PYB11Generator Documentation

Release 1.0

J. Michael Owen

Apr 23, 2020

Contents:

PYB11Generator is a python based code generator that creates pybind11 code for binding C++ libraries as extensions in Python. PYB11Generator parses input that is very close to writing the desired interface in native python, turning this into the corresponding pybind11 C++ code.

Note: Since PYB11Generator blindly generates C++ pybind11 code, it is essential to understand the pybind11 package itself as well! In other words, be sure to read and understand the pybind11 documentation before trying to go too far with PYB11Generator. The purpose of PYB11Generator is to reduce the burden of writing and maintaining redundant code when working with pybind11 (such as the trampoline classes required by Overriding virtual functions in Python), and provide a natural syntax for those already familiar with writing interfaces in Python. However, since the generated pybind11 code produced by PYB11Generator is what is actually compiled by a C++ compiler to create the corresponding python package, any errors reported by the compiler will refer to this generated code, and require understanding pybind11 itself to properly interpret.

CHAPTER 1

An important caveat about Python versions

As currently implemented, PYB11Generator assumes Python 2, and will not work with Python 3 input syntax. This is due to the fact PYB11Generator grew from an internal utility in the Spheral astrophysics modeling project, which uses Python 2.* syntax for backwards compatability with Spheral work that predates the existence of Python 3. The generated pybind11 code itself is not restricted to Python 2 however, so the generated modules should be compatible with Python 2 or 3 – only the input files to PYB11Generator need to be in Python 2 syntax.

1.1 Introduction to PYB11Generator

Using PYB11Generator to write the interface for a C++ binding is intended to emulate writing that same interface in Python, so if you're familiar with Python it should be easy to get started with PYB11Generator. As an example, if we have a header for a C++ function that looks like:

```
// A really important function!
int func();
```

We can define the PYB11Generator prescription for binding this method by writing a Python method as:

```
def func():
    "A really important function!"
    return "int"
```

Wherever possible we try to use ordinary Python syntax to correspond to pybind11/C++ constructs: Python functions correspond to and generate binding code for C++ functions as above; a Python class generates binding code for pybind11 to bind a C++ class; arguments for functions and methods in Python generate corresponding argument specifications in C++ function pointer syntax. Because Python is not a strongly typed language, we specify C++ types using strings (if needed) as above, where we specify the return `int` type by returning the string `"int"` from `func`. We also use Python decorators to annotate Python methods with uniquely C++ concepts such as `const`, `virtual`, etc., as will be discussed in succeeding sections.

1.1.1 Installation

PYB11Generator uses the Python Package Index PyPI to simplify installation of PYB11Generator, so installing PYB11Generator is simply a matter of requesting it using pip:

```
$ pip install PYB11Generator
```

This command installs PYB11Generator (and pybind11 since it is a dependency of PYB11Generator) as packages in the Python you are using (i.e., the Python associated with the `pip` command).

For those curious, the source for PYB11Generator is hosted on github.

1.1.2 The basics: how to generate pybind11 code using PYB11Generator

PYB11Generator works by starting up a Python process, importing a module containing Python definitions for functions and classes corresponding to the C++ interface to be bound, and invoking the function `PYB11generateModule` on the imported Python module, which writes out a C++ file of pybind11 statements binding that interface. This generated pybind11 C++ file is what is compiled by a C++ compiler to create the final Python shared module allowing the C++ methods and classes to be exercised from Python. As an example, if we have created a Python file `mymodule.py` containing the Python description of the C++ methods we wish to expose in a module to be called `mymodule`, we can invoke `PYB11generateModule` to create the intermediate C++ file as:

```
python -c 'from PYB11Generator import *; import mymodule;␣
↪PYB11generateModule(mymodule)'
```

resulting in the creation of a C++ source file `mymodule.cc`. A full description of the `PYB11generateModule` interface is given in *PYB11 special functions and classes*.

1.1.3 A first example start to finish

To explicitly demonstrate the stages of creating bindings using PYB11Generator, here we recreate an early example in the pybind11 documentation: Creating bindings for a simple function. Say we want to wrap the following C++ method:

```
int add(int i, int j) {
  return i + j;
}
```

We can use PYB11Generator to create the same pybind11 code used to wrap this method in the pybind11 tutorial by writing a file `simple_example.py` containing:

```
"pybind11 example plugin"

PYB11preamble = """
int add(int i, int j) {
  return i + j;
}
"""

def add():
    "A function which adds two numbers"
    return
```

Now executing the command:

```
$ python -c 'from PYB11Generator import *; import simple_example;␣
↪PYB11generateModule(simple_example, "example")'
```

creates a file `example.cc`, which looks like (omitting the boilerplate preamble code with `#include`'s):

```
int add(int i, int j) {
  return i + j;
}

//------------------------------------------------------------------------
// Make the module
//------------------------------------------------------------------------
PYBIND11_MODULE(example, m) {

  m.doc() = "pybind11 example plugin"  ;

  //.....................................................................
  // Methods
  m.def("add", &add, "A function which adds two numbers");
}
```

This is identical to the native pybind11 binding code from the pybind11 tutorial Creating bindings for a simple function, modulo some comments. This code can now be compiled to the final Python shared module as described this same pybind11 tutorial:

```
$ c++ -O3 -Wall -shared -std=c++11 -fPIC `python -m pybind11 --includes` example.cc -
↪o example.so
```

A few things worth noting:

- This example uses the fact that if the function being wrapped is unambiguous, allowing us to use a bare C++ function pointer (without the full explicit function prescription). This is reflected in the PYB11Generator syntax when we write the `def add()` function in python without arguments or a return type.
- In order to directly insert the C++ function definition into the resulting C++ file, we have used the special variable `PYB11preamble` variable. A more typical use case will require `#include`-ing the necessary C++ header files in the generated code, which is accomplished through another special variable, `PYB11includes`, described in *PYB11 reserved variables*.
- In general special variables and commands to PYB11Generator use the prefix `PYB11` such as `PYB11preamble` in this example.
- Note also that ordinary Python doc strings (both for the module and function) are picked up from `simple_example.py` and propagated to the pybind11 bindings.

This example demonstrates the steps necessary to create a usable Python module using PYB11Generator:

1. Create a Python file describing the desired interface using ordinary Python syntax, based on the C++ methods and classes to be bound.
2. Run a Python line like above to generate the pybind11 C++ code from this Python input.
3. Compile the resulting pybind11 C++ code to create the Python shared module.

In the following sections we describe the nuances of creating the PYB11 Python input files in much more detail; we will not show the compilation examples beyond this point since it is no different than using pybind11 directly, and the above example pretty much covers it.

1.2 Functions

We have already introduced a quick example of binding a function in *A first example start to finish*; this section will go into more detail on how to generate pybind11 bindings for functions, including complications such as overloaded methods and C++ templates.

1.2.1 Ordinary and overloaded functions

Suppose we have a header defining the following functions that we wish to bind for Python:

```
double unique_function(int a, double b);
double overloaded_function(int a, int b, int c);
double overloaded_function(double a, double b, double c);
```

We can use PYB11Generator to bind these functions with a file containing the following code:

```
from PYB11Generator import *      # Necessary to get decorators

def unique_function():
    "This is a unique function prescription, and so requires no details about␣
↪arguments or return types"
    return

def overloaded_function(a = "int",
                        b = "int",
                        c = "int"):
    "This is the version of overloaded_function that takes ints"
    return "double"

@PYB11pycppname("overloaded_function")
def overloaded_function1(a = "double",
                         b = "double",
                         c = "double"):
    "This is the version of overloaded_function that takes doubles"
    return "double"
```

The first function `unique_function` is trivial, since it is unambiguous and can be wrapped with an unadorned C++ function pointer as shown in *A first example start to finish*. In this case PYB11Generator assumes the C++ function name is the same as the Python function name, and all is simple.

The overloaded functions take a bit more work. The first challenge is that Python does not support the concept of function overloading: two Python functions cannot have the same name. Therefore we need to use unique Python names for the C++ `overloaded_function` Python descriptions, which is why we define `overloaded_function` and `overloaded_function1` in the source for PYB11Generator. In order to tell PYB11Generator that we really want to call `overloaded_function1` `overloaded_function` in both the C++ and Python bindings, we use our first PYB11 decorator: `PYB11pycppname`. This decorator tells PYB11Generator that that function in question is really called `overloaded_function` in C++, and we wish the Python name in the resulting binding code to call this function `overloaded_function` in Python as well. This is actually two statements, and there are two PYB11 decorators that can do these individual tasks independently if needed (`PYB11cppname` and `PYB11pyname`): `PYB11pycppname` is simply a convenient shorthand combination to cover the common case of wanting to simultaneously rename the bound method for C++ and Python. For a full listing of the PYB11 decorators see *PYB11 decorators*.

