PC204 Lecture 7

Conrad Huang
conrad@cgl.ucsf.edu
Genentech Hall, N453A
x6-0415

Topics

- Homework review
- Object-oriented design
- How to do object-oriented programming with Python dictionaries
  - Why classes and instances work the way they do
- How to do object-oriented programming with Python classes

Homework Review

- 6.1 – Rectangle functions
- 6.2 – area_difference
- Object-oriented perspective
  - 6.1 defines and implements a rectangle API
    - Methods: create, convert to string, shift and offset
    - Attributes: width, height, corner (x, y)
  - 6.2 uses the API
    - Area difference requires **two** rectangles and is not a method of a single rectangle (usually)

Object-Oriented Design and Programming

- Designing programs around “objects” instead of functions or data
- Conceptually, an object is something we think of as a single unit (eg appearance, state, behavior)
- Collectively, all objects with the same behavior form a “class”
- Programmatically, an object is represented by a set of data (“attributes”) and functions (“methods”)

OOD and OOP Criteria

- Design focuses on what are the attributes and methods (the "interface") of objects while programming focuses on how to make the interface functional (the "implementation")
- Abstraction: the external or public interface of an object or class
  - Cohesion: how much attributes and methods relate to each other
  - Coupling: how much interdependencies there are among a set of classes
- Encapsulation: hiding the internal implementation of a class

OOP Data Organization

- Organize data by:
  - Keeping data for the same object together as a single unit so that they may be accessed starting from a single reference
  - Naming individual data elements thereby providing hints as to what they represent
- Both objects and dictionaries can provide both properties
  - So why do we need both?

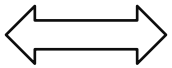
Example Data and Functions

- We will use rectangles as our example
  - Rectangles have width, height and origin
  - Function/method **area** computes and returns area from width and height
  - Function/method **shift** moves the rectangle origin

OOP with Dictionaries (cont.)

- Dictionary syntax is more cumbersome, but is effectively the same as instance attribute syntax

```
class Rectangle(object): pass
r = Rectangle()
r.width = 10
r.height = 10
r.origin = (0, 0)
```



```
r = dict()
r[“width”] = 10
r[“height”] = 10
r[“origin”] = (0, 0)
```

width	10
height	10
origin	(0, 0)

From Dictionaries to Classes

- Since we use dictionaries all the time, why create another syntax to do something similar?
  - There are common operations (like identifying the type of an object and sharing functions among multiple objects) that may be codified into class and instance syntax
  - Less code $\Longrightarrow$ more readable code and less typing (at least that's what we hope)

Why (not) Use Dictionaries?

- Advantage
  - We can use a familiar syntax for referencing data
- Disadvantage
  - Because there is only one data type, there is no easy way to figure out what any dictionary represents other than looking at its keys
- Solution
  - Reserve a "tag" key for type identification

Identify Data Types with Tags

- A tag is a convention of using a "well known" name for storing identification information

```
r = dict()
r["tag"] = "Rectangle"
r["width"] = 10
r["height"] = 10
r["origin"] = (0,0)
r2 = dict()
r2["tag"] = "Rectangle"
r2["width"] = 20
r2["height"] = 5
r2["origin"] = (2, 4)
```

tag	"Rectangle"
width	10
height	10
origin	(0, 0)

tag	"Rectangle"
width	20
height	5
origin	(2, 4)

  - Given data dictionary **d**, its data type can be found by examining **d["tag"]**

What about Functions?

- We can call functions to operate on the data dictionaries that we created
- *However*, OOP is about grouping data ***and operators*** together so that given an "object", we know about both its data ***and behavior***
  - Q: Can we do this using dictionaries?
  - A: Of course.
    - The key is to recognize that values in Python dictionaries can be references to functions

Function References in Dictionaries

- In each "data dictionary", we can store not only data, but also references to functions that can operate on the data
  - The simplest way is to reserve a tag name for each function
  - Each data dictionary may then have a set of tags assigned to the functions that can operate on it
  - Callers, when given a data dictionary, may then use the tags to find the appropriate functions

