Python Tutorial: Buffer Math

Introduction

Python is a user-friendly and straightforward programming language, and one used extensively in scientific circles. In this brief introduction, I'll be showing how to use Python to solve Henderson-Hasselbalch buffer recipe problems automagically.

To use this tutorial effectively, you'll need Python 2.7 installed on your system. There are many ways to acheive this, but one of the easiest for academic users is the Enthought Canopy distribution of Python, which is free and installs all the tools you need for effective scientific programming. You can easily download Enthought Canopy (https://store.enthought.com/) for personal use, then open up the *Canopy Code Editor* and follow along. Go on, go install it and then come back. I'll wait. Or you can read on if you really want. I can't stop you.

Python: Super Basics!

In Python, you will feed instructions to the computer (the lines shown below that begin with "**In [#]**") and it will perform computations. Sometimes, this will happen silently, without visible output. Sometimes, it will generate output (the lines below that begin with "**Out [#]**")

Core to programming is the idea of a variable, which is basically a label for a bucket of data.

```
In [1]: a = 3
```

Here, I set a variable to the value 3. I can now refer to it by that name whenever I want:

```
In [2]: a
Out[2]: 3
```

When I asked the computer for "a", it returned "3" to me, because "a" is the label that refers to whatever I stored there.

So far, this is all standard algebra. Here's where it gets more complex: Variables can hold any data, not just a number.

```
In [3]: b = "This is a string of text, usually called just a string."
```

Here, I told the computer that "b" is the sentence above. As expected, if I ask for b, it will give me the text back:

```
In [4]: b
```
```
Out[4]: 'This is a string of text, usually called just a string.'
```

You can do math to variables, as you might expect, but when they don't hold a number, the results can be surprising:

```
In [5]: a * 2
```
```
Out[5]: 6
```

```
In [6]: b * 2
```
```
Out[6]: 'This is a string of text, usually called just a string.This is a string of tex
        t, usually called just a string.'
```

The last central concept is that we can define functions, which is a set of instructions that we can reference over and over. (In the same way that a variable holds data of any type, a function holds any set of instructions. So, instead of multiplying by two manually, I could define a function:

```
In [7]: def times_two(var):
            return var * 2
```

Now, I can use that function anytime I want, instead of typing my instructions again (in this case, my instructions are very short... but they could also be pages and pages of text that I don't want to repeat).

```
In [8]: times_two(a)
```
```
Out[8]: 6
```

```
In [9]: times_two(b)
```
```
Out[9]: 'This is a string of text, usually called just a string.This is a string of tex
        t, usually called just a string.'
```

Most of programming is figuring out what variables and functions to use, and in what sequence, to achieve your goals. There's a ton more depth, and I highly recommend you check out the incredibly useful free guide A Byte of Python (http://www.swaroopch.com/notes/python/).

For now, however, we're going to learn by doing! Onward!

Buffer Recipe Calculations

In a classic Buffer Recipe question, you have available stock solutions of a buffering agent (at a given pH), access to strong acid and/or strong base to adjust its pH, and some specified final conditions that you must reach.

A quick example: > You have a stock of 1.5 M Acetic Acid, pH 4.5 (pKa 4.76), as well as access to 5 M HCl, 5 M NaOH, and water. > > How would you prepare 2 L of 0.2 M Acetic Acid, pH 5.0?

There are many ways to solve such problems, but here we want to think of programmatic solutions. That means trying to figure out what we are **actually** needing to solve.

A useful first step might be putting *everything* we know into thoughtfully labeled variables, so at least we can manipulate them later. Putting your data into variables also has the benefit that if you change it (say, on a new problem), you don't need to find and replace everywhere you entered a number-- just change the variable and go!

In [10]:
```
buffer_conc_initial = 1.5
```

In [11]:
```
buffer_conc_final = 0.2
```

In [12]:
```
total_volume = 2.0
```

In [13]:
```
HCl_stock_conc = 5.0
```

In [14]:
```
NaOH_stock_conc = 5.0
```

In [15]:
```
pKa = 4.76
```

In [16]:
```
initial_pH = 4.5
```

In [17]:
```
final_pH = 5.0
```

(Technically, we could be much more sophisticated, and put many of these into an array of values, or a database-like dictionary. For now, this will do)

Buffer Volume

One of the first things we can solve for, independent of pH adjustment, is the volume of stock buffer necessary. This invokes the tried and true chemist method: > $C_1V_1 = C_2V_2$

