

CSCI 553: Networking III

Unix Network Programming

Spring 2007



Object Oriented Programming

More on Objects



Introduction

- Previous lecture introduced basics of object-oriented programming
- This one goes a little further
 - As before, the ideas are (almost) universal, but the form varies from language to language



Length

- We've already met some special methods, like `__init__` and `__str__`
 - Usually not called directly
 - Instead, Python automatically invokes them at specific times (object creation and string creation)
- There are lots of other special methods
 - For example, if `obj` has a `__len__` method, Python calls it whenever it sees `len(obj)`

```
class Recent(object):
    def __init__(self, number=3):
        self.number = number
        self.items = []

    def __str__(self):
        return str(self.items)

    def add(self, item):
        self.items.append(item)
        self.items = self.items[-self.number:]

    def __len__(self):
        return len(self.items)

if __name__ == '__main__':
    history = Recent()

    for era in ['Permian', 'Trassic', 'Jurassic',
               'Cretaceous', 'Tertiary']:
        history.add(era)

    print len(history), history
```

```
1 ['Permian']
2 ['Permian', 'Trassic']
3 ['Permian', 'Trassic', 'Jurassic']
3 ['Trassic', 'Jurassic', 'Cretaceous']
3 ['Jurassic', 'Cretaceous', 'Tertiary']
```

Overloading Operators

- The expression "a + b" is "just" a shorthand for add(a, b)
- Or, if a is an object, for a.add(b)
- But since people might actually want to use the name add, Python spells this method `__add__`
 - Defining it is an example of operator overloading
- If a class defines a method `__add__`, it is called whenever something is '+'d to the object
 - I.e., `x + y` calls `x.__add__(y)`

```
class Recent(object):
    def __add__(self, item):
        self.items.append(item)
        self.items = self.items[-self.number:]
        return self

if __name__ == '__main__':
    history = Recent()
    for era in ['Permian', 'Trassic', 'Jurassic',
               'Cretaceous', 'Tertiary']:
        history = history + era
    print len(history), history
```

```
1 ['Permian']
2 ['Permian', 'Trassic']
3 ['Permian', 'Trassic', 'Jurassic']
3 ['Trassic', 'Jurassic', 'Cretaceous']
3 ['Jurassic', 'Cretaceous', 'Tertiary']
```



Commutativity

- $2 + x$ and $x + 2$ don't always do the same thing
 - Matrix multiplication
- Classes can define right-hand versions of operators, e.g., `__radd__` instead of `__add__`
 - If the object on the left has an `__add__` method, call that
 - Otherwise, if the object on the right has an `__radd__` method, call that
 - Otherwise, try Python's built-ins



Other Special Methods

- (Almost) every aspect of an object's behavior can be overridden by defining the right method(s)

Method	Purpose
<code>__lt__(self, other)</code>	Less than comparison; <code>__le__</code> , <code>__ne__</code> , and others are used for less than or equal, not equal, etc.
<code>__call__(self, args...)</code>	Called for <code>obj(3, "lithium")</code>
<code>__len__(self)</code>	Object "length"
<code>__getitem__(self, key)</code>	Called for <code>obj[3.14]</code>
<code>__setitem__(self, key, value)</code>	Called for <code>obj[3.14] = 2.17</code>
<code>__contains__</code>	Called for "lithium" in <code>obj</code>
<code>__add__</code>	Called for <code>obj + value</code> ; use <code>__mul__</code> for <code>obj * value</code> , etc.
<code>__int__</code>	Called for <code>int(obj)</code> ; use <code>__float__</code> and others to convert to other types



Example: Sparse Vector

- A vector is sparse if most of its entries are zero
 - Use a dictionary to record non-zero values and their indices
 - No point padding eleven actual values with nine million zeroes
- Overload operators to make the object look like a “real” vector:
 - Addition: create a new vector with a non-zero value wherever either operand had a non-zero value
 - Dot product: add up products of matching non-zero values
 - Length: return one more than the index of the largest non-zero value
 - “One more” to be consistent with Python's lists



How Long is a Sparse Vector?

