A Supplementary Guide to Computational Thinking

Ariel Feldman

07/23/18

1 Introduction

Welcome to Introduction to Computational Thinking! The goal of this course is to familiarize you with computational methods for problem solving, and though it may be daunting at first to those with little computational background, there is no reason you can't succeed.

As you continue through this course, it is important to remember that you are not learning how to *code*, but rather how to look at mathematical problems in a new light. There are numerous problems more easily solved with a computational approach that shall be highlighted through the semester. Hopefully, this course adds another weapon in your arsenal when presented with an algorithmic challenge.

This guide shall aim to provide you with resources, practice problems, and additional explanations of topics taught in class. In no way does this replace class — Dr. Rixner provides great explanations and very useful feedback, which I strongly encourage you to take advantage of. However, I'd had very little experience with computers prior to taking the course and thus encountered a steep learning curve. Thus, I hope this guide eases your transition, and I welcome you to reach out to me or any of the other TA's (whose contact info you may find on the canvas site) with any questions you may have. My email is akf1 [at] rice [dot] edu.

2 Python as a Tool

Before we begin introducing you to mathematical concepts, let's go over some basic Python. The reason we begin with Python over another language is because the syntax is somewhat easier to understand without a programmatic background than other languages, allowing us to focus more on learning the algorithmic techniques /textitbehind our problem solving solutions than on learning the language itself.

A few general links that are also provided for you on Canvas are listed below. You definitely don't need to study these, but scrolling through may help you familiarize yourself with the language:

1. Python 2.7 Documentation
2. Beginning Python
3. Learning Python
4. The *Official* Python Tutorial
5. A Beginner's Python Tutorial

And now, we'll go through some basic functions in practice. In future unit, I'll explain a bit more about the specific Python requirements you may need to know. But for now, let's stick to more general syntax and structures you'll need to know.

Let's start with defining a function. The following code can be used to define a function called colleges (by convention, we tend to name functions beginning with lowercase letters; camelcase may be used for variable names consisting of multiple words) that will return different results based on the college name:

```
def collegeComments(college):
    """
    Function to take a college name and print a comment regarding that school.
    """
    # Checking if the colleges are relevant, and printing a comment about them
    # if they are.

    if college == "Rice":
     print "unconventional"
    elif college == "Berkeley":
     print "memes"
    else:
     print "irrelevant"
```

There's a lot to unpack here for a beginner. Above is an example of documented code: the docstring explains the overall goal of the function, while the comments describe steps (since we have a very simple function, there is only one comment — more are needed for more complex functions!). The *def* is used to define a function, meaning the word after it is the function name you will call later, and the name(s) in parentheses is(/are) the input to the function.

The *if* statements, or conditionals, check if the input *college* meet the requirements set, and print a comment if so. Python is a serial language, meaning it runs the lines you write in order. Therefore, college name will first be *compared* (demarcated by the ==) to the string "Rice", then to the string "Berkeley". If neither condition is satisfied, the *else* statement will be entered. Since we are using Python 2.x, print statements do not need parentheses, but be aware that they are needed for Python 3.x versions.

It will be useful to understand basic data structures in Python. Let's break down some built-in structures into two categories: ordered vs. non-ordered.

2.1 Ordered Data Structures in Python

Lists, tuples and strings are three ordered sequences you will need to be familiar with for this course. Tuples and strings are known as "immutable types", basically meaning that the data contained within them cannot be changed. You can print altered versions of the data, and you can create other objects containing the same data — we will cover objects and references later — but you may not alter the original object containing said data. Lists, on the other hand, are very easily mutable. Lists and tuples can contain any sort of data (integers, floating point numbers, etc.), while strings contain only characters.

Lists are formed with brackets:

$$\text{sampleList} = [1, 2, 3]$$

Tuples are formed with parentheses or comas:

$$\begin{aligned} \text{sampleTuple} &= (1, 2, 3) \\ &\text{or} \\ \text{sampleTuple} &= 1, 2, 3 \end{aligned}$$

Strings are formed with quotation marks:

$$\begin{aligned} \text{sampleString} &= \text{“}character\text{”} \\ &\text{or} \\ \text{sampleString} &= \text{‘}character\text{’} \\ &\text{or} \\ \text{sampleString} &= \text{str}(character) \end{aligned}$$

You may index any of these data structures using brackets like so:

$$\text{sampleList}[0] = 1$$
$$\text{sampleTuple}[1] = 2$$
$$\text{sampleString}[2] = \text{“}a\text{”}$$