Note we have also now specified the arguments and return types for both bindings of `overloaded_function`. This is required since the C++ functions are overloaded, and in order for the C++ compiler to distinguish which one we want it is necessary to fully specify the function signatures for the function pointers in the pybind11

binding code. PYB11Generator always checks the return value for a wrapped function: if a return value is present, it should be a string describing the C++ return type (as shown here, with both `overloaded_function` and `overloaded_function1` returning the string value `"double"`). If such a return value is specified, PYB11Generator assumes a fully qualified C++ function pointer signature is required, and will also look for and generate the argument types as well. The function arguments should be named what the argument name will be in the resulting Python code, and set equal to a string with the C++ type of the argument as shown above for the `overloaded_function` descriptions. Note, a C++ `void` return value or argument should be set to the string `"void"` for PYB11Generator for such explicit specifications.

1.2.2 Default argument values

Another useful feature of pybind11 is the ability to specify default values for arguments to functions/methods in Python, and naturally PYB11Generator supports this feature as well. In order to specify a default value for an argument, we set the value of the argument in the Python binding code as a tuple, where the first element is a string describing the C++ type, and the second a string with the C++ default value. As an example suppose we wish to bind the following function that has two arguments (an `int` and a `std::string`):

```
void howToDrawADragon(int numberOfBeefyArms, std::string label name);
```

and we want to use the default values `1` and `"Trogdor"` for these arguments. The PYB11Generator code would then look like:

```
def howToDrawADragon(numberOfBeefyArms = ("int", "1"),
                     name = ("std::string", "Trogdor")):
    return "void"
```

1.2.3 C++ template functions

C++ templates present another challenge, as this another concept not found in Python. Suppose we wish to expose several instantiations of the following method:

```
template<typename ValueA, typename ValueB, typename ValueC>
ValueC
transmogrify(const ValueA& x, const ValueB& y);
```

It is always possible to explicitly (and repetitively) define the function over and over again for each template instantiation combination of (`ValueA`, `ValueB`, `ValueC`), but we would rather write the prescription once and have the computer generate the necessary redundant code. PYB11Generator has such a facility: a template method can be defined with the `@PYB11template` decorator, which takes the template arguments as a set of string arguments. The function can then be instantiated as many times as needed using the function `PYB11TemplateFunction`. The complete PYB11Generator binding code then might look like:

```
from PYB11Generator import *      # Necessary to get decorators and␣
↪PYB11TemplateFunction

@PYB11template("ValueA", "ValueB", "ValueC")
def transmogrify(x = "const %(ValueA)s&",
                 y = "const %(ValueB)s&"):
    "I'm sure this does something useful..."
    return "%(ValueC)s"

transmogrifyIntIntDouble = PYB11TemplateFunction(transmogrify, ("int", "int", "double
↪"),              pyname="transmogrify")
transmogrifyI32I32I64    = PYB11TemplateFunction(transmogrify, ("uint32_t", "uint32_t
↪", "uint64_t"), pyname="transmogrify")
```

(continues on next page)

(continued from previous page)

The first thing to note when defining a template function is that the template arguments can be used as Python string dictionary substitution variables, as shown above in the definition of `transmogrify`. Since we have defined the template parameters using the decorator `@PYB11template("ValueA", "ValueB", "ValueC")` we can use `%(ValueA)s`, `%(ValueB)s`, or `%(ValueC)s` in the body of the function, as we do in this case defining the arguments and return type.

Because we have decorated the `transmogrify` function with `@PYB11template`, PYB11 will not generate any pybind11 code directly from this function. Instead we must define instantiations of such template functions using the PYB11 function `PYB11TemplateFunction`. In this example we have created two such instantiations, and could continue making as many as we wish for different types. Note in this example we have made these different instantiations overloaded in Python by forcing them all to have the name `transmogrify` via the `pyname="transmogrify"` argument. This is not necessarily required: we must give each instantiation of the template a unique name in Python (`transmogrifyIntIntDouble` and `transmogrifyI32I32I64` in this case), and if we are happy with those being the Python names of the wrapped results we need not specify `pyname`. Such unique names in Python are safest, in that which instantiation the user wants to call down the line in the wrapped library call is unambiguous, but often it is nicer to force the Python names to match the C++ as we do in this case.

For a full description of `PYB11TemplateFunction` see *PYB11TemplateFunction()*.

Note: In this example we have used the common case of C++ templates declared with `typename` (as in `template<typename T1, typename T2>`). However, for C++ templates can also use specialized parameters, such as

```
template<int T1, double T2> func(const double x);
```

In such cases we need need to specify these template parameters appropriately to PYB11Generator. This is done by explictly declaring the types of the template parameters in `PYB11template`:

```
@PYB11template("int T1", "double T2")
def func(x = "const double"):
    "What does this function do?"
    return
```

1.2.4 Explicitly defining the binding implementation for a function

In some instances it is useful to take direct control of or modify how a given function is exposed to Python. PYB11Generator allows the user to directly specify what is passed in-place of the function pointer in such cases via the `@PYB11implementation` decorator. There are far too many possible use cases for this direct control to possibly discuss, but as an example suppose we have a function like the following that uses an exotic container type as an argument:

```
void ExoticContainer permutate(const ExoticContainer& c);
```

If pybind11 knows nothing about the `ExoticContainer` class, and we would rather expose this to Python using ordinary Python lists, we could use the following pattern to wrap a list based interface around `permutate`:

```
@PYB11implementation("""[](py::list c) -> py::list {
                                                    ExoticContainer ccopy;
                                                    for (const auto& x: c) ccopy.
↪push_back(x);
```

(continues on next page)

(continued from previous page)

```
                                                    permutate(ccopy);
                                                    py::list result;
                                                    for (const auto& x: ccopy)␣
↪result.append(x);
                                                    return result;
                                                  }""")
def permutate(c = "py::list"):
    return "py::list"
```

The resulting pybind11 code is:

```
m.def("permutate", [](py::list c) -> py::list {
                                                    ExoticContainer ccopy;
                                                    for (const auto& x: c) ccopy.push_
↪back(x);
                                                    permutate(ccopy);
                                                    py::list result;
                                                    for (const auto& x: ccopy) result.
↪append(x);
                                                    return result;1
                                                  }, "c"_a);
```

so as you can see `@PYB11implementation` allows the author to directly control the code inserted in the usual spot for a function pointer. Note that the argument spec is still generated (`"c"_a` in this example), including any default arguments defined as described above in *Default argument values*.

1.2.5 Preventing automatic casting of arguments

In C++ automatic casting of arguments which are implicitly convertible to a different type (such as calling a function that accepts a `double` argument with an `int`) is usually allowed. In pybind11 it is possible to prevent this behavior using the `.noconvert()` option to a Python argument, such as `py::arg().noconvert()`. PYB11Generator supports the idea of `noconvert` as well, though in a less granular fashion currently as it is used to decorate an entire function signature rather than individual arguments. For instance, if we wanted to bind the following method and ensure automatic conversions of the argument are prevented:

```
double munge_my_double(double x);
```

we can accomplish this using the `@PYB11noconvert` decorator:

```
@PYB11noconvert
def munge_my_double(x = "double"):
    return "double"
```

See the pybind11 discussion for more information.

1.3 Classes

Binding classes in PYB11Generator is based on writing the desired interface as a Python class, similar to the process for *Functions*. As a first example consider the example struct used as the first such example in the pybind11 class documentation Object-oriented code:

```
struct Pet {
  Pet(const std::string &name) : name(name) { }
  void setName(const std::string &name_) { name = name_; }
  const std::string &getName() const { return name; }

  std::string name;
};
```

This struct can be wrapped in straightforward fashion in PYB11Generator as:

```
class Pet:

    def pyinit(self,
               name = "const std::string&"):
        return

    def setName(self,
                name = "const std::string&"):
        return "void"

    def getName(self):
        return "const std::string"
```

Processing this Python class definition through PYB11Generator results in the following (omitting generic preamble code):

```
// Class Pet
{
  py::class_<Pet> obj(m, "Pet");

  // Constructors
  obj.def(py::init<const std::string&>(), "name"_a);

  // Methods
  obj.def("setName", (void (Pet::*)(const std::string&)) &Pet::setName, "name"_a);
  obj.def("getName", (const std::string (Pet::*)()) &Pet::getName);
}
```

which is very similar to the native pybind11 code presented in Object-oriented code. This example demonstrates a few important aspects of generating class bindings with PYB11Generator:

- A python class results in the generation of a pybind11 `class_<>` declaration.
- Binding class methods with PYB11Generator is directly analogous to binding free functions: we write the method signature in python syntax, with the arguments set equal to the C++ type as a string.
  - If the C++ class method is unambiguous (not overloaded), then just as with functions we can specify the method in python with no arguments and an empty return value.
  - If a default value for an argument is desirable, simply set the argument equal to a tuple of two strings: `arg = ("C++ type", "C++ default value")`, identically to the treatment of functions in *Default argument values*.
- Constructors are specified by any class method starting with the string `pyinit`.

