container.`**`ContainerParameter`**(*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*)

Bases: `pybindgen.container.ContainerParameterBase`

Container handlers

ctype – C type, normally 'MyClass*' name – parameter name

**`CTYPES`** = []

**`DIRECTIONS`** = [1]

**`container_type`** = <pybindgen.Container None>

**`convert_c_to_python`**(*wrapper*)

Write some code before calling the Python method.

**convert_python_to_c** (*wrapper*)
parses python args to get C++ value

**class** pybindgen.container.**ContainerParameterBase** (*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*)
Bases: `pybindgen.typehandlers.base.Parameter`

Base class for all C++ Class parameter handlers

ctype – C type, normally 'MyClass*' name – parameter name

**CTYPES = []**

**DIRECTIONS = [1]**

**container_type = <pybindgen.Container None>**

**class** pybindgen.container.**ContainerPtrParameter** (*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*, *transfer_ownership=None*)
Bases: `pybindgen.container.ContainerParameterBase`

Container handlers

**CTYPES = []**

**DIRECTIONS = [1, 2, 3]**

**container_type = <pybindgen.Container None>**

**convert_c_to_python** (*wrapper*)
Write some code before calling the Python method.

**convert_python_to_c** (*wrapper*)
parses python args to get C++ value

**class** pybindgen.container.**ContainerRefParameter** (*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*)
Bases: `pybindgen.container.ContainerParameterBase`

Container handlers

ctype – C type, normally 'MyClass*' name – parameter name

**CTYPES = []**

**DIRECTIONS = [1, 2, 3]**

**container_type = <pybindgen.Container None>**

**convert_c_to_python** (*wrapper*)
Write some code before calling the Python method.

**convert_python_to_c** (*wrapper*)
parses python args to get C++ value

**class** pybindgen.container.**ContainerReturnValue** (*ctype*, *is_const=False*)
Bases: `pybindgen.container.ContainerReturnValueBase`

Container type return handlers

override to fix the ctype parameter with namespace information

**CTYPES = []**

**container_type = <pybindgen.Container None>**

**convert_c_to_python** (*wrapper*)
see ReturnValue.convert_c_to_python

**convert_python_to_c** (*wrapper*)
see ReturnValue.convert_python_to_c

**get_c_error_return** ()
See ReturnValue.get_c_error_return

**class** pybindgen.container.**ContainerReturnValueBase** (*ctype*)
Bases: pybindgen.typehandlers.base.ReturnValue

Class return handlers – base class

**CTYPES = []**

**container_type = <pybindgen.Container None>**

**class** pybindgen.container.**ContainerTraits** (*add_value_method*, *is_mapping=False*)
Bases: object

**class** pybindgen.container.**IterNextWrapper** (*container*)
Bases: pybindgen.typehandlers.base.ForwardWrapperBase

tp_iternext wrapper

value_type – a ReturnValue object handling the value type; container – the L{Container}

**HAVE_RETURN_VALUE = True**

**generate** (*code_sink*)
code_sink – a CodeSink instance that will receive the generated code

**generate_call** ()

**reset_code_generation_state** ()

### 2.1.9 gccxmlparser: scan header files to extract API definitions

### 2.1.10 settings: pybindgen global settings

**class** pybindgen.settings.**ErrorHandler**
Bases: object

x.__init__(...) initializes x; see help(type(x)) for signature

**handle_error** (*wrapper*, *exception*, *traceback_*)
Handles a code generation error. Should return True to tell pybindgen to ignore the error and move on to the next wrapper. Returning False will cause pybindgen to allow the exception to propagate, thus aborting the code generation procedure.

pybindgen.settings.**allow_subclassing = False**
Allow generated classes to be subclassed by default.

pybindgen.settings.**automatic_type_narrowing = False**
Default value for the automatic_type_narrowing parameter of C++ classes.

pybindgen.settings.**deprecated_virtuals = None**
Prior to PyBindGen version 0.14, the code generated to handle C++ virtual methods required Python user code to define a _foo method in order to implement the virtual method foo. Since 0.14, PyBindGen changed so that virtual method foo is implemented in Python by defining a method foo, i.e. no underscore prefix is needed

anymore. Setting deprecated_virtuals to True will force the old virtual method behaviour. But this is really deprecated; newer code should set deprecated_virtuals to False.

