PyLCGDict

Jacek.Generowicz@cern.ch

Automatic C++/Python binding

- Allows the use of C++ libraries from Python.
- SWIG/Boost for the lazy and ignorant.
- Disadvantages
  - You get what you have in C++
    - Less opportunity to make it look natural in Python
    - Less opportunity to add in safety
- Advantages
  - You get what you have in C++
    - Requires no human intervention at all (... or minimizes the need)

Python

- Easy to learn
- Easy to program
- Easy to read
- Very expressive, powerful language

```
const std::vector <const Method *>::const_iterator mIt;
  const std::vector <const Class *>::const_iterator cIt;
  for (mIt = methods.begin(); mIt != methods.end(); ++mIt) {
    const std::vector <const Class *> args =  (*mIt)->argumentTypes();
    std::cout << (*mIt)->returnType()->name() << " " << (*mIt)->name() << "(";
    for (cIt = args.begin(); cIt != args.end(); ++cIt) {
      std::cout << (*cIt)->name() << ", ";
    }
    std::cout << ")" << std::endl;
 }
```

Direct translation ... including the bug

```
for m in methods:
    print '%s %s(' % (m.returnType().name(), m.name())
    for c in m.argumentTypes():
        print c.name(),
    print ')'
```

Pythonic translation (fixing bug)

```
for m in methods:
    print '%s %s(%s)' % (m.returnType(),
                         m.name(),
                         ', '.join(m.argumentTypes()))
```

First class objects

- (Almost) everything is a first class object in Python: can be stored in variables, passed to functions, returned from functions, stored in data structures.
  - functions
  - classes
  - modules

Type system

- Dynamically typed
  - Variables have no type
  - Objects know their type
- Strongly typed
  - Lots of type checking
  - Not much implicit type conversion

Creating Bindings

- Generate the LCG Dictionary for your library
  - `lcgdict MyLibrary.h`
- Compile the dictionary

```
$ >  g++ -shared -o libMyLibraryDict.so -fpic
     MyLibrary_Dict.cpp -llcg_ReflectionBuilder
     -lMyLibrary.so
```

Getting Started

(In Python) import the pylcgdict package:

```
>>> import pylcgdict
```

Load any dictionaries you need:

```
>>> pylcgdict.loadDictionary('MyLibraryDict')
Loaded dictionary MyLibraryDict
```

Get a handle on the C++ global namespace:

```
>>> g = pylcgdict.Namespace('')
<class 'pylcgdict.MyFirstClass'>
```

Classes

```
class MyFirstClass { ... };
```

Instantiating a class looks similar to the way it would be done in C++:

```
>>> mc = g.MyFirstClass(1)
```

mc is an object of type pylcgdict.MyFirstClass:

```
>>> mc
<pylcgdict.MyFirstClass object at
0x401e3d0c>

>>> type(mc)
<class 'pylcgdict.MyFirstClass'>
```

```
class MyFirstClass {
public:
  MyFirstClass();
  MyFirstClass(int i);
  int datum;
  int get_datum();
  void set_datum(int i);
};
```

Note that `dir` reports the existence of (just one, overloaded) constructor (`MyFirstClass`), the field (`datum`), and the two methods (`get_datum`, `set_datum`):

```
>>> pylcgdict.dir(mc)
['MyFirstClass', 'datum', 'get_datum', 'set_datum']
```

Methods are invoked just like they would be in C++:

```
>>> mc.get_datum()
1
>>> mc.set_datum(2)
```

Fields are accessed just like they would be in C++:

```
>>> mc.datum
2
>>> mc.datum = 3
>>> mc.get_datum()
3
```

Namespaces

```
namespace Outer {
  namespace Inner {
    class AClass {
    public:
      class InnerClass {
        // A pretty boring one
      };
    };
  }
}
```

Pylcgdict reflects the C++ namespace hierarchy found in the loaded dictionaries. The C++ scope resolution operator (`::`) maps to Python's attribute access syntax (`.`).