1.3.1 Constructors

In general PYB11Generator interprets methods of classes as ordinary methods to exposed via pybind11 – the one exception to this rule is class constructors. Any method that begins with the name `pyinit` is interpreted as a class constructor, allowing the specification of an arbitrary number of constructors. For instance, if we have a C++ class with the following constructors:

```
class A {
public:
  A();                                                  // Default constructor
  A(const std::string name);                            // Build with a name, default␣
↪priority
  A(const std::string name, const int priority);    // Build with a name and priority
};
```

We can bind these three different constructors using the following Python specification:

```
class A:
    "A class that does something with a string and an int..."

    def pyinit(self):
        "Default constructor"

    def pyinit1(self, name="const std::string"):
        "Build with a name, default priority"

    def pyinit2(self, name="const std::string", priority="const int"):
        "Build with a name and priority"
```

For constructors it does not matter what names are used past the `pyinit` string: any such name will be interpreted as a constructor. All that is required is that any class `pyinit*` name be unique – remember, python does not allow overloading, so defining successive methods with the same name simply causes the earlier method definitions to be lost. Not that the author has made such mistakes in creating my own binding code...

1.3.2 Inheritance

Class inheritance hierarchies in C++ are simple to reflect in PYB11Generator, as this is an OO concept shared by both C++ and Python: all that is required is to reflect the inheritance hierarchy in the Python PYB11 code. In order to expose the following C++ classes:

```
class A {
public:
  A();                      // Default constructor
  int func(int x);          // Some useful function of A
};

class B: public A {
public:
  B();                      // Default constructor
  double dfunc(double x); // Some useful function of B
};
```

we can simply reflect this object hiearchy in the PYB11Generator code:

```
class A:

```

(continues on next page)

(continued from previous page)

```
    def pyinit(self):
        "Default constructor"

    def func(self, x="int"):
        "Some useful function of A"
        return "int"

class B(A):

    def pyinit(self):
        "Default constructor"

    def dfunc(self, x="double"):
        "Some useful function of B"
        return "double"
```

Note: Cross module inheritance (binding a class in one module that inherits from a class bound in another) is a slightly trickier case. See the discussion in *Cross-module inheritance* for an example of how to do this.

Note: Another esoteric case is having a non-templated class inherit from a templated one. A method of handling this situation is discussed in *Non-templated class inheriting from a templated class*.

1.3.3 Methods

Class methods are wrapped much like free functions using PYB11Generator: we simply define a python class method with the desired name. If the method is unambiguous (not overloaded), we do not necessarily have to specify the return types and arguments (though full specifications are always allowed, and at times preferable to generate more explicit help in Python). The syntax for specifying C++ return types and arguments for methods is identical to that used for for *Functions*, as is evident in the examples below.

Overloaded methods

Just as with *Ordinary and overloaded functions*, overloaded methods require full call specifications, as well as unique names in python. We use the PYB11 decorators `@PYB11pyname`, `@PYB11cppname`, or `@PYB11pycppname` to link the proper C++/Python names as needed. As an example, consider the following C++ class:

```
class A {
public:
  int process(const int x);                    // Process the internal state somehow␣
↪to answer this query
  std::string label();                         // Return a string label
  std::string label(const std::string suffix); // Return a string label including a␣
↪specified suffix
}
```

In this case we have one unambiguous method (`process`), and two overloaded methods (`label`). We can write PYB11Generator bindings for these methods as:

```
class A:

    def process(self):
        "Process the internal state somehow to answer this query"
        return

    def label(self):
        "Return a string label"
        return "std::string"

    @PYB11pycppname("label")
    def label1(self, suffix="const std::string"):
        "Return a string label including a specified suffix"
        return "std::string"
```

We have chosen to bind the unambiguous `A::process` method using no method signature (i.e., no return type or arguments) for brevity. The overloaded `A::label` methods however require the complete method prescriptions be specified in order for the compiler to know which C++ `A::label` we are referring to. Because Python does not allow class methods with the same name however, we must use unique method names in our Python class binding (hence `A.label` and `A.label1`). We use the PYB11 decorator `@PYB11pycppname` on `A.label1` to indicate we want the bound Python and C++ names to be `label`. This is identical to how this overloading problem is handled for *Ordinary and overloaded functions*.

Note: In this example we have made the typical choice to overload the `label` method in Python just as in C++. We could, however, decide to leave the Python `label` and `label1` methods with unique names, removing the unpythonic overloading concept from the python interface. If we want to leave the Python name of the second binding of `A::label` as `A.label1`, we still need to tell PYB11Generator that the C++ name is `A::label` rather than `A::label1`. In this case we would simply change the decorator to specify the C++ name alone:

```
@PYB11cppname("label")
def label1(self, suffix="const std::string"):
    "Return a string label including a specified suffix"
    return "std::string"
```

Const methods

Const'ness is a concept in C++ not shared by Python, so we use a decorator (`@PYB11const`) to denote a const method when needed. For instance, the following C++ class definition:

```
class A {
public:
  int square(const int x) const { return x*x; }  // Return the square of the argument
};
```

can be specified in PYB11 using:

```
class A:

    @PYB11const
    def square(self, x="const int"):
        "Return the square of the argument"
        return "int"
```

Virtual methods

If we simply wish to expose C++ virtual methods as ordinary class methods in Python (i.e., not allowing overriding the implementation of such methods from Python), then nothing extra need be done in the method binding for PYB11. However, in pybind11 it is also possible to expose C++ virtual methods such that they *can* be overridden from Python descendants, which is a very powerful capability. Exposing such overridable virtual methods in pybind11 involves writing an intermediate "trampoline" class as described in the pybind11 documentation Overriding virtual functions in Python. PYB11Generator automates the generation of such intermediate redundant code (this was in fact the motivating factor in the creation of PYB11Generator), removing much of the bookkeeping necessary to maintain such coding in face of a changing interface. In PYB11Generator all that is required for making a virtual method overridable from Python is decorating such virtual methods with `@PYB11virtual`/`@PYB11pure_virtual` as appropriate. Consider binding the C++ example from the pybind11 documentation Overriding virtual functions in Python:

```
class Animal {
public:
    virtual ~Animal() { }
    virtual std::string go(int n_times) = 0;
};

class Dog : public Animal {
public:
    virtual std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += "woof! ";
        return result;
    }
};
```

All that is necessary to bind this code using PYB11Generator is the following:

```
class Animal:

    def pyinit(self):
        "Default constructor"

    @PYB11pure_virtual
    def go(self, n_times="int"):
        return "std::string"

class Dog(Animal):

    def pyinit(self):
        "Default constructor"

    @PYB11virtual
    def go(self, n_times="int"):
        return "std::string"
```

Now both `Animal` and `Dog` are accessible from Python, and PYB11Generator automatically generates the necessary trampoline classes such that the `go` method can be overridden by descendant Python classes as desired. Note the use of the PYB11 decorators: `PYB11virtual` and `PYB11pure_virtual`. The use of these two should evident from their names and uses in this example:

- `PYB11virtual` decorates C++ methods that are virtual (such as `Dog::go`).
- `PYB11pure_virtual` decorates C++ methods are pure virtual (such as `Animal::go`), marking such classes as abstract.

Protected methods

It is possible to bind protected class methods in pybind11 as described in the pybind11 documentation. In the pybind11 code this requires writing an intermediate C++ class to publish the protected methods. PYB11Generator automates the production of such publisher classes as needed, however, so all that is required to expose a protected class method is to decorate the PYB11 binding with `@PYB11protected`. In order to expose the protected method of the following example:

```
class A {
protected:
  void some_protected_method(const int x);      // A protected method to apply x->A␣
↪somehow
}
```

we simply provide a decorated PYB11 binding as:

```
class A:

    @PYB11protected
    def some_protected_method(self, x="int"):
        "A protected method to apply x->A somehow"
        return "void"
```

Static methods

Static C++ methods are denoted to PYB11Generator using the `@PYB11static` decorator as in the following example.

C++ class with a static method:

```
class A {
public:
  static int func(int x);     // This method does something with x
};
```

PYB11 binding code:

```
class A:

    @PYB11static
    def func(x = "int"):
        "This method does something with x"
        return "int"
```

1.3.4 Special class operators and methods

Python has a number of special methods for classes, such as `__len__`, `__add__`, etc., which allow the object behavior to be controlled for operations such as +, +=, `len()`, and so forth. pybind11 supports these operators, so naturally PYB11Generator does as well. In keeping with PYB11Generators interface, these are specified by providing these special method names in your Python class description.

Numeric operators

The numeric operators supported by PYB11Generator are `__add__`, `__sub__`, `__mul__`, `__div__`, `__mod__`, `__and__`, `__xor__`, `__or__`, `__radd__`, `__rsub__`, `__rmul__`, `__rdiv__`, `__rmod__`, `__rand__`, `__rxor__`, `__ror__`, `__iadd__`, `__isub__`, `__imul__`, `__idiv__`, `__imod__`, `__iand__`, `__ixor__`, `__ior__`, `__neg__`, and `__invert__`. Python descriptions of these methods are available here.

In the common case for binary operators where the argument is of the same type as the class we're binding, we can omit the the argument specification and return type. However, in the case where the binary operator accepts a different C++ type, we need to specify this argument type in the usual PYB11 syntax for arguments and return types.

It is also important to remember that Python does not allow us to define a method name more than once in a class, so if we have overloaded C++ math operators (say `operator+` can accept more than one type), we must give each binding a unique name, but then use decorators such as `@PYB11pyname` to force the special operator name for the method.