If we re-arrange for our unknown, we have: > $V_1 = (C_2V_2)/C_1$

Which we can assign as a new variable:

```
In [18]: buffer_volume = (buffer_conc_final * total_volume) / buffer_conc_initial
```

One of the great things about programming is that by the act of typing in our formula, we have our answer:

```
In [19]: buffer_volume

Out[19]: 0.26666666666666666
```

Next, we need to calculate the moles of buffer present:

```
In [20]: moles_buffer = buffer_volume * buffer_conc_initial
```

```
In [21]: moles_buffer

Out[21]: 0.4
```

Initial pH Determination

With that in hand, we can start solving for our pH adjustment. Our moles of buffer are either protonated or deprotonated, giving: > moles of buffer = [HA] + [A-]

We'll use that relationship later, but it's good to keep in mind for now.

To solve for our initial [HA] and [A-], we can set up our Henderson-Hasselbalch equation: > pH = pKa + log ([A-] / [HA])

But again, we need to re-arrange our equation to solve for the unknowns: > log ([A-] / [HA]) = pH - pKa

Or going even further: > [A-] / [HA] = $10^{pH - pKa}$

```
In [22]: initial_ratio = 10**(initial_pH-pKa)
```

(it's worth noting that in Python, two asterisks are used instead of the up-carrot to indicate expontential math, and similarly, instead of log we use "log10" to denote that it is not a natural log.

```
In [23]: initial_ratio

Out[23]: 0.5495408738576248
```

Now, I can return to my relationship about moles of buffer: > moles of buffer = [HA] + [A-]

Between that relationship and the ratio we just solved, we have a system of equations: two equations with two unknowns (our [A-] and our [HA]). We could solve it with an inverse matrix dot product, but for such a simple problem, we can also just use substiution:

In our *initial ratio*, Another way to arrange it is that the result (number) we found is the factor difference between [A-] and [HA]. That is: > [A-] = ratio * [HA]

Combining the two equations, we can solve: > moles of buffer = [HA] + (ratio * [HA])

And rearranging for our unknown: > [HA] = moles of buffer / (1 + ratio)

```
In [24]: initial_HA = moles_buffer / (1+ initial_ratio)
```

```
In [25]: initial_HA
```

```
Out[25]: 0.25814098017575293
```

Similar math tells us that our [A-] is the remainder:

```
In [26]: initial_A = moles_buffer - initial_HA
```

```
In [27]: initial_A
```

```
Out[27]: 0.1418590198242471
```

Final pH Determination

Now, we can apply a very similar process to our final pH values determination.

But Wait! Whenever I feel myself about to apply the same process over and over, that sounds like a function! Let's define one that, given a pH, pKa, and total moles of buffer, gives us our final HA and A values.

```
In [28]: def moles_conjugate_solver(given_pH,given_pKa,given_moles):
             given_ratio = 10**(given_pH-given_pKa)
             output_HA = given_moles / (1+ given_ratio)
             output_A = given_moles - output_HA
             return output_HA, output_A
```

That function takes all the math we did last time and plunks it into one package. Now, I can invoke that function, telling it to store it's output as my new, final value, variables:

```
In [29]: final_HA, final_A = moles_conjugate_solver(final_pH, pKa, moles_buffer)
In [30]: final_HA
Out[30]: 0.14610266597907323
```

```
In [31]: final_A
Out[31]: 0.2538973340209268
```

Wow, that was much quicker! And in fact, I could use that same function for both initial and final values (more on that later).

Volume of Titrant

Now that I know both my initial and final values, I can solve for the moles of titrant needed (*note: this is a simplication, which works very well for biochemical buffer changes, but can fail spectacularly for more complex buffer changes or larger pH adjustments*)

```
In [32]: moles_difference = abs(final_HA - initial_HA)
```

```
In [33]: moles_difference
Out[33]: 0.1120383141966797
```

Notice that I used "abs" which stands for "absolute value of". It looks a lot like a function, and indeed it is-- a built-in function. Python has thousands, and learning them can help you program faster and more efficiently.

However! In this case, the sign matters, because if [HA] is decreasing (final [HA] < initial [HA]) that indicates that I need to add NaOH, whereas if [HA] is increasing, I would achieve that by adding HCl. So let's rewrite that function:

```
In [34]: moles_difference = final_HA - initial_HA
```

```
In [35]: moles_difference
Out[35]: -0.1120383141966797
```

Now, I need to check for whether I need to add NaOH or HCl. I can use an "if/then" statement to do so.

