Chapter 0: Simplifying Repetitive Calculations

sample, Website
CAC
han

- ☐ **New concepts**
  - *and their prerequisites*

- ☐ **Giving your computer instructions using Python commands**
- ☐ **Having Python do basic math by evaluating mathematical expressions**
- ☐ **Accessing capabilities beyond basic math using functions**
- ☐ **Clarifying how your code works with comments**
- ☐ **Temporarily storing data using variables**
- ☐ **Accessing a history of code you have executed in the console**
- ☐ **Expressing data as readable text using strings**
- ☐ **Avoiding repetitive typing by making your own functions**

0.0 Problem

You recently developed a new aqueous catalyst that relies on a proton transfer step. Obviously, the efficacy of the catalyst will depend on the pH of the solution relative to the pK_a of the catalyst. However, you are also concerned that the pK_a of the catalyst will, in turn, depend on the ionic strength of the solution. This suggests you need to run multiple titration experiments, each conducted under a different (but constant) ionic strength.

While you are excited to run these experiments and lay the groundwork for your first paper, it will take some work. Fortunately, your lab has a 24 plate reader that can measure the pH of the solutions. This means that you could run multiple experiments at once, where you could use the 4 rows for different constant ionic strengths and the 6 wells in each row for different pHs. Each well's condition could be made by mixing together different amounts of stock solutions. Deciding you will titrate with HCl, you produce the following sketch of the experiment:

sample. website c4c

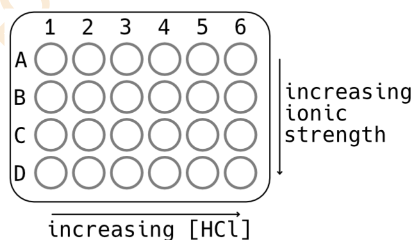

Figure 0.1: Schematic of a 24 well plate, showing the desired changes in chemical composition.

Though the plate reader will save you measurement time, it also means that you need to calculate all 24 conditions and keep track of them. You mention to your labmate that you are excited to do the experiment but are not looking forward to doing these tedious calculations and double checking them. This is particularly true since your advisor has a habit of changing their mind regarding the exact limits of reaction—and each time the limits change you will need to recalculate the conditions.

You know that you could set up a spreadsheet to do this in Microsoft Excel, but the fact that you wanted to maintain a constant volume across the experiments means that the formulas in Excel become cumbersome and it will be hard to use the spreadsheet while you're actually making the solutions. You'd rather just have a checklist of what goes in each well so that you can be sure you won't make a mistake and have to repeat the experiment.

You are talking about this in lab on morning and your labmate mentions that this really hasn't been a problem for them since they learned to code. Intrigued, you ask what they mean. They say that coding is a perfect tool for these repetitive tasks—it has reduced their mistakes *and* improved their life in lab. They say that you could write a simple program that would calculate everything you need and could adapt to whatever limits your advisor decides they want that day. They say that it should take only two lunch hours to get you to the point you can solve this problem.

They point out that the very *first* time you implement this solution might take longer than calculating by hand once, but that you will have generated a general tool that can be used again and again. So, the more frequently your advisor changes their mind, the more time you will save in the long run.

You realize that this could reclaim *years* of your life. You agree to meet over lunch, and your labmate tells you to bring your laptop with you.

0.1 Solution

When the first lunchtime comes, you walk into the break room, and your labmate is waiting. The solution to your problem, they explain, will be achieved by outlining the constraints of the experiment, expressing them in terms of calculations that you would normally do on a calculator, and then writing a bit of code that will do these calculations automatically for you. You will do so by storing information about your stock solutions and required final concentrations in the computer and then telling it how to calculate the volumes of each stock solution to add to the well. This first lunch hour, they want to get you to the point you can do this for a single well. Even this will show you how using the code can save you time over using a calculator or Microsoft Excel. Then, in the the next lunch hour, they will show you how to adapt this to the rest of the wells in a straightforward manner. This all sounds great to you, so you sit down and open your laptop.

sample, website c4c nan

0.1.0 Giving your computer instructions using Python commands

"The most basic thing to understand" Your labmate starts, "Is that programming is a set of instructions for the computer and, at its heart, a computer is not much more more than a calculator with memory. This means your code is going to be a set of instructions, but that these instructions are ultimately turned into numbers. So, you need a way to take language that *humans* understand (*i.e.,* words) and translate them into language that your *computer* understands (*i.e.,* numbers). When someone talks about programming languages, they are referring to how this translation is accomplished."

"Each coding language includes a program that performs this translation. The most basic task in coding is to write your instructions in a way that the translating program can understand." The structure and terminology required for these instructions to be understood by the translating program is called the **syntax**, and the appearance of this syntax is one of the most obvious differences between languages. That is, the exact same instructions written in two languages will look very different.

"I like the Python language," Your labmate says, "because its syntax is easy to learn and understand and there are a huge number of tools for chemistry that can be used with it. There are a lot of ways to install Python on your computer, but I recommend starting with the Anaconda Distribution, which simplifies management of Python and its extensions."[1]

"Once Anaconda is installed, the simplest thing you can do is to open it, open an "anaconda prompt" (Windows) or "terminal," (Macs) and type `idle`. This opens up Python's Integrated Development and Learning Environment (IDLE), which allows you to type Python commands and start calculating things."