2.2 Unordered Data Structures in Python

Dictionaries and sets are two unordered data structures you will need to be familiar with in this course. Since they are unordered, you cannot index them as you can with ordered sequences. However, they do have some advantages:

> Sets cannot contain duplicates. Thus, adding an item already contained within a set to said set does not *do* anything to it. Click the hyperlink above for explanations of set operations such as intersection, difference and symmetric difference.
>
> Dictionaries — refer to these as mappings in your recipes, as dictionary is not a mathematical term — do just that: map a key to a value. Thus, you can index a dictionary by a key to retrieve the related value(s). Imagine we had a dictionary called *college* of colleges in our function on page 2 mapped to the comments they print out. Calling *college*[“Rice”] would return *“unconventional”*, but calling *college*[1] would be an error.

To create a set:

$$\text{sampleSet} = set([1, 2, 3])$$

To create a dictionary:

$$\text{sampleDict} = \{\text{“}Rice\text{”} : \text{“}unconventional\text{”}, \text{“}Berkeley\text{”} : \text{“}memes\text{”}\}$$

Let’s say you try to make a set with duplicates. What happens then?

$$\text{dupSet} = set([1, 1, 2, 3])$$
$$print\ \text{dupSet}$$
$$\text{dupSet} = set([1, 2, 3])$$

Another feature of Python with which you should be familiar is loops: specifically, for loops and while loops.

2.3 For Loops

For loops are useful to repeat a set of functions or operations on multiple elements within a sequence. Let’s refer back to our list *l*, defined above.

```
'''
Adding all the elements in a list together
'''
    sum = 0
    for element in l:
     sum += element
    return sum
```

Clearly, this will return 6 (adding 1, 2 and 3 together). However, as you will see through this course, for loops can often times come in handy for more complex operations.

2.4 While Loops

A while loop will continue performing specified operations until a condition is no longer met. Let's once again refer back to our list *sampleList*.

```
"""
print numbers until length of the list sampleList
"""
num = 0
while num < len(sampleList):
 num +=1
    print num
```

Clearly, this code will print 1, then 2, and finally 3. However, since the length of the list *sampleList* is 3, once *num* is changed to 3 and we return to the top of the loop, *num* no longer meets the condition $<$ *len(sampleList)*, and thus the loop is exited. While loops may also be performed with boolean conditionals (while True or while False).

However, be cautious when creating your *while loop*s — it is important that you **always have a break case**. What does this mean? Well, let's take a quick look at another *while loop*.

```
"""
Keep printing numbers, multiplying by 2
"""
    num = 1
    while num > 0:
     num *= 2
```

So, what happens in this loop? You can see that we first initialize *num* to a value of 1 (notice how I phrased that — keep that in mind while writing your recipes), and then, while *num* has a value larger than zero, we continually multiply *num* by 2. In the first *while loop* we looked at, we saw that *num* no longer satisfied the conditions for the loop once it was equal in value to the length of the input sequence. Now reconsider the loop we defined just above. When does *num* no longer satisfy the condition for the loop? Since *num* is initialized to a positive integer, and we continually multiply it by a positive integer, *num* is never going to be negative. Thus, it will never break out of the loop — a case we refer to as an infinite loop.

3 Some Advice On Recipes

Remember several things while writing your recipes:

1. Refer to everything in mathematical terms (i.e. list as sequence, dictionary as mapping). The emphasis in this course is not on programmatic proficiency, but rather on mathematical understanding. Therefore, you should explain your reasoning in those (mathematical) terms.
2. I cannot emphasize this enough: write your recipes ***before*** writing your code. This makes writing your algorithm *so* much easier, and helps the debugging process. If you decide not to do this, you will find in later courses (and perhaps in this one as well) that you are simply unable to do so.
3. Your recipe can be similar to code in terms of indentation, i.e.

   ```
   for each element in sequence l, do:
    add 2 to the element
   ```

   This may help you keep your algorithm organized, and will make implementing it easier.

4 Approaching The Projects

Make sure you clearly read the assignment page prior to beginning the problem, including any additional materials your professor may have added. It may seem like a lot, but I promise you it'll save time in the end. Additionally, don't be afraid to come to office hours! It's what we are here for, and many students do not take advantage of us as a resource.