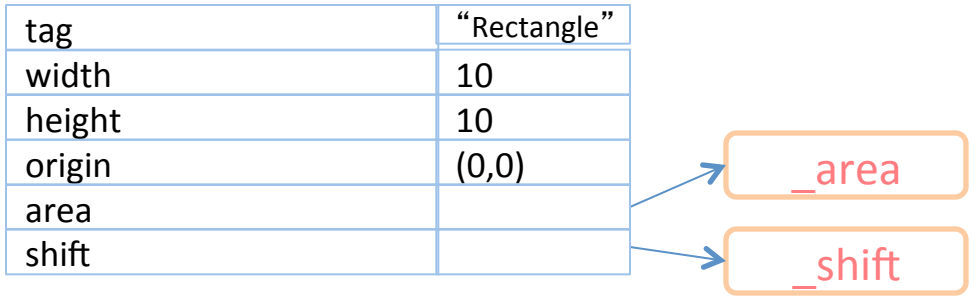
Function Reference Example

- How can we associate functions for computing area and shifting origin with "Rectangle" dictionaries?

```
def _area(r):
        return r["width"]* r["height"]
def _shift(r, dx, dy):
        r["origin"] = (r["origin"][0] + dx,
                       r["origin"][1] + dy)
def Rectangle(w, h, o):
        r= dict()
        r["tag"] = "Rectangle"
        r["width"] = w
        r["height"] = h
        r["origin"] = o
        r["area"] = _area
        r["shift"] = _shift
        return r
```

```
# Create a Rectangle data dictionary
r = Rectangle(10, 10, (0, 0))

# Compute rectangle area
print r["area"](r)
```



More Function References

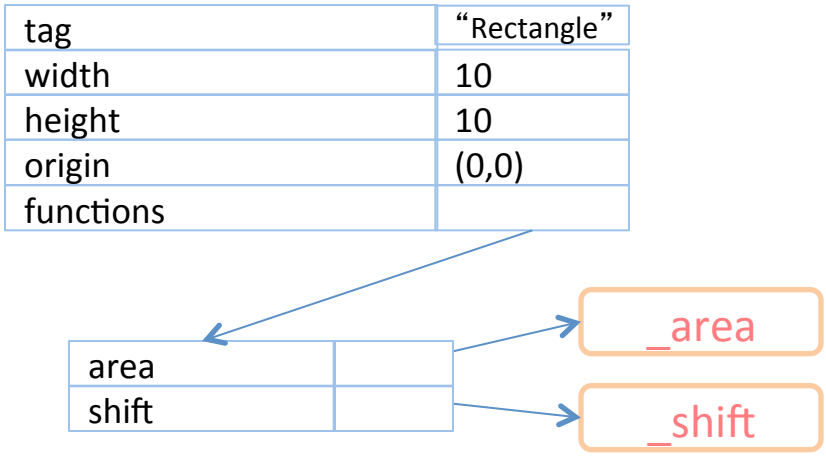
- The problem with using one tag per function is that:
  - Each data dictionary has its own set of function references, which wastes memory
  - The functions are not logically grouped together (other than in the dictionary creation function)
- Solution
  - Put related functions into their own "class dictionary" and have each "data" dictionary reference the class dictionary

Another Function Reference Example

```
def _area(r):
    return r[“width”]* r[“height”]
def _shift(r, dx, dy):
    r[“origin”] = (r[“origin”][0] + dx,
                   r[“origin”][1] + dy)
Rectangle_classDict = {
    “area”: _area,
    “shift”: _shift,
}
def Rectangle(w, h, o):
    r = dict()
    r[“tag”] = “Rectangle”
    r[“functions”] = Rectangle_classDict
    r[“width”] = w
    r[“height”] = h
    r[“origin”] = o
    return r
```

```
# Create a Rectangle data dictionary
r = Rectangle(10, 10, (0, 0))

# Compute rectangle area
print r[“functions”][“area”](r)
```