Opening IDLE, you are presented with a window that looks something like what is shown in Figure 0.2 Your labmate explains that this box is sometimes called a **shell** or a **console**. You can enter commands and then press enter and the computer will read them and try to execute them. In this case, you are going to input Python commands, and the **Python interpreter** program is going to try to execute them.

[1] We strongly recommend following the procedure outlined on the companion website for this book, https://www.codingforchemistsbook.com/, to obtain the full functionality that will be covered in this book. There are many ways to get something that will work, but you will avoid annoyance and get some extra capability installing Python our way!

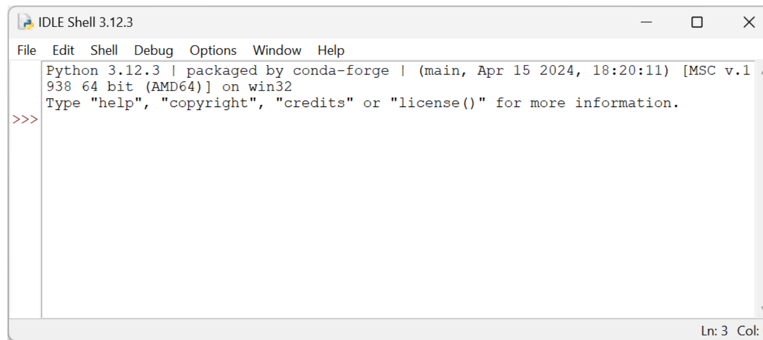

Figure 0.2: Screenshot of the IDLE interface for Python in the Windows operating system. This is one way to use Python interactively. You can enter commands with proper Python **syntax**, and they will be executed.

0.1.1 Having Python do basic math by evaluating mathematical expressions

"Remember how I said the computer is a calculator?" Your labmate begins. "Try adding 2 and 3. You can type it into the command prompt, denoted by »>."

You look at your labmate but they just raise their eyebrows and wait. So, you tentatively type:

```
>>> 2+3
```

and press enter. Immediately, a new line appears with:

```
5
```

"Great! This is the first line of code that you have ever written! What you typed into the command prompt is a mathematical **expression**, using the **operator** '+'. Python reads the expression on the input line (represented in this book by light gray, and preceded by >>>), carries out all the operations specified by operators, and prints out the result on an 'output line' (represented in this book by darker gray). Now what if you wanted to multiply them?" Here, you think a bit, but remember that other programs, like Excel, use '*' to indicate multiplication and so you try that.

```
>>> 2*3
6
```

"Ok, now, how about division? What if you divide 2 by 3?"

```
>>> 2/3
0.6666666666666666
```

"Ok," your labmate says "This is working just like a calculator. You know that 2/3 is not a rational number—that it will keep repeating forever. But a *calculator* has a limited display, so it just writes 0.666... until it runs out of display room. The computer is similar to this. It has a lot more display room than your typical calculator, but each number does require some storage in the computer, and the computer does not have infinite storage. So it has to eventually stop writing numbers. We will talk about how to control the representation of numbers later. Another thing you will run into is this..." they say while typing:

```
>>> .01/.1
0.09999999999999999
```

"We know that the answer should be 0.1, but actually, the computer doesn't know what 0.1 is. The problem is, again, that the computer has a limited amount of storage, and so it cannot store all of the real numbers—which would take an infinite amount of storage. Thus, the computer works with only a limited subset of real numbers and 0.1 isn't one of them. It is up to you to realize that 0.09999999999999999 is as close as Python will come to the correct answer. This concept is called **numerical precision** and in almost all cases, you will find that the precision of Python is close enough for your needs. You might have a strong desire to make the output look nicer, but if you can wait just a bit, we will get to that. For now, realize that as soon as you move away from situations where all the numbers involved are whole numbers, your answers will typically be *very* close to correct but not *exactly* correct.

"Now, there will be other things that are a bit different from a calculator or Excel. For instance, what if you want to calculate two to the third power? In Excel, the **operator** for exponent is ^, and many calculators use the same symbol. In Python this operator is is '**':"

```
>>> 2**3
8
```

You can see that the output is what we expect for raising 2 to the third power. Thus, this is the same operator as ^, Python just uses a different **syntax** than Microsoft Excel or your calculator.

0.1.2 Accessing capabilities beyond basic math using functions

"Now," your labmate says, "you can probably think of lots of different symbols for doing math, but sometimes programs don't have them all. Since we are considering similarities and differences with Microsoft Excel, consider taking the absolute value of a number. If you wanted to take the absolute value of a number in Excel, you use the absolute value **function** using the command `abs(-2)`. It turns out that, Python happens to use the exact same idea and notation: `abs()`."

```
>>> abs(-2)
2
```

"And here, we have introduced a new bit of **syntax**. We have taken a value we want to operate on, and placed it between two parenthesis, preceded by a name."

"In Python, the term for the the command with the parentheses is a **function**, while the number that we place between the parentheses an **argument**. Functions can evaluate to values in the same way as mathematical expressions can be evaluated to yield a single value. In this case, it evaluated to the value of '2', but not all functions evaluate to values. Functions can also be written to accept any number of arguments—including no arguments. The `abs()` function expects a single argument. But we can consider a different function—one that can accept either one or two arguments: `round()`."