`pybindgen.settings.`**`error_handler`** **= None**

Custom error handling. Error handler, or None. When it is None, code generation exceptions propagate to the caller. Else it can be a `pybindgen.settings.ErrorHandler` subclass instance that handles the error.

`pybindgen.settings.`**`gcc_rtti_abi_complete`** **= True**

If True, and GCC >= 3 is detected at compile time, pybindgen will try to use abi::__si_class_type_info to determine the closest registered type for pointers to objects of unknown type. Notably, Mac OS X Lion has GCC > 3 but which breaks this internal API, in which case it should be disabled (set this option to False).

`pybindgen.settings.`**`name_prefix`** **= ''**

Prefix applied to global declarations, such as instance and type structures.

`pybindgen.settings.`**`unblock_threads`** **= False**

Generate code to support threads. When True, by default methods/functions/constructors will unblock threads around the funcion call, i.e. allows other Python threads to run during the call.

`pybindgen.settings.`**`wrapper_registry`**

A `WrapperRegistry` subclass to use for creating wrapper registries. A wrapper registry ensures that at most one python wrapper exists for each C/C++ object.

alias of `NullWrapperRegistry`

`pybindgen.settings.`**`wrapper_registry`**

A `pybindgen.wrapper_registry.WrapperRegistry` subclass to use for creating wrapper registries. A wrapper registry ensures that at most one python wrapper exists for each C/C++ object.

**class** `pybindgen.wrapper_registry.`**`WrapperRegistry`**(*base_name*)

Bases: `object`

Abstract base class for wrapepr registries.

**`generate`**(*code_sink*, *module*, *import_from_module*)

**`generate_forward_declarations`**(*code_sink*, *module*)

**`write_lookup_wrapper`**(*code_block*, *wrapper_type*, *wrapper_lvalue*, *object_rvalue*)

**`write_register_new_wrapper`**(*code_block*, *wrapper_lvalue*, *object_rvalue*)

**`write_unregister_wrapper`**(*code_block*, *wrapper_lvalue*, *object_rvalue*)

**class** `pybindgen.settings.`**`NullWrapperRegistry`**(*base_name*)

Bases: `pybindgen.wrapper_registry.WrapperRegistry`

A 'null' wrapper registry class. It produces no code, and does not guarantee that more than one wrapper cannot be created for each object. Use this class to disable wrapper registries entirely.

**class** `pybindgen.settings.`**`StdMapWrapperRegistry`**(*base_name*)

Bases: `pybindgen.wrapper_registry.WrapperRegistry`

A wrapper registry that uses std::map as implementation. Do not use this if generating pure C wrapping code, else the code will not compile.

## 2.2 Lower layers

### 2.2.1 utils: internal utilities

**exception** `pybindgen.utils.`**`SkipWrapper`**

Bases: `exceptions.Exception`

Exception that is raised to signal a wrapper failed to generate but must simply be skipped. for internal pybindgen use

x.__init__(...) initializes x; see help(type(x)) for signature

`pybindgen.utils.`**`ascii`**(*str_or_unicode_or_None*) → str_or_None

Make sure the value is either str or unicode object, and if it is unicode convert it to ascii. Also, None is an accepted value, and returns itself.