As an example, consider the following C++ class which supports addition with itself or a `double`, multiplication by a `double`, and the unary negative operator:

```
class Vector3d {
public:
   Vector3d  operator-() const;

   Vector3d& operator+=(const Vector3d& rhs);
   Vector3d  operator+ (const Vector3d& rhs) const;

   Vector3d& operator+=(const double rhs);
   Vector3d  operator+ (const double rhs) const;

   Vector3d& operator*=(const double rhs);
   Vector3d  operator* (const double rhs) const;
};
```

We can bind these numeric operations for the Python version of `Vector3d` with PYB11Generator using normal Python operator syntax:

```
class Vector3d:

    def __neg__(self):
        return

    def __iadd__(self):
        return

    def __add__(self):
        return

    @PYB11pyname("__iadd__")
    def __iadd__double(self, rhs="const double"):
        return

    @PYB11pyname("__add__")
    def __add__double(self, rhs="const double"):
        return

    def __imul__(self, rhs="const double"):
        return "Vector3d&"
```

(continues on next page)

(continued from previous page)

```
    def __mul__(self, rhs="const double"):
        return "Vector3d"
```

Comparison operators

The comparison operators supported are `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, and `__ge__`. Usage of these methods (naturally all binary operators in this case) follow the same pattern as the numeric binary operators. As an example, suppose our `Vector3d` class in the previous example also defined comparisons with with either `Vector3d` or `double`:

```
class Vector3d {
public:
  bool operator==(const Vector3d& rhs) const;
  bool operator!=(const Vector3d& rhs) const;
  bool operator< (const Vector3d& rhs) const;

  bool operator==(const double rhs) const;
  bool operator!=(const double rhs) const;
  bool operator< (const double rhs) const;
};
```

We can expose these operations to Python similarly to the binary math operators:

```
class Vector3d:

    def __eq__(self):
        return

    def __ne__(self):
        return

    def __lt__(self):
        return

    @PYB11pyname("__eq__")
    def __eq__double(self, rhs="const double"):
        return "bool"

    @PYB11pyname("__ne__")
    def __ne__double(self, rhs="const double"):
        return "bool"

    @PYB11pyname("__lt__")
    def __lt__double(self, rhs="const double"):
        return "bool"
```

Functor (call) operator

A special class operator in Python is the `__call__` operator (corresponding to the C++ `operator()` method), which allows a class to operate like a function. If we have a C++ functor class, we can expose this functor behavior by binding the C++ `operator()` call as `__call__`. As an example, suppose we have C++ functor like the following:

```
class Transmute {
public:
  double operator()(const double x);
};
```

we can expose this functor nature of `Transmute` via this sort of PYB11 binding:

```
class Transmute:

    def __call__(self, x="const double"):
        return "double"
```

PYB11Generator automatically associates `__call__` with the C++ method `operator()`, unless overridden with something like `@PYB11implementation`.

Miscellaneous operators

Another pair other useful operators supported are `__repr__` and `__str__`. These are used to create string representations of objects for slightly different purposes, as explained in the official Python documentation – essentially `__repr__` should return a string representation of the object such that it could be reconstructed, vs. `__str__` which should produce a human friendly string.

Any function or method that produces such strings is fine to bind to these names (often via renaming such as `@PYB11pyname("__str__")`), but a very common pattern is to use lambda functions with the *`PYB11implementation()`* decorator to implement these methods directly in the binding code. As one example, we might bind useful versions of these operators for the example C++ class `Vector3d` above as:

```
class Vector3d:

    @PYB11implementation("[](const Vector3d& self) -> std::string { return "[" + self.
↪x + ", " + self.y + ", " + self.z + "]" }")
    def __repr__(self):
        return "std::string"

    @PYB11implementation("[](const Vector3d& self) -> std::string { return "Vector3d(
↪" + self.x + " " + self.y + " " + self.z + ")" }")
    def __str__(self):
        return "std::string"
```

Sequence methods

Probably the first thing to point out here is this section is *not* necessary for handling STL containers: pybind11 has built-in support for Binding STL containers, which PYB11Generator provides convenient wrappers for. In fact, so long as implicit copying of STL containers through the Python-C++ interface is acceptable, nothing need be done with STL containers at all – they will automatically be handled by pybind11 transparently.

Binding the Python sequence methods for your own C++ types can at times be a complicated process, and there is not necessarily a single solution that fits all cases. There are several methods in Python you can override to provide sequence information: `__len__`, `__getitem__`, `__setitem__`, `__getslice__`, `__setslice__`, `__iter__`, etc. PYB11Generator allows all these methods to be used via pybind11, but it definitely behooves the interested user to thoroughly understand the pybind11 and Python documentation on this subject. It will often require writing some lightweight interstitial code to translate C++ container information to Python and back, for which lambda functions and the *`PYB11implementation()`* decorator are handy.

As the bare beginning of an example, here is a version of one of the pybind11 test C++ sequence classes (stripped to just the interface) drawn from the `pybind11/tests/test_sequences_and_iterators.cpp` test code:

```
class Sequence {
  public:
      Sequence(size_t size);
      Sequence(const std::vector<float> &value);
      Sequence(const Sequence &s);

      bool operator==(const Sequence &s) const;
      bool operator!=(const Sequence &s) const;

      float operator[](size_t index) const;
      float &operator[](size_t index);

      bool contains(float v) const;

      Sequence reversed() const;

      size_t size() const;

      const float *begin() const;
      const float *end() const;
  };
```

and here is an example binding for these methods translated from the pybind11 test code in `pybind11/tests/test_sequences_and_iterators.cpp` to PYB11Generator Python syntax:

```
class Sequence:

   def pyinit0(self, size="size_t"):
       return
   def pyinit1(self, value="const std::vector<float>&"):
       return
   def pyinit2(self, s="const Sequence&"):
       return

   def __eq__(self):
       return
   def __ne__(self):
       return

   # Sequence methods
   @PYB11cppname("size")
   def __len__(self):
       return "size_t"

   @PYB11implementation("[](const Sequence &s, size_t i) { if (i >= s.size()) throw␣
↪py::index_error(); return s[i]; }")
   def __getitem__(self, i="size_t"):
       return "float"

   @PYB11implementation("[](Sequence &s, size_t i, float v) { if (i >= s.size())␣
↪throw py::index_error(); s[i] = v; }")
   def __setitem__(self, i="size_t", v="float"):
       return "void"

   # Optional sequence methods
```

(continues on next page)

(continued from previous page)

```
   @PYB11keepalive(0, 1)    # Essential: keep object alive while iterator exists
   @PYB11implementation("[](const Sequence &s) { return py::make_iterator(s.begin(),␣
↪s.end()); }")
   def __iter__(self):
       return "py::iterator"

   @PYB11cppname("contains")
   @PYB11const
   def __contains__(self, v="float"):
       return "bool"

   @PYB11cppname("reversed")
   @PYB11const
   def __reversed__(self):
       return "Sequence"

   # Slicing protocol
   @PYB11pyname("__getitem__")
   @PYB11implementation("""[](const Sequence &s, py::slice slice) -> Sequence* {
                           size_t start, stop, step, slicelength;
                           if (!slice.compute(s.size(), &start, &stop, &step, &
↪slicelength)) throw py::error_already_set();
                            Sequence *seq = new Sequence(slicelength);
                            for (size_t i = 0; i < slicelength; ++i) {
                              (*seq)[i] = s[start]; start += step;
                            }
                           return seq;
                         }""")
   def __getitem__slice(self, slice="py::slice"):
       return "Sequence*"

   @PYB11pyname("__setitem__")
   @PYB11implementation("""[](Sequence &s, py::slice slice, const Sequence &value) {
                           size_t start, stop, step, slicelength;
                           if (!slice.compute(s.size(), &start, &stop, &step, &
↪slicelength))
                               throw py::error_already_set();
                           if (slicelength != value.size())
                               throw std::runtime_error("Left and right hand size of␣
↪slice assignment have different sizes!");
                           for (size_t i = 0; i < slicelength; ++i) {
                               s[start] = value[i]; start += step;
                           }"""
                         }""")
   def __getitem__slice(self, slice="py::slice", value="const Sequence&"):
       return "void"
```

This rather in-depth example uses a few concepts not introduced yet (such as `@PYB11keepalive`) which are discussed later, but hopefully gives a flavor of what is needed. Mapping types are also supported through the same sort of overriding of built-in Python methods analogous to above.

1.3.5 Templated methods

Templated methods are handled in a very similar manner to *C++ template functions*. Suppose we want to bind the templated method in the following C++ class:

```
class A {
public:

  template<typename ValueA, typename ValueB, typename ValueC>
  ValueC
  transmogrify(const ValueA& x, const ValueB& y);

};
```

In order to bind this method we first create a python class method and decorate it with `@PYB11template` and the template types as strings. We then create however many instantiations of this method as we like using *PYB11TemplateMethod()*:

```
class A:

    @PYB11template("ValueA", "ValueB", "ValueC")
    def transmogrify(self, x="%(ValueA)s", y="%(ValueB)s"):
        "I'm sure this does something useful..."
        return "%(ValueC)s"

    transmogrifyIntIntDouble = PYB11TemplateMethod(transmogrify, ("int", "int",
↪"double"),              pyname="transmogrify")
    transmogrifyI32I32I64    = PYB11TemplateMethod(transmogrify, ("uint32_t", "uint32_
↪t", "uint64_t"), pyname="transmogrify")
```

Comparing this with the example in *C++ template functions*, we see that handling template class methods is nearly identical to template functions. The only real difference is we instantiate the template class method using `PYB11TemplateMethod` (assigned to class attributes) instead of `PYB11TemplateFunction`.

