OOP IMPLEMENTATION, SCHEME 11

COMPUTER SCIENCE 61A

July 24, 2012

1 Review: Object-Oriented Programming

In Object-Oriented Programming, we have the following main components:

classes
A "blueprint" of something we want to model. This will include the declaration of all **methods**, **instance attributes**, and **class attributes**.

instances
An object created from a class. This object has its own state (instance attributes) and behavior (methods).

So far, we have used Python's built-in object-oriented syntax in order to define and use classes. For instance, a `Person` class could look something like:

```
class Person(object):
    population = 0                # class var
    def __init__(self, name):     # constructor method
        self.name = name          # instance attribute 'name'
        Person.population = Person.population + 1
    def greet(self):              # method 'greet'
        return "Hi, I'm " + self.name
```

This week, we will learn how to implement our own object-oriented system, using only the concepts from this course.

1.1 What's in a class?

Before we begin implementing classes in our new system, we should ask ourselves: what do classes need to be able to do? After some thought, we can reduce it to three basic tasks:

get
A class needs to be able to retrieve (get) its stored attributes. This includes class attributes and methods.

set
A class needs to be able to set class attributes, or create new class attributes.

instantiate
We need to be able to create (instantiate) instances of this class.

We will implement this behavior via the use of a **dispatch dictionary**:

```
def make_class(attributes, base_class=None):
    """Return a new class.

    attributes -- class attributes
    base_class -- a dispatch dictionary representing a class
    """
    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)
    def set_value(name, value):
        attributes[name] = value
    def new(*args):
        return init_instance(cls, *args)
    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
```

The important thing to note is that a class is simply represented as a **dictionary** that contains three keys: `get`, `set`, and `new`. When we want to get the value of an attribute from the class, we pass it the `get` message, which returns to us an internally-defined function `get_value` that we can use to get the value of an attribute.

Similarly, we can use `set_value` to modify an existing attribute within the class or, if it hasn't been set yet, create a new instance attribute.

Finally, if we want to actually create an instance, then we pass in the `new` message, which returns the `new` function that calls `init_instance`:

```
def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args."""
    instance = make_instance(cls)
    init = cls['get']('__init__')
    if init:
        init(instance, *args)
    return instance
```

As one can see, `init_instance` checks to see if the class has the `__init__` method defined, and if it does, to call it on the newly-created `instance`.

As a concrete example, let's convert the `Person` Python class definition into the equivalent definition within our object-oriented system:

```
def make_person_class():
    def __init__(self, name):
        self['set']('name', name)
        Person['set']('population', Person['get']('population') + 1)
    def greet(self):
        return "Hi, I'm " + self['get']('name')
    attrs = {'population': 0, '__init__': __init__,
             'greet': greet}
    Person = make_class(attrs)
    return Person
```

```
>>> Person = make_person_class()
>>> Person['get']('population')
0
>>> joy = Person['new']('Joy')
>>> Person['get']('population')
1
```

The above interaction is effectively equivalent to the following interaction:

```
>>> Person.population
0
>>> joy = Person('joy')
>>> Person.population
1
```

1.2 What is an object?

What are the fundamental behaviors of objects that we need to capture in our Object-Oriented implementation?

get

An object needs to be able to retrieve (get) its stored attributes. This includes instance/class attributes, in addition to methods.

set

An object needs to be able to modify the value of previously-set attributes, or create new instance attributes.

Once again, we will implement this behavior via a **dispatch dictionary**:

```
def make_instance(cls):
    """Return a new object instance."""
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)
    def set_value(name, value):
        attributes[name] = value
    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
```

An instance is simply a **dictionary** of two keys: `get` and `set`. When we want to get the value of an attribute from an instance, we pass it the `get` message, which returns to us an internally-defined function `get_value` that we can use.

Similarly, we can use `set_value` to modify an existing attribute within the instance or, if it hasn't been set yet, create a new instance attribute.

Here's a comparison between the Python object-oriented system and our own system:

```
>>> # Our way
>>> joy = Person['new']('joy')
>>> joy['get']('name')
'joy'

>>> # Python's way
>>> joy = Person('joy')
>>> joy.name
'joy'
```

1. In which `attributes` dictionary are methods stored? Are they stored in the instance

dispatch-dictionary, or the class dispatch-dictionary?