`pybindgen.utils.`**`call_with_error_handling`**(*callback*, *args*, *kwargs*, *wrapper*, *exceptions_to_handle=(<class 'pybindgen.typehandlers.base.TypeConfigurationError'>, <class 'pybindgen.typehandlers.base.CodeGenerationError'>, <class 'pybindgen.typehandlers.base.NotSupportedError'>)*)

for internal pybindgen use

`pybindgen.utils.`**`eval_param`**(*param_value*, *wrapper=None*)

`pybindgen.utils.`**`eval_retval`**(*retval_value*, *wrapper=None*)

`pybindgen.utils.`**`get_mangled_name`**(*base_name*, *template_args*)

for internal pybindgen use

`pybindgen.utils.`**`mangle_name`**(*name*)

make a name Like<This,and,That> look Like__lt__This_and_That__gt__

`pybindgen.utils.`**`param`**(**args*, ***kwargs*)

Simplified syntax for representing a parameter with delayed lookup.

Parameters are the same as L{Parameter.new}.

`pybindgen.utils.`**`parse_param_spec`**(*param_spec*)

`pybindgen.utils.`**`parse_retval_spec`**(*retval_spec*)

`pybindgen.utils.`**`retval`**(**args*, ***kwargs*)

Simplified syntax for representing a return value with delayed lookup.

Parameters are the same as L{ReturnValue.new}.

`pybindgen.utils.`**`write_preamble`**(*code_sink*, *min_python_version=None*)

Write a preamble, containing includes, #define's and typedef's necessary to correctly compile the code with the given minimum python version.

### 2.2.2 typehandlers.base: abstract base classes for type handlers and wrapper generators

Base classes for all parameter/return type handlers, and base interfaces for wrapper generators.

**class** `pybindgen.typehandlers.base.`**`BuildValueParameters`**

Bases: `object`

Object to keep track of Py_BuildValue (or similar) parameters

```
>>> bld = BuildValueParameters()
>>> bld.add_parameter('i', [123, 456])
>>> bld.add_parameter('s', ["hello"])
>>> bld.get_parameters()
['"is"', 123, 456, 'hello']
>>> bld = BuildValueParameters()
>>> bld.add_parameter('i', [123])
>>> bld.add_parameter('s', ["hello"], prepend=True)
>>> bld.get_parameters()
['"si"', 'hello', 123]
```

**add_parameter** (*param_template*, *param_values*, *prepend=False*, *cancels_cleanup=None*)

Adds a new parameter to the Py_BuildValue (or similar) statement.

**Parameters**

- **param_template** – template item, see documentation for Py_BuildValue for more information
- **param_values** – list of C expressions to use as value, see documentation for Py_BuildValue for more information
- **prepend** – whether this parameter should come first in the tuple being built
- **cancels_cleanup** – optional handle to a cleanup action, that is removed after the call. Typically this is used for 'N' parameters, which already consume an object reference

**clear** ()

**get_cleanups** ()

Get a list of handles to cleanup actions

**get_parameters** (*force_tuple_creation=False*)

returns a list of parameters to pass into a Py_BuildValue-style function call, the first paramter in the list being the template string.

**Parameters** **force_tuple_creation** – if True, Py_BuildValue is instructed to always create a tuple, even for zero or 1 values.

**class** pybindgen.typehandlers.base.**CodeBlock** (*error_return*, *declarations*, *predecessor=None*)

Bases: object

An intelligent code block that keeps track of cleanup actions. This object is to be used by TypeHandlers when generating code.

CodeBlock constructor

```
>>> block = CodeBlock("return NULL;", DeclarationsScope())
>>> block.write_code("foo();")
>>> cleanup1 = block.add_cleanup_code("clean1();")
>>> cleanup2 = block.add_cleanup_code("clean2();")
>>> cleanup3 = block.add_cleanup_code("clean3();")
>>> cleanup2.cancel()
>>> block.write_error_check("error()", "error_clean()")
>>> block.write_code("bar();")
>>> block.write_cleanup()
>>> print block.sink.flush().rstrip()
foo();
if (error()) {
    error_clean()
    clean3();
    clean1();
    return NULL;
}
bar();
clean3();
clean1();
```

**Parameters**

- **error_return** – code that is generated on error conditions (detected by write_error_check()); normally it returns from the wrapper function, e.g. return NULL;
- **predecessor** – optional predecessor code block; a predecessor is used to search for additional cleanup actions.