`g.Outer` is a `pylcgdict.Namespace` whose name is "Outer":

```
>>> g.Outer
<pylcgdict.Namespace Outer object at 0x401ea7cc>
```

It contains the `Inner` namespace:

```
>>> pylcgdict.dir(g.Outer)
['Inner']
>>> g.Outer.Inner
<pylcgdict.Namespace Inner object at 0x401ea88c>
```

```
namespace Outer {
  namespace Inner {
    class AClass {
    public:
      class InnerClass {
        // A pretty boring one
      };
    };
  }
}
```

... which, in turn, contains `AClass`

```
>>> pylcgdict.dir(g.Outer.Inner)
['AClass']
>>> g.Outer.Inner.AClass
<class 'pylcgdict.AClass'>
```

... and the latter contains an inner class

```
>>> pylcgdict.dir(g.Outer.Inner.AClass)
['AClass', 'InnerClass']
>>> g.Outer.Inner.AClass.AClass
<pylcgdict.Constructor object at 0x401ea9cc>
>>> g.Outer.Inner.AClass.InnerClass
<class 'pylcgdict.InnerClass'>
```

Templates

```
namespace Templates {
  template<class T, class U>
  class Parametrized {
    ...
}
```

The `g.Templates` namespace contains a template called "Parametrized":

```
>>> g.Templates.Parametrized
<Template 'Templates::Parametrized<>'>
```

We will store it in a Python variable to save typing:

```
>>> gTP = g.Templates.Parametrized
>>> gTP
<Template 'Templates::Parametrized<>'>
```

```
namespace Templates {
  template<class T, class U>
  class Parametrized {
     ...
}
```

```
>>> gTP
<Template 'Templates::Parametrized<>'>
```

The angled brackets which enclose template arguments in C++, map to parentheses in Python: `Parametrized<int, double>` becomes `Parametrized(int, double)`.
The instantiated templates appear as Python classes:

```
>>> gTP(int, float)
<class 'pylcgdict.Parametrized<int,float>'>

>>> gTP(int, double)
<class 'pylcgdict.Parametrized<int,double>'>

>>> gTP(g.Outer.Inner.AClass, g.MyFirstClass)
<class 'pylcgdict.Parametrized<Outer::Inner::AClass,MyFirstClass>'>
```

```
template<class T, class U>
class Parametrized {
public:
  T t;
  U u;
  Parametrized() {}
  Parametrized(T tt, U uu): t(tt), u(uu) {}
  T get_tea() { return t; }
  U get_you() { return u; }
};
```

```
>>> gTP
<Template 'Templates::Parametrized<>'>
```

As the template instances are classes themselves, they can be instantiated:

```
>>> Pif = gTP(int, float)(1, 2.5)
>>> Pif
<pylcgdict.Parametrized<int,float> object at 0x401ea7ec>
```

... and their instances can be used just like those of any other class:

```
>>> Pif.get_tea()
1
>>> Pif.get_you()
2.5
```

```
template<class T, class U>
class Parametrized {
  ...
  Parametrized(T tt, U uu): t(tt), u(uu) {}
  T get_tea() { return t; }
  U get_you() { return u; }
};
```

Python has only one precision for floating point numbers (equivalent to C's `double`).
Pylcgdict provides both `float` and `double` as symbols for use in template arguments:

```
>>> gTP(int, float )(1, 0.8).get_you(); type(_)
0.80000001192092896
<type 'float'>

>>> gTP(int, double)(1, 0.8).get_you(); type(_)
0.8000000000000004
<type 'float'>
```

Note the loss of precision in the case of the `float` template argument.

Overloaded methods

```
class Overload {
public:
  std::string overloaded()                    { return "void"; }
  std::string overloaded(int)                 { return "int"; }
  std::string overloaded(double)              { return "double"; }
  std::string overloaded(MyFirstClass)        { return "MyFirstClass"; }
  std::string overloaded(int, int)            { return "int, int"; }
  std::string overloaded(int, MyFirstClass) { return "int, MyFirstClass"; }
};
```

You can use Python's `help` to see the signatures of methods:

```
>>> help(g.Overload.overloaded)
Help on method dispatching_method in module pylcgdict:

dispatching_method(self, *args) unbound pylcgdict.Overload method
    std::basic_string<char>  ()
    std::basic_string<char>  (int)
    std::basic_string<char>  (double)
    std::basic_string<char>  (MyFirstClass)
    std::basic_string<char>  (int, int)
    std::basic_string<char>  (int, MyFirstClass)
```

The overloaded methods can be called just like they would be called in C++; Pylcgdict takes care of invoking the one which matches the actual types of the arguments:

```
>>> ov = g.Overload()
>>> ov.overloaded()
'void'
>>> ov.overloaded(1)
'int'
>>> ov.overloaded(1.0)
'double'
>>> ov.overloaded(g.MyFirstClass())
'MyFirstClass'
>>> ov.overloaded(1, 1)
'int, int'
>>> ov.overloaded(1, g.MyFirstClass())
'int, MyFirstClass'
```