- What is the length of `v` after the following operations?

```
v = SparseVector() # all values initialized to 0.0
v[27] = 1.0        # length is now 28
v[43] = 1.0        # length is now 44
v[43] = 0.0        # is the length still 44, or 28?
```

- This isn't really a programming question
 - “Largest current index” and “largest index ever seen” can both be implemented
 - The latter is easy, so we'll use that



Vector Behavior

- Construction creates an empty sparse vector
- Define `__len__`, `__getitem__`, and `__setitem__` to make it behave like a list
 - Exercise: implement `del sparse[index]`

```
class SparseVector(object):  
    '''Implement a sparse vector.  If a value has not been set  
    explicitly, its value is zero.'''  
    def __init__(self):  
        '''Construct a sparse vector with all zero entries.'''  
        self.data = {}  
    def __len__(self):  
        '''The length of a vector is one more than the largest index.'''  
        if self.data:  
            return 1 + max(self.data.keys())  
        return 0  
    def __getitem__(self, key):  
        '''Return an explicit value, or 0.0 if none has been set.'''  
        if key in self.data:  
            return self.data[key]  
        return 0.0  
    def __setitem__(self, key, value):  
        '''Assign a new value to a vector entry.'''  
        if type(key) is not int:  
            raise KeyError, 'non-integer index to sparse vector'  
        self.data[key] = value
```



Dot Product

- The other object (on the right side of "*") is usually called other
 - No reason to insist that it be a sparse vector
 - Could equally well be a list of values
- So loop over our indices, and multiply by corresponding values in other object
 - Any index not encountered in this loop doesn't matter, since it corresponds to something that's zero
- And make `__rmul__ = __mul__` do the same thing as `__rmul__`

```
def __mul__(self, other):  
    '''Calculate dot product of a sparse vector with something else.'''  
  
    result = 0.0  
    for k in self.data:  
        result += self.data[k] * other[k]  
    return result  
  
def __rmul__(self, other):  
    return self.__mul__(other)
```



Addition

- Trickier than multiplication: result is non-zero wherever either argument is non-zero
- Don't want to loop over all the zeroes of either argument
- Solution: if the other object is a sparse vector, cheat
 - I.e., reach inside it, and rely on details of its implementation

```
def __add__(self, other):
    '''Add something to a sparse vector.'''
    # Initialize result with all non-zero values from this vector.
    result = SparseVector()
    result.data.update(self.data)
    # If the other object is also a sparse vector, add non-zero values.
    if isinstance(other, SparseVector):
        for k in other.data:
            result[k] = result[k] + other[k]
    # Otherwise, use brute force.
    else:
        for i in range(len(other)):
            result[i] = result[i] + other[i]
    return result

# Right-hand add does the same thing as left-hand add.
__radd__ = __add__
```



Testing

- The class isn't written until the tests are finished

- Exercise:
replace the print statements with assertions

```
if __name__ == '__main__':  
    x = SparseVector()  
    x[1] = 1.0  
    x[3] = 3.0  
    x[5] = 5.0  
    print 'len(x)', len(x)  
    for i in range(len(x)):  
        print '...', i, x[i]  
  
    y = SparseVector()  
    y[1] = 10.0  
    y[2] = 20.0  
    y[3] = 30.0  
  
    print 'x + y', x + y  
    print 'y + x', y + x  
  
    print 'x * y', x * y  
    print 'y * x', y * x  
  
    z = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]  
  
    print 'x + z', x + z  
    print 'x * z', x * z  
    print 'z + x', z + x
```

```
len(x) 6  
... 0 0.0  
... 1 1.0  
... 2 0.0  
... 3 3.0  
... 4 0.0  
... 5 5.0  
x + y [0.0, 11.0, 20.0, 33.0, 0.0, 5.0]  
y + x [0.0, 11.0, 20.0, 33.0, 0.0, 5.0]  
x * y 100.0  
y * x 100.0  
x + z [0.0, 1.1, 0.2, 3.3, 0.4, 5.5]  
x * z 3.5  
z + x [0.0, 1.1, 0.2, 3.3, 0.4, 5.5]
```



Static Data Members

- Sometimes want to share data between all instances of a class
 - Constants, a count of the number of class instances created, etc.
- Any data members defined inside the class block belong to the class as a whole

```
class Counter(object):  
  
    num = 0      # Number of Counter objects created.  
  
    def __init__(self, name):  
        Counter.num += 1  
        self.name = name  
  
if __name__ == '__main__':  
    print 'initial count', Counter.num  
    first = Counter('first')  
    print 'after creating first object', Counter.num  
    second = Counter('second')  
    print 'after creating second object', Counter.num
```

```
initial count 0  
after creating first object 1  
after creating second object 2
```



Static Methods

- Can also create static methods
 - Just like a function, but put inside the class definition for clarity
- Define the method without the self parameter
 - Since it isn't tied to any particular instance of the class
- Put @staticmethod in front of it
 - A decorator
 - Powerful, but beyond the scope of this course

```
class Experiment(object):
    already_done = {}

    @staticmethod
    def get_results(name, *params):
        if name in Experiment.already_done:
            return Experiment.already_done[name]
        exp = Experiment(name, *params)
        exp.run()
        Experiment.already_done[name] = exp
        return exp

    def __init__(self, name, *params):
        self.name = name
        self.params = params

    def run(self):
        # ...

if __name__ == '__main__':
    first = Experiment.get_results('anti-gravity')
    second = Experiment.get_results('time travel')
    third = Experiment.get_results('anti-gravity')
    print 'first ', id(first)
    print 'second', id(second)
    print 'third ', id(third)

first 5120204
second 5120396
third 5120204
```