1.3.6 Attributes

C++ structs and classes can have attributes, such as:

```
struct A {
  double x;                // An ordinary attribute
  const double y;          // A readonly attribute
  static double xstatic;   // A static attribute
};
```

Attributes in pybind11 are discussed in Instance and static fields; PYB11Generator exposes these kinds of attributes via the special PYB11 types `PYB1readwrite` and `PYB11readonly`. We can expose the attributes of `A` in this case via PYB11Generator using:

```
class A:
    x = PYB11readwrite(doc="An ordinary attribute")
    y = PYB11readonly(doc="A readonly attribute")
    xstatic = PYB11readwrite(static=True, doc="A static attribute")
```

In this example we have used the optional arguments `doc` to add document strings to our attributes, and `static` to indicate a static attribute – for the full set of options to these functions see *PYB11readwrite()* and *PYB11readonly()*.

1.3.7 Properties

A related concept to attributes is class properties, where we use getter and setter methods for data of classes as though they were attributes. Consider the following C++ class definition:

```
class A {
  public:
  double getx() const;      // Getter for a double named "x"
  void setx(double val);    // Setter for a double named "x"
};
```

There are at least two ways we can go about creating `A.x` as a property.

Option 1: use `PYB11property`

The most convenient method (or at least most succinct) to treat `A.x` as a property is via the `PYB11property` helper type. In this example we could simply write:

```
class A:
    x = PYB11property(getter="getx", settter="setx",
                      doc="Some helpful description of x for this class")
```

This minimal example demonstrates that using `PYB11property` we can expose properties in a single line like this – see full description of *`PYB11property()`*.

Option 2: use an ordinary python property definition

Python has native support for properties via the built-in `property()`; PYB11Generator is able to interpret use of this function to define pybind11 properties as well. We can use this method to create `A.x` as follows:

```
class A:

    def getx(self):
        return

    def setx(self):
        return

    x = property(getx, setx, doc="Some helpful description of x for this class")
```

This method has the advantage we are using all ordinary python constructs, which PYB11Generator is able to parse and create the property as desired.

Note: In this second example we have also exposed the `getx` and `setx` methods to be bound in pybind11. If this is not desired, we can decorate these methods with `@PYB11ignore`, allowing these methods to be used in the `property()` definition while preventing them from being directly exposed themselves.

1.3.8 Dynamic attributes

By default pybind11 classes are immutable from Python, so it is an error to try and insert new attributes to an instance of a pybind11 bound C++ class. This is different than default behavior in Python however, which allows instances of classes to be modified with new attributes. For example, the following is legal Python code:

```
>>> class Strongbadia:
...     headOfState = "Strong Bad"
...
>>> country = Strongbadia()
>>> country.population = "Tire"    # Valid, we just dynamically added a new attribute
```

pybind11 allows us to specify if we want classes to be modifiable in this way (see pybind11 docs), which is reflected in PYB11Generator by using the decorator `@PYB11dynamic_attr`. So if we wanted to modify one of our class bindings for `A` above to allow dynamic attributes, we can simply decorate the class declaration like:

```
@PYB11dynamic_attr
class A:
...
```

1.3.9 Singletons

Suppose we have declared a C++ class to be a singleton object (i.e., declared all constructors and destructors private) like so:

```
class Asingleton {
public:
  static A* instance() { return instanceptr; }

private:
  static A* instanceptr;
  A();
  A(const A&);
  A& operator=(const A&);
  ~A();
};
```

pybind11 (via its use of `std::unique_ptr` to hold Python instances) assumes bound objects are destructible, but for singletons such as `Asingleton` above the destructor is private. We must notify pybind11 that singletons such as this are different (as discussed in pybind11 for Non-public destructors) – PYB11Generator accomplishes this via the decorator `@PYB11singleton` like so:

```
@PYB11singleton
class Asingleton:

    @PYB11static
    @PYB11returnpolicy("reference")
    def instance(self):
        return "Asingleton*"
```

This example also involves setting a policy for handling the memory of the `Asingleton*` returned by `A.instance`: these sorts of memory management details are discussed in *Return value policies*.

1.3.10 Templated classes

PYB11 handles C++ class templates similarly to *C++ template functions*: first, we decorate a class definition with `@PYB11template`, which takes an arbitrary number of string arguments representing the template parameters; second, we use the *`PYB11TemplateClass()`* function to create instantiations of the template class. Consider a C++ template class definition:

```
template<typename Scalar>
class Vector {
public:
  Scalar x, y, z;                          // Coordinate attributes

  Vector(Scalar x, Scalar y, Scalar z);  // Constructor
  Scalar magnitude() const;              // Compute the magnitude (norm)
};
```

We can create PYB11Generator instantiations of this class for `double` and `float` types using:

```
@PYB11template("Scalar")
class Vector:
    "A simple three-dimensional Vector type using %(Scalar)s coordinates"

    x = PYB11readwrite()
    y = PYB11readwrite()
    z = PYB11readwrite()

    def pyinit(self, x="%(Scalar)s", y="%(Scalar)s", z="%(Scalar)s"):
        "Construct with specified coordinates"

    @PYB11const
    def magnitude(self):
        "Compute the magnitude (norm)"
        return "%(Scalar)s"

FloatVector = PYB11TemplateClass(Vector, template_parameters="float")
DoubleVector = PYB11TemplateClass(Vector, template_parameters="double")
```

Just as is the case with template functions, classes decorated with `@PYB11template` are implicitly ignored by PYB11Generator until an instantiation is created with *`PYB11TemplateClass()`*. Additionally, template parameters specified in `@PYB11template` become named patterns which can be substituted with the types used to instantiate the templates. So, in the `Vector` example above, `%(Scalar)s` becomes `float` in the first instantiation and `double` in the second. See *`PYB11template()`* and *`PYB11TemplateClass()`* for further details.

Complications can arise with inheritance of templated classes, particularly if the template parameters change between the base and descendant types. See the discussion in *Non-templated class inheriting from a templated class* and *Templated class inheritance with template parameter changes* for further details.

1.4 Enums

C++ enums are handled in a fairly straightforward manner as discussed in the pybind11 docs. Suppose we have the following enum in C++:

```
// A collection of adorable rodents
enum Rodents {
  mouse = 0,
  rat = 1,
  squirrel = 2,
  hamster = 3,
  gerbil = 4,
  capybara = 5
};
```

PYB11 uses the special method `PYB11enum` to declare enums, directly corresponding to the pybind11 construct `py::enum_`. We can bind our enumeration of Rodents using:

```
Rodents = PYB11enum(("mouse", "rat", "squirrel", "hamster", "gerbil", "capybara"),
                    doc="A collection of adorable rodents")
```

See *PYB11enum()* for the full set of options to `PYB11enum`.

It is also straightforward to declare an enum type that is inside a class scope; if we have a C++ class with an enum like the following:

```
class Homestararmy {
  public:

  enum members {
    HomestarRunner = 0,
    StrongSad = 1,
    Homsar = 2,
    PaintingOfGuyWithBigKnife = 3,
    FrankBennedetto = 4,
  };
};
```

We can bind this enum using PYB11Generator with:

```
class Homestararmy:

    members = PYB11enum(("HomestarRunner", "StrongSad", "Homsar",
↪"PaintingOfGuyWithBigKnife", "FrankBennedetto"))
```

1.5 Memory management

Generally memory management "just works" when binding C++ and Python with pybind11 without worrying about how the memory/lifetime of the objects is handled. However, since C++ allows memory and objects to be allocated/deallocated in a variety of ways, at times it is necessary to pay attention to this issue. In this section we discuss the pybind11 methods of handling memory and object lifetimes PYB11Generator provides wrappers for. In order to understand this section it is advisable to read the pybind11 documentation on the use of smart pointers, pybind11 Return value policies, and Additional call policies.

1.5.1 Class holder types

When pybind11 creates a new instance of a bound C++ class, it uses a smart pointer type to hold and manage that instance. The default type used by pybind11 for this purpose is `std::unique_ptr`, which means new objects created in this manner by Python will be deallocated when their reference count goes to zero. In most circumstances this is fine, but some C++ applications may have a smart pointer type they already are working with. In such cases it might be preferable to make pybind11 manage these object using the same sort of smart pointer. In PYB11 we specify this by decorating class declarations with `@PYB11holder`. For instance, to make pybind11 use `std::shared_ptr` to hold a class type `A`:

```
@PYB11holder("std::shared_ptr")
class A:

    def pyinit(self):
        "Default constructor"
```

This tells pybind11 any new instance of `A` created from python should be managed by `std::shared_ptr`. pybind11 supports `std::unique_ptr` and `std::shared_ptr` without further work. It is possible to use any reference counting smart pointer for this purpose, but types other than `std::unique_ptr` and `std::shared_ptr` require more information be specified to pybind11. PYB11 does not provide any convenience methods for adding that additional information, but it can be done directly with pybind11 as described in the pybind11 documentation.

