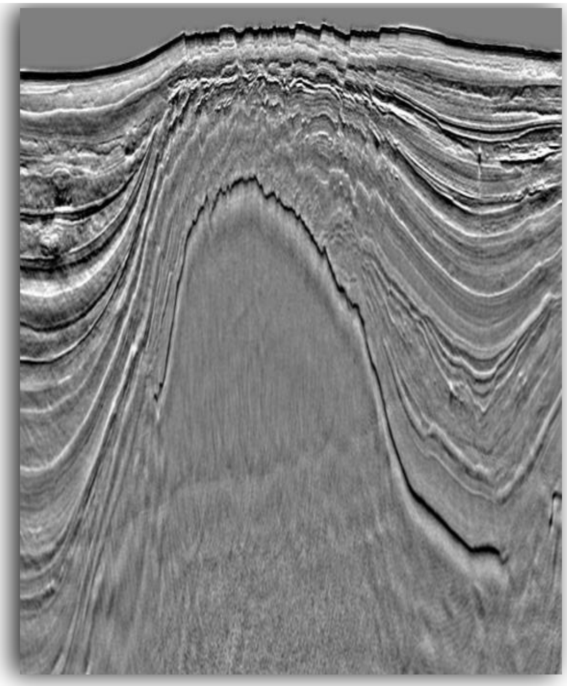
Tornado - scripting



D, Doledec
Liu, Yanhua

Houston

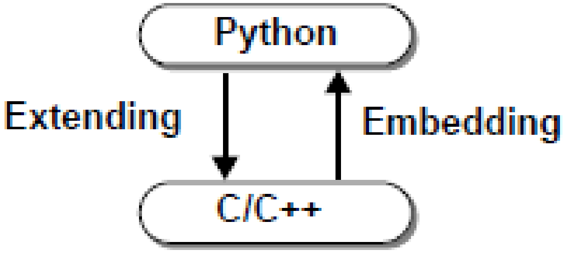
Passion for Geoscience

What Python Brings to C/C++

- **An interpreted high-level programming environment**
- Flexibility.
- Interactivity.
- Scripting.
- Debugging.
- Testing
- Rapid prototyping.
- **Component gluing**
- A common interface can be provided to different C/C++ libraries.
- C/C++ libraries become Python modules.
- Dynamic loading (use only what you need when you need it).
- **The best of both worlds**
- Performance of C
- The power of Python

