Scientific Computing (Phys 2109/Ast 3100H)
I. Scientfic Software Development

SciNet HPC Consortium

University of Toronto

Winter 2013

Part I

Introduction to Software Development

Lecture 8

Course project

Object-Oriented Programming in Python

Mixing C++ and Python

Course project: Where are we going?

- Two dimensional diffusion equation for density field $\rho(\mathbf{r},\mathbf{t})$

$$\frac{\partial \rho}{\partial t} = D\left(\frac{\partial^2 \rho}{\partial x^2} + \frac{\partial^2 \rho}{\partial y^2}\right).$$

- Tracer particle satisfies ODE

$$m\ddot{R} = F - \alpha(\rho(R))\dot{R},$$

where $\mathbf{m}$ is mass, $\mathbf{F}$ is force and α is viscosity.

- Ad hoc form for density dependent friction constant α:

$$\alpha(\rho) = \alpha_0(1 + a\rho).$$

- Ad hoc form for force, like a constant electric field:

$$F = qE\hat{x}.$$

SciNet compute • calcul CANADA

Course project: Where are we going?

- Periodic boundary conditions in all directions

$$\mathbf{r} \sim \mathbf{r} + \mathbf{L}(\mathbf{k}\hat{\mathbf{x}} + \mathbf{l}\hat{\mathbf{y}}).$$

  L is the length of the side of the box, **k** and **l** integer.

- Initial conditions:

$$\rho(\mathbf{r}) = \exp\left[-\frac{\|\mathbf{r}\|^2}{2\sigma_0^2}\right],$$

$$\mathbf{R}(0) = \mathbf{R}_0,$$

$$\dot{\mathbf{R}}(0) = \mathbf{V}_0.$$

- Values for the parameters (arbitrary) are

$$\mathbf{D} = 1;\ \mathbf{m} = 1;\ \alpha_0 = 1;\ \mathbf{a} = 15;\ \mathbf{qE} = 1;\ \mathbf{L} = 10;$$
$$\sigma_0 = 1;\ \mathbf{R}_0 = 0;\ \mathbf{V}_0 = 10\hat{\mathbf{y}}$$

Classes in Python

- As in C++, Python uses classes to group together data and code, accessing them with '.' operator
- We could also do this with modules. But there can be only one instance of a module, and many instances of a class.
- Inheritance: multiple base classes, derived class can override any methods of its base class or classes, and method can call a base class method with the same name.
- Objects can contain arbitrary amounts and kinds of data.
- Classes partake of the dynamic nature of Python: created at runtime, and can be modified further after creation.

Easy cases

Source 1

```
class Apple:
   type = "Delicious"
   colour = "Green"
```

- Collection of variables
- Source 1: `apple1` and `apple2` share `colour` (class variable); tricky.
- Source 2: works, but now we have to assign each member.
- Anything more workable requires writing a constructor.

Easy cases

- Collection of variables
- Source 1: apple1 and apple2 share colour (class variable); tricky.
- Source 2: works, but now we have to assign each member.
- Anything more workable requires writing a constructor.

Source 1

```
class Apple:
    type = "Delicious"
    colour = "Green"

apple1 = Apple()
apple2 = Apple()
Apple.colour = "Golden"
print apple1.colour
[Golden]
```

SciNet compute • calcul CANADA

Easy cases

- Collection of variables
- Source 1: `apple1` and `apple2` share `colour` (class variable); tricky.
- Source 2: works, but now we have to assign each member.
- Anything more workable requires writing a constructor.

Source 1

```
class Apple:
   type = "Delicious"
   colour = "Green"

apple1 = Apple()
apple2 = Apple()
Apple.colour = "Golden"
print apple1.colour
[Golden]
```

Source 2

```
class Apple:
   pass
```

Easy cases

- Collection of variables
- Source 1: `apple1` and `apple2` share `colour` (class variable); tricky.
- Source 2: works, but now we have to assign each member.
- Anything more workable requires writing a constructor.

