PC204 LECTURE 7

CONRAD HUANG

CONRAD@CGL.UCSF.EDU

GENENTECH HALL, N453A, X60415

TOPICS

- Homework review
- Object-oriented design
- Object-oriented programming with Python

HOMEWORK REVIEW

- **6.1** - Rectangle functions
- **6.2** - area_difference
- Object-oriented perspective
  - 6.1 defines and implements 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 (*i.e.* state and behavior)
- Collective, all objects that share the same behavior (but not necessarily the same data) form a **class**
- Programmatically, an object is represented by **attributes** (data/state) and **methods** (function/behavior)

OOD AND OOP CRITERIA

- Object-oriented design (OOD) focuses on *what* are the attributes and methods (the **interface**) of objects/classes while object-oriented programming (OOP) focuses on *how* to make the interface functional (the **implementation**).
- **Abstraction**: semantics defined by the interfaces of one or more classes. The quality of abstraction is discussed in terms of:
  - Cohesion: how much attributes and methods in a single class is related to each other (higher is better).
  - Coupling: how much interdependencies there are among a set of classes (lower is better).
- **Encapsulation**: concealment of implementation methods for a class. The degree of encapsulation is measured by how much internal data structures are accessible externally. (Appropriate degree of encapsulation depends on the application.)

WHY IS OOD/OOP A GOOD PARADIGM?

- For similar reasons why modules are a good idea:
  - Divide-and-conquer approach allows developers to focus on smaller problems.
  - Small, independent chunks of code (whether module or class) are easier to write and debug.
  - Independent chunks of code may be tested separately (**unit testing**) to verify proper operation before attempting **integration testing** for the entire system.
- And for reasons beyond them:
  - Classes may **inherit** from other classes to form class hierarchies with a low degree of code duplication.
  - Classes may be **polymorphic** so that they may be used interchangeably in generic algorithms.

DESIGNING "OBJECT-ORIENTED-LY"

- Design is still an art. People with different ideas come up with different abstractions.
- Programmatic objects often correspond to real-world objects.
- Frequently, enumerating a variety of subtasks helps identify classes. Nouns are class candidates. Nouns that occur in multiple contexts are better candidates because creating such a class would reduce code duplication by being shared by multiple subtasks.
- When the design may be achieved using several different sets of classes, compare cohesion and coupling among the solutions.
- In general, large classes that do many things should be reviewed to see whether they can be logically split into smaller classes. Several small classes that refer to each other very frequently should be reviewed to see whether they can be combined into a single class with a smaller interface.

OOP IN PYTHON

- Implementing a **Rectangle** class in Python:

```
class Rectangle(object):
  NUMBER_OF_SIDES = 4
  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]
```

- Using the class:

```
r1 = Rectangle(10, 20, (0,0))
r2 = Rectangle(20, 10, (-10,5))
print r1.origin
r1.shift(5, 12)
print "r1", r1.origin, r1.area()
print "r2", r2.origin, r2.area()
```

OBJECTS, CLASSES, INSTANCES

- In Python, *everything* is an **object**.
  - In Python 2, an object has a **type** and possibly a **class**
  - In Python 3, an object's type is its class.
- A **class** object defines, among other things, a collection of **methods** (verbosely **instance methods**) that define how **instances** of the class behave.
- An **instance** is an object that is associated with a **class** object. There may be many instances associated with the same class object.
  - All instances of the same class share the same methods defined by their class object.
  - Each instance has its own attribute values that are distinct from attribute values of other instances.
  - When a method is defined in a class, it can refer to instance attributes.
  - A method *cannot* be called unless it is **bound** to an instance. This is because references to instance attributes are undefined unless there is a definitive instance in which to search for attribute values.

CLASSES

- A class object is defined by a **class** statement:
  - A class statement is introduced by the **class** keyword:

```
class Rectangle:
```

  - In Python 2, the above statement creates an **old-style class** while the preferred **new-style class** is defined with:

```
class Rectangle(object):
```

- **def** statement in a class statement define methods of the class.
  - Method definitions look exactly like function definitions, except the first argument of a method is conventionally named **self**.
- Assignment statements in class statement define **class attributes** not **instance attributes**.
  - Class attributes are associated with the class object, not instances.
  - Unlike instance attributes (which are unique for each instance), all instances of the class share the same class attribute.

METHODS

- Defining a method is exactly analogous to defining a function, except the **def** statement is indented inside a class statement.
- All methods take at least one argument. The first argument refers to the instance in which to look for any referenced instance attributes. By convention, the first argument is named **self**.
- In our example:

```
class Rectangle(object):
  [...]
  def area(self):
    return self.width * self.height
  [...]
```

  we define the method **area**. When **area** is invoked, the first argument, **self**, is assigned to a **Rectangle** instance, and values for the **width** and **height** attributes from that instance are used to calculate the return value.
- Python will make sure that the first argument is an instance of the correct class. For example, if we have two classes, **Rectangle** and **Triangle**, that both define **area** methods, Python will ensure that **Rectangle.area** will not be called with an instance of **Triangle** as **self**.