Extending and Embedding Python

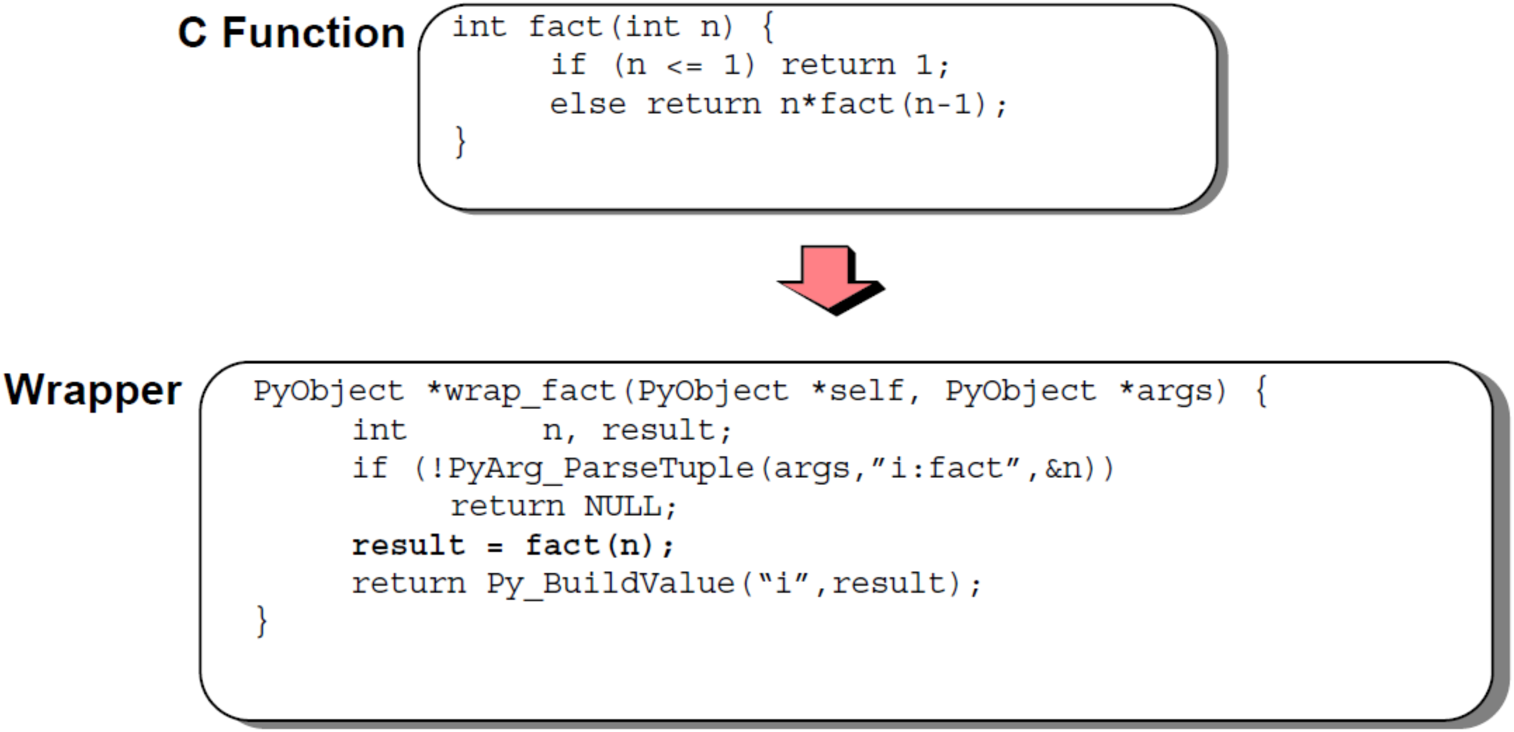
- There are two basic methods for integrating C/C++ with Python
  - Extension writing.
    - Python access to C/C++.
  - Embedding
    - C/C++ access to the Python interpr



We are primarily concerned with “Embedding”.

Writing Wrapper Functions

- “wrapper” functions are needed to access C/C++
  - Wrappers serve as a glue layer between languages.
  - Need to convert function arguments from Python to C++
  - Need to return results in a Python-friendly form



C Function

```
int fact(int n) {
     if (n <= 1) return 1;
     else return n*fact(n-1);
}
```

Wrapper

```
PyObject *wrap_fact(PyObject *self, PyObject *args) {
     int       n, result;
     if (!PyArg_ParseTuple(args,”i:fact”,&n))
          return NULL;
     result = fact(n);
     return Py_BuildValue(“i”,result);
}
```