If the arguments passed do not match any of the signatures, then a `pylcgdict.NoMatchingMethod` exception is raised:

```
>>> ov.overloaded(1, g.MyFirstClass(), 2.3)
Traceback (most recent call last):
      ...
NoMatchingMethod: Overload::overloaded(<type 'int'>,
<class 'pylcgdict.MyFirstClass'>, <type 'float'>)
Candidates:
   ()
   (int)
   (double)
   (MyFirstClass)
   (int, int)
   (int, MyFirstClass)
```

Note how Pylcgdict reports the types which were passed as arguments, as well as list of the sets of argument types which are acceptable.

`NoMatchingMethod` is a subclass of `TypeError`.

Static methods

```
class Staticmethods {
public:
  static std::string staticmethod()    { return "void"; }
  static std::string staticmethod(int) { return "int"; }
};
```

Static methods can be called through an instance of the class:

```
>>> sm = g.Staticmethods()
>>> sm.staticmethod()
'void'
>>> sm.staticmethod(1)
'int'
```

... or without any instance, through the class:

```
>>> g.Staticmethods.staticmethod()
'void'
>>> g.Staticmethods.staticmethod(1)
'int'
```

Sequence protocol

Load the dictionary for the C++ STL, which is provided in SEAL releases

```
>>> pylcgdict.loadDictionary('SealSTLDict')
Loaded dictionary SealSTLDict
```

Instantiate `vector<double>` with 3 (default) elements:

```
>>> vd = g.std.vector(double)(3)
```

... and add some interesting elements to the vector:

```
>>> vd.push_back(1); vd.push_back(2)
```

By default, Pylcgdict displays the contents of the vector in the printed representation of the instances:

```
>>> vd
<vector<double>:[0.0, 0.0, 0.0, 1.0, 2.0]>
```

The Python sequence protocol is automatically supported:

```
>>> vd[3]
1.0
>>> len(vd)
5
```

Iterator protocol

The iterator protocol is supported. Python for loops can loop over the object:

```
>>> [x*x for x in vd]
[0.0, 0.0, 0.0, 1.0, 4.0]
```

Iterators can be acquired explicitly, when necessary:

```
>>> vd2 = g.std.vector(double)()
>>> vd2.push_back(15)
>>> vd2.push_back(16)
>>> ivd = iter(vd2)
>>> ivd.next()
15.0
>>> ivd.next()
16.0
>>> ivd.next()
Traceback (most recent call last):
  File "<console>", line 1, in ?
StopIteration
```

Note: the `StopIteration` exception is Python's standard mechanism for signalling that an iterator has been exhausted.

Inheritance

```
class Base { ... };

class Derived : public Base { ... };
```

Pylcgdict reflects the class hierarchies it finds in the original library:

```
>>> g.Derived.__bases__
(<class 'pylcgdict.Base'>,)
>>> isinstance(g.Derived(), g.Base)
True
```

```cpp
class Base {
public:
  virtual std::string whoami() {
    return "Base";
  }
  virtual std::string inheritme() {
    return "This was defined in Base";
  }
};

class Derived : public Base {
  std::string whoami() {
    return "Derived";
  }
};
```

Methods (and data) are inherited and overridden as one expects in C++:

```
>>> b = g.Base()
>>> d = g.Derived()
>>> b.whoami()
'Base'
>>> d.whoami()
'Derived'
>>> b.inheritme()
'This was defined in Base'
>>> d.inheritme()
'This was defined in Base'
```


```
struct Base {
  virtual void whoami() {
    cout << “Base” << endl;
  }
};

void callback(Base* obj) {
  obj->whoami();
}
```

```
>>> class Derived(g.Base):
...     def whoami(self):
...         print ‘Derived’
...
>>> g.callback(Derived())
```

We want:

```
'Derived'
```

By default we get:

```
'Base'
```

Callbacks from C++ into Python-overridden methods need extra support. This has not been automated yet.