I'll go further, too, and wrap the whole thing into another function:

```
In [36]: def titrant_solve(given_final_HA, given_initial_HA, given_HCl_conc, given_NaOH_
         conc):
             difference = given_final_HA - given_initial_HA
             if difference < 0:
                 titrant = "NaOH"
                 difference = abs(difference)
                 volume = difference / given_NaOH_conc
             else:
                 titrant = "HCl"
                 volume = difference / given_HCl_conc
             return volume, titrant
```

Now, I can feed in my values:

```
In [37]: volume_of_titrant , which_titrant = titrant_solve(final_HA, initial_HA, HCl_sto
         ck_conc, NaOH_stock_conc)
```

```
In [38]: volume_of_titrant

Out[38]: 0.02240766283933594
```

```
In [39]: which_titrant

Out[39]: 'NaOH'
```

Final Answer:

Putting these pieces together, we find:

```
In [40]: buffer_volume

Out[40]: 0.26666666666666666
```

```
In [41]: volume_of_titrant

Out[41]: 0.02240766283933594
```

```
In [42]: water = total_volume - (buffer_volume + volume_of_titrant)
```

```
In [43]: water

Out[43]: 1.7109256704939975
```

Can we Automate the whole process?

Now that we've seen it step-by-step, let's put together one **BIG** function to do everything for us!

```
In [44]: def Buffer_Solver(buffer_conc_initial, buffer_conc_final, buffer_pKa, total_vol
         ume, HCl_stock_conc, NaOH_stock_conc, initial_pH, final_pH):
             # First find moles of buffer and volume of buffer:
             buffer_volume = (buffer_conc_final * total_volume) / buffer_conc_initial
             moles_of_buffer = buffer_volume * buffer_conc_initial
             # Then, find initial conditions:
             initial_ratio = 10**(initial_pH-buffer_pKa)
             initial_HA = moles_of_buffer / (1+ initial_ratio)
             initial_A = moles_of_buffer - initial_HA
             # Then, final conditions:
             final_ratio = 10**(final_pH-buffer_pKa)
             final_HA = moles_of_buffer / (1+ final_ratio)
             final_A = moles_of_buffer - final_HA
             # Then, solve for titrant:
             difference = final_HA - initial_HA
             if difference < 0:
                 titrant = "NaOH"
                 difference = abs(difference)
                 volume_titrant = difference / NaOH_stock_conc
             else:
                 titrant = "HCl"
                 volume_titrant = difference / HCl_stock_conc
             volume_water = total_volume = (volume_titrant + buffer_volume)
             return "Buffer recipe: add %s liters stock buffer, %s liters of stock %s, a
         nd %s liters of water" % (
             buffer_volume, volume_titrant, titrant, volume_water)
```

```
In [45]: Buffer_Solver(1.0,0.2,4.76,2,5.0,5.0,4.5,5.0)
```

```
Out[45]: 'Buffer recipe: add 0.4 liters stock buffer, 0.0224076628393 liters of stock Na
         OH, and 0.422407662839 liters of water'
```

Wow! Now, I can solve buffer problems all day long! For instance, what is I had this situation: > Given 2 M Tris, pH 7.55 (pKa 8.0), 4 M NaOH, 4 M HCl, and water > > What is the recipe for 1.2 L of 0.05 M Tris, pH 7.3?

```
In [46]: Buffer_Solver(2.0,0.05,8.0,1.2,4.0,4.0,7.55,7.3)
```

```
Out[46]: 'Buffer recipe: add 0.03 liters stock buffer, 0.00143330154965 liters of stock
         HCl, and 0.0314333015496 liters of water'
```

What Next?

Now that you can write a function to solve buffer problems for you, you could make it prettier and more useful. For instance, to use **Buffer_Solver**, you need to know the order of terms to enter.

One easy next step would be to set the function to ask you for the values, record them, use them, and then return an answer. After that, you could write a graphical interface, publish the code as a single file that will work on any computer, or so much more!

Another useful next step would be to rewrite our function to use a more general solution to the Henderson-Hasselbalch equation, that doesn't fail for large changes of pH or large volumes of titrant. It could also incorporate poly-protic acids and other behavior.

```
In [ ]:
```