ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

Module definition

the identifier `module_definition` refers to a `struct`

```
// the module documentation string
const char * const __doc__ = "sample module documentation string";
// the module definition structure
PyModuleDef gsl::module_definition = {
   // header
   PyModuleDef_HEAD_INIT,
   // the name of the module
   "gsl",
   // the module documentation string
   __doc__,
   // size of the per-interpreter state of the module
   -1, // don't touch
   // the methods defined in this module
   gsl::module_methods
};
```

Method table

the method table is an array of method meta-data

```
// the module method table
PyMethodDef module_methods[] = {
   // random numbers
   { rng::avail__name__, rng::avail, METH_VARARGS, rng::avail__doc__ },
   { rng::alloc__name__, rng::alloc, METH_VARARGS, rng::alloc__doc__ },
   { rng::get__name__, rng::get, METH_VARARGS, rng::get__doc__ },
   { rng::name__name__, rng::name, METH_VARARGS, rng::name__doc__ },
   { rng::range__name__, rng::range, METH_VARARGS, rng::range__doc__ },
   { rng::set__name__, rng::set, METH_VARARGS, rng::set__doc__ },
   { rng::uniform__name__, rng::uniform, METH_VARARGS, rng::uniform__doc__ },
};
```

we have placed all `RNG` related symbols in their own namespace, so we can shorten their names without conflicts

Getting the set of available generators

```
// get the name of all the generators known to GSL
const char * const gsl::rng::avail__name__ = "rng_avail";
const char * const gsl::rng::avail__doc__ =
    "return the set of all known generators";

PyObject *
gsl::rng::avail(PyObject *, PyObject * args) {
    // unpack the argument tuple
    int status = PyArg_ParseTuple(args, ":rng_avail");
    // raise an exception if something went wrong
    if (!status) return 0;

    // make a frozen set to hold the names
    PyObject *names = PyFrozenSet_New(0);

    // iterate over the registered names
    for (
        gsl::rng::map_t::const_iterator i = gsl::rng::generators.begin();
        i != gsl::rng::generators.end();
        i++ ) {
       // add the name to the set
       PySet_Add(names, PyUnicode_FromString(i->first.c_str()));
    }

    // return the set of names
    return names;
}
```

Allocating generators

```
1  // allocation
2  const char * const gsl::rng::alloc__name__ = "rng_alloc";
3  const char * const gsl::rng::alloc__doc__ = "allocate a rng";
4
5  PyObject *
6  gsl::rng::alloc(PyObject *, PyObject * args) {
7     // place holders for the python arguments
8     char * name;
9     // unpack the argument tuple
10    int status = PyArg_ParseTuple(args, "s:rng_alloc", &name);
11    // raise an exception, if something went wrong
12    if (!status) return 0;
13
14    // get the rng type
15    const gsl_rng_type *algorithm = gsl::rng::generators[name];
16    // if it's not in table
17    if (!algorithm) {
18       PyErr_SetString(PyExc_ValueError, "unknown random number generator");
19       return 0;
20    }
21
22    // allocate one
23    gsl_rng * r = gsl_rng_alloc(algorithm);
24    // wrap it in a capsule and return it
25    return PyCapsule_New(r, capsule_t, free);
26 }
```

Deallocating generators

capsules are python objects that hold pointers to low level entities; the name is used to check that you received the capsule you expected

```
// capsules
namespace gsl {
   // rng
   namespace rng {
      const char * const capsule_t = "gsl.rng";
      void free(PyObject *);
   }
}
```

the destructor is an example

```
// destructor
void gsl::rng::free(PyObject * capsule)
{
   // bail out if the capsule is not valid
   if (!PyCapsule_IsValid(capsule, gsl::rng::capsule_t)) return;
   // get the rng
   gsl_rng * r = static_cast<gsl_rng *>(
      PyCapsule_GetPointer(capsule, gsl::rng::capsule_t));
   // deallocate
   gsl_rng_free(r);
   // and return
   return;
}
```

Generating a random number

```
1  // a random double in [0,1)
2  const char * const gsl::rng::uniform__name__ = "rng_uniform";
3  const char * const gsl::rng::uniform__doc__ =
4     "return the next random integer with the range of the generator";
5
6  PyObject *
7  gsl::rng::uniform(PyObject *, PyObject * args) {
8     // the arguments
9     PyObject * capsule;
10    // unpack the argument tuple
11    int status = PyArg_ParseTuple(args, "O!:rng_uniform", &PyCapsule_Type, &capsu
12    // raise an exception if something went wrong
13    if (!status) return 0;
14    // bail out if the capsule is not valid
15    if (!PyCapsule_IsValid(capsule, capsule_t)) {
16       PyErr_SetString(PyExc_TypeError, "invalid rng capsule");
17       return 0;
18    }
19
20    // get the rng
21    gsl_rng * r = static_cast<gsl_rng *>(PyCapsule_GetPointer(capsule, capsule_t)
22
23    // get a random number and return it
24    return PyFloat_FromDouble(gsl_rng_uniform(r));
25 }
```

Filling out the top layer

```
1  import pyre, itertools, gsl
2  from .PointCloud import PointCloud
3
4  class GSL(pyre.component, family="gauss.meshes.gsl", implements=PointCloud):
5      """
6      A point generator implemented using the large set of random number
7      generators available as part of the gnu scientific library (GSL)
8      """
9
10     # public state
11     seed = pyre.properties.float(default=0)
12     algorithm = pyre.properties.str(default="ranlxs2")
13
14     # interface
15     @pyre.export
16     def points(self, n, box):
17         """
18         Generate {n} random points in the interior of {box}
19         """
20         # unfold the bounding box
21         intervals = tuple(box.intervals()) # realize, so we can reuse in the loop
22         # loop over the sample size
23         while n > 0:
24             # make a point and yield it
25             yield tuple(itertools.starmap(self.rng.uniform, intervals))
26             # update the counter
27             n -= 1
28         # all done
29         return
```

Filling out the top layer – continued

```
    # meta methods
    def __init__(self, **kwds):
        # chain up
        super().__init__(**kwds)
        # build the RNG
        self.rng = gsl.rng_alloc(self.algorithm)
        # and seed it
        gsl.rng_set(self.rng, int(self.seed))
        # all done
        return

    # private data
    rng = None
```