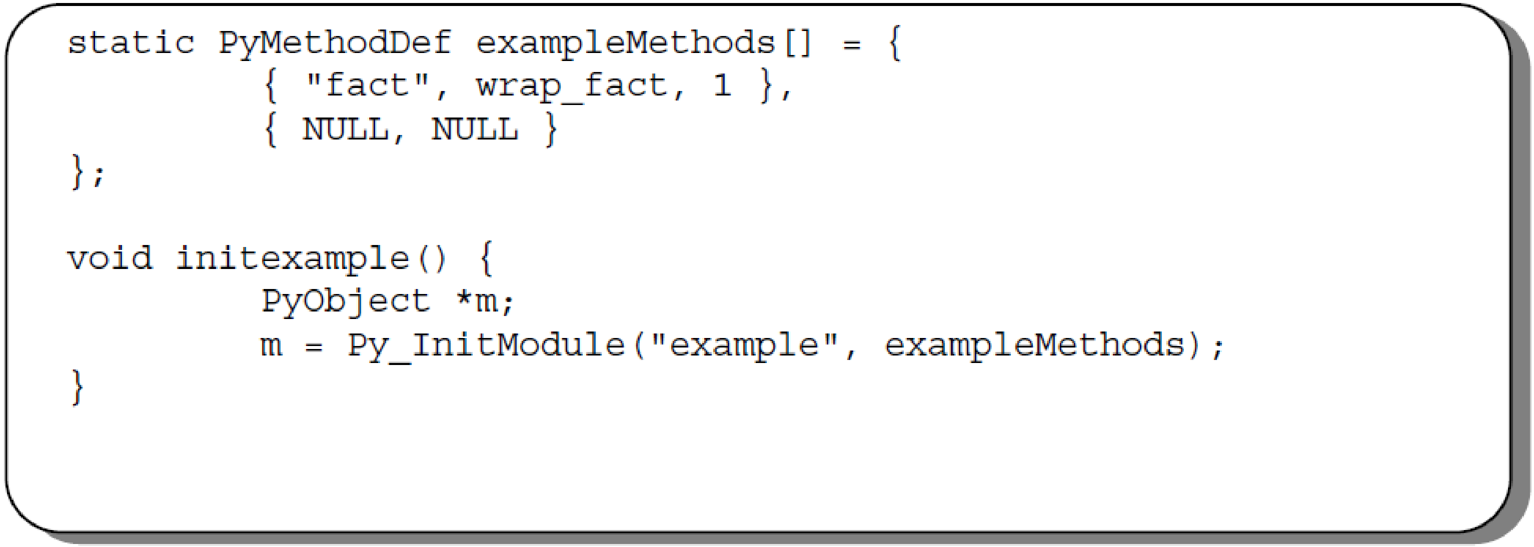
Conversion

- The conversion of data between Python and C is performed using two functions
  - int **PyArg_ParseTuple**(PyObject *args, char *format, ...)
  - PyObject ***Py_BuildValue**(char *format, ...)
- For each function, the format string contains conversion codes:
  - PyArg_ParseTuple(args,”iid”,&a,&b,&c); // Parse an int,int,double
  - Py_BuildValue(“d”,value); // Create a double

s = char *
i = int
l = long int
h = short int
c = char
f = float
d = double

Module

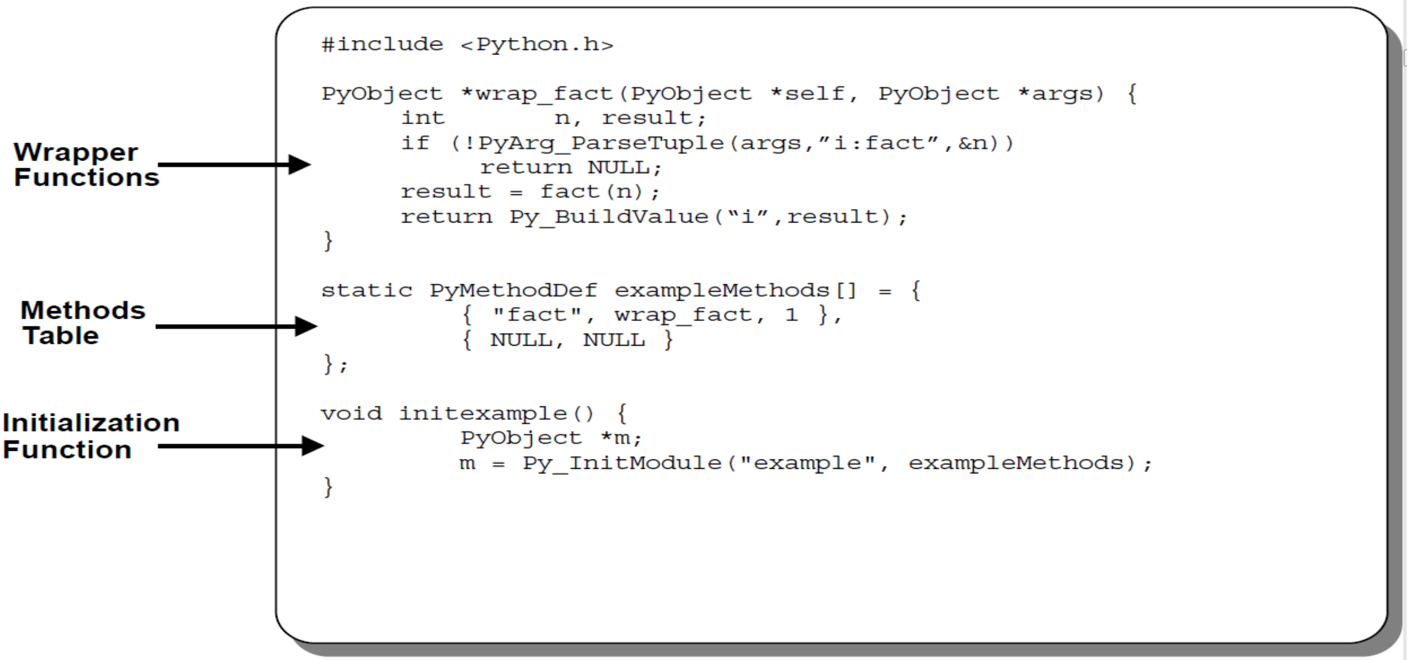
- **All extension modules need to register wrappers with Python**
  - An initialization function is called whenever you import an extension module.
  - The initialization function registers new methods with the Python interpreter.



```
static PyMethodDef exampleMethods[] = {
         { "fact", wrap_fact, 1 },
         { NULL, NULL }
};

void initexample() {
         PyObject *m;
         m = Py_InitModule("example", exampleMethods);
}
```