You can google mathematical concepts or ask a friend if you need extra help, but remember to abide by the honor code, described on Canvas. Like I said before, please write the recipes ***before*** implementing in Python. It'll be *much* easier to get help from a TA if you can clearly explain and reason through your algorithm.

While on the topic of seeking help from TAs — please, please, please give your variables useful names! Though you can *technically* name your variables, functions and classes anything you'd like (so long as it is not already defined by Python, such as str), it makes both debugging on your own and explaining what you did to a TA *so* much easier. As well, it is good practice to do so outside of your courses. Many people I work with spend so much more time debugging because they forget what *exactly* certain steps of their algorithms were attempting to do — which should also signify to you commenting and adding docstrings is applicationally relevant! — and I, too, have found well named variables, functions and classes incredibly helpful whilst writing algorithms. Thus, I'd encourage you to name a list more than just *l* or *string*, but rather something like *colleges* or *courses*, so long as it is relevant to the problem at hand.

It is helpful to test your code on OwlTest as you're debugging, as it will give examples of edge cases that may not be accounted for by your algorithm. As always, it is very helpful to think through your algorithm and try to identify edge cases to test before submitting, as not all possible edge cases are likely in OwlTest. OwlTest (or any method to check your work prior to final submission) is a resource you may not have both in later courses and in practice, so learning how to debug now will benefit you tremendously.

Since Module 1 is relatively well guided, and is just to get you familiarized with thinking in a computational manner, let's skip ahead to Module 2.

5 Module 2

We already discussed conditionals briefly above, but let's dive a little deeper. Conditional statements are useful when you want to perform specific operations on a subset of cases, i.e. when some specified requirements are met. Let's breakdown that algorithm we wrote on page 2, *collegeComments*, into English terms. Assume the user passes in a college name we did not account for, such as *Harvard*, so that we shall encounter each of the three conditional statements.

1. First, we will check if the college name passed in is Rice. Note that we use the '==' here, indicating that we are comparing the values of the objects, not the actual objects themselves (a topic for a later unit). Harvard isn't Rice, sadly for them, so we move onto the next statement.
2. Second, we'll move into the first (and in this case, only) "else, if"" statement (written *elif*). Is Harvard the same as Berkeley? Nope. Time to move onto the next statement.
3. Since there are no more "else, if" statements, we would exit the conditionals. But wait! Before we do, we check if there is an else statement — in our case, there is! The else statement will provide default operations for any cases that do not meet those specified in the "if" and "else, if" cases. Since Harvard did not satisfy those cases, we perform the default operation and print "irrelevant".

5.1 Modular Arithmetic

Perhaps a new mathematical concept to you is modular arithmetic. Not to worry! It's actually quite a simple (and useful tool) to think about. Modular (represented by $\%$), simply determines a type of remainder. Let's say, for example, I have $18\%10$. When I divide 18 by 10, I have 8 numbers left over that 10 does not fit nicely into. Thus, $18\%10 = 8$. It may be useful to think of this like a clock, in which the hours of the day are evaluated using $\%12$.

The hours of the day, then sort of *wrap around* this number — 17 hours into the day is referred to as 5 P.M., since $17 \% 12 = 5$. An applicational use of this can be observed in a modified version of DeepLabCut, a laboratory solution for animal video tracking in which the objects to be tracked must be initially labeled in training images (for a neural network) in the exact same order, as the labels are assigned values using modular arithmetic.

In Python, you can think of *mod* as an operator. However, mathematically speaking, *mod* is a sort of modifier to the equality assertion, in the sense that $5 = 17$ when the modulus is 12. It's a small difference that probably does not matter in this course, but personally I prefer to know proper definitions — you never know when the proper definition may just make all the pieces of a problem click.