2. Modify the following `Person` class implementation to add a new method `nom` that returns the same three strings in order: `"om"`, `"nom"`, and `"nom!"`:

```
>>> eric = Person['new']('eric')
>>> eric['get']('nom')()
'om'
>>> eric['get']('nom')()
'nom'
>>> eric['get']('nom')()
'nom!'
>>> eric['get']('nom')()
'om'
```

```
def make_person_class():
    def __init__(self, name):
        self['set']('name', name)
        Person['set']('population', Person['get']('population') + 1)
    def greet(self):
        return "Hi, I'm " + self['get']('name')
    """ YOUR CODE HERE """
```

3. What if we modified the `get_value` function inside of `make_instance` to not call `bind_method`, i.e. `make_instance` becomes:

```
def make_instance(cls):
    """Return a new object instance."""
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
```

```
            value = cls['get'](name)
            # return bind_method(value, instance)  # remove this!
            return value
    def set_value(name, value):
        attributes[name] = value
    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
```

What changes? In particular, what happens in the following interaction?

```
>>> bruce = Person['new']('Bruce')
>>> bruce['get']('greet')()
________ # ?
```

4. Translate the `Account` Python class definition to an equivalent definition using our object-oriented system:

```
class Account:
    tax = 0.01
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 0
    def deposit(self, amt):
        new_balance = self.balance + amt
        self.balance = new_balance
    def withdraw(self, amt):
        if amt > self.balance:
            return "Not enough funds."
        else:
            self.balance -= amt
            return amt * Account.tax

def make_account_class():
    """ YOUR CODE HERE """
```

1.3 Inheritance, Done Our Way

To finish things off, let's examine how inheritance is handled within this object-oriented system. In Python's object-oriented system, inheritance worked in the following way. Say we have class `A`, and class `B` is a subclass of `A`. When we access an attribute `attr` of an instance of `B`, if `attr` isn't found within the class `B`, then we look in the parent class `A` for the attribute `attr`, and so on if `A` itself is a subclass of another class.

As a concrete example, let's define the `TA` class that behaves just like a `Person`, but only responds to every-other invocation of the `greet` method (this delay is presumably because TA's stay up late preparing discussion notes):

```
class Person(object):
    population = 0
    def __init__(self, name):
        self.name = name
        Person.population = Person.population + 1
    def greet(self):
        return "Hi, I'm " + self.name

class TA(Person):
    def __init__(self, name):
        Person.__init__(self, name)
        self.greet_count = 0
```

```
    def greet(self):
        if self.greet_count % 2 == 1:
            val = Person.greet(self)
        else:
            val = '...hm...'
        self.greet_count += 1
        return val
```

```
>>> albert = TA('Albert')
>>> albert.greet()
"...hm..."
>>> albert.greet()
"Hi, I'm Albert"
>>> albert.greet()
"...hm..."
```

To use inheritance in our own object system, when we define the `TA` class, we will also pass in the `Person` class as the `base_class` argument to `make_class`:

```
def make_ta_class(parentclass):
    def __init__(self, name):
        parentclass['get']('__init__')(self, name)
        self['set']('greet_count', 0)
    def greet(self):
        if self['get']('greet_count') % 2 == 1:
            val = parentclass['get']('greet')(self)
        else:
            val = "...hm..."
        self['set']('greet_count', self['get']('greet_count') + 1)
        return val
    attrs = {'__init__': __init__, 'greet': greet}
    return make_class(attrs, parentclass)
```

1. Using our object-oriented system, define the `CS61AStudent` class that behaves just like a `Person`, but repeats their `greet` phrase twice in a row (presumably because of all the coffee and all-nighters being pulled):

   ```
   >>> fry = CS61AStudent('Fry')
   >>> fry['get']('greet')()
   "Hi, I'm Fry Hi, I'm Fry"
   ```

   ```
   def make_CS61AStudent_class(parentclass):
       """ YOUR CODE HERE """
   ```

2. What if we changed the last few lines of `make_ta_class` to instead be:

```
def make_ta_class(parentclass):
    def __init__(self, name):
        parentclass['get']('__init__')(self, name)
        self['set']('greet_count', 0)
    def greet(self):
        if self['get']('greet_count') % 2 == 1:
            val = parentclass['get']('greet')(self)
        else:
            val = "...hm..."
        self['set']('greet_count', self['get']('greet_count') + 1)
        return val
    attrs = {'__init__': __init__, 'greet': greet}
    parentclass = make_person_class()    # Added this line
    return make_class(attrs, parentclass)
```

What would change about the `ta_class`, if anything? In particular, what would be following interactions print out?

```
>>> Person = make_person_class()
>>> TA = make_ta_class()
>>> joe = Person['new']('Joe')
>>> Person['get']('population')
_________   # value?
>>> TA['get']('population')
_________    # value?
```