"Invoking the code in a function is referred to as **call**ing the function. To start with, we include, only a single argument in the function call: the number to be rounded. Once called, the function evaluates to the argument value, rounded to a whole number."

```
>>> round(3.14159)
3
```

```
>>> round(3.5)
4
```

sample: Website c40 nan

```
>>> round(4.5)
4
```

"Take a close look at these last two examples,' Your labmate says, pointing at their computer's monitor, "you might be surprised that both numbers were rounded to 4. This is probably not how you were taught to round. Most people in the USA were taught to round up, when presented with a 5 at the decimal place where they are rounding. However, Python uses a different rule, known as 'bankers' rounding. Because always rounding a 5 upwards will tend to bias in reported statistics for measurements, a rule is created to eliminate this bias. If the digit before the 5 is odd, we round up. If the data before the 5 is even, we round down. Though this may look strange, it produces a result that, in the long term, has less bias."

"We can exercise greater control over the behavior of the round() function by using another argument. This second argument comes after the number we wish to round, and specifies the number of decimal places to keep after rounding."

```
>>> round(3.14159, ndigits = 2)
3.14
```

```
>>> round(1.61803, ndigits = 3)
1.618
```

This could be used to solve the issue we encountered above with **numerical precision** and the answer to 0.01 divided by 0.1. For instance, if we wanted we could use the following

```
>>> round(0.01/0.1, 1)
0.1
```

"And so now we are getting the answer *displayed* as we expect, but there are two things to realize here. First, the answer the computer calculated for 0.01/0.1 was not 0.1. We we saw that before (Section 0.1.1). We have asked it to round to give the answer we expect... so really we are just asking for the answer to a precision of 1 decimal place. This is a technical point, and it is, but it is worth keeping in mind. Second, you will notice that we did math *within* an argument passed to a function. That is, we specified a mathematical operation, and this was evaluated, and then the result of this evaluation was used as the argument. This is our first hint that, when coding, we are going to be able to combine many different operations in a single line of code!

"One more important thing about function arguments—you notice that we always need to supply a value for the number that we are rounding, but we don't always have to supply the digits to round to. We say that the former is an required argument, while the latter is an optional argument. Required arguments are arguments without which the function cannot run. When you call the function, all required arguments must have values. However, for optional arguments, the function has defined a default value that it will use, if you do not pass a value for it. In the case of round, ndigits uses the default value that essentially acts as if it were 0—meaning that the number will be rounded to its integer value.

"Another thing that we see above is that we don't always need to use the construction ndigits = <value>. In the last input line, we simply gave the value we wanted. This is the behavior of **positional argument**s. For these, the order in which they are passed determines how they are used in the function. However, we can also name the **parameter** that we wish to assign the value to, and this is what we were doing when we used ndigits = <value>."

"All of this might seem a bit overwhelming at first, and there is certainly a lot of terminology to learn. But Python also has some built-in functions to help. If you want to know what the required and optional

arguments of a function are, the easiest way to do it is to type help(<name of function>), where <name of function> is just the name of the function. In this case, we could type:"

```
>>> help(round)
```

```
1 Help on built-in function round in module builtins:
2
3 round(number, ndigits=None)
4     Round a number to a given precision in decimal digits.
5
6     The return value is an integer if ndigits is omitted or None.  Otherwise
7     the return value has the same type as the number.  ndigits may be negative.
```

"Let's have a look at this output."

"Line 1 just tells you what you asked for. It gives you the name of the function, and where it is located. In this case builtins indicates this is part of the base Python."

"Line 3 is the start of the function—what is called the function 'signature'. We see the name of the function, and then information between parentheses. These are the things that the arguments get passed to—basically **variable**s within the function. The technical name for this is **parameters**. So, parameters exist in the function (as variable names) and arguments are the values that are passed (assigned) to them, when you **call** a function. In this case, we see that there is a parameter number, and a parameter ndigits. The ndigits parameter is also followed by an assignment (=None)[2], which is the default value used, if no value is passed as an argument. The parameter number does not have this assignment, and so has no default value. This is why number is a required argument and ndigits is an optional argument in the function call."

"The rest of the lines gives details about how the function works and what values it will **return**. For instance, look at the very end of Line 7. This states that ndigits can accept a negative number! But it doesn't say what this does, so let us see!"

```
>>> round(42, ndigits = -1)
```

```
40
```

```
>>> round(180.156, ndigits = -2)
```

```
200.0
```

"So, you can see that negative values passed to ndigits will result in integers that are rounded to the specified number of digits before the decimal."

"This could be a useful way to show significant figures, but be careful—as you can see in the last example, when rounding a number with a decimal place to negative decimal places, the number will still contain a decimal place. This might lead to confusion regarding significant figures. There are ways to solve this," Your labmate says. "But that is going to have to wait until our next lunch."

"For now, let's start using this cool calculator for some chemistry!"

[2]A thing to note here: the default value is actually None, not 0. None is the value specific to a **data type** in Python. You don't need to know more about this yet, but if you are curious, this is discussed in the chapter on data types (Chapter B.