A complete example



Wrapper Functions

Methods Table

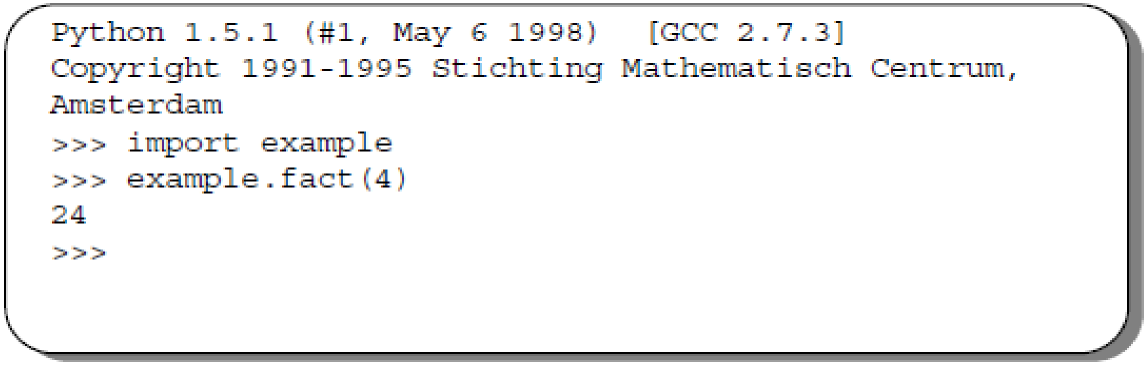
Initialization Function

```
#include <Python.h>

PyObject *wrap_fact(PyObject *self, PyObject *args) {
     int        n, result;
     if (!PyArg_ParseTuple(args,”i:fact”,&n))
          return NULL;
     result = fact(n);
     return Py_BuildValue(“i”,result);
}

static PyMethodDef exampleMethods[] = {
         { "fact", wrap_fact, 1 },
         { NULL, NULL }
};

void initexample() {
         PyObject *m;
         m = Py_InitModule("example", exampleMethods);
}
```



```
Python 1.5.1 (#1, May 6 1998)  [GCC 2.7.3]
Copyright 1991-1995 Stichting Mathematisch Centrum,
Amsterdam
>>> import example
>>> example.fact(4)
24
>>>
```

Python variables

- "Assignment" in Python
  - Variables are references to objects.

```
>>> a = [1,2,3]
>>> b = a
>>> b[1] = -10
>>> print a
[1, -10, 3]
```

- A C++ global variable is not a reference to an object, it is an object.
- To make a long story short, assignment in Python has a meaning that doesn’t translate to assignment of C global variables.