Pointers

```
class Expose {
private:
  std::string text;
public:
  Expose(std::string t): text(t) {}
  std::string passvalue    (Expose  e) { return e.text; }
  std::string passreference(Expose& e) { return e.text; }
  std::string passpointer  (Expose* e) {
    if (e) {return e->text;}
    return "NULL pointer";
  }
  ...
};
```

There are no pointers in Python: hide them where possible

Values can be passed where the formal type is a value or a reference:

```
>>> e1 = g.Expose('one')
>>> e2 = g.Expose('two')
>>> type(e1)
<class 'pylcgdict.Expose'>
>>> e1.passvalue(e2)
'two'
>>> e1.passreference(e2)
'two'
```

... but also where a pointer is required:

```
>>> e1.passpointer(e2)
'two'
```

Sometimes a true pointer type is required. Pylcgdict provides a Pointer metatype, which can create pointer types, or pointers to existing instances.

Create the Python proxy for the type `Expose*`:

```
>>> pylcgdict.Pointer(g.Expose)
<class 'pylcgdict.Pointer(Expose)'>
```

Make a `NULL` pointer of that type:

```
>>> epNULL = pylcgdict.Pointer(g.Expose)(0)
>>> epNULL
<pylcgdict.Pointer(Expose) object at 0x4035c88c>
```

Make a pointer to an existing object:

```
>>> ep2 = pylcgdict.Pointer(g.Expose)(e2)
>>> ep2
<pylcgdict.Pointer(Expose) object at 0x4035c82c>
```

```
std::string passpointer  (Expose* e) {
  if (e) {return e->text;}
  return "NULL pointer";
}
```

Such pointers can be passed to functions expecting `Expose*`:

```
>>> e1.passpointer(epNULL)
'NULL pointer'
>>> e1.passpointer(ep2)
'two'
```

... and they can also be passed to functions expecting `Expose` values:

```
>>> e1.passvalue(ep2)
'two'
>>> e1.passreference(ep2)
'two'
```

But you do have to be careful with pointers:

```
>>> e1.passvalue(epNULL) # Don't try this at home !
Segmentation Fault
```

```
std::string passvalue(Expose  e) { return e.text; }
```

The pointer types try to behave as the underlying type whenever possible:

```
>>> ep2
<pylcgdict.Pointer(Expose) object at
0x4035c82c>
>>> ep2.passvalue(e1)
'one'
```

Note that the above is equivalent to `ep2->passvalue(e1)` in C++. In general the C++ dereference operator "`->`" is mapped to Python's attribute access syntax "`.`"

Sometimes the pointer shadows the object's behaviour with pointer specific behaviour (`[]`, `->` etc.) Use `pylcgdict.Deref` to get the underlying type or object.

Raw pointers are made unavoidable by interfaces which require pointers as output parameters.

```
>>> ep3 = pylcgdict.Pointer(g.Expose)(0)
```

Be careful! ep3 is a NULL pointer at this point

```
>>> ep3.whoami() # Don't try this at home, kids !
Segmentation Fault
```

But this pointer can be readjusted when passed as an output parameter:

```
>>> e1.outputparameter('three', ep3)
```

... making the pointer point to a valid object, and the same operation is now safe:

```
>>> ep3.whoami()
'three'
```

```
class Expose {
  ...
  std::string text;
  ...
  std::string whoami() { return text; }
};
```

```
class Expose {
  ...
  Expose  getvalue()     { return *this; }
  Expose& getreference() { return *this; }
  Expose* getpointer()   { return  this; }
};
```

Objects returned from methods always appear as values, regardless of whether the formal return type is a value, reference or pointer:

```
>>> type(e1.getvalue())
<class 'pylcgdict.Expose'>

>>> type(e1.getreference())
<class 'pylcgdict.Expose'>

>>> type(e1.getpointer())
<class 'pylcgdict.Expose'>
```

Use `pylcgdict.Pointer` when you really need the pointer.

```
Foo* null() {
  return 0;
}
```

There are two alternative modes for NULL pointer treatment.

User must check validity of returned pointer:

```
>>> p = null()
<class 'pylcgdict.Expose'>
>>> if not p: print 'Careful !'
Careful !
```

Python warns of returned NULL pointers:

```
>>> p = null()
Traceback (most recent call last):
  File "<console>", line 1, in ?
pylcgdict.NullPointer
```

Select the behaviour you want at module import time (interface still under development).

