TUPLES AND RECURSIVE LISTS 5

COMPUTER SCIENCE 61A

July 3, 2012

1 Sequences

From the Pig project, we discovered the utility of having structures that contain multiple values. Today, we are going to cover a few ways we can model and create these structures, which are called *sequences*. A sequence is an ordered collection of data values. A sequence can have an arbitrary (but finite) number of ordered elements. A sequence is not a particular abstract data type, but instead a collection of behaviors that different types share. That is, there are many kinds of sequences, but they all share certain properties. In particular:

1. *Length*: A sequence has a finite length
2. *Element selection*: A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

2 Tuples

Tuples are a kind of sequence, which we saw during lab. Since we spent a lot of time on them in lab, we will only briefly review them here. You can make a tuple by enclosing a comma separated set of elements inside parentheses.

```
>>> (1, 2, 3)
(1, 2, 3)
```

Since tuples are a type of sequence, that means we should be able to get their length:

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

We should also have a way to pull elements out. Note that tuples are zero-indexed, which means that the first element is at index zero, the second element is at index one and so on:

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

In fact we've seen tuples in t he first project when you return multiple values from a function:

```
>>> def foo(a, b):
...     return a, b
...
>>> tup = foo(1, 2)
>>> tup
(1, 2)
```

A useful tool to extract a subset of elements from a tuple is using the slicing operator, which uses a colon:

```
>>> x = (10, 20, 50, 7, 300, 30, 40)
>>> x[1:]
(20, 50, 70, 300, 30, 40)
>>> x[3:]
(7, 300, 30, 40)
>>> x[:6]
(10, 20, 50, 7, 300, 30)
>>> x[3:5]
(7, 300)
>>> x[3:3]
()
```

1. Write a function `sum` that uses a `while` loop to calculate the sum of the elements of a tuple. *Note*: `sum` is actually already built into Python!

```
def sum(tup):
    """ Sums up the tuple.

    >>> sum((1, 2, 3, 4, 5))
    """
```

2. Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) tuples and returns a single tuple with all of the elements of the two tuples, in ascending order. Use recursion.

Hint: If you can figure out which list has the smallest element out of both, then we know that the resulting merged tuple will have that smallest element, followed by the merge of the two tuples with the smallest item removed. Don't forget to handle the case where one tuple is empty!

```
def merge(s1, s2):
```

3. Consider the subset sum problem: you are given a tuple of integers and a number k. Is there a subset of the tuple that adds up to k? For example:

```
>>> subset_sum((2, 4, 7, 3), 5)          # 2 + 3 = 5
True
>>> subset_sum((1, 9, 5, 7, 3), 2)
False
```

Note: You can use the `in` operator to determine if an element belongs to a tuple:

```
>>> 3 in (1, 2, 3)
True
>>> 4 in (1, 2, 3)
False
```

```
def subset_sum(seq, k):
```

4. We will now write one of the faster sorting algorithms commonly used, known as *merge sort*. Merge sort works like this:

   1. If there is only one (or zero) item(s) in the sequence, it is already sorted!
   2. If there are more than one item, then we can split the sequence in half, sort each half recursively, then merge the results, using the `merge` procedure from earlier in the notes). The result will be a sorted sequence.

Using the algorithm described, write a function `mergesort(seq)` that takes an unsorted sequence and sorts it.

```
def mergesort(seq):
```

3 Sequence Iteration with For Loops

In many of our sequence questions so far, we have ended up with code that looks like:

```
i = 0
while i < len(sequence):
    elem = sequence[i]
    # do something with elem
    i += 1
```

This particular construct happens to be incredibly useful because it gives us a way to look at each element in a sequence. In fact, iterating through a sequence is so common that Python actually gives us a special piece of syntax to do it, called the *for-loop*:

```
for elem in sequence:
    # do something with elem
```

Look at how much shorter that is! More generally, `sequence` can be any expression that evaluates to a sequence, and `elem` is simply a variable name. In the first iteration through this loop, `elem` will be bound to the first element in sequence in the current environment. In the second iteration, `elem` will be rebound to the second element in the sequence. This process repeats until `elem` has been bound to each element in the sequence, at which point the for-loop terminates.

1. Implement `sum` one more time, this time using a for-loop.

```
def sum(sequence):
```

2. Now use a for-loop to write a function `filter` that takes a predicate of one argument and a sequence and returns a tuple. (A predicate is a function that returns `True` or `False`.)

   This tuple should contain the same elements as the original sequence, but without the elements that do not match the predicate, i.e. the predicate returns `False` when you call it on that element.