**class** `CleanupHandle`(*code_block*, *position*)

Bases: `object`

Handle for some cleanup code

Create a handle given code_block and position

`cancel`()

Cancel the cleanup code

`code_block`

`get_position`()

returns the cleanup code relative position

`position`

`CodeBlock.add_cleanup_code`(*cleanup_code*)

Add a chunk of code used to cleanup previously allocated resources

Returns a handle used to cancel the cleanup code

`CodeBlock.clear`()

`CodeBlock.declare_variable`(*type_*, *name*, *initializer=None*, *array=None*)

Calls declare_variable() on the associated DeclarationsScope object.

`CodeBlock.get_cleanup_code`()

return a new list with all cleanup actions, including the ones from predecessor code blocks; Note: cleanup actions are executed in reverse order than when they were added.

`CodeBlock.indent`(*level=4*)

Add a certain ammount of indentation to all lines written from now on and until unindent() is called

`CodeBlock.remove_cleanup_code`(*handle*)

Remove cleanup code previously added with add_cleanup_code()

`CodeBlock.unindent`()

Revert indentation level to the value before last indent() call

`CodeBlock.write_cleanup`()

Write the current cleanup code.

`CodeBlock.write_code`(*code*)

Write out some simple code

`CodeBlock.write_error_check`(*failure_expression*, *failure_cleanup=None*)

Add a chunk of code that checks for a possible error

**Parameters**

- **failure_expression** – C boolean expression that is true when an error occurred
- **failure_cleanup** – optional extra cleanup code to write only for the the case when failure_expression is true; this extra cleanup code comes before all other cleanup code previously registered.

`CodeBlock.`**`write_error_return`**`()`

Add a chunk of code that cleans up and returns an error.

**exception** `pybindgen.typehandlers.base.`**`CodeGenerationError`**

Bases: `pybindgen.typehandlers.base.CodegenErrorBase`

Exception that is raised when wrapper generation fails for some reason.

x.__init__(...) initializes x; see help(type(x)) for signature

**exception** `pybindgen.typehandlers.base.`**`CodegenErrorBase`**

Bases: `exceptions.Exception`

x.__init__(...) initializes x; see help(type(x)) for signature

**class** `pybindgen.typehandlers.base.`**`DeclarationsScope`**`(`*parent_scope=None*`)`

Bases: `object`

Manages variable declarations in a given scope.

Constructor

```
>>> scope = DeclarationsScope()
>>> scope.declare_variable('int', 'foo')
'foo'
>>> scope.declare_variable('char*', 'bar')
'bar'
>>> scope.declare_variable('int', 'foo')
'foo2'
>>> scope.declare_variable('int', 'foo', '1')
'foo3'
>>> scope.declare_variable('const char *', 'kwargs', '{"hello", NULL}', '[]')
'kwargs'
>>> print scope.get_code_sink().flush().rstrip()
int foo;
char *bar;
int foo2;
int foo3 = 1;
const char *kwargs[] = {"hello", NULL};
```

**Parameters** **parent_scope** – optional 'parent scope'; if given, declarations in this scope will avoid clashing with names in the parent scope, and vice versa.

**`clear`**`()`

**`declare_variable`**`(`*type_*, *name*, *initializer=None*, *array=None*`)`

Add code to declare a variable. Returns the actual variable name used (uses 'name' as base, with a number in case of conflict.)

**Parameters**

- **type** – C type name of the variable
- **name** – base name of the variable; actual name used can be slightly different in case of name conflict.

- **initializer** – optional, value to initialize the variable with
- **array** – optional, array size specifiction, e.g. '[]', or '[100]'

**get_code_sink**()

Returns the internal MemoryCodeSink that holds all declararions.

**reserve_variable**(*name*)

Reserve a variable name, to be used later.

