ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

Recap: what we know so far

- `pyre` components are evolved python objects
  - the factories have family names, the instances have names
  - these names are unique strings in hierarchical namespaces delimited by periods
  - collections of components form packages *implicitly*, based on the topmost level in their namespace
- components have properties that are under the control of the *user*
  - they look and behave like regular attributes
  - they are *typed* to enable conversions from strings
  - they have default values and other metadata
- configuration is partly about assigning values to component properties
  - a requirement for supporting user interfaces
  - intuitive syntax for the command line
  - simple configuration files inspired by the Microsoft Windows `.ini` format
- configuration is automatically handled by the framework and requires no explicit involvement on the part of the component author

A simple component

```python
import pyre

class Disk(pyre.component, family="gauss.shapes.disk"):

    # public state
    radius = pyre.properties.float()
    radius.default = 1
    radius.doc = 'the radius of the disk'

    center = pyre.properties.array()
    center.default = (0,0)
    center.doc = 'the location of the center of the circle'

    # interface
    @pyre.export
    def interior(self, points):
        """
        Filter {points} by removing exterior ones
        """
        ...
```

A simple configuration file

- a file named `gauss.cfg` with the following

```
[ gauss.shapes.disk ] ; the family name
radius = 2
center = (-1,-1)

[ disk1 ] ; the name of an instance
center = (-1,1) ; leave {radius} alone

[ disk2 ] ; the name of another instance
radius = .5
center = (1,1)
```

  could get loaded automatically when `Disk` is first imported, or loaded by naming it explicitly on the command line
- the `.cfg` extension allows the framework to deduce which configuration file parser should be used to process the contents

Configuration files

- currently, there are two file formats for configuration information
  - `.cfg`: the format in the examples
  - `.pml`: an `XML` based format that is a bit more powerful but not as user friendly
- `pyre` looks for configuration files in the following places
  - explicitly provided on the command line

    ```
    gauss.py --config=sample.cfg
    ```

  - the current directory
  - the `.pyre` subdirectory of the current user's home directory
  - a special subdirectory wherever `pyre` is installed

  in order of priority
- settings on the command line have the highest priority, and override each other from left to right
- when a property is assigned a value multiple times, the highest priority setting wins
  - the framework keeps track of all changes in the value of properties and the source of the assignment, so if a property doesn't end up with the value you expected, you can get its complete history

Properties

- properties make sense for both classes and instances
  - the class holds the default value that gets used in case the component instance does not have explicit configuration
  - each instance gets its own private value when it gets configured
  - identical to regular python attributes
- there is support for
  - simple types: `bool`, `int`, `float`, `str`
  - containers: `tuple`, `array`
  - higher level: `date`, `time`, `inputfile`, `outputfile`, `inet`
  - units: `dimensional`
  - easy enough to implement your own; the requirements are very simple
- metadata:
  - `doc`: a simple and short documentation string
  - `default`: the default value, in case the user doesn't supply one
  - `converters`: a chain of preprocessors of the string representation
  - `normalizers`: a chain of post-processors of the converted value
  - `validators`: a tuple of predicates that get called to ensure the property value satisfies the specified constraints
  - you can add your own; the framework passes them through to your component

Units

- `dimensional` properties have units
- the low level support is in `pyre.units`
  - full support for all SI base and derived units
  - all common abbreviations and names from alternative systems of units
  - correct arithmetic; proper handling of functions from `math`

```
from math import cos
from pyre.units.SI import meter, second, radian

A = 2.5 * meter
t = 1.5 * second
ω = 4.2 * radian/second

x = A * cos(ω * t)
```

if the units in the argument to `cos` do not cancel, leaving a pure `float` behind, an exception is raised; `x` has dimensions of meters

Connecting components

the real power is in wiring components to other components

```
import pyre

class MonteCarlo(pyre.component, family="gauss.integrators.montecarlo"):
    """
    A Monte Carlo integrator
    """

    # public state
    samples = pyre.property.int(default=10**5)
    region = ????
    ...
```

- `MonteCarlo` should be able to specify what constitutes an acceptable region
- `Disk` should be able to advertise itself as being an acceptable region
- the user should have natural means for specifying that she wants to wire an instance of `Disk` as the region of integration
- and be able to configure that particular instance of `Disk` in a natural manner
- the framework should check the consistency of this assignment

Interfaces

the component version of our abstract base class is the *interface*

```
import pyre

class Shape(pyre.interface, family="gauss.shapes"):
    """
    The obligations of implementations of geometrical shapes
    """

    # my default implementation
    @classmethod
    def default(cls):
        """
        The default {Shape} implementation
        """
        # use a box
        from .Box import Box
        return Box # if you return an instance, it will be shared by all...

    # interface
    @pyre.provides
    def measure(self):
        """
        Compute my measure: length, area, volume, etc
        """

    @pyre.provides
    def interior(self, points):
        """
        Filter out {points} that are on my exterior
        """
```

`pyre` prohibits the instantiation of interfaces, so you don't have to worry about the implementation of the methods

Declaring compatibility with an interface

`Disk` can inform the framework that it intends to implement `Shape`

```
import pyre
from Shape import Shape

class Disk(pyre.component, family="gauss.shapes.disk", implements=Shape):
    """
    A representation of a circular disk
    """

    ...
```

an exception is raised if `Disk` does not conform fully to `Shape`

- missing methods or missing attributes

also, proper namespace design simplifies many things for the user

- `Shape` declared its family as `gauss.shapes`
- `Disk` declared its family as `gauss.shapes.disk`

we'll see how later when it's time to put all this together

Specifying assignment requirements

MonteCarlo can now specify it expects a Shape compatible object to be assigned as its region of integration

```
import pyre
from Shape import Shape

class MonteCarlo(pyre.component, family="gauss.integrators.montecarlo"):
    """
    A Monte Carlo integrator
    """

    # public state
    samples = pyre.property.int(default=10**5)
    region = pyre.facility(interface=Shape)
    ...
```

the default value is whatever Shape returns from its default class method

Component specification

- and now for the real trick: converting some string provided by the user into a live instance of `Disk`, configuring it, and attaching it to some `MonteCarlo` instance
- the syntax is motivated by `URI`, the universal resource identifiers of the web; the general form is

```
    <scheme>://<authority>/<path>#<identifier>
```

  where most of the segments are optional
- if your component is accessible from your python path, you could specify

```
    import:gauss.shapes.disk
```

- if your component instance is somewhere on your disk, you would specify

```
    file:/tmp/shapes.py/disk
```

- in either case, `disk` is expected to be a name that resolves into a component class, a component instance or be a callable that returns one of these

The user does the wiring

with our definition of `MonteCarlo`, an appropriately structured package `gauss` on the python path, and the following configuration file

```
[ gauss.shapes.disk ] ; change the default values for all disks
radius = 1
center = (0,0)

[ mc ] ; configure our Monte Carlo integrator instance
region = import:gauss.shapes.disk
...
```

the following python code in some script `sample.py`

```
  ...
  mc = MonteCarlo(name="mc")
  ...
```

builds a `MonteCarlo` instance and configures it so that its region of integration is a `Disk` instance; similarly, from the command line

```
   sample.py --mc.region=import:gauss.shapes.disk
```

or, thanks to the consistency in our namespace layout, simply

```
   sample.py --mc.region=disk
```