CS61A Notes – Week 8: OOP Below the Line, Multiple Representations, and Generic Functions

So far, we've been working with objects by defining classes and creating instances. Now, we will dive below the abstraction barriers that object-oriented programming creates to see how we can implement an object system. We'll also take a look at implementing data types using multiple representations and generic functions for working with many different, but similar data types.

OOP Below the Line

The python OOP system probably seems pretty magical right now. OOP is flexible enough to allow you to do a wide variety of things, while still being simple and intuitive to use. Today we are going to explore how one might implement an OOP system.

The first thing we should ask ourselves is, fundamentally, what is it that we need objects to be able to do? Surprisingly, we can boil it down to three basic tasks - we need to be able to set the value of an attribute of an object, we need to be able to get the value of an attribute of an object (remember: methods, i.e. functions, are values too!), and we need to be able to make (instantiate) new objects. One way that we can implement all three tasks is to use a dispatch dictionary.

Let's look at how we implement a single instance of a class:

```
def make_instance(cls):
    """Return a new object instance."""

    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)

    def set_value(name, value):
        attributes[name] = value

    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
```

There's a fair amount of stuff going on here, but the most important thing to note is that an instance is simply represented by a dictionary that contains two keys: '**get**' and '**set**'. When we want to get a value of an attribute from an instance, we pass it a 'get' message (i.e. look up 'get'), which returns to us an internally defined get_value function that we can now invoke on the name of the attribute to retrieve its value. This is helpful, as it means we no longer have to worry about how get_value is implemented.

Now to move on to classes:

```
def make_class(attributes, base_class=None):
    """Return a new class."""

    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)

    def set_value(name, value):
        attributes[name] = value

    def new(*args):
        return init_instance(cls, *args}

    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
```

The first thing we want to note is that make_class is strikingly similar to make_instance. There are three differences to point out:

1. make_instance takes a class as argument. make_class takes two arguments, attribute, which is a dictionary of default attributes, and base_class, which is the class you are inheriting from.
2. Whenever an instance is asked to get a name that isn't in its own attribute dictionary, it will ask its class. Whenever a class is asked to get a name that isn't in its own attribute dictionary, it will ask its base_class. Think about what this parallels in Python's regular OOP.
3. A class understands an additional third key in its dictionary, 'new', which can be used to make new instances of the class.

And here's an example of using make_class to define the Account class:

```
def make_account_class():
    """Return the Account class."""

    def __init__(self, account_holder):
        self['set']('holder', account_holder)
        self['set']('balance', 0)

    def deposit(self, amount):
        """Increase the account balance by amount."""
        new_balance = self['get']('balance') + amount
        self['set']('balance', new_balance)
        return self['get']('balance')
```

```
    def withdraw(self, amount):
        """Decrease the account balance by amount."""
        balance = self['get']('balance')
        if amount > balance:
            return 'Insufficient funds'
        self['set']('balance', balance - amount)
        return self['get']('balance')
```

```
    return make_class({'__init__': __init__, 'deposit': deposit,
                       'withdraw': withdraw, 'interest': 0.02})
```

Questions

1. In which attributes dictionary are method definitions stored? (instance or class dictionary)

   Method definitions are stored in the class dictionary

2. The definition of make_account_class above is incomplete. Fill in the definition of withdraw to complete it.

3. Using make_class, redefine the Person class from lab last week. Here's the regular OOP version for reference:

```
class Person(object):

    def __init__(self, name):
        self.name = name

    def say(self, stuff):
        return stuff

    def ask(self, stuff):
        return self.say("Would you please " + stuff)

    def greet(self):
        return self.say("Hello, my name is " + self.name)
```

```
def make_person_class():
    def __init__(self, name):
        self['set']('name', name)

    def say(self, stuff):
        return stuff

    def ask(self, stuff):
        return self['get']('say')("Would you please " + stuff)

    def greet(self):
        return self['get']('say)('Hello, my name is '+ self['get']('name'))

    return make_class({'__init__': __init__, 'say': say,
           'ask': ask, 'greet': greet})
```

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.

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?

```
>>>for elem in [3, 4, 5]:
...    print(elem)
3
4
5

>>>for elem in {3:'a', 4:'b', 5:'c'}:
...    print(elem)
3
4
5
```

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

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

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

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()
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?

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

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.

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.

Question: 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 Hamilton's and Richard's student records.

1. Hamilton and Richard 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, grade is a
           grade object”””
        self.student_info = [name, grade]

class RL_record(object):
    """A student record formatted via Richard's standard"""
    def __init__(self, name, grade):
    “””name is a string containing the student’s name, 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', RL_record: 'RL'}

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['RL'] = lambda x: x.student_info['name']

get_grade.implementations = { }
get_grade.implementations['HN'] = lambda x: x.student_info[1]
get_grade.implementations['RL'] = lambda x: x.student_info['grade']
```

2. Hamilton and Richard 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 RL_grade(object):
    def __init__(self, total_points):
```

```
        self.grade_info = total_points
```

John needs you to 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.

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

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, John needs you to convert all student records into the format that he uses. Unlike Hamilton and Richard, 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_record(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"""

    O#F3jrjfw%783023$*($#%)@NIFVN#*R#k329r9F#jrPfj3sh83
```

Write a function convert_to_JD which takes a list of student records formatted either using Hamilton's or Richard's standard, and returns a list of the same student records but now formatted using John's standard.

```
def convert_to_JD(records):
    list_of_JD = []
    for rec in records:
        name = get_name(rec)
        points = get_points(get_grade(rec))
        list_of_JD.append(JD_record(name, points))

    return list_of_JD
```