Design Patterns

- Style guidelines describe what code should look like line by line
- Design patterns are how we describe larger patterns
 - A standard solution to a commonly-occurring problem
 - That isn't specific enough to be captured once and for all in a library routine or framework
- Idea developed from the 1960s on by the (building) architect Christopher Alexander
 - For example, it's hard to define what a porch is, but the basic idea comes up everywhere the climate is warm
- Introduced to programmers in [Gamma et al 1995]
 - Still a bestseller, but not particularly approachable



The Singleton Pattern

- Problem: want to ensure that there's only ever one instance of a particular class
 - E.g. the controller for a radio telescope antenna
- Considerations:
 - There must be exactly one instance of the class
 - All objects that use the class must have access to that instance
- Solution:
 - Create objects by calling a function instead of the class's constructor
 - Have the function store a reference to the first object it creates
 - Have it return that same object on every subsequent call



Singleton Implementation

```
class AntennaClass(object):
    '''Singleton that controls a radio telescope.'''

    # The unique instance of the class.
    instance = None

    # The constructor fails if an instance already exists.
    def __init__(self, max_rotation):
        assert AntennaClass.instance is None, 'Trying to create a second instance!'
        self.max_rotation = max_rotation
        AntennaClass.instance = self

# Make the creation function look like a class constructor.
def Antenna(max_rotation):
    '''Create and store an AntennaClass instance, or return the one
    that has already been created.'''
    if AntennaClass.instance:
        return AntennaClass.instance
    return AntennaClass(max_rotation)
```

Singleton Pattern Demonstration



```
first = Antenna(23.5)
print 'first instance:', id(first)
second = Antenna(47.25)
print 'second instance:', id(second)
```

first instance: 10685200

second instance: 10685200



The Visitor Pattern

- Problem: want an easy way to walk around a complex structure
 - E.g. visit each value in a list of lists of lists exactly once
- Considerations:
 - Many different operations may need to be performed
 - Structure is complex enough that visiting elements is error-prone
 - The types of objects in the structure, and the ways they are connected, are fixed
- Solution:
 - Create a class that knows how to get to each value in turn
 - Give it an empty method that is called once for each value
 - Users derive from this class and fill in the method
 - An all-in-one version of the framework shown earlier



Visitor Implementation

```
class NestedListVisitor(object):
    '''Visit each element in a list of nested lists.'''
    def __init__(self, data):
        '''Construct, but do not run.'''
        assert type(data) is list, 'Only works on lists!'
        self.data = data
    def run(self):
        '''Iterate over all values.'''
        self.recurse(self.data)
    def recurse(self, current):
        '''Loop over a particular list or sub-list (not meant
        to be called by users).'''
        if type(current) is list:
            for v in current:
                self.recurse(v)
        else:
            self.visit(current)
    def visit(self, value):
        '''Users should fill this method in.'''
        pass
```



Visitor Demonstration

```
class MaxOfN(NestedListVisitor):
    def __init__(self, data):
        NestedListVisitor.__init__(self, data)
        self.max = None
        self.count = 0
    def visit(self, value):
        self.count += 1
        if self.max is None:
            self.max = value
        else:
            self.max = max(self.max, value)

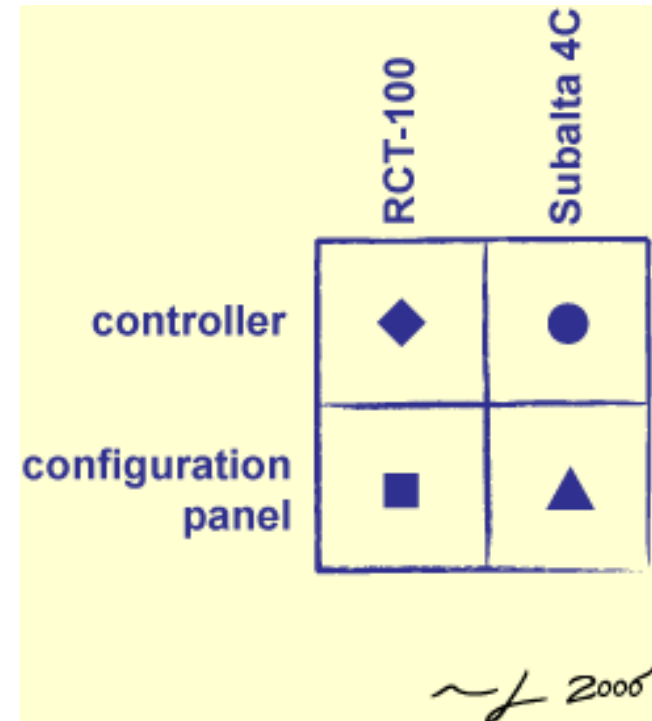
test_data = [['gold', 'lead'], 'zinc', [['silver', 'iron'], 'mercury']]
test = MaxOfN(test_data)
test.run()
print 'max:', test.max
print 'count:', test.count
```