0.1.3 Clarifying how your code works with comments

"Even now, you already know enough to calculate chemically-relevant things like the number of molecules present in 1 g of water. You also already know how to do this on paper; it would look like:"

$$\#\text{ of molecules of water} = 1\text{ g}\left(\frac{1\text{ mole}}{18\text{ g}}\right)\left(\frac{6.022\times10^{23}\text{ molecules}}{1\text{ mole}}\right)$$

"We can translate these numbers and mathematical operations directly to code, with one important difference: *Python does not know about units by default.* Like your calculator, it just does math on numbers and you will have to keep track of units yourself. Keeping this in mind, we can calculate the number of molecules in 1 g of water:"

```
>>> 1*1/18.02*6.022*10**23
3.341842397336293e+22
```

"It is important to talk about how the answer is formatted. We input Avogadro's number using the sort of scientific notation that most scientists are taught—the general format <#>$\times 10^{<\#>}$. However, even though we might expect our final answer to also be formatted using this same notation, Python represented it differently." Instead of using '*10**<##>' for the order of magnitude, it used 'e<##>,' which saved some room, and might even be easier to read.[3] A *critical* point here is that there are no spaces between the digits and the 'e.'

"And, remember, Python doesn't include any units in the answer. But *we* know units are important. Rather than needing to always remember what the units are, we can we can leave ourselves notes about this using **comment**s. In Python, we can add a comment to any line using '#' and then adding text after it. Once a comment is started, that will be the last thing on the line. In other words, if you want the line to have both **executable instructions** and comments, the comments need to come last." Your labmate leans over and does the typing this time:

```
>>> 1*1/18.02 # final answer in mols
0.05549389567147614
```

"You see three things here. This is the same format that Python output last time. First, the comment is a different color from the numbers, which are also different colors from the mathematical operator symbols you supplied (like multiplication, $*$). They say this is called **syntax highlighting** and is meant to make it easier to read code. Second, I have added a comment to the end of the expression. Third, the output is the same, but we now have a record that the units were molecules."

"You could also make a line entirely a comment." They also lean over and type:

```
>>> # We want to calculate number of molecules in 1 g water.
>>> 1 * 1 / 18.02 * 6.022e23 # final answer in units of molecules
3.341842397336293e+22
```

Your labmate stops here and says that there are again three things they want you to notice. First, the line that was entirely a comment was run, but it didn't result in an output. Second, I used the e notation for the order of magnitude. Finally, they point out that, though you may not have noticed, they added spaces between several of the numbers and the operations. They say that Python largely ignores spaces *between*

[3] This is sometimes referred to as **e notation**.

sample Website c4c nan

numbers and operators in an expression and so you can add in these space to help increase the readability of the code—but it is *critical* to not add a space between the number and 'e' when you are using this to add an order of magnitude, because this represents a single number only when they are given together without spaces.

"There are other rules for using spaces, but I will help you learn them as you go."

"Before we move on, there's one more thing," your labmate says. "Everything we have been doing so far is evaluating **expression**s, where expressions are basically a complete set of instructions. In our case, the result the evaluation was printed below our input line, and this is the default behavior in Python. In other words, unless we go out of our way to direct the result to something else, it just goes to the command line output.[4] However, there are other options. Above (Section 0.1.2), we passed the result of the expression 0.01/0.1 to the `round()` **function**. Another option is that we can transfer the result to the computer's **memory**, so that we can use it later."

sample, website c4c man

0.1.4 Temporarily storing data using variables

"Up to this point, you may not really see the major advantage of using coding. You may be asking yourself why not just use a calculator?" Your labmate says. However, they point out that the computer has better **memory** than a calculator, and this can be very useful. For instance, what if you didn't want to type out the value of Avogadro's number (N_A) all the time? You can give it a name, and then use that name. This is known as creating a **variable**, which is a name that you can use in your code to avoid having to keep track of the value yourself. You can pick almost any name you want, as long as you follow a few rules, such as the name cannot start with a number.[5] For our variable, `N_A` might be a nice choice, since it looks a bit like N_A. Using sensible and descriptive names is good practice. Your future self will definitely be grateful that your past self did so."

```
>>> N_A = 6.022e23
>>> N_A
6.022e+23
```

"On the first input line, you created a **variable** called '`N_A`' and **assign**ed it a value of 6.022e23 using the assignment **operator** (`=`). The number `6.022e23` is not a variable, it is the actual value that you want to use in the expression, and this turns out to be known as a **literal**. Literals are not stored and have to be retyped every time."

"This assignment **statement** does not result in an output, because the assignment operator (`=`) redirects the result of evaluating the **expression** (on the right side of the equals sign) from the output to instead be stored by the `N_A`. However, in the next input line, you ask for the value of the variable `N_A`. Now, Python evaluates the expression '`N_A`,' and passes the result (`6.022e23`) to the output."

"Rather than passing the value of the variable to the output, we can pass it to an operator, like this:"

```
>>> N_A * 1 * 1 / 18.02
3.3418423973362933e+22
```

[4] This is not actually always the case, but for this moment in time, we will pretend it is. Later in this chapter, we will learn a different way to ensure this always happens—the `print()` function.

[5] If you want to know the full set of rules, you can read technical Section A.1.1.1.

"Here, multiple things happened in a single line. First, `N_A` was first evaluated effectively replacing the variable with its value. Then, the leftmost multiplication was evaluated. The value resulting from that evaluation was used in the next multiplication. That multiplication was evaluated and *it's* values was used in the division. Finally, the division operation was evaluated and the result was displayed in the output line."

"We can also take the result of expressions with variables in them and assign the result to a variable. For instance:"

```
>>> N_water = N_A * 1 * 1 / 18.02
```

"Then, asking for the value of of `N_water` will return the value we saw above:"

```
>>> N_water
3.3418423973362933e+22
```

"It is worth noting something about the order of all these operations. Essentially, the math follows standard order of operations[6]. But in Python, the part of the expression to the right of the equals sign is evaluated first, and then passed to the assignment operator ('='), which then assigns the value to the variable `N_water`."

