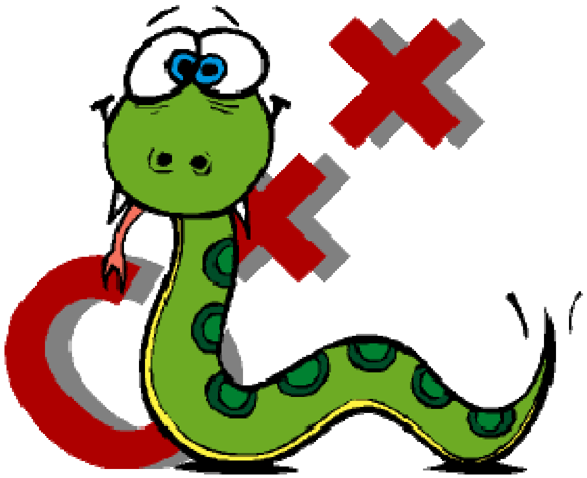
How to use CGAL-Python?

from CGAL import *

Naceur MESKINI.
GEOMETRICA.
INRIA Sophia Antipolis.

Overview:



- Introduction to Python:
  - Philosophy of Python.
  - What we should know to start using Python.
- CGAL-Python:
  - Modules currently in CGAL-Python.
  - How to use it?
  - Examples.
  - Getting help?
  - Doing a simple demo.
- How to expose the CGAL code to python?
  - Using Pyste.
  - Using Boost-Python.
  - How to deal with the C++ iterators and circulators?
  - How to deal with functions that takes references as parameters?

✓ Introduction to Python

- Philosophy of Python:
  - Is an interpreted programming language.
  - Is object oriented.
  - Python is fully dynamically typed.
  - Python uses automatic memory management.
  - Python uses indentation, rather than braces, to delimit blocks.

- The Python programming language is actively used in industry and academia for a wide variety of purposes.

What we should know to start using Python.

- Basic data types.
- Functions
- Classes
- Modules.

What we should know to start using Python.

- Basic data types:
  - Python Supports:
    - integers: 0, 1, 2, 3, -1, -2, -3
    - floats: 0., 3.1415926, -2.05e30, 1e-4
    - strings: s = 'abcd' or "abcd"
    - lists: l = [1, 3.3, Point_2(),…]
    - dictionaries (associative arrays): vec = {key1:value1,key2:value2, …..}
    - …

What we should know to start using Python.

- **Lists:**

Lists are essentially heterogeneous arrays

- declaration & initialization:

```
>>> days = [ 'mon' ,  'tue', 'wed',  'thu',  'fri']
```

- Individual elements are accessed using the normal square bracket format, with indices starting at 0:

```
>>> days[2]
'wed'
```

- looping over a list using “for statement”:

```
for d in days:
    print d
```

- len(listname) → returns length of list
- listname.append(item) → inserts data at the end of list
- listname.sort() → sorts the list in place in ascending order
- listname.reverse() → reverses the list order

What we should know to start using Python.

- Dictionaries :

Associative arrays with keys and values

```
>>>  week = { 1: 'mon',2:'tue',3:'wed',4:'thu',5:'fri'}
>>> week.keys()                       → returns a list of keys
[1,2,3,4,5]
>>> week.values()                     → returns a list of values.
['mon', 'tue', 'wed', 'thu', 'fri']
>>> week.has_key(2)
True
```

- Looping through dictionaries:

```
>>> for i in week:
        print i ,week[i]
1 mon
2 tue
3 wed
4 thu
5 fri
```

What we should know to start using Python.

- Functions:

```
>>> def addition(x,y):
        return x+y

>>> addition(2,3)
>>> 5
>>> addition('nom','prenom')
>>> 'nomprenom'
>>> addition(1.25,5.36)
>>> 6.6100000000000003
```

What we should know to start using Python.

➢ Classes :

```
>>> class Complex:
...         def __init__(self, realpart, imagpart):
...             self.r = realpart
...             self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

← class instantiation uses function notation.

- Inheritance:

The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    <statement-N>
```

What we should know to start using Python.

- Modules :

A module is a file containing definitions of functions and classes.