Pointers

- Pointer management is critical!
  - Arrays
  - Objects
  - Most C programs have tons of pointers floating around.
- The type-checked pointer model
  - C pointers are handled as opaque objects.
  - Encoded with type-information that is used to perform run-time checking.
  - Pointers to virtually any C/C++ object can be managed by SWIG.
- Advantages of the pointer model
  - Conceptually simple.
  - Avoids data representation issues (it’s not necessary to marshal objects between a Python and C representation).
  - Efficient (works with large C objects and is fast).
  - It is a good match for most C programs.

Pointer Encoding and Type Checking

- Pointer representation
  - Currently represented by Python strings with an address and type-signature.

```
>>> f = example.fopen("test","r")
>>> print f
  _f8e40a8_FILE_p
>>> buffer = example.malloc(8192)
>>> print buffer
  _1000afe0_void_p
>>>
```

  - Pointers are opaque so the precise Python representation doesn't matter much.

- Type errors result in Python exceptions

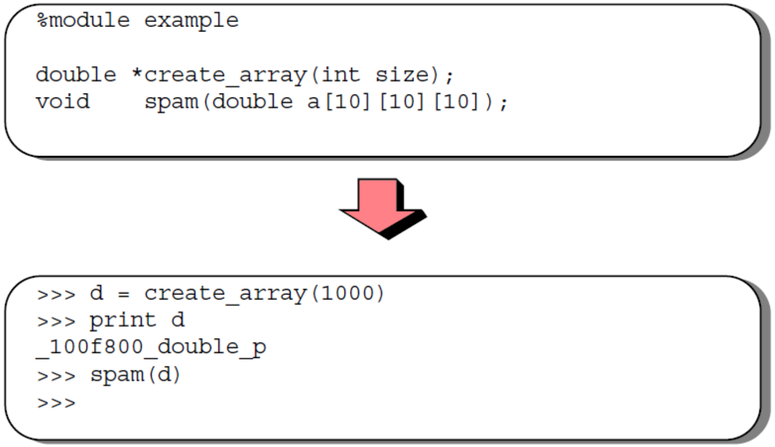
```
>>> example.fclose(buffer)
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: Type error in argument 1 of fclose. Expected _FILE_p.
>>>
```

- Type-checking prevents most of the common errors.
- Has proven to be extremely reliable in practice.

Array Handling

Arrays are pointers

- Same model used in C (the "value" of an array is a pointer to the first element).
- Multidimensional arrays are supported.
- There is no difference between an ordinary pointer and an array.
- However, not perform bounds or size checking.
- C arrays are not the same as Python lists or tuples!



```
%module example

double *create_array(int size);
void    spam(double a[10][10][10]);
```

```
>>> d = create_array(1000)
>>> print d
_100f800_double_p
>>> spam(d)
>>>
```

For array manipulation, we may need to check out the Numeric Python extension.

Manipulating Objects

- The pointer model
  - Most C/C++ programs pass objects around as pointers.
  - In many cases, writing wrappers and passing opaque pointers is enough.
  - However, in some cases you might want more than this.
- Issues
  - How do you create and destroy C/C++ objects in Python?
  - How do you access the internals of an object in Python?
  - How do you invoke C++ member functions from Python?
  - How do you work with objects in a mixed language environment?