"Whew! That is a lot of detail," your labmate says, "and not anything we need to talk about over and over again, but it does show that all Python is doing is evaluating expressions and then passing results along. Coding is basically planning out expressions and describing how their results are passed around. And variables are going to play a key role in how we manage all of that."

"You can use variables for anything you would like the computer to remember, like holding the value of the number of grams you have or the molecular mass of water."

```
>>> g_water = 1
>>> MW_water = 18.01528 # g/mol
>>> g_water / MW_water * N_A
3.3427179594211138e+22
```

"There is a point worth making here. The number 18.01528 might look funny as the mass of water, but it shows yet another advantage of coding and computer **memory**: since we can save a value and use it over and over again, we don't waste time typing all those decimal places over and over, letting use as high precision as we like quite easily."

"Now imagine we had a different amount of water. We can simply change the value of the `g_water` variable."

```
>>> g_water = 4.3121
```

"We could just type out the whole expression again, but we can actually do something much faster, as we will see next!"

0.1.5 Accessing a history of code you have executed in the console

"So far, we have just been typing out everything we want to write. But the **console** keeps a record of what we have already typed, so that if we want to repeat a previous command—or modify a small part of a previous command— we can save some time by accessing this record."

"If you are ready to type a command into the console, you can instead press the up arrow key on your keyboard. The cursor will then move through commands you already entered, from the most recent to

[6] That you probably learned as PEMDAS - parentheses, exponent, multiplication, division, addition, subtraction.

the farthest back—moving up through the IDLE display. If you move the cursor to the line that reads "`g_water / MW_water * N_A` and press enter, this will place those commands on the input line, and if you press enter again, it will repeat the calculation. However, between the first time we entered these commands, and now, we have changed the value of the `g_water` **variable**. This time, the expression will use the new value for `g_water`, and will give you the following result:"

```
1.4414134112819785e+23
```

"A variable can only have a single value at a time. If we reassign the value of a variable, we can only access the current value."

"Okay," Your labmate says, "We are not *quite* in a place to completely solve your original problem, but we are in a place to start making progress. For instance, we can create variables to store your stock concentrations so that you don't have to remember them, like this:"

```
>>> conc_stock_cat = 0.1 # M
>>> conc_stock_HCl = 6.0 # M
>>> conc_stock_NaCl = 3.0 # M
```

"We can also store the final volume in a different variable:"

```
>>> vol_final = 1.0 # mL
```

"Now you never have to think about that again! Next, we need to figure out the math that we need to do and translate that to code. We can calculate how much we will need using the $M_1V_1 = M_2V_2$ approach:"

$$V_2 = \frac{M_1V_1}{M_2}$$

"If you were to do this on a calculator and you wanted the answer in mL, you would compute:"

$$V_{\text{stock catalyst}} = \frac{0.01\text{ mol catalyst}}{1\text{ L}}(0.001\text{ L final volume})\frac{1\text{ L stock solution}}{0.1\text{ mole catalyst}}\left(\frac{1000\text{ mL}}{1\text{ L}}\right)$$

"We can translate this equation into Python code by replacing the math variables with Python variables. Let's say you want the final catalyst concentration to be 0.01 M, the final HCl concentration to be 0.006 M, and the ionic strength to be 0.02 M. Let's go ahead and make those variables too."

```
>>> conc_cat = 0.01 # M
>>> conc_HCl_final = 0.006 # M
>>> I = 0.02 # M
```

"You could get the volume of stock catalyst solution needed by:"

```
>>> conc_cat * vol_final / conc_stock_cat # mL
0.09999999999999999
```

"If you are paying close attention, you will notice that the code we typed in does not exactly map onto the solution we outlined above. In particular, we are missing the part where we multiply by 0.001 L fo final volume and then the final conversion between L and mL. The reason for this is that these two steps cancel out, so as long as we leave them *both* out we will still get the correct final answer. This also means that we had three calculations that we did not ask Python to perform, and this speeds up the calculation. Of course, here, the difference would not be noticeable. But if you were going to repeat simple calculations a billion times or more (as you will find yourself doing surprisingly often), you might start to notice the difference. So, simplifying the number of steps can be good, as long as you can justify it."

"Since the calculation is so similar for HCl, we'll just use the up arrow to select the previous line, press Enter, and then edit the line to replace `con_cat` and `conc_stock_cat` with `conc_HCl_final` and `conc_stock_HCl`, respectively:"

```
>>> conc_HCl_final * vol_final / conc_stock_HCl # mL
0.001
```

"Ionic strength is only a little more complicated. You calculated the ionic strength as"

$$I = \frac{1}{2}\left([H^+] + [Cl^-] + [Na^+] + [Cl^-]\right)$$

"so you can get the volume of NaCl stock solution needed by"

$$V^{stock}_{NaCl} = \frac{(I - [HCl]) * V_{sol'n}}{[NaCl]_{stock}}$$

"So we just type"

```
>>> (I - conc_HCl_final) * vol_final / conc_stock_NaCl # mL
```

"The last question is how much water you need to make up the total volume. We need to calculate"

$$V_{water} = V_{sol'n} - V_{cat} - V_{HCl} - V_{NaCl}$$

"This would be easy if we stored all of the results we already got as variables, otherwise we have to either write them down or recalculate them. We can go back and type"

```
>>> vol_cat = conc_cat_final * vol_final / conc_stock_cat # mL
>>> vol_HCl = conc_HCl_final * vol_final / conc_stock_HCl # mL
```

"which then allows us to do this:"

```
>>> vol_NaCl = (I - conc_HCl_final) * vol_final / conc_stock_NaCl # mL
>>> vol_water = vol_final - vol_cat - vol_HCl - vol_NaCl # mL
>>> vol_water
0.8943333333333333
```

"This is convenient, it would still be nice to be able to output numbers in context, with units attached to them, so let's talk about using text in Python."