Dynamic cast

```
class DynamicDerived : public DynamicBase {
public:
  DynamicBase* getBase() { ... }
    return new DynamicBase;
  }
  DynamicBase* getDerived() {
    return new DynamicDerived;
  }
  ...
```

In C++ the actual (dynamic) type of an object may differ from its formal (static) type. It is up to the programmer to discover the true dynamic type by guessing it, performing a `dynamic_cast` and checking the validity of the resulting pointer. Such low-level details are none of the programmer's concern in Python, so Pylcgdict downcasts all returned values to their true dynamic type:

```
>>> type(g.DynamicDerived().getBase())
<class 'pylcgdict.DynamicBase'>
```

... note how the dynamic type matches the formal return type of the method in the above, but not in the following ...

```
>>> type(g.DynamicDerived().getDerived())
<class 'pylcgdict.DynamicDerived'>
```

```
void modify(int select, DynamicBase*& it) {
    it = new DynamicDerived;
}
```

If, however, the dynamic type of an existing Python proxy object is changed:

```
>>> db = g.DynamicBase()
>>> dd.modify(1, db)
```

such changes are too subtle for Pylcgdict to handle efficiently, and it does not perform a type change automatically:

```
>>> type(db)
<class 'pylcgdict.DynamicBase'>
```

In such cases the conversion to the new dynamic type must be requested explicitly:

```
>>> pylcgdict.dynamic_downcast(db)
>>> type(db)
<class 'pylcgdict.Pointer(DynamicDerived)'>
```

NOTE ... the explicit dynamic casting interface and behaviour will change (improve!) in the future.

Python Training

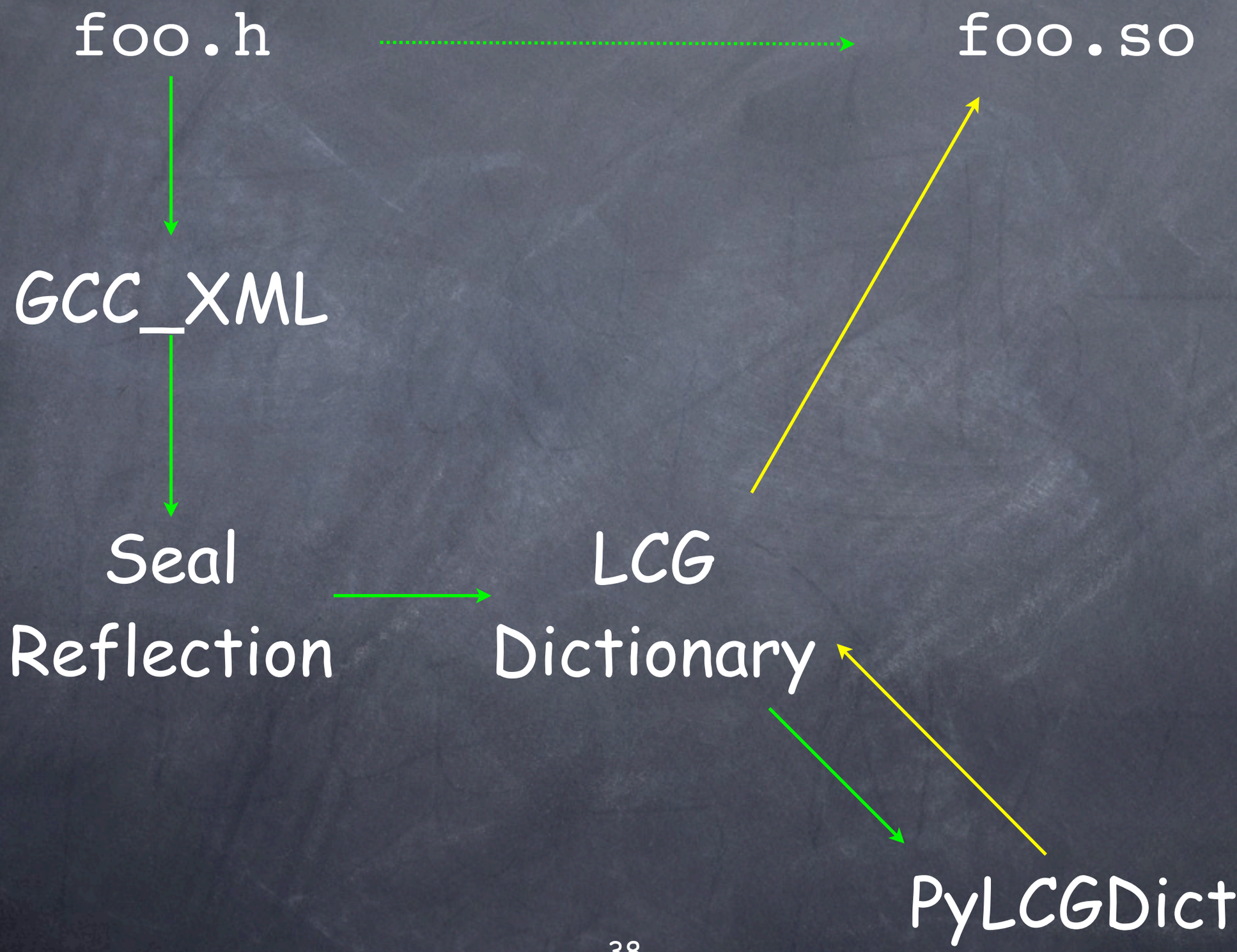
- Hands-on Introduction to Python Programming
  - Coondinated by CERN Technical Training
  - Very well received by students
- Physics Analysis in Python
  - In preparation
  - Should be available later this year

