NEWTON'S METHOD AND SEQUENCES 3

COMPUTER SCIENCE 61A

September 17, 2012

1 Sequences

In Friday's lecture, you were introduced to *sequences*. Today, we are going to cover a few ways we can model and create sequences. A sequence is an ordered collection of items. 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 and properties that different implementations 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 in Lab 4. 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)
```

You can also create an empty tuple

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

and a tuple with only one element

```
>>> (1, )
(1, )
```

Since tuples are a type of sequence, we can get their length with the built-in `len` function:

```
>>> 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. In general, to get the ith item, use index $i - 1$.

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

We can also use the `in` operator to determine if a particular element is in a tuple.

```
>>> 1 in (1, 2, 3)
True
>>> b = (3, 4, 5)
>>> 6 in b
False
```

We can also return tuples from functions just like any other value:

```
>>> 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. The number to the left of the colon is the start index and the number to the right of the colon is the end index. If you don't provide a start index, it's 0 by default. Similarly, if you don't provide an end index, it's the index of the last element by default.

```
>>> 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]
()
```

Finally, we can add two tuples together. Tuple addition is very simple - you take all the elements from one tuple, all the elements from the other tuple, and you combine them to form a new tuple.

```
>>> a = (1, 2, 3)
>>> b = (4, 5, 6)
>>> c = a + b
>>> c
(1, 2, 3, 4, 5, 6)
```

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))
    15
    """
```

2. Write a function `min_element` that returns the minimum element in a tuple.

```
def min_element(tup):
    """ Returns the minimum element in tup.

    >>> a = (1, 2, 3, 2, 1)
    >>> min_element(a)
    1
    """
```

3. Fill in the definition of map_tuple. map_tuple takes in a function and a tuple as arguments and applies the function to each element of the tuple.

```
def map_tuple(func, tup):
```

```
    """Applies func to each element of tup and returns a new tuple.

    >>> a = (1, 2, 3, 4)
    >>> func = lambda x: x * x
    >>> map_tuple(func, a)
    (1, 4, 9, 16)
    """
```

4. Fill in the definition of cartesian_product. cartesian_product takes in two tuples and returns a tuple that is the Cartesian product of those tuples. To find the Cartesian product of tuple X and tuple Y, you take the first element in X and pair it up with all the elements in Y. Then, you take the second element in X and pair it up with all the elements in Y, and so on.

```
def cartesian_product(tup_1, tup_2):
    """Returns a tuple that is the cartesian product of tup_1 and tup_2.

    >>> X = (1, 2)
    >>> Y = (4, 5)
    >>> cartesian_product(X, Y)
    ((1, 4), (4, 1) (1, 5), (5, 1), (2, 4), (4, 2) (2, 5), (5, 2))
    """
```

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):
```

3. Rewrite the cartesian_product function with for loops.

4 Newton's Method

Newton's method is an algorithm that is widely used to compute the zeros of functions. It can be used to approximate a root of any continuous, differentiable function.

Intuitively, Newton's method works based on two observations:

- At a point $P = (x, f(x))$, a root of the function f is in the same direction relative to P as the root of the linear function L that not only passes through P, but also has the same slope as f at that point.
- Over any *very small* region, we can approximate f as a linear function. This is one of the fundamental principles of calculus.

Starting at an initial guess $(x_0, f(x_o))$, we estimate the function f as a linear function L, solve for the zero $(x', 0)$ of L, and then use the point $(x', f(x'))$ as the new guess for the root of f. We repeat this process until we have determined that $(x', f('x))$ is a zero of f.

Mathematically, we can derive the update equation by using two different ways to write the slope of L:

Let x be our current guess for the root, and x^* be the point we want to update our guess to. Let L be the linear function tangent to f at $(x, f(x))$.

Remember that x^* is the root of L. So, we know two L passes through, namely $(x, f(x))$ and $(x^*, 0)$.

We can write the slope of L as

$$L'(x) = \frac{0 - f(x)}{x^* - x} = \frac{-f(x)}{x^* - x} \tag{1}$$

We also know that L is tangent to f as x, so:

$$L'(x) = f'(x) \tag{2}$$

We can equate these to get our update equation:

$$\frac{-f(x)}{x^* - x} = f'(x) \Rightarrow x^* = x - \frac{f(x)}{f'(x)} \tag{3}$$

We know $f(x)$, and from calculus, for some very small ε:

$$f'(x) = \frac{f(x+\varepsilon)}{(x+\varepsilon) - x} = \frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{4}$$

From the above, we get this algorithm:

```
def approx_deriv(fn, x, dx=0.00001):
    return (fn(x+dx)-fn(x))/dx

def newtons_method(fn, guess=1, max_iterations=100):
    ALLOWED_ERROR_MARGIN = 0.0000001
    i = 1
    while abs(fn(guess)) > ALLOWED_ERROR_MARGIN and i <= max_iterations:
        guess = guess - fn(guess) / approx_deriv(fn, guess)
        i += 1
    return guess
```

We can generalize this idea into a framework known as *iterative improvement*. Basically, you start out by guessing a value, and then continuously update the guess until it is a reasonable approximation of the value we are looking for. Here is an implementation for `iter_improve`. The `update` function takes the current guess, and returns an updated guess. The `isdone` function also takes the current guess, and returns `True` if and only if the current guess is "good enough", according to some set criterion.

```
def iter_improve(update, isdone, guess=1, max_iterations=100):
    i = 1
    while not isdone(guess) and i <= max_iterations:
        guess = update(guess)
        i += 1
    return guess

def newtons_method2(fn, guess=1, max_iterations=100):
    def newtons_update(guess):
        return guess - fn(guess) / derivative(fn, guess)
    def newtons_isdone(guess):
        ALLOWED_ERROR_MARGIN= 0.0000001
        return abs(fn(guess)) <= ALLOWED_ERROR_MARGIN
    return iter_improve(newtons_update,
                        newtons_isdone,
                        max_iterations)
```

1. Write a function `cube_root` that computes the cube root of the input number `x`. (*Hint*: Use `newtons_method` with a function that is zero at the cube root of the input.)

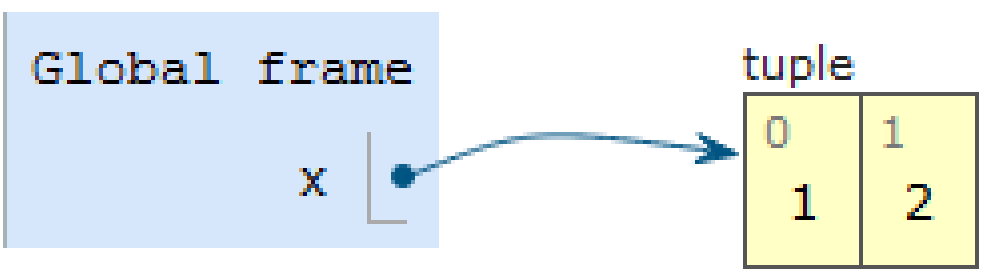
```
def cube_root(x):
```

2. Newtons method converges very slowly (or not at all) if the algorithm happens to land on a point where the derivative is very small. Modify the implementation that uses `iter_improve` to return `None` if the derivative is under some threshold, say 0.001.

```
def newtons_method2(fn, guess=1, max_iterations=100):
    def newtons_update(guess, min_size=0.001):




    def newtons_done(guess):




    return iter_improve(newtons_update, newtons_done, guess,
        max_iterations)
```

5 "Box-and-Pointer" notation

When you want to draw a sequence, such as a tuple, on paper, computer scientists often use "box-and-pointer" diagrams. For example, the tuple `(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 sequence very clearly. The steps to construct such a diagram are as follows:

1. Represent a sequence as horizontally adjacent boxes. The number of boxes that you need to draw is just the number of elements in the sequence.
2. If an element of the sequence is a primitive value, just write the value directly in the box.
3. Else, draw an arrow from inside the box to the contents of the sequence, which can also be other sequences.
4. Write the type of the sequence directly above the boxes that represent the sequence.
5. 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.
6. 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 an environment diagram for the following lines of code:

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

>>> y = ((1, 2), (4, 5), (3, 2))

>>> z = (1, (2, 3), (4, (5, (6, 7))))

>>> a = (1, (2, (3, (4, (5, 6)))))
```