   ```
   >>> filter(lambda x: x % 2 == 0, (1, 4, 2, 3, 6))
   (4, 2, 6)
   ```

   ```
   def filter(pred, sequence):
   ```

4 Nested Pairs and "Box-and-Pointer" notation

In computer science, a *pair* is a data structure that can contain two elements. There are many ways to define constructors and selectors to create a pair structure. Here is an intuitive one, where we construct a 2-element tuple, along with its selectors:

```
def make_pair(first, second):
    return (first, second)

def get_first(pair):
    return pair[0]

def get_second(pair):
    return pair[1]
```

Alternatively, we could define `make_pair` using lambda:

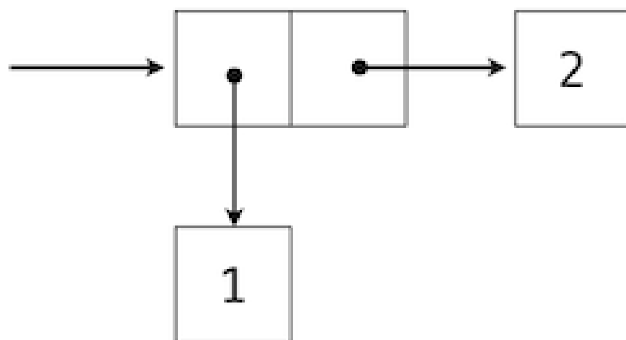
```
def make_pair(first, second):
    def dispatch(which):
        if which == 'first':
            return first
        elif which == 'second':
            return second
        else:
            return 'Huh?'
             # We will cover raising errors later in the course.
    return dispatch

def get_first(pair):
    return pair('first')

def get_second(pair):
    return pair('second')
```

The above two definitions are both perfectly valid ways of creating a pair data structure.

When you want to draw a pair (or other sequencing structures) on paper, computer scientists often use "box-and-pointer" diagrams. For example, the pair `(1, 2)` would be represented as the following box and pointer diagram:



Box-and-pointer diagrams are useful, since they show the structure and elements of a pair or sequence very clearly. The steps to construct such a diagram are as follows:

1. Represent a pair as two horizontally adjacent boxes.
2. Draw arrows from inside the boxes to the contents of the pair, which can also be pairs (or any other value).
3. Draw a *starting arrow* to the first entry in your diagram. Note that the starting arrow in the above diagram points to the *entirety of the two boxes* and not just the first box.

4. Don't forget your starting arrow! Your diagram will spontaneously burst into flames without it.

The arrows are also called *pointers*, indicating that the content of a box 'points' to some value.

1. Draw a box-and-pointer diagram for the following code:

```
>>> x = make_pair(make_pair(42, 28), make_pair(2, 3))
```

2. Given the definition of `x` above, what will `get_first(get_second(x))` return?

5 Recursive Lists

In lab, we created our own type of sequence, called an *immutable recursive list* (or *IRList*). Recursive lists are chains of pairs. Each pair will hold a value in our list, referred to as the *first* element of the pair, and a pointer to the *rest* of the recursive list (another pair). In lab, we defined the following constructors and selectors:

```
empty_irlist = ()           # The empty immutable recursive list.
def make_irlist(first, rest=empty_irlist):
    return (first, rest)
def irlist_first(irlist):
    return irlist[0]
def irlist_rest(irlist):
    return irlist[1]
```

Remember to use the name `empty_irlist` (instead of `None`) when working with recursive lists to respect the abstraction.

To formalize our definition, an IRList is something that fits either of the following two descriptions:

1. A pair whose first element can be anything, and whose second element is itself an IRList.

2. The empty IRList.

Since we want recursive lists to be a *sequence*, we should be able to find the *length* of a recursive list and be able to *select any element* from the recursive list. The relevant functions are in the lecture notes, but let us try re-implementing them in the exercises below:

1. Draw a box-and-pointer diagram for this IRList:

```
make_irlist(1,
    make_irlist(2,
        make_irlist(make_irlist(3, empty_irlist),
        make_irlist(4,
            make_irlist(5, empty_irlist)))))
```

2. Using calls to `make_irlist`, create the IRList that is printed as:

```
    <3, 2, 1, 'Blastoff'>
```

3. Write the function `sum_irlist` that takes an IRList of numbers and returns the sum of the numbers. Assume that `irlist_select` is not defined.

```
def sum_irlist(irlist):
```

6 Extra Practice with Immutable Dictionaries

It is often useful to pair values in a sequence with names. If we do that, then instead of having to remember the index of the value you are interested in, you only need to remember the name associated with that value (think: IDicts). In this section, we will build an abstract data type called an *association list*, or an *alist*.

We can implement this as a sequence of 2-tuples. The first element (index 0) of the tuple will be the 'key' associated with the value, and the second element (index 1) will be the 'value' itself.

1. Write a constructor `make_alist` that takes a sequence of keys and a sequence of corresponding values, and returns an association list (that is, a sequence of key-value tuples).

```
>>> make_alist(('a','b','c'), (1, 2, 3))
(('a', 1), ('b', 2), ('c', 3))
```

```
def make_alist(keys, values):
```

2. Write a selector `lookup` that takes an alist and a key and returns the value that corresponds to that key, or `None` if the key is not in the alist.

```
def lookup(alist, key_to_find):
```

3. Finally, write a procedure `change` that takes an alist, a key, and a value. It should return a new alist that matches the original, but with the following difference:

    1. If the key is already present in the original alist, replace its corresponding value with the argument value.
    2. Otherwise, add the key and value to the end of the alist.

```
def change(alist, key_to_change, new_value):
```

4. What are the differences between our immutable dictionary implementation (IDict) and the association list data type you just defined?

   The key take-away: Using the very few data structures we have learned so far, we are able to create abstract data structures that can organize information in very meaningful ways! Dictionaries, which an IDict is an implementation of, are extremely important in computer science, and we just implemented a simple version of one.