Short term future

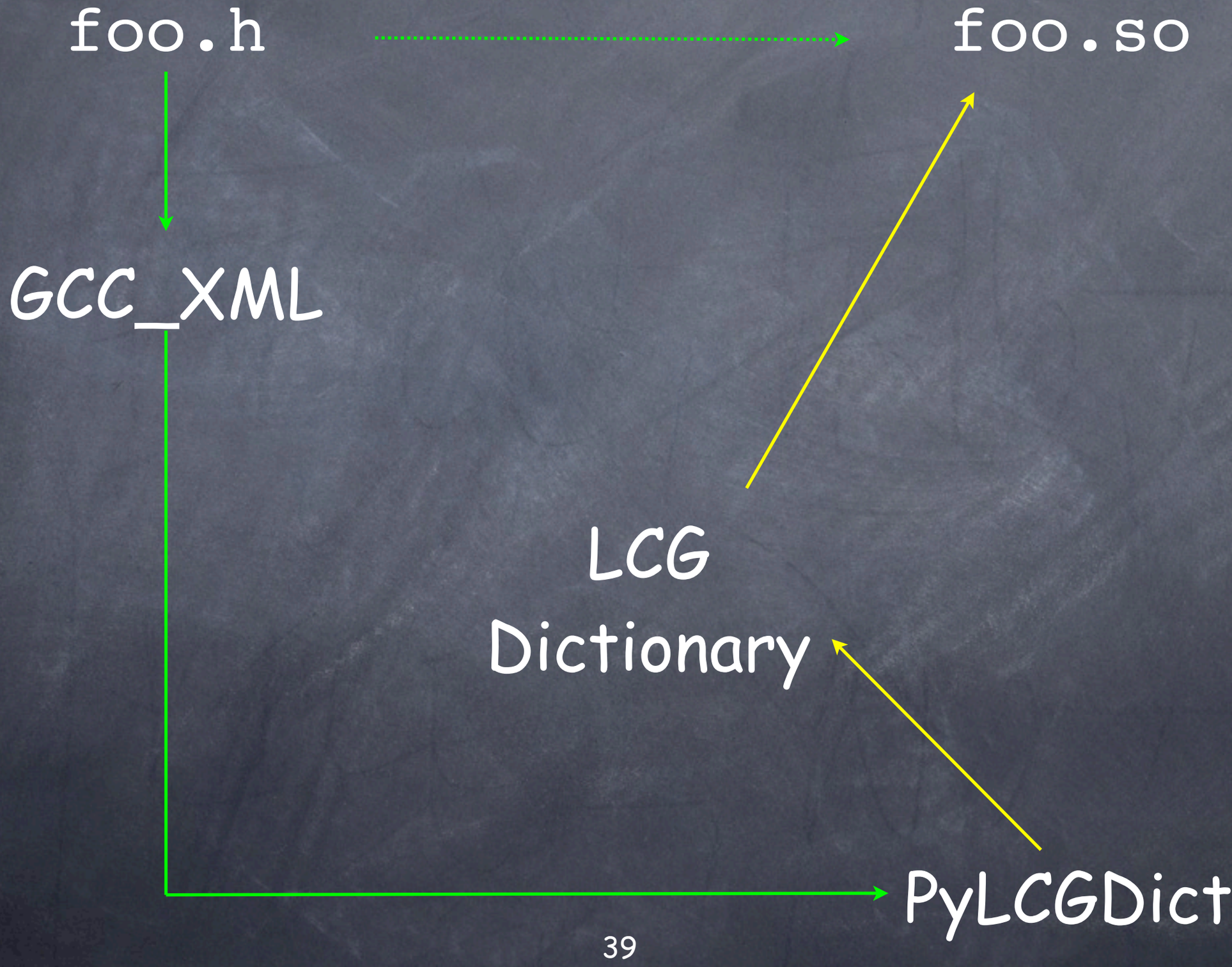
- Automate callback support
- Stop depending on Boost

