ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

Namespaces

- names are placed in *namespaces* in order to avoid collisions
  - no specific type or construct: anything that supports the `.` operator
  - examples: classes, modules, packages
- modules are objects created when requesting access to the names from a different file
  - python sources, which are byte-compiled on first import
  - shared libraries, which are dynamically loaded on first request
  - folders on the filesystem that contain the marker `__init__.py`
  - statically linked when the interpreter was compiled
- the interpreter has a *search path* for modules, which is controlled
  - at interpreter compile time
  - by the current working directory of the process
  - by reading user settings at interpreter start up
    - the `PYTHONPATH` environment variable on unix, the registry on windows

Namespace access

- names within a namespace are accessed with the `import` implicit assignment statement

```
1    import <namespace>
2    from <namespace> import <name>
3    from <namespace> import *
4    from <namespace> import <name> as <alias>
```

- namespaces may be nested

```
1    from sys.path import abspath
```

so *name qualifications* allow fine tuning of the list of imported symbols

- folders, and their sub-folders and files, become a hierarchy of nested namespaces automatically
  - files with the `.py` extension
  - folders with the `__init__.py` special file

Namespaces as objects

- modules and packages are objects:

```
    def load(material):
        'load the named {material} model'
        # build the comman string
        cmd = 'from materials import {} as model'.format(material)
        # get the interpreter to do its thing
        exec(cmd)
        # if all goes well, return the loaded module
        return model

    # load the material model
    model = load(material='perfectGas')
    # ask for an equation of state
    eos = model.newMaterial()
```

  dynamic programming!

- this example is brittle; one can do much better...

Classes

- classes are object factories
  - they introduce new types with state and behavior
  - using the name of a class in a call expression invokes the constructor
  - each instance has access to all the class attributes
  - assignments in the class declaration create class attributes
  - assignments to `self` create per-instance attributes

```
1   class Shape:
2       'the basis of all shapes'
3
4       # public data
5       name = 'generic shape'
6
7       # interface
8       def kind(self): return self.name
9
10      # meta methods
11      def __init__(self, **kwds):
12          super().__init_(**kwds)
13          self.rep = None
14          return
```

Class records and class instances

- the class declaration is an implicit assignment to a *class record*
- class records are a built-in type

```
print(Shape)
print(Shape.name)
print(Shape.kind)
```

- to make an *instance*, use the name of the class in a call expression

```
shape = Shape()
print(shape.name)
print(shape.kind())
```

- of course, neither `Shape` nor its instances are very interesting

Methods

- the class declaration creates a class record and assigns it to whatever name you used for the class
- invoking the class name as a function builds new instances of that class
- *methods* are functions defined within the class declaration; they provide behavior for the instances
  - they must take at least one parameter to receive the instance through which they were invoked
  - this special parameter is named `self`, by *convention*

Inheritance

- specialization through inheritance
  - super-classes must be listed in the class declaration
  - derived classes inherit all the attributes of their ancestors
  - instances inherit attributes from all accessible classes

```
1     class Disk(Shape):
2        'the shape bounded by a circle'
3
4        # public data
5        name = 'disk'
6        radius = 1
7        center = (0,0)
8
9        # meta methods
10       def __init__(self, radius=radius, center, **kwds):
11          super().__init_(**kwds)
12          self.radius = radius
13          self.center = center
14          return
```

- all classes inherit from `object`

Class glossary

- *class*
  - a blueprint for the construction of new types of objects
- *instance*
  - an object created using a class constructor
- *member*
  - an attribute of a class instance
- *method*
  - an attribute of a class instance that is bound to a function object
- *self*
  - the conventional name give to the method parameter that receives the referenced instance

Overloading operators in classes

- *Don't!*
- most python operations involving instances can be intercepted and customized
- through methods that have special names

method	*purpose*	*method*	*purpose*
`__init__`	construction: `x = X()`	`__getattr__`	member access: `x.name`
`__del__`	destruction	`__getitem__`	indexing: `x[5]`
`__str__`	string coercion: `str(x)`	`__setitem__`	indexing: `x[5] = 0`
`__repr__`	representation: `repr(x)`	`__add__`	addition: `x + other`
`__len__`	size, truth tests: `len(x)`	`__radd__`	addition: `other + x`
`__cmp__`	comparisons: `cmp(x), x < other`	`__and__`	logic: `x and other`
`__call__`	function class: `x()`	`__or__`	logic: `x or other`

Namespace rules

- a more complete story
  - unqualified names are looked up in a chain of lexical namespaces
  - qualified names conduct a search in the indicated namespace
  - scopes initialize object namespaces: packages/modules, classes, instances
- unqualified names
  - are global on read
  - are local on write, unless explicitly marked `global`
- qualified names, e.g. `instance.name`, are looked up in the indicated namespace
  - module and package
  - instance, then class record, then ancestors as specified in the `__mro__`
- namespace dictionaries
  - `__dict__`
  - name qualification is a dictionary lookup