CS61A Notes – Week 5: Sequences

From the pig project, we have discovered the utility of having structures that contain multiple values. Today, we are going to cover some ways we can model and create these structures, which are called sequences.

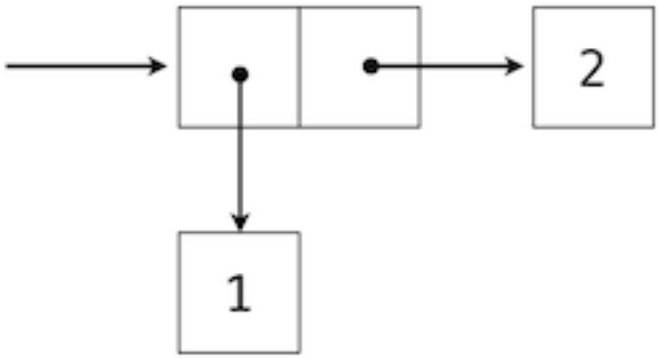
A sequence is an ordered collection of data values. Unlike a pair, which has exactly two elements, a sequence can have an arbitrary (but finite) number of ordered elements.

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.

Nested pairs and "box and pointer" notation

We have seen pairs implemented using both tuples and functions. 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:

a. Represent a pair as two horizontally adjacent boxes.
b. Draw arrows from inside the boxes to the contents of the pair, which can also be pairs (or any other value).
c. 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.
d. 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.

EXERCISES

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 `getitem_pair(getitem_pair(x, 1), 0)` return?

Recursive Lists

Using nested pairs, we are going to create our own type of sequence, called a **recursive list** (or **rlist**). 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). Thus, we have the following constructors and selectors:

```
def make_rlist(first, rest):
    return (first, rest)

def first(s):
    return s[0]

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

However, we still need a way to represent the end of a recursive list. For this example, we will use the Python constant, `None`. We can make the following definition as well:

```
empty_rlist = None
```

Remember to use the name `empty_rlist` when working with recursive lists in order to respect the abstraction, or John will set your paper on fire.

To formalize our definition, a recursive list is something that fits either of the following two descriptions:

1. A pair whose `first` item can be anything, and whose `rest` item is itself a recursive list.
2. The Python value `None`, when used in the context of recursive lists.

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. Functions to find such things are in the lecture notes, but let's try reimplementing them in the exercises below:

EXERCISES

1. Draw the box and pointer diagram for this rlist:

```
make_rlist(1,
  make_rlist(2,
    make_rlist(make_rlist(3, empty_rlist),
      make_rlist(4,
        make_rlist(5, empty_rlist)))))
```

2. Using calls to `make_rlist`, create the rlist that is printed as :

```
(3, (2, (1, ('Blastoff', None))))
```

3. Write the function `len_rlist` that takes an rlist and returns its length. (This is in the lecture notes as well.)

4. Write the function `last` that takes an rlist and returns the last item in that rlist (assume the function `getitem_rlist` is not yet defined).

5. Write the function `getitem_rlist` that takes an rlist and an index and returns the element at that index:

```
>>> x = make_rlist(2, make_rlist(3, empty_rlist))
>>> getitem_rlist(x, 1)
3
```

Tuples

Tuples are another kind of sequence. You can make a tuple by enclosing a comma separated set of elements inside parentheses. (The parentheses are optional at times, but it's generally safe to include them anyway).

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

Since tuples are a type of sequence, that means we should have a way to get the length of the sequence:

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

And we should have a way to pull elements out:

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

A note on constructing tuples: constructing a tuple with 2 or more elements is easy, and creating an empty tuple is easy too (simply don't include anything between your parentheses). However, you may run into problems trying to create a tuple of one element:

```
>>> (3)
3
```

What happened? Well, Python sees these parentheses as grouping parentheses, like the ones you'd use in a math expression, rather than parentheses that denote a tuple. We can tell Python to create a one element tuple by adding an extra comma:

```
>>> (3,)
(3,)

>>> (3,)[0]
3
```

Also note that we've actually seen tuples before, you just didn't know they were there! Take the following function:

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

That's right! Functions that return multiple values actually return tuples.

EXERCISES

1. Write a function, `sum`, that takes a tuple and returns the sum of its elements. (Note: `sum` is actually already built into Python!)

```
>>> sum((1, 2, 3, 4, 5))
15
```

Sequence Iteration with For Loops

In a lot of our sequence questions so far, we've ended up with code that looks like this:

```
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 by any expression that evaluates to a sequence, and `elem` is simply a variable name. On the first iteration through this loop, `elem` will be bound to the first element in sequence in the current environment. On the second iteration, `elem` will be bound to the first element in sequence in the current environment. On the second iteration, `elem` gets rebound to the second element. This process repeats until `elem` has been bound to each element in the sequence, at which point the for loop terminates.

EXERCISES

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

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)
```

3. It's 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. (This has proven to be so useful that Python actually has something like this built-in, called dictionaries, which you will learn about later.)

   We can implement this type of sequence, which we will call an association list, 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. In this question, you will define constructors and selectors for an association list ADT:

   a. 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'), (1, 2))
(('a', 1), ('b', 2))
```

   b. 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.

   c. 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:
      i. If the key is already present in the original alist, replace its corresponding value with the argument value
      ii. Otherwise, add the key and value to the end of the alist.