More Function References

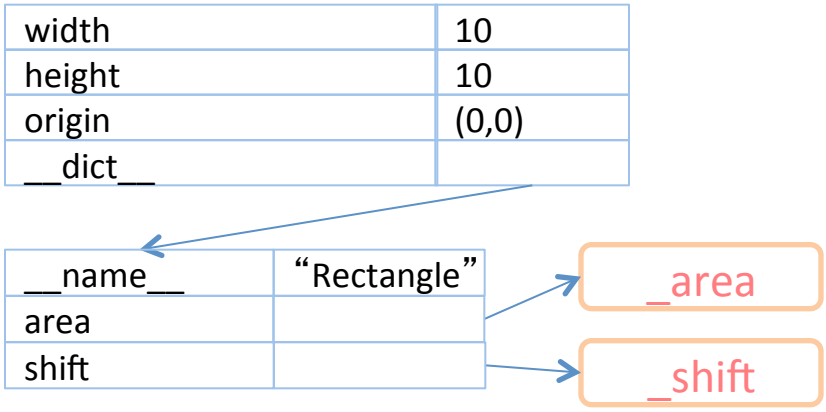
- The data/class-dictionary system has disadvantages
  - There is an extra dictionary: "Rectangle_classDict" in the previous example
  - To call a function for a dictionary, we need to do two lookups: "functions" and "area"
- But it also has advantages:
  - The class dictionary tells everyone that the functions are related and operate on the same type of data dictionaries
  - All data dictionaries now have exactly two reserved keys: "tag" and "functions"
  - In fact, since all function calls go through the "functions" tag, we do not really need "tag" for much of anything
  - We can optimize things so that there is only one reserved key for each dictionary

More Function References (cont.)

```
def _area(r):
    return r["width"]* r["height"]
def _shift(r, dx, dy):
    r["origin"] = (r["origin"][0] + dx,
                   r["origin"][1] + dy)
Rectangle_classDict = {
    "__name__": "Rectangle",
    "area": _area,
    "shift": _shift,
}
def Rectangle(w, h, o):
    r = dict()
    r["__dict__"] = Rectangle_classDict
    r["width"] = w
    r["height"] = h
    r["origin"] = o
    return r
```

```
# Create a Rectangle data dictionary
r = Rectangle(10, 10, (0, 0))

# Compute rectangle area
print r["__dict__"]["area"](r)
```



OOP with Classes

- Python has syntax and built-in rules that aids in object-oriented programming

```
def _area(r):
        return r[“width”]* r[“height”]
def _shift(r, dx, dy):
        r[“origin”] = (r[“origin”][0] + dx,
                        r[“origin”][1] + dy)
Rectangle_classDict = {
        “__name__”: “Rectangle”,
        “area”: _area, “shift”: _shift,
}
def Rectangle(w, h, o):
        r = dict()
        r[“__dict__”] = Rectangle_classDict
        r[“width”] = w
        r[“height”] = h
        r[“origin”] = o
        return r
```

```
class Rectangle(object):
        def __init__(self, w, h, o):
                self.width = w
                self.height = h
                self.origin = o
        def area(self):
                return self.width * self.height
        def shift(self, dx, dy):
                self.origin = (self.origin[0] + dx,
                                self.origin[1] + dy)
```

```
r = Rectangle(10, 10, (0,0))
print r.area()
```

Class and Instance Syntax

- The class dictionary is created from the variables and functions defined within the **class** statement
- The class name doubles as the name of the function for creating *instances* of the class (dictionaries in our old terminology)
- Newly created instances are automatically tagged with the class dictionary

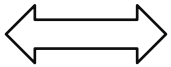
Class and Instance Syntax (cont.)

- Assigning to instance attributes are the equivalent to assigning values to keys in the data dictionary
- References to instance attributes are the equivalent to looking up keys in the data dictionary, ***except***
  - if the key is not in data dictionary, the class dictionary is automatically searched for a key of the same name
  - This is *inheritance*

Class and Instance Syntax (cont.)

- Using attributes and methods in instances is (almost) the same as using values out of a data dictionary

```
r = Rectangle()
r.width = 10
r.height = 10
r.origin = (10, 10)
print r.area()
```



```
r= dict()
r[“width”] = 10
r[“height”] = 10
r[“origin”] = (0,0)
print r[“__dict__”][“area”](r)
```

Class and Instance Syntax (cont.)