Performance

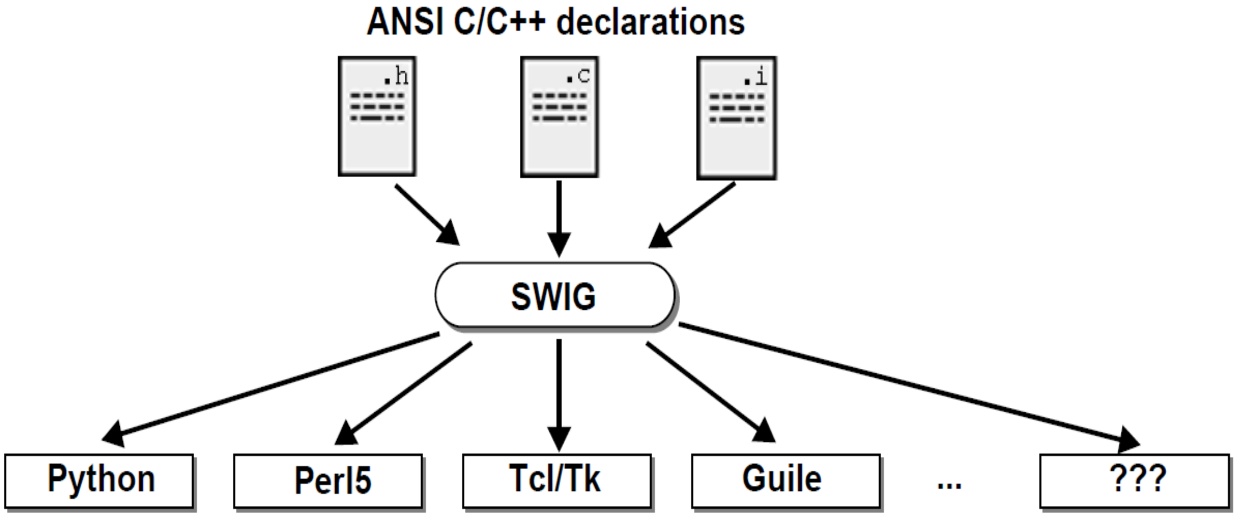
- Python introduces a performance penalty
  - Decoding
  - Dispatch
  - Execution of wrapper code
  - Returning results
- These tasks may require thousands of CPU cycles
- Rules of thumb
  - The performance penalty is small if your C/C++ functions do a lot of work.
  - If a function is rarely executed, who cares?
  - Don’t write inner loops or perform lots of fine-grained operations in Python.

Extension Building Tools

- **Stub Generators (e.g. Modulator)**
  - Generate wrapper function stubs and provide additional support code.
  - You are responsible for filling in the missing pieces and making the module work.
- **Dynamic python binding**
  - PyhonQt: offers an easy way to embed the Python scripting language into your C++ Qt applications (this not PyQt).
- **Automated tools (e.g. SWIG)**
  - Automatically generate Python interfaces from an interface specification.
  - Easy to use, but somewhat less flexible than hand-written extensions.
- **Distributed Objects (e.g. ILU)**
  - Concerned with sharing data and methods between languages
  - Distributed systems, CORBA, COM, ILU, etc...

SWIG (Simplified Wrapper and Interface Generator)

- A compiler that turns ANSI C/C++ declarations into scripting language interfaces.
- Completely automated (produces a fully working Python extension module).
- Language neutral. SWIG can also target Tcl, Perl, Guile, MATLAB, etc...
- Attempts to eliminate the tedium of writing extension modules.



Actual implementation in Tornado

- Wrapping
  - Direct wrapping (just link against python)
  - PythonQt
  - Boost (implementation not implemented yet)
- Run
  - script from command line

    tornado **-script** /<..>/copyHorizon.py **–nodisplay**
  - Dialog
    - Execute python file
    - Python console

Python (initialization)

```
vlPythonInterpreter::vlPythonInterpreter()
{
  const char* module="tornado";
  Py_SetProgramName(const_cast<char*>(module));
  Py_Initialize();

  Py_InitModule(const_cast<char*>(module), module_load);
}
```

```
static PyMethodDef module_load[] = {
  {"loadAttribute", pyLoadAttribute, METH_VARARGS, "Load Attribute in tornado"},
  {"loadSeismic", pyLoadSeismic, METH_VARARGS, "Load Seismic in tornado"},
  {"loadHorizon", pyLoadHorizon, METH_VARARGS, "Load Horizon in tornado"},
  {"saveHorizon", pySaveHorizon, METH_VARARGS, "Save Horizon in tornado"},
  {"loadGather", pyLoadGather, METH_VARARGS, "Load Gather in tornado"},
  {"interpHorizon", pyInterpHorizon, METH_VARARGS, "Linear interpolation of the
horizon"},
  {"showHrzPicking", pyShowHrzPicking, METH_VARARGS, "Open horizon picking
window"},
  {"rmvHorizon", pyRmvHorizon, METH_VARARGS, "Remove Horizon"},
  {NULL, NULL, 0, NULL}
};
```

