Inheritance, Multiple Representations, Generic Functions 4a

Computer Science 61A

July 16, 2013

1 Inheritance

So far, we've been working with objects by defining classes and creating instances. Now that you are familiar with how object system works, let's explore another powerful tool that comes with objects system–inheritance.

Consider writing `Dog` and `Cat` classes. You can imagine that they'd both have `name`, `age`, and `owner` instance variables, and also `eat` and `talk` methods. That's a lot of effort for writing the same code! This is where Inheritance steps in. In Python, you can create a class and have it inherit the instance variables and methods of a *parent* class without typing it all out again. All of our classes thus far have been inheriting from the `object` class. They are *children* of the `object` class. `Object` is the top-level, generic mack-daddy of all classes. It provides basic functionality for all objects. This is an example of *code reusability*, the idea that you shouldn't reinvent the wheel if at all possible.

When do you want to inherit? The rule-of-thumb is when there is a hierarchical relationship between two classes, where one is a type or sub-categorization of the other. This is commonly know as a "is a" relationship. A truck "is a" type of vehicle and thus could be a child class of a `Vehicle` class. Make sure you don't get this confused with "has a" relationship. A truck has a color, and therefore color would be an instance variable of `Truck`, not a child class.

Python has some particular syntax when it comes to inheritance. Take a look at this partial implementation of animals:

```
current_year = 2013

class Animal(object):
```

```
    def __init__(self):
        self.is_alive = True # It's alive!!!

class Pet(Animal):
    def __init__(self, name, year_of_birth, owner=None):
        Animal.__init__(self) # call the parent's constructor
        self.name = name
        self.age = current_year - year_of_birth
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print("...")

class Dog(Pet):
    def __init__(self, name, yob, owner, color):
        Pet.__init__(self, name, yob, owner)
        self.color = color
    def talk(self):
        print("Woof!")
```

1.1 Questions

1. What does the following code do?

```
>>> fido = Dog('Fido', 1993, 'Joe', 'golden')
>>> clifford = Dog('Clifford', 1963, 'Emily', 'red')
>>> fido.age


>>> fido.talk()


>>> fido.owner


>>> clifford.owner


>>> clifford.color


>>> clifford.eat('bone')
```

2. Now write a `Cat` class that inherits from `Pet`. Use parent methods wherever possible:

```
class Cat(Pet):
    def __init__(self, name, yob, owner, lives=9):
        """Assume lives is a positive integer."""




    def talk(self):
        """A cat says 'Meow!' when asked to talk."""



    def lose_life(self):
        """A cat can only lose a life if they have
        at least one life. When lives reach zero,
        the 'is_alive' variable becomes False.
        """
```

3. More Cats!

```
class NoisyCat(Cat):
    """A class that behaves just like a Cat, but always
    repeats things twice.
    """
    def __init__(self, name, yob, owner, lives=9):



    def talk(self):
        """A NoisyCat will always repeat what he/she said
        twice.
        """
```

2 Multiple Representations

The ability to represent data using different representations without breaking the modularity of a program rests on our ability to define a common message interface for the data type.

So what exactly is an interface? An *interface* is the set of messages that a data type understands and can respond to. If we are talking about an object, then we can say that its interface is made up of all of its methods and attributes. For instance, the interface for the Person class defined in the previous section consists of the name attribute, the say, ask, and greet methods, as well as the attributes and methods of its ancestor classes.

When implementing a common interface for an abstract data type that has multiple representations, there must be a subset of messages that both representations understand. This set of common messages is the common interface. A system that uses multiple data representations and is designed with common interfaces is modular because one can add any number of different representations without needing to change code already written. All the implementer needs to do is to ensure that the new representation understands the messages required by the interface.

2.1 Questions

1. What do Python strings, tuples, lists, dictionaries, ranges, etc. all have in common? Hint: What happens when you toss one of these data types into a for loop?

2. Why can't you put something else, say an integer, into the for loop?

```
>>>for elem in 5:
  print(elem)
Error!
```

3. Suppose that these datatypes all implement a common interface called Iterable that supports the attributes **'current'** and **'next'**. The 'current' attribute starts out being the

first element in the datatype. Each time we call the 'next' method on the datatype, 'current' becomes the next element in the Iterable datatype. If 'current' is the last element, then calling 'next' will cause 'current' to be set to *None*.