- Create your own module:

```
def addition(x,y):
    # do something
def function_2():
    # do something else
class Complex():
    …..
```

my_module.py

What we should know to start using Python.

- Modules :
  - To use our module we have to import it:

```
>>> import my_module
>>> v = my_module.addition(2,4)
>>> c = my_module.Complex()
>>> from my_module import addition      ( To import a part of my_module )
>>> from my_module import *             ( To import the whole module )
>>> import my_module as m       (the safe one )
>>> m.addition(3,4)
```

- Resources :

  - Official Python homepage & documentation
    www.python.org

  - Good, brief online overview
    http://www.hetland.org/python/instant-python.php

  - A software for mathematics, science, and engineering.
    www.SciPy.org

✓ Introduction to CGAL-Python:

- CGAL-Python is:
  - A collection of modules:
    - Each module corresponds to one package of CGAL.

  - Currently CGAL-Python contains the following modules:
    - CGAL.Kernel:
      - Point_2 , Point_3 ….
    - CGAL.Triangulations_2:
      - Triangulation_2, Delaunay_triangulation_2, Constrained_triangulation_2 …
    - CGAL. Triangulations_3:
      - Triangulation_3, Delaunay_triangulation_3
    - CGAL.Mesh_2
    - CGAL. Geometric_Optimisation:
      - Min_annulus_2, Min_annulus_3, Min_circle_2, Min_ellipse_2 , Min_sphere_2, Min_sphere_3
    - CGAL.Alpha_shapes_2(3)
    - CGAL.Polyhedron
    - CGAL.Convex_hull_2

How to use it?:

- To use CGAL as Python package you have to:
  - Import the module you expect to use by typing :
    - from CGAL import *   # this will import the whole CGAL.
    - from CGAL.Kernel import *   # you have to import this module.
- Example:

In Python

```
from CGAL.Triangulations_2 import *
from CGAL.Kernel import Point_2
dt = Triangulation_2()
Pts = [Point_2(0,0),Point_2(1,0),Point_2(0,1),Point_2(1,1)]
for p in Pts:
    dt.insert(p)
print dt.number_of_vertices()
```

In C++

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_2.h>
struct K : CGAL::Exact_predicates_inexact_constructions_kernel {};
typedef CGAL::Triangulation_2<K>        Triangulation;
typedef K::Point_2            Point_2;
int main() {
        Point_2
        Pts[]={Point_2(0,0),Point_2(1,0),Point_2(0,1),Point_2(1,1)};
        Triangulation t;
        for (int i = 0; i < 4 ; i++ ) t.insert(Pts[i]);
        std::cout<< t.number_of_vertices() << std::endl;
        return 0;
}
```

Example:

```
from CGAL.Kernel import *
from CGAL.Triangulations_2 import *
from random import *
dt = Delaunay_triangulation_2()
vert = {}
for i in range(20):
   vert[i] = dt.insert(Point_2(random(),random()))
# the finite vertices:
for v in dt.vertices:
   print v.point()
# the finite edges
for e in dt.edges:
   print e.vertex()
# the finite faces
for f in dt.faces:
     for i in range(3):
          print f.vertex(i).point()
# compute the finite faces incident to Vertex = vert[0]
finites_faces = []
f1 = cir_faces.next()
if dt.is_infinite(f1) == False:
   finites_faces.append(f1)
for f in cir_faces:
   if f == f1:
      break
   finites_faces.append(f)
```

Getting help:

- From the web site of cgal-python
  - http://cgal-python.gforge.inria.fr/
- By typing help(class_name)

Doing a simple demo:

- In this demo we will use the following:
  - Classes:
    - dt = Delaunay_triangulation_2()
    - Point_2
    - Ray_2
    - Segment_2
  - Functions:
    - dt.insert()
    - dt.remove()
    - dt.nearest_vertex()
    - dt.clear()
    - dt.line_walk()
    - dt.triangle()
    - dt.segment()
  - Iterators:
    - dt.edges
    - dt.faces
    - dt.points

How to expose the CGAL code to Python?

- Using Pyste:
  - Pyste is a Boost.Python code generator.
  - The user specifies the classes and functions to be exported using a simple interface file.
  - Pyste uses GCCXML to parse all the headers and extract the necessary information to automatically generate C++ code.

interface_file.pyste:

```
Triangulation_2 = template(CGAL::Triangulation_2, ``triangulation.h``)
Triangulation_2([K TDS],`` Triangulation_2 ``)
Bbox_2 = class(CGAL::Bbox_2, ``Bbox_2.h``)
…..
```

The command:

python pyste.py –module = my_wrapper interface_file.pyste ⇨ my_wrapper.cpp

How to expose the CGAL code to Python?

- Using Boost-Python.
  - Step 1: writing a corresponding Boost.Python C++ Wrapper.