max: zinc

count: 6

The Abstract Factory Pattern

- Problem: application doesn't know the specific types of objects it wants to create until runtime
 - If the chromatograph is an RCT-100, create an RCT-100 controller and an RCT-100 configuration panel
 - If it's a Subalta 4C, create a Subalta 4C controller and configuration panel
- Considerations:
 - Objects can be grouped by category and family
 - New categories or families may appear later
- Solution:
 - Create a class that knows how to build an instance of each category for a particular family
 - Create another class that stores instances of these builder classes, and calls their methods when asked to
 - Adding a new family is easy, but adding a new category requires changes to every builder





Abstract Factory Builder

```
class AbstractFamily(object):  
    '''Builders for particular families derive from this.'''  
  
    def __init__(self, family):  
        self.family = family  
  
    def get_name(self):  
        return self.name  
  
    def make_controller(self):  
        raise NotImplementedError('make_controller missing')  
  
    def make_configuration_panel(self):  
        raise NotImplementedError('make_configuration_panel missing')
```



Abstract Factory Manager

```
class FactoryManager(object):
    '''Manage builders by family.'''
    def __init__(self, current_family=None):
        self.builders = {}
        self.family = family
    def set_family(self, family):
        assert family, 'Empty family'
        self.family = family
    def add(self, builder):
        name = builder.get_name()
        self.builders[name] = builder
    def make_controller(self):
        self._check_state()
        return self.builders[self.family].make_controller()
    def make_configuration_panel(self):
        self._check_state()
        return self.builders[self.family].make_configuration_panel()
    def _check_state(self):
        assert self.family, 'No family specified'
        assert self.family in self.builders, 'Unknown family:', self.family
```



Factory Demonstration

```
factory = FactoryManager()  
factory.add(RCT100Factory())  
factory.add(Subalta4CFactory())  
factory.set_family('Subalta4C')  
controller = factory.make_controller()  
configuration_panel = factory.make_configuration_panel()
```



The Command Pattern

- Problem: want to be able to control the operation of a complex object
 - Turn on the robot arm, move it, lower it, move it again, etc.
- Considerations:
 - Do not want to have to write an entirely new program for each sequence of operations
 - Want to be able to add new operations
 - Would like to be able to undo operations
- Solution:
 - Create one class for each distinct operation
 - Give the class do, undo, and redo methods
 - Create instances of these classes to represent particular commands
 - Create lists of these instances to control the robot arm



Base Command Class

```
class AbstractCommand(object):
    '''Base class for commands.'''
    def is_undoable(self):
        return False # by default, can't undo/redo operations
    def do(self, robot):
        raise NotImplementedError("Don't know how to do %s" % self.name)
    def undo(self, robot):
        pass
    def redo(self, robot):
        pass
```



A Particular Command

```
class MoveCommand(AbstractCommand):
    '''Move the robot arm.'''

    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def is_undoable(self):
        return True

    def do(self, robot):
        robot.translate(self.x, self.y, self.z)

    def undo(self, robot):
        robot.translate(-self.x, -self.y, -self.z)

    def redo(self, robot):
        self.do(robot)
```



Command Demonstration

```
robot = Robot()  
commands = [MoveCommand(5.0, 2.0, 2.3),  
             RotateCommand(-90.0, 0.0, 0.0),  
             MoveCommand(1.0, 2.0, 2.0),  
             CloseHandCommand()]  
  
for c in commands:  
    c.do(robot)
```



A Few Other Patterns

- Cache: store temporary copies of objects locally to improve performance
- State: record state of program as object so that it can be re-started
- Null Object: use an object that does nothing in place of null
 - Saves testing that object isn't null before doing operations
- Adapter: wrap one object in another to give the first a different interface
 - Usually used to give a new library an interface that's compatible with an old one
- Proxy: use one object as an interface to another
 - Typically, the proxy is local, and the real object is on another machine



Summary

- Overloading, design patterns, and other advanced concepts serve two purposes:
 - Communication: a concise way for designers to communicate with each other
 - Education: gives them a way to communicate what they know to newcomers
- Don't expect to connect them all to your own experience the first time
 - But keep them in mind as you look at new problems