INHERITANCE, MULTIPLE REPRESENTATION, GENERIC FUNCTIONS 8

COMPUTER SCIENCE 61A

March 13, 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, (it's subtle). 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
```

Solution:

19

```
>>> fido.talk()
```

Solution:

```
Woof!
```

```
>>> fido.owner
```

Solution:

```
'Joe'
```

```
>>> clifford.owner
```

Solution:

```
'Emily'
```

```
>>> clifford.color
```

Solution:

```
'red'
```

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

Solution:

```
Clifford ate a 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):
```

Solution:

```
        Pet.__init__(self, name, yob, owner)
        self.lives = lives
```

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

Solution:

```
print("Meow!")
```

```
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.
    """
```

Solution:

```
if self.lives > 0:
    self.lives -= 1
    if self.lives == 0:
        self.is_alive = False
else:
    print("This cat has no more lives to lose :(")
```

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):
```

Solution:

```
Cat.__init__(self, name, yob, owner, lives)
```

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

Solution:

```
Cat.talk(self)
```

```
        Cat.talk(self)
```

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?

   Solution: All of these data types implement a common interface that allows easy iteration over all of its elements.

2. Why cant you put something else, say an integer, into the for loop?

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

   Solution: The int type does not implement the standard interface that the other data types implement.

3. Suppose that these datatypes all implement a common interface called Iterable that expects the messages **'current'** and **'next'**. The 'current' attribute starts out being the first element in the datatype. Each time we pass the 'next' message to the datatype, current becomes the 'next' element in the Iterable datatype. If 'current' is the last element, then passing '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 pass messages to the datatype using dot notation. (The task here is simple, but the ideas are important. We can use this common interface to iterate over both lists, tuples, and ranges, which are sequences, as well as dictionaries, which are NOT sequences.)

   ```
   data = create_data()
   ```

   Solution:

   ```
   while data.current != None:
       print(data.current)
       data.next
   ```

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 8th week discussion section. On a high level, what do you need to do?

   Solution: You must ensure that your new datatype implements the common Iterable interface. In this example, the interface includes the 'current' and 'next' messages

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 dont 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 spe-

cific data types. We can simply look up the arguments data type in the table, which will return to us a function that we know will work with the arguments data type.

3.1 Type Dispatching

Revisiting the complex number example, we have:

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

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

Now `type_tag.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))`? Lets refer to the two complex numbers as $z1$ and $z2$. `type_tag` looks up the tag for each them and returns com and com. We then look up (com, com) in our table of supported implementations of add and see that we should use `add_complex`. We then invoke `add_complex(z1, z2)` which works without a hitch because all the data types match up.

3.2 Questions

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

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

```
class HN_record(object):
"""A student record formatted via Hamilton'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 JO_record(object):
"""A student record formatted via Julia'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.

```
type_tag.tags = {HN_record: 'HN', JO_record: 'JO'}
```

Solution:

```
def get_name(record):
    data_type = type_tag(record)
    return get_name.implementations[data_type](record)

def get_grade(record):
    data_type = type_tag(record)
    return get_grade.implementations[data_type](record)

get_name.implementations = {}
get_name.implementations['HN'] = lambda x: x.student_info[0]
get_name.implementations['JO'] = lambda x: x.student_info['name']
get_grade.implementations = { }
get_grade.implementations['HN'] = lambda x: x.student_info[1]
get_grade.implementations['JO'] = lambda x: x.student_info['grade']
```

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

```
class HN_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 JO_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.

Solution:

```
type_tag.tags[HN_grade] = 'HN'
type_tag.tags[RL_grade] = 'JO'

def get_points(grade):
    data_type = type_tag(grade)
    return get_points.implementations[data_type](grade)

def compute_average_total(records):
    total = 0
    for rec in records:
        grade = get_grade(rec)
        total += get_points(grade)
    return total / len(records)
```

3. Lastly, Amir needs you to convert all student records into the format that he uses. Unlike Hamilton and Julia, Amir 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 AK_grade(object):
"""A student record formatted via John'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_AK` which takes a list of student records formatted either using Hamilton's or Julia's standard, and returns a list of the same student records but now formatted using Amirs standard.

```
def convert_to_AK(records):
```

Solution:

```
    list_of_AK = []
    for rec in records:
        name = get_name(rec)
        points = get_points(get_grade(rec))
        list_of_AK.append(AK_record(name, points))
    return list_of_AK
```