```
template < class kernel>
void Py_Delaunay_triangulation_2()
{
      typedef   ……       Tds;
      typedef CGAL::Triangulation_2<kernel,Tds>                                  Triangulation_2;
      typedef CGAL::Delaunay_triangulation_2<kernel,Tds>            Delaunay_triangulation_2;

   class_< Delaunay_triangulation_2, bases< Triangulation_2 >  >("Delaunay_triangulation_2", init< >())
      .def(init< const Delaunay_triangulation_2& >())
      .def("is_valid", &Delaunay_triangulation_2::is_valid, is_valid_overloads_0_2())
      .def("nearest_vertex", &Delaunay_triangulation_2::nearest_vertex,nearest_vertex_overloads_1_2())
  ;
}
```

How to expose the CGAL code to Python?

```
void export_Delaunay_triangulation_2()
{
     #define CGAL_KERNEL(Type)          \
    {                                   \
         Py_Delaunay_triangulation_2<Type>();     \
    }
    #include <include/Kernel.h>
}
```

- Step 2: building our module as a shared library.

```
extern void export_Delaunay_triangulation_2();
BOOST_PYTHON_MODULE(Triangulations_2)
{
     export_Delaunay_triangulation_2();
     export_Constrained_triangulation_2();
     export_Constrained_Delaunay_triangulation_2();
}
```

Triangulations_2_module.cpp → make → Triangulations_2.so (.dll)

How to deal with iterators and circulators:

points_begin ()

points_end()

→ Point_iterator

1

```
template<class Iterator>
class simple_python_iterator
{
     public:
   simple_python_iterator(std::pair<Iterator, Iterator> p)
   : orig_first(p.first), first(p.first), last(p.second) { }

  typename std::iterator_traits<Iterator>::value_type next()
  {
      using boost::python::objects::stop_iteration_error;
      if (first == last) stop_iteration_error();
      return *first++;
  }
    private:
   Iterator orig_first,first,last;
}
```

2

```
template<class Iterator,class Triangulation >
simple_python_iterator<iterator> py_points(const Triangulation& dt)
{
     std::pair< iterator , iterator > p( dt.points_begin(), dt.points_end() );
     return simple_python_iterator<iterator>(p);
}
```

↓

3

```
class_< Triangulation_2>(" Triangulation_2 ", init< >())
      .def("points", &py_points< Point_iterator, Triangulation_2 >)

=====================

Typedef simple_python_iterator< Point_iterator > Point_iterator ;

class_< Point_iterator  >(" Point_iterator ", no_init< >())
      .def("next", & Point_iterator::next())
```

What about the Output iterators?

In C++

get_conflicts(..) ⇨ *OutputItFaces*

In Python, we will return a python list.

```
template <class Delaunay_triangulation_2,class Point_2>
boost::python::list py_get_conflicts(const Delaunay_triangulation_2&
dt,Point_2 pt)
{
    boost::python::list result;
    typedef typename Delaunay_triangulation_2::Face_handle  Face_handle;
    std::list<Face_handle>  liste;
    dt.get_conflicts(pt, std::back_inserter(liste));
    typename std::list<Face_handle>::iterator iter;

   for(iter = liste.begin(); iter != liste.end(); iter++)
   result.append(*iter);

   return result;
}
```

How to deal with functions that takes references as parameters?

- python doesn't support native-type (float, int, string, …) reference.

locate(Point query,Locate_type & lt, **int** & li) ⇒ py_locate(Point query, boost::python::list l)

is_edge(Vertex_handle va,Vertex_handle vb,Face_handle& fr,**int** & i)

⇒ py_is_edge(Vertex_handle va ,Vertex_handle vb, Edge& e)

Thank you.