Python (wrapping)

```
//load horizon
static PyObject* pyLoadHorizon(PyObject *self, PyObject *args)
{
  char * input;
  // parse arguments
  if (!PyArg_ParseTuple(args, "s", &input)) {
    return NULL;
  }
  // run the actual function
  long res=(long)vlPythonAPIWrapper::getInstance()->hrzLoad(input);

  // build the resulting string into a Python object.
  return Py_BuildValue("l", res);
}
```

```
//load horizon
static PyObject* pyLoadHorizon(PyObject *self, PyObject *args)
{
  char * input;
  // parse arguments
  if (!PyArg_ParseTuple(args, "s", &input)) {
    return NULL;
  }
  // run the actual function
  long res=(long)vlPythonAPIWrapper::getInstance()->hrzLoad(input);

  // build the resulting string into a Python object.
  return Py_BuildValue("l", res);
}
```

Python (code)

```
=========== Python demo of the horizon interpolation
import tornado

help(tornado)
dir(tornado)

print tornado.loadHorizon("/data2/devtest/tornado/yanhliu/test.hrz")
tornado.showHrzPicking()
tornado.interpHorizon()
print "test load"
```

PythonQt (initialization)

```
vlPythonInterpreter::vlPythonInterpreter() {

    PythonQt::init(PythonQt::IgnoreSiteModule | PythonQt::RedirectStdOut);

    PythonQt::self()->registerCPPClass("HorizonGroup", "","horizon",

PythonQtCreateObject<HorizonGroupWrapper>);
    PythonQt::self()->registerCPPClass("Horizon", "","horizon",
                                                            PythonQtCreateObject<HorizonWrapper>);
}
```

Python (wrapping)

```
class HorizonGroupWrapper : public QObject {
   Q_OBJECT
public slots:
   // add a constructor
   vgHorizonGroup* new_HorizonGroup(const QString& first);
   // add a destructor
   void delete_HorizonGroup(vgHorizonGroup* o);

   /** Load/Save a group of horizons from a horizon file */
   bool load(vgHorizonGroup* o, const char* path);

   /** Save a group of horizons from a horizon file. format is for binary version only */
   bool save(vgHorizonGroup* o, const char* path);

   // add access methods
   QString getName(vgHorizonGroup* o);
};
```

```
// Constructor
vgHorizonGroup* HorizonGroupWrapper::new_HorizonGroup(const QString& first)
{
   vgHorizonGroup *hg = new vgHorizonGroup;
   return hg;
}
// A method
bool HorizonGroupWrapper::load(vgHorizonGroup* hg, const char* path)
{
   return hg->load(path);
}
```

Ptython (code)

```
=== The python (copy a horizon) ====
from PythonQt.horizon import *

# create a new object
hg = HorizonGroup("Horizon group")

# print the object (to see how it is wrapped)
print hg

# print the methods available
print dir(hg)

hg.load("/s0/scr/doledec/FromYanhliu.hrz")
hg.save("/s0/scr/doledec/FromYanhliu2.hrz")

parameters = horizon.algorithms.Parameters;
parameters.setMethod(1)
parameters.setXXX(xxx)
<….>
hg_interpolated= horizon.algorithms.interpolate(hg, parameters)
```

SWIG - analysis

- Cons
  - Executables
    - Version exist on db7 but do not works (dependencies)
    - Need to recompile (!! version of python)
- Pros
  - No dependencies
  - Simple to add in compilation chain

PythonQt

- Cons
  - Libraries
    - Recompilation (static) against tornado lib (qt)
- Pros