CLASS ATTRIBUTES

- Although used infrequently, class attributes come in handy for keeping track of information about the class object (not instances) and for defining shared constants
- In our example:

```
class Rectangle(object):
  NUMBER_OF_SIDES = 4
  [...]
```

  we define a class attribute **NUMBER_OF_SIDES**. There is only one **NUMBER_OF_SIDES** shared by the class object and all instances of **Rectangle**, *i.e.*, changing the value of **NUMBER_OF_SIDE**:

```
Rectangle.NUMBER_OF_SIDES = 6
```

  will change it for all instances as well as the class object.
- (**NUMBER_OF_SIDES** is all caps to conform with Python naming convention. Google for "Python PEP 8".)

CLASS NAMESPACE

- Each class has its own namespace. Different classes may use the same method and class attribute names without worrying about ambiguity. You can think of classes as something like mini-modules.
- Methods and class attributes share the same namespace within a class.
  - If there are multiple **def** statements defining methods with the same name, last one wins.
  - If there are multiple assignment statements defining class attributes with the same name, last one wins.
  - If there definitions of methods and class attributes with the same name, last one wins.

  The moral of the story: pick unique names within a class.

INSTANCES

- A class statement defines a single class object. To create instances of a class, the class object is used as if it were a function:

```
class Rectangle(object):
  NUMBER_OF_SIDES = 4
  def __init__(self, w, h, o):
    self.width = w
    self.height = h
    self.origin = o
  [...]
r1 = Rectangle(10, 20, (0,0))
r2 = Rectangle(20, 10, (-10,5))
[...]
```

- When a class object is called as a function:
  1. it creates an instance of the class;
  2. if there are arguments to the function call, the **__init__** method is invoked with **self** set to the newly created instance and the function call arguments passed as additional arguments to **__init__**.
- As its name suggests, **__init__** is where instance initialization should occur, *e.g.*, setting instance attributes according to passed arguments. All methods and class attributes may be used as part of instance initialization.

INSTANCES (CONT.)

- An instance has access to:
  - its own instance attributes (data unique to the instance and not shared with neither its class object nor other instances of the class),
  - class attributes (data associate with the class object), and
  - methods (functions defined in the class object).
- Instance attributes, class attributes, and methods are all accessed using the same syntax of *instance.name*, where *name* is the name of an attribute or method.

INSTANCES (CONT.)

- In this example:

```
class Rectangle(object):
  NUMBER_OF_SIDES = 4
  def __init__(self, w, h, o):
    self.width = w                    # set instance attribute
    self.height = h
    self.origin = o
  def area(self):
    return self.width * self.height   # use instance attributes
  [...]
r1 = Rectangle(10, 20, (0,0))         # create instance
print r1.width, r1.height             # use instance attribute
print r1.NUMBER_OF_SIDES              # use class attribute
print r1.area()                       # call method
```

  we define a class **Rectangle**, create an instance of the class, and then proceed to print some values for the instance.
- When we create **r1**, **__init__** is implicitly called with **self** set to the new instance and **w**, **h** and **o** set to 10, 20 and (0,0), respectively.
- The three **print** statements consecutively access instance attributes (set in call to **__init__**), a class attribute (defined by class statement), and call a method. Even though the syntax for all three statements are similar, they access different types of data associated with **r1**.

INSTANCE ATTRIBUTES

- Because an expression *instance.name* may potentially refer to several types of data associated with the instance, we need some precedence rules to take care of possible ambiguities.
- For retrieving data:
  1. If *name* matches an instance attribute, the value of the instance attribute is used;
  2. If *name* matches a class attribute or method in the class object, use that value (note that the name cannot match both a class attribute *and* a method since defining one overrides the other);
  3. Raise **AttributeError** exception.
- For defining data (usually an assigment statment):
  1. If *name* matches an instance attribute, the attribute is updated with the new value;
  2. If *name* does not match an instance attribute, *create* an instance attribute with the new value.
- Note the asymmetry of accessing and setting attributes.

INSTANCE ATTRIBUTES (CONT.)

- Each instance is its own namespace.
- For retrieving an attribute value, the instance namespace is checked first for the attribute name. If the name is not found in the instance namespace, the class object namespace is checked. (This is similar to the idea of LSGB scoping where a name is searched in progressively less specific namespaces.)
- For assigning an attribute value, *only* the instance namespace is used. The assignment either replaces an existing value, or creates a new value. There are two ramifications:
  - New attributes may be created for each instance, independently of the class object or other instances. So different instances may have different attributes. (That is not generally considered a good idea, but Python allows it.)
  - Instance attributes can **shadow** class attributes (as the example below shows).