**Parameters name** – base name of the variable; actual name used can be slightly different in case of name conflict.

**class** pybindgen.typehandlers.base.**ForwardWrapperBase**(*return_value*, *parameters*, *parse_error_return*, *error_return*, *force_parse=None*, *no_c_retval=False*, *unblock_threads=False*)

Bases: object

Generic base for all forward wrapper generators.

Forward wrappers all have the following general structure in common:

1. **'declarations' – variable declarations; for compatibility** with older C compilers it is very important that all declarations come before any simple statement. Declarations can be added with the add_declaration() method on the 'declarations' attribute. Two standard declarations are always predeclared: '<return-type> retval', unless return-type is void, and 'PyObject *py_retval';
2. **'code before parse' – code before the** PyArg_ParseTupleAndKeywords call; code can be freely added to it by accessing the 'before_parse' (a CodeBlock instance) attribute;
3. **A PyArg_ParseTupleAndKeywords call; uses items from the** parse_params object;
4. **'code before call' – this is a code block dedicated to contain** all code that is needed before calling the C function; code can be freely added to it by accessing the 'before_call' (a CodeBlock instance) attribute;
5. **'call into C' – this is realized by a C/C++ call; the list of** parameters that should be used is in the 'call_params' wrapper attribute;
6. **'code after call' – this is a code block dedicated to contain** all code that must come after calling into Python; code can be freely added to it by accessing the 'after_call' (a CodeBlock instance) attribute;
7. A py_retval = Py_BuildValue(...) call; this call can be customized, so that out/inout parameters can add additional return values, by accessing the 'build_params' (a BuildValueParameters instance) attribute;
8. Cleanup and return.

Object constructors cannot return values, and so the step 7 is to be omitted for them.

Base constructor

**Parameters**

- **return_value** – type handler for the return value
- **parameters** – a list of type handlers for the parameters
- **parse_error_return** – statement to return an error during parameter parsing
- **error_return** – statement to return an error after parameter parsing
- **force_parse** – force generation of code to parse parameters even if there are none
- **no_c_retval** – force the wrapper to not have a C return value

- **unblock_threads** – generate code to unblock python threads during the C function call

**HAVE_RETURN_VALUE = False**

**PARSE_TUPLE = 1**

**PARSE_TUPLE_AND_KEYWORDS = 2**

**generate_body**(*code_sink*, *gen_call_params=()*)
Generate the wrapper function body code_sink – a CodeSink object that will receive the code

**generate_call**()
Generates the code (into self.before_call) to call into Python, storing the result in the variable 'py_retval'; should also check for call error.

**get_py_method_def_flags**()
Get a list of PyMethodDef flags that should be used for this wrapper.

**reset_code_generation_state**()

**set_parse_error_return**(*parse_error_return*)

**write_close_wrapper**(*code_sink*)

**write_open_wrapper**(*code_sink*, *add_static=False*)

**exception** pybindgen.typehandlers.base.**NotSupportedError**
Bases: pybindgen.typehandlers.base.CodegenErrorBase

Exception that is raised when declaring an interface configuration that is not supported or not implemented.

x.__init__(...) initializes x; see help(type(x)) for signature

**class** pybindgen.typehandlers.base.**NullTypeTransformation**
Bases: object

Null type transformation, returns everything unchanged.

x.__init__(...) initializes x; see help(type(x)) for signature

**create_type_handler**(*type_handler_class*, **args*, ***kwargs*)
identity transformation

**get_untransformed_name**(*name*)
identity transformation

**transform**(*type_handler*, *declarations*, *code_block*, *value*)
identity transformation

**untransform**(*type_handler*, *declarations*, *code_block*, *value*)
identity transformation

**class** pybindgen.typehandlers.base.**Parameter**(*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*)
Bases: pybindgen.typehandlers.base._Parameter

