MULTIPLE REPRESENTATION, GENERIC FUNCTIONS 8

COMPUTER SCIENCE 61A

October 24, 2013

1 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 a Person class might consist 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.

1.1 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 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()
```

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?

2 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 we know will work with the argument's data type.

2.1 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 `add(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 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.

2.2 Questions

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

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

```
class KM_record(object):
    """A student record formatted via Keegan'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 = {KM_record: 'KM', JO_record: 'JO'}
```

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

```
class KM_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.

3. Lastly, John needs you to convert all student records into the format that he uses. Unlike Keegan and Julia, John 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 JD_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_JD` which takes a list of student records formatted either using Keegan's or Julia's standard, and returns a list of the same student records but now formatted using John's standard.

```
def convert_to_JD(records):
```