3. What if I modified the `__init__` method of the `Person` implementation to be:

```
def make_person_class():
    def __init__(self, name):
        self['set']('name', name)
        self['set']('population',
                    self['get']('population') + 1) # changed
    def greet(self):
        return "Hi, I'm " + self['get']('name')
    attrs = {'population': 0, '__init__': __init__,
             'greet': greet}
    Person = make_class(attrs)
    return Person
```

What, if anything, will change? In particular, what will the following interactions return?

```
>>> Person = make_person_class()
>>> cecilia = Person['new']('cecilia')
>>> tajel = Person['new']('tajel')
>>> Person['get']('population')
_________
>>> cecilia['get']('population')
_________
>>> tajel['get']('population')
_________
```

2 The Scheme Language

In the next part of the course, we will be working with the **Scheme** programming language. In addition to learning how to write Scheme programs, we will eventually write a Scheme interpreter in Project 4.

Scheme is a dialect of the **Lisp** programming language, a language dating back to 1958. The popularity of Scheme within the programming language community stems from its simplicity – in fact, previous versions of CS 61A were taught in the Scheme language.

2.1 The Scheme Interpreter

Like Python, Scheme features an interpreter where you can have an interactive session On the 61A class accounts, you can start a Scheme interactive session by running the `stk` program:

```
star [16] ~ # stk
Welcome to the STk interpreter version 4.0.1-ucb1.3.6
Copyright (c) 1993-1999 Erick Gallesio
Modifications by UCB EECS Instructional Support Group
STk>
```

We can ask it to evaluate a few simple arithmetic expressions:

```
STk> 42
42
STk> (+ 1 2)
3
STk> (* 2 (- 5 3) (+ 3 1 0))
16
```

In the last line, we see that the arithmetic functions can take any number of arguments.

2.2 An Example Scheme Program

Let's take a look at the following Scheme code:

```
(define (factorial n)
  (if (= n 1)
      n
      (* n (factorial (- n 1)))))
```

Here, we have defined a function `factorial` that, given an argument `n`, computes the factorial of n. We can call it in the same way we called the arithmetic functions:

```
STk> (factorial 3)
6
STk> (factorial 4)
24
STk> (factorial (+ 2 3))
120
```

As you can see, without explicitly going over the Scheme syntax we can look at the above `factorial` definition and see the similarities to the equivalent Python definition:

```
def factorial(n):
    if n == 1:
        return n
    else:
        return n * factorial(n - 1)
```

2.3 My Little Scheme Exercises

1. What will Scheme print? For the following expressioins, write down what Scheme will display.

   ```
   STk> (+ 1 2 3 4)

   STk> (factorial (+ (factorial 2) (* (- 2 1) 1)))

   STk> (+ (* (- 5 1) 4 2) 3)

   STk> (> 44 2)

   STk> (and #t #t #f)

   STk> (or (= 3 5) #f (> 2 3) (<= 5 5))
   ```

2. Translate the following Scheme functions into its equivalent Python function definition:

   a.)

   ```
   (define (sum num)
     (if (= num 0)
         num
         (+ num (sum (- num 1)))))

   def sum(num):
   ```

   b.)

```
(define (fib n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

def fib(n):
```

2.4 Types of Scheme Expressions

We can separate Scheme expressions into three different groups: *primitive* expressions, *call* expressions, and *special forms*.

A **primitive** expression include "simple" things like numbers, variables, and strings. Note that in Scheme, `#t` and `#f` stand for True and False respectively. In the last line below, we see how Scheme displays function values:

```
STk> 100
100
STk> "hi there"
"hi there"
STk> #t
#t
STk> #f
#f
STk> +
#[closure arglist=args 196920]
```

A **call** expression is an expression that takes on the form: `( <function name> <arg1> ... <argN> )`. You can call any user-defined or built-in function this way:

```
STk> (+ 4 3)
7
STk> (/ (- 22 2) (+ 2 (* 4 2)))
2
STk> (factorial 3)
6
```

Finally, **special forms** are language constructs that allow for features such as function definitions, conditional expressions, variable assignment, and quoting. We've already seen the first two special forms already, and we'll get to the others shortly.

2.5 More Scheme Practice

1. Given the following Python session, translate each line into equivalent Scheme code:

```
>>> 1 + 2 + 3
>>> 2 * ((3 + 4) - 8)
>>> def maxfn(a, b):
...     if a > b:
...         return a
...     else:
...         return b
>>> maxfn(5, 2)
>>> maxfn(3, maxfn(5, maxfn(7, 9001)))
```