0.1.6 Expressing data as readable text using strings

"Even though Python doesn't know the units associated with the numbers, *we* do. So we can add them to the answer if we like. When we want to give Python text (like letters, words, and numerals) instead of code and numbers, we place the text between apostrophes (`'`), quotation marks (`"`), triple apostrophes (`'''`), or triple sets of quotation marks `"""`). This lets Python know that we are dealing with text, rather than numbers or code. From Python's perspective, numbers and text are different kinds of data, or different **data type**s. The technical term for textual data is a **string**. We can create these strings and assign them to **variable**s:"

```
>>> greeting = `Hello there.'
>>> greeting
`Hello there.'
```

"Here you can see that Python indicates that the value output is a string by placing it between apostrophes. Of course, you might not want to see the apostrophes when Python writes out the string, and this can be done by using the `print()` **function**. This function takes 1 or more arguments and outputs the value of the **argument** to the standard output, which is in this case the output in our **console** window (the thing we have been seeing as the output).

```
>>> print(greeting)
Hello there.
```

You can see the difference is that Python has output the value assigned to the variable `greeting` (Hello there.), without an indication of the type of data this is (*i.e.*, there are no bracketing apostrophes).

"Of course, we can also provide data types besides strings as arguments to the `print()` function. For instance, we could use one of the variables that has numbers associated with it."

```
>>> print(MW_water)
18.01528
```

"Another thing that is useful about `print()` is that it can accept multiple arguments, as long as we separate them by commas. It will then display each one sequentially, separated by a space."

```
>>> water_molecules = g_water / MW_water * N_A
>>> print(water_molecules, 'molecules of water.')
1.4414134112819785e+e23 molecules of water.
```

"Suppose we wanted this output to be on two different lines. If we type them separately, then this happens"

```
>>> print('Molecules of water:')
Molecules of water:
```

```
>>> print(water_molecules)
1.4414134112819785e+e23
```

"But the outputs are now broken by a line of input, which isn't great. Luckily, we can make one print statement that prints multiple lines of text:"

```
>>> print('Molecules of water:\n', water_molecules)
Molecules of water:
1.4414134112819785e+e23
```

"Here, We used `\n` in our string, and this really gets to the heart of how computers see text.[7] The computer has no concept of paragraphs. Even though it is automatic for us to end a paragraph and start a new line, we have to *tell* the computer where to do this. We do this by using a symbol to indicate this. The symbol is the **newline character** and, in Python, it is `\n`. So, we are using this character to make the output more readable. There are many other such characters that specify formatting, with the other one you will likely need being tab: `\t`."

"So you can see that we get the output we wanted. The next time we meet, we will talk about how to format this even more. For today, let's end by looking at everything we have done so far and talk about how to tie it all up so it is easy to use."

[7]If you want to read more about how text works in detail, see Technical Chapter C. The information is not needed for this chapter, but we point it out in case you are interested.

0.1.7 Avoiding repetitive typing by making your own functions

"You have already used a couple of **function**s that are built in to Python: `print()` and `round()`. These functions allow you to easily accomplish common tasks. Essentially, you can think of them as taking a series of instructions that accomplish this task and bundling them together, so that you only need to write out the name of the function, rather than all the instructions. This is certainly very convenient! And wouldn't it be nice if you had a function that did this for the amounts of stock solution you need to add for your experiment?"

"Well, you are in luck! While Python may not automatically include such a function, you can make your own functions that do basically anything that you can code up! To do so, you simply take code you write and then define a function to use it. Then, you can just **call** the function, and the code you wrote is hidden away inside of it so that only Python has to deal with it. Functions have four basic parts:"[8]

- A name—function names need to follow the same conventions as variable names.[9]
- Zero or more **parameter**s that define the data that the function will use. These hold that values that are passed as arguments.
- The code in the function—this needs to be indented to make it clear that the code belongs to the function instead of the rest of the code. Indentation is one type of **white space**, used to tell Python how code should be executed. For any function, all of the contiguous lines of code after a function definition that have the same indentation are interpreted to be part of the code inside the function. The first line of code where there is no longer indentation is outside of the function and is the signal to the **Python interpreter** that the code is no longer part of the function. As you will see later, there are reasons why you will need to use additional indents to specify more complex code inside of functions, but this is enough for now.
- The value(s) that the function will **return** when executed. Sometimes there are no values returned, and this is fine. Remember when we used `round()`? The rounded value was the value that was returned from that function when it was evaluated.

"When Python evaluates the line of code with the function call, it will evaluate the code in the function and then replace the function call with the returned value."

"Using the IDLE (as we have so far), you *could* choose to create a function by typing the function into the **console**, line-by-line. However, a *much* simpler way is to create a file that contains the code we want to use and then read the code from this file. If you click File–>New File in IDLE, you will get a new window to define your function."