Source 1

```
class Apple:
   type = "Delicious"
   colour = "Green"
apple1 = Apple()
apple2 = Apple()
Apple.colour = "Golden"
print apple1.colour
[Golden]
```

Source 2

```
class Apple:
   pass
apple1 = Apple()
apple1.type = "Delicious"
apple1.colour = "Green"
apple2 = Apple()
apple2.type = "Delicious"
apple2.colour = "Golden"
print apple1.colour
[Green]
```

Using a constructor

- Collection of variables
- Same `def` keyword to define methods.
- Constructor name is `__init__`

```
class Apple:
   def __init__(self):
      self.type = "Delicious"
      self.colour = "Green"
apple1 = Apple()
apple2 = Apple()
print apple1.colour
[Green]
```

Class syntax in Python

- Methods take a first argument that is an instance of the class
- This argument is explicit (`self`) in definition but implicit in calls.
- In methods, refer to member fields as `self.field`.
- No separation interface/implementation

```
class Apple:
    def __init__(self):
        self.type = "Delicious"
        self.colour = "Green"
    def describe(self):
        print self.type,
        print self.colour

apple1 = Apple()
apple2 = Apple()
print apple1.colour
[Green]
apple1.describe()
[Delicious Green]
```

More special methods

- `__del__`
  A kind of destructor.
- `__str__`
  Converts object to a string for output. Used by print.
  Intended to be readable by users.
- `__repr__`
  Returns a string representation for the object. Used by python (e.g., if you just type the name of an object).
  Intended to be understandable by developers.

Example: Tracer Particle

```
class Tracer:
   def __init__(self,x0,y0,vx0,vy0):
      self.t = 0.0
      self.x = x0
      self.y = y0
      self.vx = vx0
      self.vy = vy0
   def timeStep(self,dt):
      self.t += dt
      self.x += d*self.vx
      self.y += d*self.vy
   def write(self):
      print self.t, self.x, self.y
```

```
tr = Tracer(0.0,1.0,-1.0,2.0)
while tr.t < 10.0:
   tr.timeStep(0.1);
   tr.write()
```

Inheritance in Python

- Need to discuss this for completeness' sake
- Put classes to derive from between parenthesis.

- Two kinds of classes: old and new style
- For multiple inheritance and operator overloading.
- To get new style, inherit from `object` class

Inheritance

```
class NamedTracer(Tracer):
   def __init__(self,a,b,c,d,name):
      Tracer.__init__(self,a,b,c,d)
      self.name = name

t = NamedTracer(1.,2.,-1.,0.,"Al")
```

Inheritance in Python

- Need to discuss this for completeness' sake
- Put classes to derive from between parenthesis.

- Two kinds of classes: old and new style
- For multiple inheritance and operator overloading.
- To get new style, inherit from `object` class

Inheritance

```
class NamedTracer(Tracer):
   def __init__(self,a,b,c,d,name):
      Tracer.__init__(self,a,b,c,d)
      self.name = name

t = NamedTracer(1.,2.,-1.,0.,"Al")
```

New style class

```
class Tracer(object):
   #...
```

SciNet compute • calcul CANADA

Mixing C++ and Python

Mixing C++ and Python

- Python is versatile and quick to write in
- C++ is fast
- Let's combine them: best of both worlds
- Ideally:
  - Have a blazingly fast module in C++
  - Compile it
  - Import it into Python, and start playing
  - Can then write test and driver code in Python
- Simple idea. Implementation is a harder. And non-unique.

Why isn't this straightforward?

- Objects in Python very different from objects in C/C++
- The Python C-API exposes all the nitty gritty of making Python work.
- A .o file is not a Python module.
- C++ compiler and Python have to be binary compatible.

What's involved in getting this to work?

- Need to create a 'Python extension module'
- That module needs to load a dynamic library (if it isn't a dynamic library itself).
- So we need to build a dynamic library from the C++ code.
- And we create a Python extension module.
- For which we'll have to write some wrapper code.

Many automating frameworks. . .

- Python C-API
- SWIG
- Boost.Python
- Cython
- . . .

This is getting hairy... and yet somehow this is very popular.

Boost Python

- Boost: large collection of useful c++ libraries. (so useful that some parts have made it into the next c++ standard)
- Boost Python: framework for interfacing Python and C++.
- C++ specific, but same issues for other interfaces.
- Should be able to translate a C++ class structure into a python class structure.
- Boost likes the bjam automated build systems, but we can just use g++ or make.