Note: Overriding the holder smart pointer type can result in subtleties that lead to hard to understand memory errors. If using this capability, read the pybind11 description carefully!

1.5.2 Return value policies

Return value policies are important but at times tricky for C++ types returned by reference or pointer. By default pybind11 assumes Python will take ownership of returned values, implying Python can delete those objects when the Python reference count falls to zero. If a C++ library returns a pointer to memory it expects to manage, however, the result of this conflict over who can manage (delete) that memory is an error. For this reason pybind11 provides Return value policies, allowing the developer to explicitly state how memory returned from C++ should be handled. Before using these policies it is *critical* to read and understand these policies from pybind11. These return value policies are applied (for functions or methods) using the `@PYB11returnpolicy` decorator, with allowed values `take_ownership`, `copy`, `move`, `reference`, `reference_internal`, `automatic`, and `automatic_reference`. The default policy is `automatic`.

Consider the example C++ function from the pybind11 documentation:

```
/* Function declaration */
Data *get_data() { return _data; /* (pointer to a static data structure) */ }
```

If we want to tell pybind11 the C++ side will manage the memory for the returned `Data*` from this method, the `reference` return policy is appropriate. We can express this by decorating the function binding as:

```
@PYB11returnpolicy("reference")
def get_data():
    return "Data*"
```

Decorating return values from class methods is identical to functions: simply use `@PYB11returnpolicy` to decorate the method declaration.

1.5.3 Call policies

While *Return value policies* are specific to return types from functions or methods, call policies allow the user to tie together the lifetimes of return values and/or arguments. This is discussed in depth in the pybind11 documentation Additional call policies. PYB11 provides the decorator `@PYB11keepalive(a, b)` for direct access to the pybind11 command `py::keep_alive<a, b>`. The arguments to the decorator `a` and `b` are integers indicating arguments in the call signature by position index, with the convention:

- 0 denotes the return value of the function/method.
- If decorating a class method, index 1 is the `this` (or `self`) pointer.

To recreate the example from the pybind11 documentation, if we have a custom `List` class which we are binding in Python, we might want to decorate the `append` method like:

```
class List:
```

(continues on next page)

(continued from previous page)

```
    @PYB11keepalive(1, 2)
    def append(self, x):
        return "void"
```

This tells pybind11 to keep the the second argument (`x`, the element being appended) alive so long as the first argument (`self`, our container class) is alive.

1.5.4 Call guards

Another variation on wrapping functions/methods is to provide `call_guard` as discussed in the pybind11 call_guard documentation. Call guards must be default constructable C++ types, and will be built in scope just before calling the wrapped method. One typical usage would be to release the Python Global Interpreter Lock (GIL) before calling a wrapped method, so something like:

```
@PYB11call_guard("py::gil_scoped_release")
def my_wrapped_method():
    "Some C++ method that needs to have the GIL released."
    return
```

1.6 STL containers

In many cases STL containers (such as `vector`, `deque`, `map`, etc.) can be used transparently with pybind11: python lists will automatically convert to `std::vector` and vice versa for example with no extra work or notation needed. The main caveat to this convenience, however, is that this is accomplished by copying the data in the container, which has two potential drawbacks: for large containers this may not be practical; second, any attempt to change data on either the C++ or Python side will be lost due to the fact you would be operating on a copy. Even if your function/method specification is passing STL containers by reference, this silent copying will make modifying them across the Python/C++ interface impossible without further work.

If you need to pass an STL container without all this magic copying, it becomes necessary to directly bind such containers and define the behavior you want. PYB11Generator provides some interfaces to pybind11's machinery for such bindings, but it is essential to first read and understand what pybind11 is doing for STL types.

PYB11Generator currently only provides Python convenience methods for handling two STL containers: `std::vector` via *`PYB11_bind_vector()`*, and `std::map` with *`PYB11_bind_map()`*. It is possible to use the pybind11 semantics directly in C++ in combination with PYB11Generator to handle cases beyond `std::vector` and `std::map` of course, it simply involves using the pybind11 C++ interface directly.

1.6.1 std::vector

Suppose we want to bind `std::vector<int>` and `std::vector<Aclass>` in our module such that they will be modifiable through the interface (no copying). We can accomplish this by adding two lines to our Python module definition:

```
vector_of_int = PYB11_bind_vector("int", opaque=True)
vector_of_Aclass = PYB11_bind_vector("Aclass", opaque=True)
```

When we import the resulting compiled module it will now include the types `vector_of_int` and `vector_of_Aclass` explicitly, and we need to deal in those types rather than the more convenient Python lists for arguments of those types. The `opaque` argument here is what makes pybind11 treat these vector's as mutable

references through the Python/C++ interface. We also have the option of making these bindings local to the module or global: see *PYB11_bind_vector()* for the full description.

1.6.2 std::map

Making `std::map` instances mutable through the Python/C++ interface (opaque as described in pybind11 terms) is similar to our treatment of `std::vector`. If in our module we need to use `std::map<std::string, Aclass>` as an opaque argument we simply add a line:

```
map_of_int_to_Aclass = PYB11_bind_map("int", "Aclass", opaque=True)
```

Just as in our prior `std::vector` examples, our module will now include a type `map_of_int_to_Aclass` which we can use explicitly to pass this container between Python and C++ mutably.

1.7 Complications and corner cases

1.7.1 Cross-module inheritance

For the most part having C++ types exposed in different modules is transparent: so long as you import all the necessary modules once they are all compiled and bound, everything just works. However, the exception to this rule is if you want to bind a class in one module that inherits from a class bound in another. Suppose for instance we have two C++ classes (`A` and `B`), defined in two different headers `A.hh` and `B.hh`, as follows.

A.hh:

```
struct A {
  A()                           { printf("A::A()\n"); }
  virtual ~A()                  { printf("A::~A()\n"); }
  virtual int func(const int x) { printf("A::func(%d)\n", x); return x; }
};
```

B.hh:

```
#include "A.hh"

struct B: public A {
  B(): A()                                  { printf("B::B()\n"); }
  virtual ~B()                              { printf("B::~B()\n"); }
  virtual int func(const int x) override { printf("B::func(%d)\n", x); return x*x; }
};
```

We want to expose these two class in two different modules, `Amodule` and `Bmodule`. We will now need to annotate the bindings for the `A` class with one piece of new information – the module it will be bound in. This is accomplished with a new decorator: `PYB11module`, and the bindings for `Amodule` might look like:

```
from PYB11Generator import *

PYB11includes = ['"A.hh"']

@PYB11module("Amodule")        # <--- This is our new annotation
class A:

    def pyinit(self):
```

(continues on next page)

(continued from previous page)

```
        "Default constructor"

    @PYB11virtual
    def func(self, x="int"):
        "A::func"
        return "int"
```

Let's suppose the above binding source is stored in file `Amodule_bindings.py`. We can now write our binding source for `Bmodule` as normal, but we need to import `Amodule_bindings` so we can express the inheritance relation between `B` and `A`:

```
from PYB11Generator import *

import Amodule_bindings

PYB11includes = ['"B.hh"']

class B(Amodule_bindings.A):

    def pyinit(self):
        "Default constructor"

    @PYB11virtual
    def func(self, x="int"):
        "B::func"
        return "int"
```

The `@PYB11module` decoration on `A` tells PYB11Generator how to generate the pybind11 code to correctly import `A` rather than generate `A` locally, as described in the pybind11 documentation.

Note: It is critical here in the bindings for `Bmodule` that we use `import Amodule_bindings`, and do *not* import `A` into the local scope using `from Amodule_bindings import A`! If we put `A` in the top-level scope of our bindings for `B`, the binding code for `A` will be generated redundantly in the new bindings, and cause a conflict when we try to import the two modules together.

1.7.2 Non-templated class inheriting from a templated class

PYB11Generator needs to know template parameters for templated classes in order to create concrete instantiations, but since Python does not have the concept of templates we have adopted a two-stage process for creating template class instantiations in PYB11 as described in *Templated classes*. However, if we have a non-templated class which inherits from a templated base, there is no longer the second-stage of this procedure using *`PYB11TemplateClass()`* to instantiate the base with the proper template parameters.

It is possible to handle this situation, but it requires two decorations be applied to the non-templated descendant:

1. Because the descendant will inherit the template decoration of the base class, we must explicitly state that the descendant has no template parameters with `@PYB11template()`.
2. We still need to specify what template parameters should be used for the base class. Template parameters in PYB11Generator are specified using python dictionary matching, so we can directly insert the proper template parameter choices in the appropriate dictionary for our non-templated descendant using `@PYB11template_dict`.

