Bilkent University
Computer Engineering Department
CS 315 - Programming Languages

Spring 2016
Due Apr 4, 2016 23:59
Note that plagiarism is not allowed and we will strictly check your homeworks

Homework II

In this homework you will implement currying using OO techniques and operator overloading in Python. Currying is the technique of transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions. This is better explained using an example.

Say we have a function `add` that adds a bunch of numbers. Rather than writing `add(3, 5, 4, 1)` we want to use currying to create an adder function that can be extended using a chain of calls. We would then have `adder(3)(5)(4)(1)()`. Let us assume we have the currying function that can create this adder given the `add2` function (the binary version of `add`) and a start value. Let us call it `curry`. Then we have `adder = curry(add2, 0)`.

In the general case, `curry(add2, ival)` returns us an `adder` function. That `adder` function can return two results:

1. If a parameter is supplied, returns another adder function like itself by adding the parameter value to its current value (e.g. `adder(aval)`),
2. If no parameter is supplied, returns its current value. A few examples follow:

```
add2 = lambda x, y : x + y
adder = curry(add2, 5)    # creates a 5 adder
adder = adder(3)          # creates an 8 adder
print adder()             # prints 8
print adder(2)()          # prints 10
print adder(1)()          # prints 9
adder = adder(3)          # creates an 11 adder
print adder()             # prints 11
```

Now assume that the `curry` function takes an optional parameter called `kind`. If its value is `'Eager'` (which is the default), then the resulting curried function behaves such that when a new value is added via (`val`) the new result is computed right away, eagerly. If the parameter value is `'Lazy'`, then the resulting curried function behaves such that when a new value is added via (`val`) it is just *remembered* and the entire result is computed lazily, when the `()` operation is performed for the first time. Here is an example:

```python
    adder0 = curry(add2, 0)          # Eager
    adder18 = adder0(3)(4,5)(6)
    print adder18()                  # prints 18
    print adder18(2)()               # prints 20
    print

    lazyAdder0 = curry(add2, 0, 'Lazy')
    lazyAdder18 = lazyAdder0(3)(4,5)(6)
    print lazyAdder18()              # prints 18
    print lazyAdder18(2)()           # prints 20
```

The two code segments above would behave exactly the same, because the laziness of the evaluation does not change the result. However, consider the following alternative:

```python
def sleepingAdd(x, y):
        import time
        for i in xrange(0, y):
            print ".",
            time.sleep(1);
        return x + y;

    sadder0 = curry(sleepingAdd, 0)
    print "Hey",
    sadder10 = sadder0(3)(4,2)(1) # sleeps 10 secs (prints 10 dots)
    print "Yo"
    print sadder10()       # prints 10
    print "Hey",
    sadder10()             # returns 10
    print "Yo"
    print

    lazySadder0 = curry(sleepingAdd, 0, 'Lazy')
    print "Hey",
    lazySadder10 =  lazySadder0(3)(4,2)(1)
    print "Yo"
    print lazySadder10() # sleeps 10 secs (prints 10 dots), prints 10
    print "Hey",
    lazySadder10()       # returns 10
    print "Yo"
```

This time, we replaced the `add2` function that was curried with the `sleepingAdd` function. The latter is a function with a side effect, as it prints dots to the output. A consequence of this is that, the two outputs corresponding to the above two code segments would be different. Here is what the output looks like:

```
Hey . . . . . . . . . . . . Yo
10
Hey Yo

Hey Yo
. . . . . . . . . . . . 10
Hey Yo
```

Deliverables

There are two deliverables:

1. Write a report explaining what you understand from eager and lazy evaluation. Describe why the above example gives two different outputs.
2. Implement the `curry` function. Some hints follow:
   - The `curry` function will return an object. This object would be an instance of a class you write. In fact, you have to write two classes: one for the eager version and one for the lazy version.
   - The class you write will overload the `()` operator. Depending on the number of arguments, it will behave differently. If there are no arguments, then the result of the computation is returned. If there are one or more arguments, then a new object that incorporates additional values are returned.
   - In summary, you would be using objects to emulate functions, via overloading the `()` operator.

Note: This homework requires learning about the following things in Python:

- Defining functions and lambdas (only used in the example code above)
- Handling variable number of arguments in functions
- Defining classes, instantiating and using objects
- Operator overloading

Logistics

Put your code and report under a directory named `lastname_name_hw2` and make an archive from that directory. Your report should be named `lastname_name_hw2.pdf` (or `.txt`). For example, the following Unix commands could be used:

```
$ mkdir lastname_name_hw2
$ cd lastname_name
...
# (edit and test your files in this directory)
...
$ cd ..
$ tar -cvzf lastname_name_hw2.tar.gz lastname_name_hw2
```

Then e-mail this newly generated file (named `lastname_name_hw2.tar.gz`) with subject `CS315-HW2` to Arda Unal ([firstname].[lastname]@bilkent.edu.tr). Reports in formats other than `.txt` and `.pdf` are not accepted.

Note that plagiarism is not allowed and we will strictly control for that.