INSTANCE ATTRIBUTES (CONT.)

- The following example illustrates the pitfalls of undisciplined use of instance attribute names:

```
class Rectangle(object):
  NUMBER_OF_SIDES = 4
  def __init__(self, w, h, o):
    self.width = w
    self.height = h
    self.origin = o
  def area(self):
    return self.width * self.height

r = Rectangle(20, 10, (0,0))
print r.NUMBER_OF_SIDES          # use class attribute
r.NUMBER_OF_SIDES = 6            # define _instance_ attribute
print r.NUMBER_OF_SIDE           # uses new instance attribute!
print r.area()                   # call method
r.area = 300                     # define instance attribute
print r.area()                   # call 300 as function?!
```

CALLING METHODS

- Calling a method is slightly more complicated than calling a function. The following example shows two different ways of calling a method:

```
class Rectangle(object):
  NUMBER_OF_SIDES = 4
  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(20, 10, (0,0))
print r.area()                  # Call _bound_ method
print Rectangle.area(r)         # Call _unbound_ method
```

CALLING METHODS (CONT.)

- A method may be found in two ways:
  - Associated with an instance, *e.g.*, **r.area**. In this case, the found method is called a **bound method** because there is already an instance, **r**, associated with how the method was found.
  - Associated with a class, *e.g.*, **Rectangle.area**. In this case, the found method is called an **unbound method** because there is no instance associated with how the method was found.
- When an bound method is called, Python implicitly inserts the instance used to find the method as the method's first argument, **self**. That is why even though **area** is defined to take one argument, the call **r.area()** passes zero arguments.
- When an unbound method is called, Python has no idea what instance should be used and therefore does *not* insert the first argument. That is why we must explicitly pass **r** as the first argument in **Rectangle.area(r)**.
- Bound methods are the preferred way to call methods because the class is not named explicitly. The advantage of this will be discussed in the **inheritance** and **polymorphism** topics.

CALLING METHODS (CONT.)

- A very common error with using methods is shown below:

```
>>> class C(object):
...     def m(self, v):
...             print self, v
...
>>> i = C()
>>> i.m()
Traceback (most recent call last):
  File "", line 1, in
TypeError: m() takes exactly 2 arguments (1 given)
>>> i.m(i, 10)
Traceback (most recent call last):
  File "", line 1, in
TypeError: m() takes exactly 2 arguments (3 given)
```

- Remember to account for the implicit first argument in a call to a bound method.

SPECIAL METHODS

- In addition to executing methods when they are explicitly called, Python also calls "special methods" (if they exist) under certain conditions. All special methods have names that begin with __ (double underscore).
  - __ prefixes get special treatment from Python and you should not use it unless you know exactly what you are doing.
- For example, when an instance is created, Python calls a method **__init__** if it is defined in the class.
- Another special method is **__str__** which is called to convert an instance into its string representation.

__STR__

- **__str__** overrides the default string representation that Python uses for instances:

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

- A better choice may be to include the rectangle origin and size in the output.

OPERATOR OVERLOADING

- Python can call methods when standard operators (*e.g.*, **+**, **–**, ***** and **/**) are used
- If one of the operands of the operator is an instance of a class that defines a corresponding special method, then the method is called with the operand(s) as arguments. The return value of the special method is expected to "make sense".
- For example:

```
>>> 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):
...     "Overload + operator"
...     return Vector(self.x + other.x, self.y + other.y)
...
>>> print Vector(1, 2) + Vector(2, 4)
(3, 6)
```

- The **__add__** method is called automatically when the left operand of **+** is an instance of **Vector**

OPERATOR OVERLOADING (CONT.)

- Other operators that can be overloaded include list or map lookup (**[]**), function call (**()**), comparison (**<**, **>**, *etc.*) and even getting or setting attributes (**.**).
- See the *Data model* page (**Python 2**, **Python 3**) for the full list of operators that you can overload and the names of their corresponding special methods.
- Operator overloading should only be used when the operator "makes sense", *e.g.* overriding **+** for vectors. Gratuitous use of operator overloading can easily lead to completely inscrutable code.

DEBUGGING

- **Invariants** are data consistency requirements
- In general, we can use **assert** statements in our methods to make sure that the invariants hold
- Two special cases are **pre-conditions**, which are invariant tests made just after a method is called, and **post-conditions**, which are tests made just before a method returns. Together, they (try to) guarantee that an object is always in a consistent state.
- Wide use of pre- and post-conditions helps developers detect inconsistencies early, and minimize red herrings that derive from propagation of bad data.

HOMEWORK

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