Creates a parameter object

**Parameters**

- **ctype** – actual C/C++ type being used
- **name** – parameter name
- **direction** – direction of the parameter transfer, valid values are DIRECTION_IN, DIRECTION_OUT, and DIRECTION_IN|DIRECTION_OUT

**CTYPES = NotImplemented**

**class** `pybindgen.typehandlers.base.`**`ParameterMeta`**(*mcs*, *name*, *bases*, *dict_*)

Bases: `type`

Metaclass for automatically registering parameter type handlers

metaclass __init__

**class** `pybindgen.typehandlers.base.`**`ParseTupleParameters`**

Bases: `object`

Object to keep track of PyArg_ParseTuple (or similar) parameters

```
>>> tuple_params = ParseTupleParameters()
>>> tuple_params.add_parameter('i', ['&foo'], 'foo')
1
>>> tuple_params.add_parameter('s', ['&bar'], 'bar', optional=True)
2
>>> tuple_params.get_parameters()
['"i|s"', '&foo', '&bar']
>>> tuple_params.get_keywords()
['foo', 'bar']
```

```
>>> tuple_params = ParseTupleParameters()
>>> tuple_params.add_parameter('i', ['&foo'], 'foo')
1
>>> tuple_params.add_parameter('s', ['&bar'], 'bar', prepend=True)
2
>>> tuple_params.get_parameters()
['"si"', '&bar', '&foo']
>>> tuple_params.get_keywords()
['bar', 'foo']
```

```
>>> tuple_params = ParseTupleParameters()
>>> tuple_params.add_parameter('i', ['&foo'])
1
>>> print tuple_params.get_keywords()
None
```

**`add_parameter`**(*param_template*, *param_values*, *param_name=None*, *prepend=False*, *optional=False*)

Adds a new parameter specification

**Parameters**

- **param_template** – template item, see documentation for PyArg_ParseTuple for more information
- **param_values** – list of parameters, see documentation for PyArg_ParseTuple for more information
- **prepend** – whether this parameter should be parsed first
- **optional** – whether the parameter is optional; note that after the first optional parameter, all remaining parameters must also be optional

**`clear`**()

**`get_keywords`**()

returns list of keywords (parameter names), or None if none of the parameters had a name; should only be called if names were given for all parameters or none of them.

**get_parameters**()

returns a list of parameters to pass into a PyArg_ParseTuple-style function call, the first paramter in the list being the template string.

**is_empty**()

**class** pybindgen.typehandlers.base.**PointerParameter**(*ctype*, *name*, *direction=1*, *is_const=False*, *default_value=None*, *transfer_ownership=False*)

Bases: pybindgen.typehandlers.base.Parameter

Base class for all pointer-to-something handlers

**CTYPES = NotImplemented**

**class** pybindgen.typehandlers.base.**PointerReturnValue**(*ctype*, *is_const=False*, *caller_owns_return=None*)

Bases: pybindgen.typehandlers.base.ReturnValue

Base class for all pointer-to-something handlers

**CTYPES = NotImplemented**

**class** pybindgen.typehandlers.base.**ReturnValue**(*ctype*, *is_const=False*)

Bases: pybindgen.typehandlers.base._ReturnValue

Creates a return value object

Keywork Arguments:

**Parameters** **ctype** – actual C/C++ type being used

**CTYPES = NotImplemented**

**class** pybindgen.typehandlers.base.**ReturnValueMeta**(*mcs*, *name*, *bases*, *dict_*)

Bases: type

Metaclass for automatically registering parameter type handlers

metaclass __init__

**class** pybindgen.typehandlers.base.**ReverseWrapperBase**(*return_value*, *parameters*, *error_return=None*)

Bases: object

Generic base for all reverse wrapper generators.

