CS61A Lab 4: Data Abstraction and Intro to Sequences

Week 4, 2012

Today we get to explore the ideas of DATA ABSTRACTION AND SEQUENCES!!!

Exercise 1: Data Derping

Try typing these into python, and think about the results:

```
x = (4, 5)
x[0]
x[1]
x[2]
y = ('hello', 'goodbye')
z = (x, y)
z[1]
(z[1])[0]
z[1][0]
z[1][1]
```

Exercise 2: Data Herp Derping

Predict the result of each of these before you try it:

```
z[0][1]
(8, 3)[0]
z[0]
3[0]
```

Exercise 3: Abstracting Rational Numbers

Enter these definitions into Python:

```
def make_rat(num, den):
    return (num, den)

def num(rat):
```

```
    return rat[0]

def den(rat):
    return rat[1]

def mul_rat(a, b):
    new_num = num(a) * num(b)
    new_den = den(a) * den(b)
    return make_rat(new_num, new_den)

def str_rat(x):  #from lecture notes
        """Return a string 'n/d' for numerator n and denominator d."""
        return '{0}/{1}'.format(num(x), den(x))
```

Now try this, be sure to predict the results first!

```
str_rat(make_rat(2, 3))
str_rat(mul_rat(make_rat(2, 3), make_rat(1, 4)))
```

Exercise 4: div_rat

Define a procedure div_rat to divide two rational numbers in the same style as mul_rat above

Exercise 5: Segments

Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor make-segment and selectorsstart-segment and end-segment that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the x coordinate and the y coordinate. Accordingly, specify a constructor make-point and selectors x-point and y-point that define this representation. Finally, using your selectors and constructors, define a procedure midpoint-segment that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints)

Exercise 6: Tuples

0.) (Note: remember, computer scientists usually start counting from 0!) Try to guess what

Python will do for the following expressions. Check your answer using the Python interpreter.

```
>>> for item in (1, 2, 3, 4, 5):
...      print(item * item)
...

>>> len((1, 2, 3, 4, 5))

>>> (1, 3, 9) * 3

>>> len((1, 5, 10) * 2)

>>> len(1, 2, 3, 4)

>>> (1, 2, 3) + (4, 3, 2, 1)

>>> ((1, 2, 3) + (4, 3, 2, 1))[4]
```

1.) For each of the following `tuples`, give the correct expression to get 7.

```
>>> x = (1, 3, (5, 7), 9)
>>> #YOUR EXPRESSION INVOLVING x HERE
7
>>> y = ((7,),)
>>> #YOUR EXPRESSION INVOLVING y HERE
7
>>> z = (1, (2, (3, (4, (5, (6, 7))))))
>>> #YOUR EXPRESSION INVOLVING z HERE
7
```

2.) Write the `reverse` procedure which operates on tuples. For a description of its behavior, see the docstring given below:

```
def reverse(seq):
    """Takes an input tuple, seq, and returns a tuple with the same items in
    reversed order. Does not reverse any items in the tuple and does not modify the
    original tuple.

    Arguments:
    seq -- The tuple for which we return a tuple with the items reversed.
```

```
    >>> x = (1, 2, 3, 4, 5)
    >>> reverse(x)
    (5, 4, 3, 2, 1)
    >>> x
    (1, 2, 3, 4, 5)
    >>> y = (1, 2, (3, 4), 5)
    >>> reverse(y)
    (5, (3, 4), 2, 1)
    """
    "*** Your code here. ***"
```

Exercise 7: Rlists

Remember this from lecture?

```
empty_rlist = None

def make_rlist(first, rest = empty_rlist):
    return first, rest

def first(r):
    return r[0]

def rest(r):
    return r[1]
```

The above is a functional representation of a recursive list from lecture. The big idea here is recursive structure. Why would we want a data structure like this? (Quiz your TA!)

It'd be convenient to have a procedure `tuple_to_rlist`, which takes a tuple and converts it to the equivalent rlist. Finish the implementation below. (*Hint*: an easy solution uses `reverse` from earlier in the lab).

```
def tuple_to_rlist(tup):
    """Takes an input tuple, tup, and returns the equivalent representation of
    the sequence using an rlist.

    Arguments:
    tup -- A sequence represented as a tuple.

    >>> tuple_to_rlist((1, 2, 3, 4, 5, 6))
```

```
    (1, (2, (3, (4, (5, (6, None))))))
    """
    "*** Your code here. ***"
```

You are now a professional Data Abstracter, now go derp around with your new toys.... :D