Your labmate types into the IDLE text editor:

```
1 # A function to calculate the volumes of stock solutions needed to create
    solutions with given catalyst, HCl, and ionic strength
2 # All concentrations are in M
3 # The final volume is in mL and assumed to be 1 unless provided
4
5 def calcPlateVols(conc_cat, conc_HCl, I, vol_final = 1):
6
7     # Define our stock solutions
8     conc_stock_cat = 0.1 # M
```

[8] A more complete discussion of how functions work can be found in the technical chapter on functions (Chapter G.
[9] See Section A.1.1.1.

Sample, Website c4c nan

```
9      conc_stock_HCl = 6 # M
10     conc_stock_NaCl = 3 # M
11
12     # Calculate the volumes needed
13     vol_cat = conc_cat / conc_stock_cat * vol_final
14     vol_HCl = conc_HCl / conc_stock_HCl * vol_final
15     vol_NaCl = (I - conc_HCl) / conc_stock_NaCl * vol_final
16
17     # Calculate the water needed to make up 1 mL
18     vol_water = vol_final - vol_cat - vol_HCl - vol_NaCl
19
20     print('[ ] catalyst solution (mL)\n', vol_cat)
21     print('[ ] HCl (mL)\n', vol_HCl)
22     print('[ ] NaCl (mL)\n', vol_NaCl)
23     print('[ ] water (mL)\n', vol_water)
```

and saves the file as calcPlateVols.py.

"Let's talk through this code line by line."

1–3: We use **comment**s to let people know what this code does and how to use it. The first comment explains what the code does. The next two give the assumptions the code makes.

4: This line in blank. Though not required, using blank lines can increase the readability of the code. We often use them to indicate separation of tasks.

5: The definition of the **function** starts with the `def` **keyword**, which tells Python that what follows is the name of the function and that the lines that follow will have the instructions that comprise the working of the function. The name of the function is the name that you will need to type in Python when you want to use the function, like "print" for the `print()` function or "round" for the `round()` function. After that, you have parentheses, and everything between the parentheses is a **parameter**... the names of the **variable**s inside the function that **argument**s will be **pass**ed to when the function is **call**ed. This is like how you provided values to the `round()` function. In the case of this function, you can type `calcPlateVols(.01, .006, .02)` and in the function, there will be three variables: `conc_cat` (holding the value 0.01), `conc_HCl` (holding the value 0.006), and `conc_NaCl` (holding the value 0.02). We will set the final volume of the wells, `vol_final`, to be a **keyword argument**. This allows us to set a default value for the volume (in this case 1mL), but also allow the user to easily change the value when the function is called.

7: Another comment to tell us what this part of the function does.

8–10: Here we are defining variables to hold the concentrations of our stock solutions, just like we did when using the interpreter. This type of usage is known as **hard coding**, because we cannot change the values without changing the code. A more general usage would be to add these as keyword parameters in the function definition, with these default values assigned. However, since this was a quick tool for a specific situation, and you commonly use these stock concentrations, they are hard-coded.

12: Another comment to tell us what this part of the function does. It is good practice to leave comments that help a user understand the steps you are trying to accomplish.

13–15: Now we can compute the volumes of stock solutions needed the same way as we did earlier in this chapter.

17: Another comment, because we are now moving on to a new step in our solution.

18: We're calculating the volume of water needed.

20–23: This is a series of print statements to output the resulting arrays in human-readable form. Each one prints two lines, one that has a checkbox and a description of the solution to be added, and a second that prints the volume that needs to be added. Even though the print statement is on a single line, it will output two lines, because we are using the newline character, `\n`.

"Now, when you run this code by clicking Run–>Run Module, Python will step through this code line by line and run each line in the interpreter. When it is finished, Python will now know that there is a new function and how it should behave. So, we can use it like any other function:"

```
>>> calcPlateVols(.01, .006, .02)
```

```
[ ] catalyst solution (mL)
 0.09999999999999999
[ ] HCl (mL)
```

```
 0.001
[ ] NaCl (mL)
 0.004666666666666667
[ ] water (mL)
 0.8943333333333333
```

"Now, you just have to press the up key and enter to copy the last line into the input, change whichever concentration you want to, press enter again to run it, and get a new calculation! You can use this to calculate the value for all the wells in your experiment."

"In my experience" your labmate says, "this is *already* better than using a calculator or Excel, but if we learn one more concept, we can make these calculations even easier. Like, we can apply this function to every well of your 24 well plate. And if we plan it out correctly, we could use the same approach for a 96 well plate, or a plate of any size!"

Your labmate glances at her watch and says, "But I have to get back to the lab to finish up an experiment, we can pick this up tomorrow at lunch!"

Given the amount you have learned in only this hour, you can hardly wait.

The Python Kernel

Each session of Python involves the 'kernel.' The Python Kernel is the core execution environment that runs Python code in interactive computing environments like the IDLE. The kernel is what keeps track of variable names, variable values, defined functions, and so on. If you close IDLE, this closes the kernel. The reason to mention this is that, if you close IDLE, and then open it and immediately try to run `calcPlateVols()` you will get an **error** message:

```
NameError: name 'calcPlateVols' is not
defined
```

which indicates that Python does not know what this function is. You must first run the code that defines the function, before you can use it.