These two steps are best demonstrated by an example – consider the following C++ class hierarchy:

```
template<typename Value1, typename Value2>
class A {
public:
  A();
  virtual ~A();
  virtual std::string func(const Value1& x, const Value2& y) const;
};

class B: public A<double, int> {
public:
  B();
  virtual ~B();
  virtual std::string func(const double& x, const int& y) const;
};
```

PYB11Generator can represent this hierarchy with:

```
@PYB11template("Value1", "Value2")
class A:

    def pyinit(self):
        "Default A()"

    @PYB11virtual
    @PYB11const
    def func(self, x="const %(Value1)s&", y="const %(Value2)s&"):
        "Default A::func"
        return "std::string"

@PYB11template()                                                    # <--- force not to␣
↪inherit template parameters from A
@PYB11template_dict({"Value1" : "double", "Value2" : "int"}) # <--- specify the␣
↪template parameter substitutions
class B(A):

    def pyinit(self):
        "Default B()"

    @PYB11virtual
    @PYB11const
    def func(self, x="const double&", y="const int&"):
        "B::func override"
        return "std::string"

# We still need to instantiate any versions of A that we need/use.
A_double_int = PYB11TemplateClass(A, template_parameters=("double", "int"))
```

1.7.3 Templated class inheritance with template parameter changes

Another variation on the above is the templated class inheritance where the template parameters are changed between the base and descendant types. For example, consider the following class hierarchy:

```
template<typename Value1, typename Value2>
class A {
...
};
```

(continues on next page)

(continued from previous page)

```
template<typename Value2, typename Value3>
class B: public A<unsigned, Value2> {
...
};
```

In this case the descendant `B` class inherits from `A`, but specializes one of the template arguments to `unsigned`. Binding instantiations of `A` is straightforward using the methods described in *Templated classes*, but how should we create instantiations of `B`? There are two choices: we can use `PYB11template_dict` as above to specify the `Value1` template parameter for `B`, or we can explicitly give a dictionary for the template parameters in the instantiation of `B`, including the definition for `Value1`. The first pattern can be written as:

```
@PYB11template("Value2", "Value3")
@PYB11template_dict({"Value1" : "unsigned"})
class B(A):
    ...

B_double_int = PYB11TemplateClass(B, template_parameters=("double", "int")
```

Alternatively, we could choose to specify the exact same instantiation of `B_double_int` using an explicit dictionary for `template_parameters` in the instantiation:

```
@PYB11template("Value2", "Value3")
class B(A):
    ...

B_double_int = PYB11TemplateClass(B, template_parameters={"Value1" : "unsigned",
                                                          "Value2" : "double",
                                                          "Value3" : "int"})
```

The end result for binding `B_double_int` is identical, so the choice of which pattern to use is up to the developer and their preference.

1.8 PYB11 reserved variables

For the most part Python variables declared in module bindings are ignored by PYB11Generator. There are a few exceptions to this rule though – some variables are used to communicate information to PYB11Generator as described below.

PYB11includes = [...] A list of strings, each of which represents a file that should be `#include`-ed in the final C++ generated file. For instance, if we needed the C++ to have the following include statements:

```
#include "A.hh"
#include <vector>
```

We would put this in our file for PYB11Generator:

```
PYB11includes = ['"A.hh"', '<vector>']
```

PYB11namespaces = [...] A list of strings for C++ namespaces we wish to use in the generated C++ – as an example, the following statement:

```
PYB11namespaces = ["extreme", "measures"]
```

results in the following in the generated pybind11 C++ source:

```
using namespace extreme;
using namespace measures;
```

PYB11scopenames = [...] A list of C++ types we want to directly declare with `use` (more focused than importing an entire namespace). In order to decare we want to use `std::vector` and `MyNameSpace::A`, we could use:

```
PYB11scopenames = ["std::vector", "MyNameSpace::A"]
```

which simply inserts the following in the generated C++:

```
using std::vector
using MyNameSpace::A
```

PYB11preamble = "..." PYB11preamble is used to specify a string of arbitrary C++ code that will be inserted near the top of the generated pybind11 source file. `PYB11preamble` is a bit of catch-all, we could for instance directly perform the tasks of `PYB11includes` and `PYB11namespace` using `PYB11preamble` by simply typing the final C++ code in here. One typical usage of this preamble variable is to insert small inline utility methods directly in the final C++ code. For instance, if we had need of a simple function we want to use in the subsequent bindings, we could do something like:

```
PYB11preamble = """
namespace JustForBindings {
  inline int square(int x) { return x*x; }
}"""
```

and now our generated code will include this function.

PYB11modulepreamble = "..." PYB11modulepreamble is used to specify a string of arbitrary C++ code that will be inserted following the `PYBIND11_MODULE` statement, so inside module scope. A typical use of this variable is to insert macros such as `PYBIND11_NUMPY_DTYPE(...)`, for native support of user defined types with Numpy.

PYB11opaque = [...] A list of C++ types we want to be treated as opaque: typically STL types declared as opaque and global in another module. See *STL containers* for further information. As an example, to declare that `std::vector<int>` and `std::vector<std::string>` are declared as opaque in another module:

```
PYB11opaque = ["std::vector<int>", "std::vector<std::string>"]
```

which results in the following inserted into the generated C++:

```
PYBIND11_MAKE_OPAQUE(std::vector<int>)
PYBIND11_MAKE_OPAQUE(std::vector<std::string>)
```

Note all of these reserved variables affect the start of the generated pybind11 C++ code, coming before any of the function, class, module, or other pybind11 declarations that are subsequently generated. The order that these methods are executed in is the same as they are listed above: first any `PYB11includes`, then `PYB11namespaces`, `PYB11scopenames`, `PYB11preamble`, and finally `PYB11opaque`. If we were to include all of the above examples (in any order) in a single source code for instance like so:

```
PYB11includes = ['"A.hh"', '<vector>']
PYB11namespaces = ["extreme", "measures"]
PYB11scopenames = ["std::vector", "MyNameSpace::A"]
PYB11preamble = """
namespace JustForBindings {
```

(continues on next page)

(continued from previous page)

```
  inline int square(int x) { return x*x; }
}
"""
PYB11opaque = ["std::vector<int>", "std::vector<std::string>"]
```

the generated pybind11 code would look like:

```
...
#include "A.hh"
#include <vector>

using namespace extreme;
using namespace measures;

using std::vector
using MyNameSpace::A


namespace JustForBindings {
  inline int square(int x) { return x*x; }
}


PYBIND11_MAKE_OPAQUE(std::vector<int>)
PYBIND11_MAKE_OPAQUE(std::vector<std::string>)

//------------------------------------------------------------------------
// Make the module
//------------------------------------------------------------------------
...
```

1.8.1 PYB11 variables for classes

There is also one reserved variable for class scope in PYB11:

PYB11typedefs = "..." A string of C++ to be inserted at the beginning of a class declaration. This is a bit of a misnomer; the string can be any valid C++ for use in the class declaration scope. Suppose for instance we are defining a class `A`, and we want to declare some typedefs only for use in the scope of `A`:

```
class A:

    PYB11typedefs = """
typedef int    IntType;
typedef double ScalarType;
"""
```

which results in the following C++ code:

```
//.....................................................................
// Class A
{

  typedef int    IntType;
  typedef double ScalarType;
```

(continues on next page)

(continued from previous page)

```
  class_<A> obj(m, "A");
  ...
}
```

so now `IntType` and `ScalarType` are avaiable as types in the scope where we are defining A.

1.9 PYB11 decorators

`@PYB11ignore`
Specifies that the decorated object should be ignored by PYB11Generator, i.e., not processed to produce any pybind11 binding output.

`@PYB11template`(*"type1"*, *"type2"*, ...)
Indicates the object should be treated as a C++ template. Accepts any number of strings which represent the names of the template arguments.

The succeeding python class or function can use the specified template argument strings as patterns for substitation with python dictionary string replacement. So if we are binding a C++ templated function:

```
template<typename T>
T manipulate(const T& val);
```

The corresponding PYB11 template specfication would look like:

```
@PYB11template("T")
def manipulate(val = "const %(T)s"):
    return "%(T)s"
```

`@PYB11template_dict`
Explicitly specifies the dictionary of template args to values for use with `@PYB11template` types.

NOTE: this is a highly unusual pattern to need/use. It is preferable to use the ordinary PYB11 template instantion methods `PYB11TemplateClass`, `PYB11TemplateMethod`, *or* `PYB11TemplateFunction`.

`@PYB11singleton`
Specifies that the decorated object should be treated as a C++ singleton.

`@PYB11holder`(*holder_type*)
Specify a special C++ holder for the generated type in `pybind`, rather than the usual default `std::unique_ptr`. See pybind11 documentation on using shared_ptr as a holder type.

`@PYB11dynamic_attr`
Make the wrapped class modifiable, i.e., allow attributes to be added dynamically to an instance of the class in python. See pybind11 documentation about dynamic attributes.

`@PYB11namespace`(*"val"*)
Set the namespace the C++ type should be found in.

`@PYB11cppname`(*"val"*)
Give a value for the C++ name of the decorated function, class, or method. Overrides the default assumption that the C++ name is the same as that given for the object in the PYB11 python binding file.

`@PYB11pycppname`(*"val"*)
Simultaneously set the Python and C++ name of the decorated function, class, or method. Shorthand for specifying both `@PYB11pyname` and `@PYB11cppname` to the given `"val"`.