Write a code snippet that can implement a for loop that prints out each element using this common interface. You may use the 'current' and 'next' attributes through dot notation. (The task here is simple, but the ideas are important. We can use this common interface to iterate over lists, tuples, and ranges, which are sequences, as well as dictionaries, which are NOT sequences.)

```
data = create_data()
# assume 'data' is our datatype, which
# implements Iterable
```

4. After acing CS61A and becoming a renowned professor, you invent a new datatype with magical properties. Because of the fond memories you have of your first computer science course at Berkeley, you decide that the new datatype should implement the Iterable interface described during your 4th week discussion section. On a high level, what do you need to do?

3 Generic Operators

In the previous section, we saw how to work with multiple representations of data, by forcing each of the representations to use a common method interface. But suppose we wanted to generalize this further. Could we write functions that work with arguments that don't even work with a common interface?

We are going to employ *type dispatching*. The idea: our generic functions will see arguments of various data types. We can inspect what type of data the argument is. Now suppose we have been keeping a table that holds functionality for interacting with specific data types. We can simply look up the argument's data type in the table, which will return to us a function that will work with the argument's data type.

3.1 Type Dispatching

Revisiting the complex number example, we have:

```
def type_tag(x):
    return tags[type(x)]

tags = {ComplexRI: 'com', ComplexMA: 'com', Rational: 'rat'}
```

Now `tags` is a dictionary that associates data types (specifically, a class name) with a key word that we can use to look up the type tag.

Next, we can implement a generic add function:

```
def add(z1, z2):
    types = (type_tag(z1), type_tag(z2))
    return add_implementations[types](z1, z2)

add_implementations = {}
add_implementations[('com', 'com')] = add_complex
add_implementations[('com', 'rat')] = add_complex_and_rational
add_implementations[('rat', 'com')] = lambda x, y:
add_complex_and_rational(y, x)
add_implementations[('rat', 'rat')] = add_rational
```

So what happens when we call `(ComplexRI(2, 3), ComplexRI(4, 5))`? Let's refer to the two complex numbers as $z1$ and $z2$. `type_tag` looks up the tag for each of them and returns `com` and `com`. We then look up (`com`, `com`) in our dictionary of supported implementations of add and see that we should use `add_complex`. The function `add` then calls `add_complex(z1, z2)`, which works without a hitch because all the data types match up.

3.2 Questions

The TAs have broken out into a cold war; apparently, at the last midterm-grading session, someone ate the last slice of Cheeseboard and refused to admit it. It is near the end of the semester, and Albert really needs to enter the grades. Unfortunately, the TAs represent the grades of their students differently, and refuse to change their representation to match someone else's. Albert has asked you to look into writing generic functions for Rohan's and Leonard's student records.

1. Rohan and Leonard have agreed to release their implementations of student records, which are given below:

   ```
   class RC_record(object):
   ```

```
"""A student record formatted via Rohan's standard"""
    def __init__(self, name, grade):
    """name is a string containing the student's name,
    and grade is a grade object"""
        self.student_info = [name, grade]

class LT_record(object):
"""A student record formatted via Leonard's standard"""
    def __init__(self, name, grade):
    """name is a string containing the student's name,
    and grade is a grade object"""
        self.student_info = {'name': name, 'grade': grade}
```

Write functions `get_name` and `get_grade`, which take in a student record and return the name and grade, respectively.

```
def type_tag(x):
    return tags[type(x)]

tags = {RC_record: 'RC', LT_record: 'LT'}
```

2. Rohan and Leonard also use their own grade objects to store grades. Here are the definitions for their grade class:

```
class RC_grade(object):
    def __init__(self, total_points):
        if total_points > 90:
            letter_grade = 'A'
        else:
            letter_grade = 'F'
        self.grade_info = (total_points, letter_grade)

class LT_grade(object):
    def __init__(self, total_points):
        self.grade_info = total_points
```

Write a function `compute_average_total`, which takes in a list of records (that could be formatted via either standard) and computes the average total points of all the students in the list.

3. Lastly, Albert needs you to convert all student records into the format that he uses. Unlike Rohan and Leonard, Albert is actually helpful and provides the class definition of his formatted student records. Unfortunately, his email was corrupted so you can only see the first few lines of his class definition:

```
class AW_grade(object):
"""A student record formatted via Albert's standard"""
    def __init__(self, name_str, grade_num):
    """NOTE: name_str must be a string, grade_num must be a number"""
```

Write a function `convert_to_AW` which takes a list of student records formatted either using Rohan's or Leonard's standard, and returns a list of the same student records but now formatted using Albert's standard.

```
def convert_to_AW(records):
```