Sample Website C4C nan

0.2 Exercises

Targeted exercises

Exercises for Section 0.1.0

0. Using Anaconda Navigator, verify that the following Python **libraries** are installed, and if not, install them. If you want directions you can use the resources at https://www.codingforchemistsbook.com/
   a) numpy
   b) scipy
   c) lmfit
   d) plotly
   e) kaleido
   f) pillow
   g) openpyxl
   h) python-docx
   i) rdkit
   j) nbformat
   k) codechembook
1. Run plotly-test.py from the codechembook library. If you need more directions, refer to the resources at https://www.codingforchemistsbook.com/.

Exercises for Section 0.1.1

2. Compute the following outside of Python, and then have Python compute them. Compare the answers:

a) $9^{1/2}$
b) $9^{0.5}$
c) $5 + \frac{7}{8}$
d) $5 + 7 \div 8$
e) $(5 + 7) \div 8$
f) $(4 + 5)^{1/2}$
g) $9 \div 3^2$
h) 2^{3+1}

Exercises for Section 0.1.2

3. Compute the following outside of Python, and then have Python compute them. Compare the answers:
   a) $2 - 3$
   b) $|3 - 2|$
   c) $4.5 + 3.2 \times 7.8 - 12.1$
   d) $(4.5 + 3.2) \times (7.8 - 12.1)$
   e) $(4.5 + 3.2) \times (|7.8 - 12.1|)$
4. A arithmetic operator that is included in Python which you haven't used yet in this chapter is the modulo operator (%). This produces the remainder of a division operation. Calculate the following outside of Python and using Python, and then compare the answers:
   a) 13%10
   b) 10%13
   c) 7%2
   d) 6%2
5. A function included in Python that you haven't yet seen is `pow()`, which raises one number to the power of another and accepts two **positional arguments** and one **keyword argument**. Use the `help()` function to get information on how to use `pow()`, and then use it to compute the following:
   a) $2^{3.14}$
   b) $\sqrt{2}$
   c) $7^{3/4}$
   d) A rendering of Avogadro's number.
6. In Excercise 4, you learned modulo, and in Excercise 5, you learned to use the `pow()` function. Using the `pow()` function only, check to see if 7^3 is a factor of 3.
7. Using the `round()` function, reproduce the answers to Excercises 0-2, rounded to the nearest tenths.

Exercises for Section 0.1.3

8. Produce comments for all the code you used to solve the exercises in 0.1.2.

Exercises for Section 0.1.4

9. Which of the following are valid variable names in Python? Fore invalid variable names, propose a new name that is acceptable.
   a) `2nd_trial`
   b) `length*width`
   c) `^volume`
   d) `import`
   e) `__hidden_variable`
   f) `_`
   g) `fast component`
10. Create valid Python variables for the masses of carbon, hydrogen, oxygen, and nitrogen. Then using these variables, compute the molar masses of:
    a) NO_2
    b) glucose

sample: Website CAC nan

c) glycine
d) benzene
e) pyridine
f) cyclohexane
g) 2-hexene
h) water

sample, Website c4c nan

Exercises for Section 0.1.5

11. Using only the mass of hydrogen, carbon, nitrogen, and oxygen, compute the mass of hexaglycine simply by adding up all the masses of the elements.
12. Solve Exercise 11 and then, without retyping any code, press up until you highlight the line for calculating the mass of glycine. Then modify this line to store it as a variable. Then use this to calculate hexaglycine.

Exercises for Section 0.1.6

13. Enter each of the following into the console and press return, then also put the value inside a `print()` statement and compare the outputs.
    a) 2
    b) '2'
    c) 2.0
    d) '2.0'
    e) 2/2
14. Using a single line of code with a single print command, output a grammatically correct sentence that answers the following questions (perform the math, if necessary, and output with correct units):
    a) What is the number of moles of water in 14.7g of water?
    b) What is the density of toluene in g/mL?
    c) How many mL are in 1 gallon?
    d) What is 72.5 °F in °C?

Exercises for Section 0.1.7

15. Perform the following actions:
    a) Make a function that calculates the concentration of protons at any given pH.
    b) Modify the function to print this value
    c) Modify the function return this value to a print statement that is outside the function.
    d) In one line of code, without further modifying the function, use the value returned by the function to calculate the number of protons in 1L of neutral water.
    e) In one line of code, without further modifying the function, calculate how many more protons there are at pH=7 versus pH=10

Comprehensive exercises

16. Write a function that can solve dilution problems ($M_1V_1 = M_2V_2$) for M_1.
17. Write a function that can solve dilution problems ($M_1V_1 = M_2V_2$) for V_1.
18. Another operator in Python is the modulo operator (`%`). Using this operator, write a function that will take two masses, and compute how many water molecules they differ by. Write your function so that the order of masses supplied as arguments to the function does not matter. If you need help understanding the modulo operator, you can read the technical chapter on operators in our book, or you can read the Python documentation, or you can ask an AI tool about it.
19. You have a spectroscopic detector that you need to cool to 6 K using liquid He. You know the heat capacity of the detector is $125\text{J} \cdot \text{K}^{-1}$. Write a function that will take the current (room) temperature of the detector and calculate the number of liters of liquid He it will take to cool down the detector. Assume that it requires 5L to simply cool down the transfer line, before you start cooling the detector. Using this function, create a print statement that converts the function's output to a cost in dollars to cool

down the detector. At the time of this writing, the cost of liquid helium is $40/L. If you like, you can try to update this for the cost at the time of your reading.

sample, Website c4c nan