@PYB11virtual
Mark a class method as virtual.

@PYB11pure_virtual
Mark a class method as pure virtual, making the class abstract.

@PYB11protected
Mark a class method as protected.

@PYB11const
Mark a class method as const.

@PYB11static
Mark a class method as static.

@PYB11noconvert
Applies `py::noconvert` to all the arguments of a method to prevent automatic conversion. See pybind11 discussion of py::.

@PYB11implementation(*"val"*)
Give an implementation for the bound function or method. This is typically used to specify lambda function implementations, or explicitly call a helper method.

@PYB11returnpolicy(*"val"*)
Specify a pybind11 return policy for the return value of a function or method. This is a tricky topic that if misused can create memory errors, but is at times absolutely necessary to get the expected behavior from the underlying C++ code and types. Before using this method carefully read the pybind11 discussion about Return value policies.

@PYB11keepalive(*a*, *b*)
Tie the lifetime of objects in the return value/argument spec together, where the arguments (`a`, `b`) are integers indicating the order of the arguments to tie together (0 refers to the return value). This is another way of specifying memory policies, similar to *returnpolicy*. Carefully read the pybind11 discussion of the `keep_alive` directive in Additional call policies.

@PYB11call_guard(*"val"*)
Specify a pybind11 call_guard for a function or method. See the discussion of *pybind11:call_policies* for examples of call_guards.

@PYB11module(*"val"*)
Indicate the object should be imported from the specified python module. This is useful for classes wrapped in one module which are needed in another, such as for inheritance.

1.10 PYB11 special functions and classes

This section describes the special functions and classes defined in PYB11Generator for use in createing python bindings. Note we use the convention that PYB11 internals always start with the `PYB11` prefix.

PYB11generateModule(*pymodule*[, *basename=None*])
Inspect the function and class definitions in `pymodule`, and write a C++ file containing pybind11 statements to bind those interfaces.

- `pymodule`: the module to be introspected for the interface
- `"basename"`: a basename for the generated C++ file. If specified, the output is written to `basename.cc`, otherwise output will be written to `pymodule.cc`

PYB11TemplateFunction (*func_template*, *template_parameters*[, *cppname = None*, *pyname = None*, *docext = ""*])

Instantiate a function template (`func_template`) that was decorated by `@PYB11template`.

- `func_template`: The template function definition
- `template_parameters`: A single string (for a single template parameter function) or tuple of strings (for multiple template parameters), one for each template parameter defined by `@PYB11template` on `func_template`.
- `cppname`: The name of the C++ function template, if different from that used for `func_template`.
- `pyname`: The name of the resulting Python function; defaults to the name of the instance created for this invocation of `PYB11TemplateFunction`.
- `docext`: An optional string extension to be applied to the docstring associated with `func_template`.

PYB11attr ([*value=None*, *pyname=None*])

Create an attribute in a module; corresponds to the pybind11 command `attr`.

- `value`: define the C++ name this variable corresponds to. If `None`, defaults to the name of the local python variable.
- `pyname`: define the generated python attribte name. If `None`, defaults to the name of the local python variable.

PYB11readwrite ([*static=False*, *pyname=None*, *cppname=None*, *returnpolicy=None*, *doc=None*])

Define a readwrite class attribute; corresponds to pybind11 `def_readwrite`.

- `static`: If `True`, specifies the bound attribute is static.
- `pyname`: Optionally specify the Python name of the attribute. If `None`, assumes the Python name is the name of Python variable instance.
- `cppname`: Optionally specify the C++ name of the attribute. If `None`, assumes the C++ name is the name of Python variable instance.
- `returnpolicy`: Specify a special return policy for how to handle the memory of the return value. Read pybind11 documentation at Return value policies.
- `doc`: Optionally give a docstring.

PYB11readonly ([*static=False*, *pyname=None*, *cppname=None*, *returnpolicy=None*, *doc=None*])

Define a readonly class attribute; corresponds to pybind11 `def_readonly`.

- `static`: If `True`, specifies the bound attribute is static.
- `pyname`: Optionally specify the Python name of the attribute. If `None`, assumes the Python name is the name of Python variable instance.
- `cppname`: Optionally specify the C++ name of the attribute. If `None`, assumes the C++ name is the name of Python variable instance.
- `returnpolicy`: Specify a special return policy for how to handle the memory of the return value. Read pybind11 documentation at Return value policies.
- `doc`: Optionally give a docstring.

PYB11property ([*returnType = None*, *getter = None*, *setter = None*, *doc = None*, *getterraw = None*, *setterraw = None*, *getterconst = True*, *setterconst = False*, *static = None*, *returnpolicy = None*])

Helper to setup a class property.

- `returnType`: Specify the C++ type of the property
- `getter`: A string with the name of the getter method. If `None`, assumes the getter C++ specification looks like `returnType (klass::*)() const`.
- `setter`: A string with the name of the setter method. If `None`, assumes the setter C++ specification looks like `void (klass::*)(returnType& val)`.
- `doc`: Specify a document string for the property.
- `getterraw`: Optionally specify raw coding for the getter method. Generally this is used to insert a C++ lambda function. Only one of `getter` or `getterraw` may be specified.
- `setterraw`: Optionally specify raw coding for the setter method. Generally this is used to insert a C++ lambda function. Only one of `setter` or `setterraw` may be specified.
- `getterconst`: Specify if `getter` is a const method.
- `setterconst`: Specify if `setter` is a const method.
- `static`: If `True`, make this a static property.
- `returnpolicy`: Specify a special return policy for how to handle the memory of the return value. Read pybind11 documentation at Return value policies.

PYB11TemplateMethod(*func_template*, *template_parameters*[, *cppname = None*, *pyname = None*, *docext = ""*])

Instantiate a class method (`func_template`) that was decorated by `@PYB11template`.

- `func_template`: The template method definition
- `template_parameters`: A single string (for a single template parameter method) or tuple of strings (for multiple template parameters), one for each template parameter defined by `@PYB11template` on `func_template`.
- `cppname`: The name of the C++ method template, if different from that used for `func_template`.
- `pyname`: The name of the resulting Python method; defaults to the name of the instance created for this invocation of `PYB11TemplateMethod`.
- `docext`: An optional string extension to be applied to the docstring associated with `func_template`.

PYB11TemplateClass(*klass_template*, *template_parameters*[, *cppname = None*, *pyname = None*, *docext = ""*])

Instantiate a class template (`klass_template`) that was decorated by `@PYB11template`.

- `klass_template`: The template class definition
- `template_parameters`: A single string (for a single template parameter class) or tuple of strings (for multiple template parameters), one for each template parameter defined by `@PYB11template` on `klass_template`.
- `cppname`: The name of the C++ class template, if different from that used for `klass_template`.
- `pyname`: The name of the resulting Python class; defaults to the name of the instance created for this invocation of `PYB11TemplateClass`.
- `docext`: An optional string extension to be applied to the docstring associated with `klass_template`.

PYB11enum(*values*[, *name=None*, *namespace=""*, *cppname=None*, *export_values=False*, *doc=None*])

Declare a C++ enum for wrapping in pybind11 – see pybind11 docs.

- `values`: a tuple of strings listing the possible values for the enum
- name: set the name of enum type in Python. `None` defaults to the name of the instance given this enum declaration instance.

- namespace: an optional C++ namespace the enum lives in.
- cppname: the C++ name of the enum. `None` defaults to the same as `name`.
- export_values: if `True`, causes the enum values to be exported into the enclosing scope (like an old-style C enum).
- doc: an optional document string.

PYB11_bind_vector (*element*[, *opaque=False*, *local=None*])
Bind an STL::vector explicitly. This is essentially a thin wrapper around the pybind11 `py::bind_vector` function (see Binding STL containers).

- `element`: the C++ element type of the `std::vector`
- `opaque`: if `True`, causes the bound STL vector to be "opaque", so elements can be changed in place rather than accessed as copies. See Binding STL containers.
- `local`: determines whether the binding of the STL vector should be module local or not; once again, see Binding STL containers.

PYB11_bind_map (*key*, *value*[, *opaque=False*, *local=None*])
Bind an STL::map explicitly. This is a thin wrapper around the pybind11 `py::bind_map` function (see Binding STL containers).

- `key`: the C++ key type
- `value`: the C++ value type
- `opaque`: if `True`, causes the bound STL map to be "opaque", so elements can be changed in place rather than accessed as copies. See Binding STL containers.
- `local`: determines whether the binding of the STL map should be module local or not; once again, see Binding STL containers.

PYB11_inject (*fromcls*, *tocls*[, *virtual=None*, *pure_virtual=None*])
Convenience method to inject methods from class `fromcls` into `tocls`. This is intended as a utility to help avoiding writing redundant methods common to many classes over and over again. Instead a convenience class can be defined containing the shared methods (typically screened from generation by `@PYB11ignore`), and then `PYB11_inject` is used to copy those methods into the target classes.

- `fromcls`: Python class with methods we want to copy from.
- `tocls`: Python class we're copying methods to.
- `virtual`: if `True`, force all methods we're copying to be treated as virtual.
- `pure_virtual`: if `True`, force all methods we're copying to be treated as pure virtual.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

P