Research

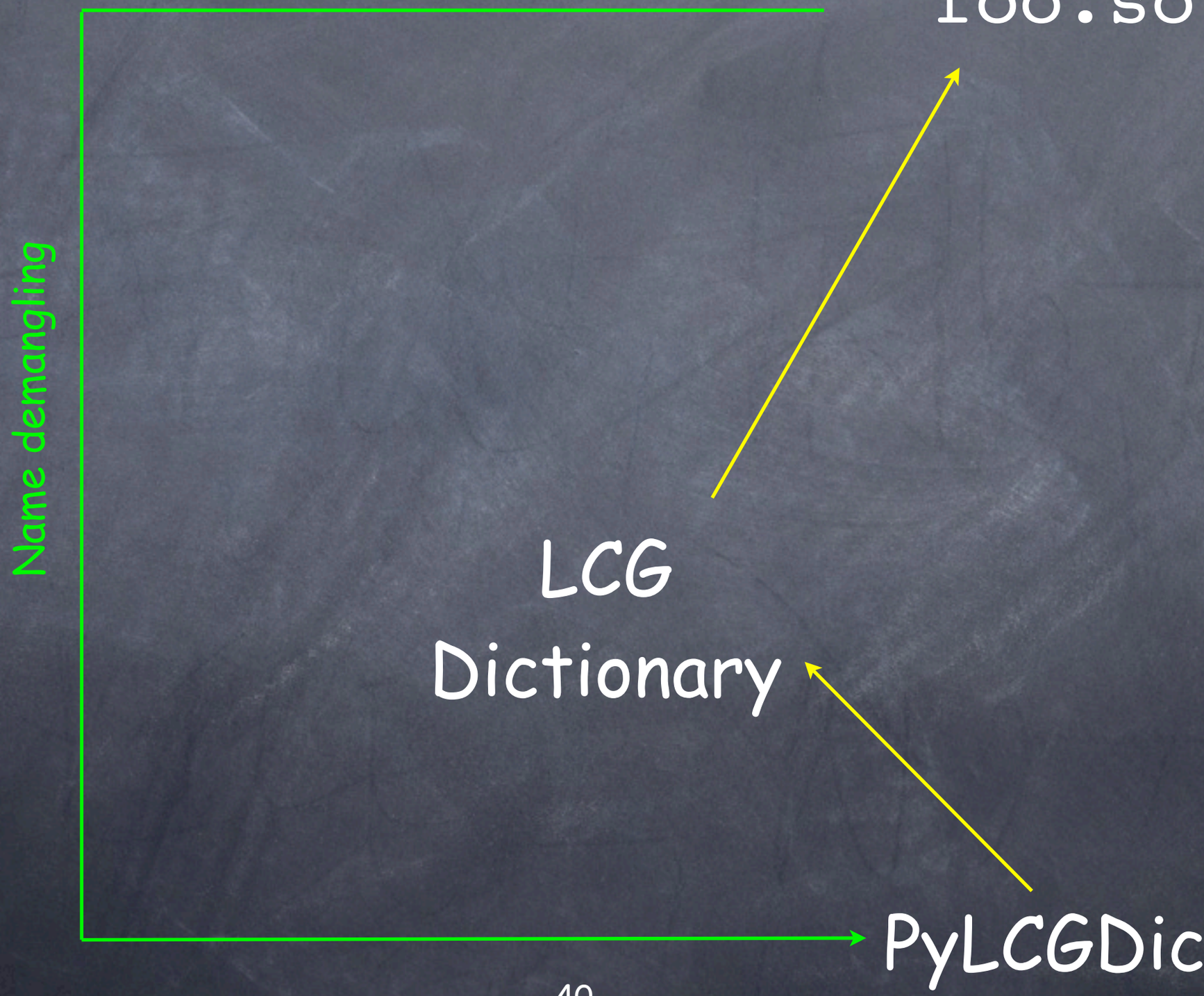
- Dynamic template instantiation
- Reduce dependence on other packages