Boost Python - example 1

- C++ code:

```
//hi.h
#ifndef HIH
#define HIH
char const* greet();
#endif
```

```
//hi.cc
#include "hi.h"
char const* greet() {
   return "hi world";
}
```

- Python code:

```
#usehi.py
import hi
print hi.greet()
```

Boost Python - example 1

- C++ code:

```
//hi.cc
#include "hi.h"
char const* greet() {
   return "hi world";
}
```

Boost Python - example 1

- C++ code:

```
//hi.cc
#include "hi.h"
char const* greet() {
   return "hi world";
}
```

- Step 1: Write glue code for extension module:

```
//hix.cc
#include "hi.h"
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(hi) {
   using namespace boost::python;
   def("greet", greet);
}
```

Boost Python - example 1

- C++ code:

```
//hi.cc
#include "hi.h"
char const* greet() {
   return "hi world";
}
```

- Step 1: Write glue code for extension module:

```
//hix.cc
#include "hi.h"
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(hi) {
   using namespace boost::python;
   def("greet", greet);
}
```

- Step 2: Compile

```
$ g++ -c hi.cc  -fPIC
$ g++ -c hix.cc -fPIC -I/usr/include/python2.7
```

Boost Python - example 1

- C++ code:

```
//hi.cc
#include "hi.h"
char const* greet() {
   return "hi world";
}
```

- The glue code:

```
//hix.cc
#include "hi.h"
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(hi) {
   using namespace boost::python;
   def("greet", greet);
}
```

- Step 3: Create a dynamically loadable library

```
$ g++ -o hi.so hi.o hix.o -shared -lboost_python
```

Boost Python - example 1

- C++ code:

```
char const* greet() {
   return "hi world";
}
```

- Python code:

```
#usehi.py
import hi
print hi.greet()
```

- Step 4: Use it

```
$ python usehi.py
hi world
```

Boost Python - example 2

- C++ code:

```
//tracer.h
#ifndef TRACERH
#define TRACERH
class Tracer {
   public:
      Tracer(float x, float y, float vx, float vy);
      void timeStep(float dt);
      void write();
      float t;
      ...
};
#endif
```

Boost Python - example 2

- C++ code:

```
//tracer.h
#ifndef TRACERH
#define TRACERH
class Tracer {
   public:
      Tracer(float x, float y, float vx, float vy);
      void timeStep(float dt);
      void write();
      float t;
      ...
};
#endif
```

- Python code:

```
#usetracer.py
from tracer import Tracer
tr = Tracer(0.0,1.0,-1.0,2.0)
while tr.t < 10.0:
   tr.timeStep(0.1)
   tr.write()
```

Boost Python - example 2

- C++ code:

```
class Tracer {
   public:
      Tracer(float x, float y, float vx, float vy);
      void timeStep(float dt);
      void write();
      float t;
};
```

SciNet compute • calcul CANADA

Boost Python - example 2

- C++ code:

```
class Tracer {
   public:
      Tracer(float x, float y, float vx, float vy);
      void timeStep(float dt);
      void write();
      float t;
};
```

- Glue code:

```
#include "tracer.h"
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(tracer) {
   using namespace boost::python;
   class_<Tracer>
      ("Tracer", init<float ,float ,float ,float
      >()) .def("timeStep",&Tracer::timeStep)
      .def("write",&Tracer::write)
      .def_readonly("t", &Tracer::t);
}
```

Compile and use...

```
$ g++ tracer.cc -fPIC
$ g++ tracerx.cc -fPIC -I/usr/inlude/python2.7
$ g++ -o tracer.so tracer.o tracerx.o -shared -lboost_python
$ python
>>> from tracer import Tracer
>>> tr = Tracer(0.0, 1.0, -1.0, 2.0)
>>> while tr.t < 10.0:
...     tr.timeStep(0.1)
...     tr.write()
...
```

Good as long as it works. . .

- One wrapper, one .so
- Need to remember -fPIC
- If there's something wrong, hard to figure out where.
- Some things are still hard in Boost Python, such as passing back numpy arrays.
- Still requires substantial amount of glue.
- Other approaches may need less glue at first (SWIG), but if you want anything that is not yet automated, you are still glueing.

Scientific software development:
What have we learned?

Recap Part I of Scientific Computing

- Choose the tools for the jobs: C++ for performance, python for flexibility, fast development, and visualization.
- Version control
- Modular programming
  - header files/implementation files
  - make
  - object-oriented programming
- Defensive programming (assert)
- Unit testing
- Debugging