Reverse wrappers all have the following general structure in common:

1. ‘declarations’ – variable declarations; for compatibility with older C compilers it is very important that all declarations come before any simple statement. Declarations can be added with the add_declaration() method on the ‘declarations’ attribute. Two standard declarations are always predeclared: ‘<return-type> retval’, unless return-type is void, and ‘PyObject *py_retval’;

2. ‘code before call’ – this is a code block dedicated to contain all code that is needed before calling into Python; code can be freely added to it by accessing the ‘before_call’ (a CodeBlock instance) attribute;

3. ‘call into python’ – this is realized by a PyObject_CallMethod(...) or similar Python API call; the list of parameters used in this call can be customized by accessing the ‘build_params’ (a BuildValueParameters instance) attribute;

4. ‘code after call’ – this is a code block dedicated to contain all code that must come after calling into Python; code can be freely added to it by accessing the ‘after_call’ (a CodeBlock instance) attribute;

5. A ‘return retval’ statement (or just ‘return’ if return_value is void)

Base constructor

**Parameters**

- **return_value** – type handler for the return value
- **parameters** – a list of type handlers for the parameters

**NO_GIL_LOCKING = False**

**generate** (*code_sink*, *wrapper_name*, *decl_modifiers=('static', )*, *decl_post_modifiers=()*)
Generate the wrapper

**Parameters**

- **code_sink** – a CodeSink object that will receive the code
- **wrapper_name** – C/C++ identifier of the function/method to generate
- **decl_modifiers** – list of C/C++ declaration modifiers, e.g. 'static'

**generate_python_call** ()
Generates the code (into self.before_call) to call into Python, storing the result in the variable 'py_retval'; should also check for call error.

**reset_code_generation_state** ()

**set_error_return** (*error_return*)

**exception** pybindgen.typehandlers.base.**TypeConfigurationError**
Bases: pybindgen.typehandlers.base.CodegenErrorBase

Exception that is raised when a type handler does not find some information it needs, such as owernship transfer semantics.

x.__init__(...) initializes x; see help(type(x)) for signature

**class** pybindgen.typehandlers.base.**TypeHandler** (*ctype*, *is_const=False*)
Bases: object

**SUPPORTS_TRANSFORMATIONS = False**

**ctype_no_const**

**set_tranformation** (*transformation*, *untransformed_ctype*)

**set_transformation** (*transformation*, *untransformed_ctype*)
Set the type transformation to use in this type handler

**exception** pybindgen.typehandlers.base.**TypeLookupError**
Bases: pybindgen.typehandlers.base.CodegenErrorBase

Exception that is raised when lookup of a type handler fails

x.__init__(...) initializes x; see help(type(x)) for signature

**class** pybindgen.typehandlers.base.**TypeMatcher**
Bases: object

Type matcher object: maps C type names to classes that handle those types.

Constructor

**add_type_alias** (*from_type_name*, *to_type_name*)

**items** ()
Returns an iterator over all registered items

**lookup** (*name*) → type_handler, type_transformation, type_traits

**Parameters** **name** – C type name, possibly transformed (e.g. MySmartPointer<Foo> looks up Foo*)

**Returns** a handler with the given ctype name, or raises KeyError.

Supports type transformations.

`register`(*name*, *type_handler*)
Register a new handler class for a given C type

**Parameters**

- **name** – C type name
- **type_handler** – class to handle this C type

`register_transformation`(*transformation*)
Register a type transformation object

**class** `pybindgen.typehandlers.base.`**`TypeTransformation`**
Bases: `object`

Type transformations are used to register handling of special types that are simple transformation over another type that is already registered. This way, only the original type is registered, and the type transformation only does the necessary adjustments over the original type handler to make it handle the transformed type as well.

This is typically used to get smart pointer templated types working.

x.__init__(...) initializes x; see help(type(x)) for signature

`create_type_handler`(*type_handler_class*, **args*, ***kwargs*)
Given a type_handler class, create an instance with proper customization.

**Parameters**

- **type_handler_class** – type handler class
- **args** – arguments
- **kwargs** – keywords arguments

`get_untransformed_name`(*name*)
Given a transformed named, get the original C type name. E.g., given a smart pointer transformation, MySmartPointer:

```
get_untransformed_name('MySmartPointer<Foo>') -> 'Foo\*'
```

`transform`(*type_handler*, *declarations*, *code_block*, *value*)
Transforms a value expression of the original type to an equivalent value expression in the transformed type.