foo.h
foo.so
GCC_XML
Seal
Reflection
LCG
Dictionary
PyLCGDict

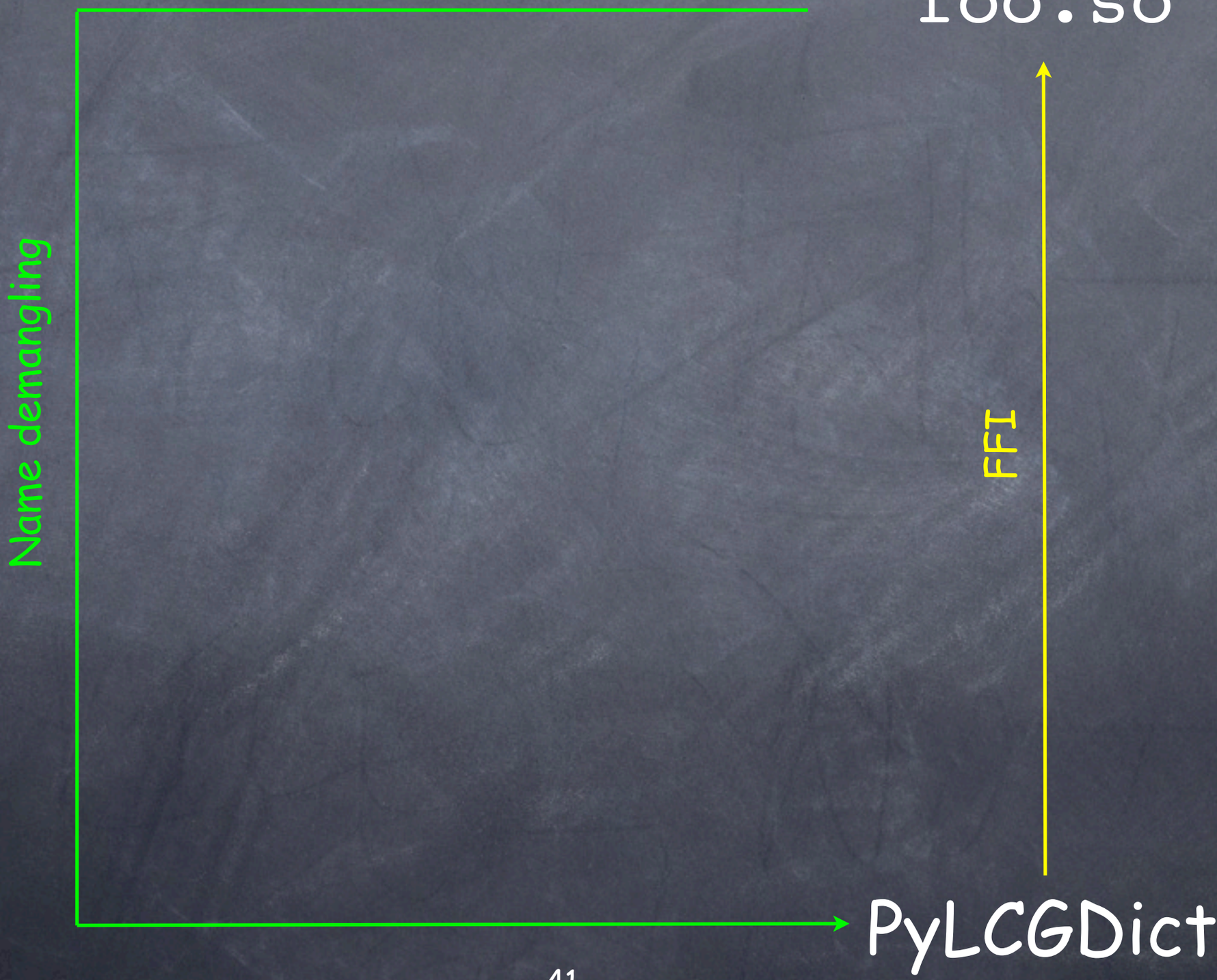
Remove Dictionary from front-end



Remove GCC_XML from front-end



Remove Dictionary from back-end



C'est tout