6 Data Abstraction

In Lab 4, you were introduced to the concept of data abstraction. Data abstraction is simply a methodology by which functions are used to enforce an abstraction barrier between the *use* and *representation* of a data object. These functions are called *constructors* and *selectors*. Constructors are functions that construct the data object and selectors are functions that query the data object for the values that it contains. To illustrate how these functions are used, let's suppose that you want to represent a person object in your program. To make this example simple, we'll just represent a person in terms of his/her first name, last name, and age. The constructor is defined as follows:

```
def make_person(first, last, age):
    return (first, last, age)
```

Note that all make_person does is return a tuple that contains the arguments passed to it. So, to create a person object, you could write the following:

```
>>> Bob = make_person("Bob", "Smith", 45)
>>> Bob
('Bob', 'Smith', 45)
>>> Alice = make_person("Alice", "Douglas", 36)
>>> Alice
('Alice', 'Douglas', 36)
```

Since a person has three properties - his/her first name, last name, and age - we need to define three selector functions.

```
def get_first(person):
    return person[0]

def get_last(person):
    return person[1]

def get_age(person):
    return person[2]
```

So, to find out all about Alice, we would write something like this:

```
>>> first_name = get_first(Alice)
>>> first
'Alice'
>>> last_name = get_last(Alice)
'Douglas'
>>> age = get_age(Alice)
```

```
36
```

So, where does data abstraction come in? Well, let's suppose that instead of using the constructors and selectors that we defined, your program makes the assumption that person objects are represented as tuples and uses the bracket syntax to get the first name, last name, and age of a person. Parts of your program might look something like this:

```
first_name_a = Alice[0]
last_name_a = Alice[1]
age_a = Alice[2]
...
...
...
...
...
first_name_b = Bob[0]
last_name_b = Bob[1]
age_b = Bob[2]
```

Now, let's suppose that you're getting pretty tired of tuples and want to represent person objects as dispatch functions. If you wanted to make this change, you would also have to change every single line in your program that used the bracket syntax. Why? Well, the bracket syntax can only be used on sequences, not functions. If you were writing an extremely large program (tens of thousands of lines of code), then you would have to make A LOT of changes to your program. If you had instead used the constructor and selector functions

```
first_name_a = get_name(Alice)
last_name_a = get_last(Alice)
age_a = get_age(Alice)
...
...
...
...
first_name_b = get_name(Bob)
last_name_b = get_last(Bob)
age_b = get_age(Bob)
```

then you would only have to make changes to your constructor and selector functions in order for your program to work properly. Data abstraction is all about separating the use of a data object from its representation. Your program only needs to know how to interact with a data object, i.e., how to construct the object and get its attributes. The representation of that data object should be hidden from your view.

1. The following constructor and selector functions are defined for you

```
def make_point(x, y):
    return (x, y)

def get_x(point):
    return point[0]

def get_y(point):
    return point[1]

def make_seg(pt1, pt2):
    return (pt1, pt2)

def start_pt(seg):
    return seg[0]

def end_pt(seg):
    return seg[1]
```

Using these constructors and selectors, fill in the definition of the following functions(remember, use data abstraction!):

```
def reflect_across_y(seg):
    """Returns a line segment that is the y-axis reflection of seg.

    >>> pt1 = make_point(0, 0)
    >>> pt2 = make_point(4, 5)
    >>> line_seg = make_seg(pt1, pt2)
    >>> reflect_across_y(line_seg)
    ((0, 0), (-4, 5))
```

Below, we've written a function that calculates the midpoint of a line segment. Cross out every line of code that violates an abstraction barrier and fix them.

```
def midpoint(seg):
    first_pt = seg[0]
    sec_pt = seg[1]

    new_x = (first_pt[0] + sec_pt[0]) // 2
    new_y = (first_pt[1] + sec_pt[1]) // 2
```

```
    return (new_x, new_y)
```