**Example, with the transformation::** ‘T*’ -> ‘boost::shared_ptr<T>’

**Then::** transform(wrapper, ‘foo’) -> ‘boost::shared_ptr<%s>(foo)’ % type_handler.untransformed_ctype

`untransform`(*type_handler*, *declarations*, *code_block*, *value*)
Transforms a value expression of the transformed type to an equivalent value expression in the original type.

**Example, with the transformation::** ‘T*’ -> ‘boost::shared_ptr<T>’

**Then::** untransform(wrapper, ‘foo’) -> ‘foo->get_pointer()’

`pybindgen.typehandlers.base.`**`add_type_alias`**(*from_type_name*, *to_type_name*) pybindgen.typehandlers.base.**join_ctype_and_name**(*ctype*, *name*)

Utility method that joins a C type and a variable name into a single string

```
>>> join_ctype_and_name(’void*’, ’foo’)
’void *foo’
>>> join_ctype_and_name(’void *’, ’foo’)
’void *foo’
>>> join_ctype_and_name("void**", "foo")
’void **foo’
>>> join_ctype_and_name("void **", "foo")
’void **foo’
>>> join_ctype_and_name(’C*’, ’foo’)
’C *foo’
```

### 2.2.3 cppclass_typehandlers: type handlers for C++ classes (or C structures)

### 2.2.4 typehandlers.codesink: classes that receive generated source code

Objects that receive generated C/C++ code lines, reindents them, and writes them to a file, memory, or another code sink object.

**class** pybindgen.typehandlers.codesink.**CodeSink**

Bases: object

Abstract base class for code sinks

Constructor

```
>>> sink = MemoryCodeSink()
>>> sink.writeln("foo();")
>>> sink.writeln("if (true) {")
>>> sink.indent()
>>> sink.writeln("bar();")
>>> sink.unindent()
>>> sink.writeln("zbr();")
>>> print sink.flush().rstrip()
foo();
if (true) {
    bar();
zbr();
```

```
>>> sink = MemoryCodeSink()
>>> sink.writeln("foo();")
>>> sink.writeln()
>>> sink.writeln("bar();")
>>> print len(sink.flush().split("\n"))
4
```

**indent**(*level=4*)

Add a certain ammount of indentation to all lines written from now on and until unindent() is called

**unindent**()

Revert indentation level to the value before last indent() call

**writeln**(*line=’‘*)

Write one or more lines of code

**class** pybindgen.typehandlers.codesink.**FileCodeSink**(*file_*)

Bases: pybindgen.typehandlers.codesink.CodeSink

A code sink that writes to a file-like object

**Parameters** **file** – a file like object

**writeln**(*line=''*)
Write one or more lines of code

**class** `pybindgen.typehandlers.codesink.`**`MemoryCodeSink`**
Bases: `pybindgen.typehandlers.codesink.CodeSink`

A code sink that keeps the code in memory, and can later flush the code to another code sink

Constructor

**flush**()
Flushes the code and returns the formatted output as a return value string

**flush_to**(*sink*)
Flushes code to another code sink :param sink: another CodeSink instance

**writeln**(*line=''*)
Write one or more lines of code

**class** `pybindgen.typehandlers.codesink.`**`NullCodeSink`**
Bases: `pybindgen.typehandlers.codesink.CodeSink`

A code sink that discards all content. Useful to ‘test’ if code generation would work without actually generating anything.

Constructor

**flush**()
Flushes the code and returns the formatted output as a return value string

**flush_to**(*sink*)
Flushes code to another code sink :param sink: another CodeSink instance

**writeln**(*line=''*)
Write one or more lines of code

CHAPTER 3

# Indices and tables

- *genindex*
- *modindex*
- *search*

# Python Module Index