- Note that the dictionary-style function call

  r[“__dict__”][“area”](r)

  was translated to

  r.area()
- In neither case is there a reference to **Rectangle** since everything may be found from the instance
- In the dictionary-style call, we use **r** twice:
  - Once to find the function to call
  - Once to pass as the argument so that the function can find the dictionary on which to operate
- In the OO-style call, we only use **r** once:
  - Python automatically passes the instance, **r**, as the first argument to the called method, **area**

Methods

- This is because Python treats functions defined inside of class statements as **methods**, and the first argument to any method must be an instance of the class
  - By convention, the first argument is named "self"
- In fact, the code

  r.area()

  is equivalent to

  Rectangle.area(r)

Methods (cont.)

- The reason **r.area()** is preferred over **Rectangle.area(r)** is that the latter requires us to know the class of **r** whereas the former does not
  - The former only requires knowledge of the instance and the method
  - The latter requires instance, method and class (even when we can get class from instance)
  - This is important later in **polymorphism** where instances of *different* classes behave similarly

Special Methods

- In addition to executing methods when they are explicitly called, Python also calls “special methods” (if they exist) under certain conditions
- For example, when an instance is created, Python will call a method named “__init__” if it is defined in the class
- The initialization method is a perfect place for setting initial values for attributes (whether they are constant or supplied by the caller)

Special Methods (cont.)

- Instance initialization usually consists of setting some attributes

```
>>> class D1(object):
...     def __init__(self, size):
...             self.count = 0
...             self.size = size
...
>>> d1 = D1(12)
>>> d1
<__main__.D1 object at 0x00D39310>
>>> print d1.count, d1.size
0 12
>>>
```

- Any arguments in the instance creation call is passed through to the **__init__** method

Special Methods (cont.)

- Using an **__init__** method is easier to read and less prone to error (like forgetting to initialize an attribute)

```
class Rectangle(object):
        def __init__(self, w, h, o):
                self.width = w
                self.height = h
                self.origin = o
r = Rectangle(10, 10, (0,0))
r2 = Rectangle(20, 5, (2,4))
```

>

```
class Rectangle(object):
        pass
r = Rectangle()
r.width = 10
r.height = 10
r.origin = (0,0)
r2 = Rectangle() …
```

- Note that **__init__** is called ***after*** the instance has been tagged with the class dictionary so all methods are available for use

More Special Methods (cont.)

- Python calls the **__str__** method when printing an instance

```
>>> class Rectangle(object):
...     pass
...
>>> r = Rectangle()
>>> print r
<__main__.Rectangle object at 0x100599fd0>
>>> class Rectangle2(object):
...     def __str__(self):
...             return "I'm a Rectangle2 instance"
...
>>> r2 = Rectangle2()
>>> print r2
I'm a Rectangle2 instance
```

Operator Overloading

- Python can call functions when standard operators (such as +, -, * and /) are used
- For example, Python calls the **__add__** method when the left operand of an addition is an instance and **__add__** is defined for the class
- The **__radd__** method is called when the right operand of an addition is an instance
- If both sides are instances, **__add__** wins

Operator Overloading (cont.)

```
>>> class Vector(object):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __str__(self):
...         return "(%g,%g)" % (self.x, self.y)
...     def __add__(self, other):
...         return Vector(self.x + other.x, self.y + other.y)
...
>>> print Vector(1, 2) + Vector(2, 1)
(3,3)
```

- Operator overloading should only be used when the operator makes sense, eg vector addition
- Using them in other contexts only makes the code harder to understand

Debugging

- Invariants are data consistency requirements
- If we can identify invariants for our classes, we can put in **assert** statements in our methods to make sure that the invariants hold
- There may also be conditions that must be true at the start (pre-conditions) and end (post-conditions) of methods that we can test to make sure that, for example, arguments are reasonable and data consistency is maintained
- The earlier we can detect errors, the fewer red herrings we need to deal with

Homework

- 7.1 – convert 6.1 to use classes
- 7.2 – implement and use special method **__add__**