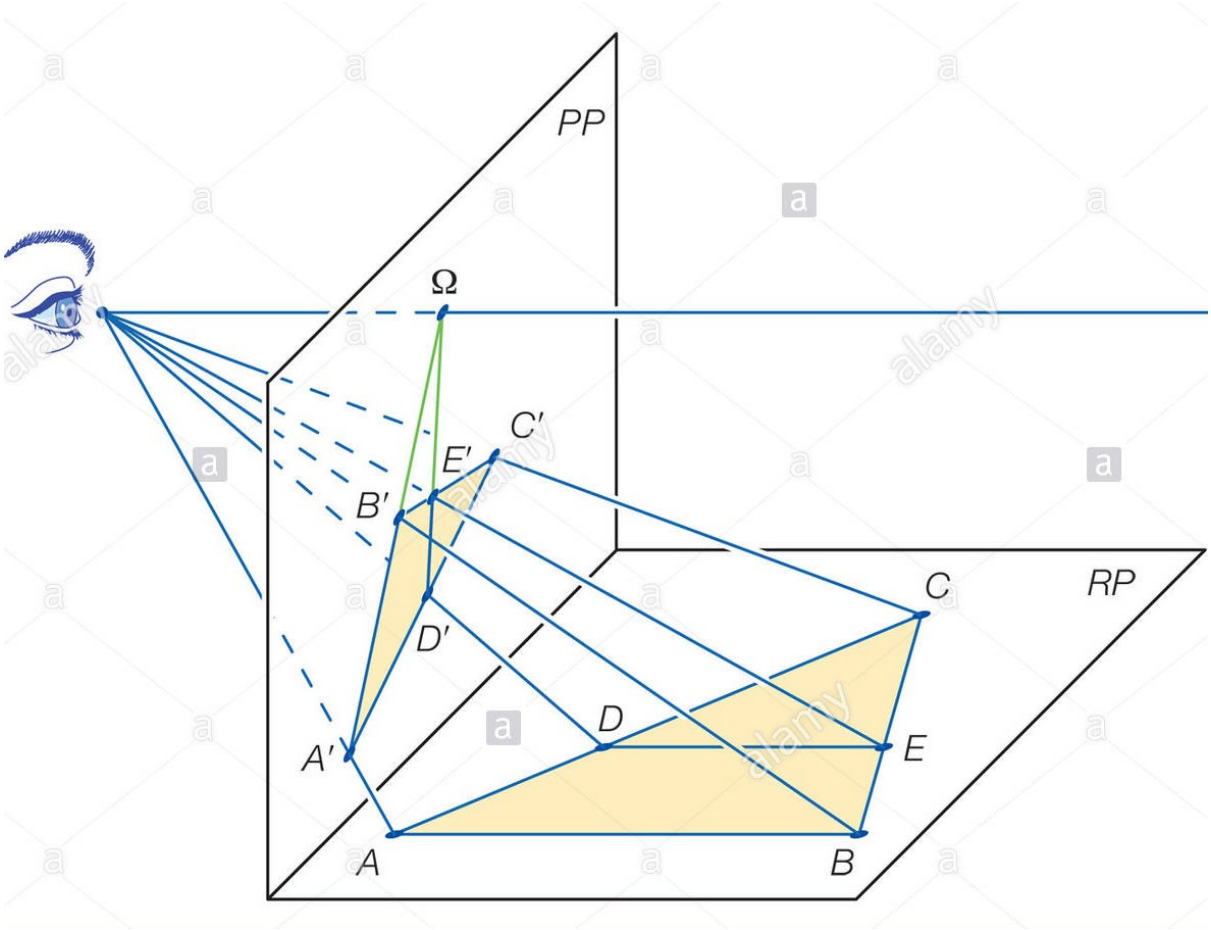
5.2 Projective Geometry

Projective geometry was, for me, arguably the hardest concept (besides perhaps recursion) to understand. So, let's spend a bit of time on this one. Imagine you're looking at a painting, like the one below by Leonid Afremov called City by the Lake.

You may be wondering how a scene this beautiful (isn't it something?) may relate to projective geometry. But, if you're looking at the painting like a projective mathematician, you'll probably be looking at the road by the lake, as opposed to the city. Why's that? Think about an ideal road — no, not the lack of potholes, but rather how perfectly parallel the sides are. Imagine the road in the painting is one of the "ideal" ones. Would the sides still appear to converge in the distance? Of course they would. It's a basic rule of art: perspective. Could two parallel lines converge in Euclidean space (the 2-dimensional geometric plane, and 3-dimensional space, on which all geometry

you've likely learned prior is based)? No! In fact, that's the definition of parallel. However, in projective space, it's very possible. In fact, you can just think of projective geometry as a mathematical backing to the concept of perspective.

But how does projective geometry rationalize this? Consider the next image.



We know from Euclidean geometry's fundamental theorem of similarity that $\overline{DE}$ of triangle *ABC* splits $\overline{AC}$ and $\overline{BC}$ proportionally. Now let's project triangle *ABC* (from plane *RP*) onto plane *PP*. Take a look at $\overline{A'B'}$ and $\overline{D'E'}$. Still parallel? They don't seem to be — in fact, they seem to converge at some point along the horizon, labeled Ω in the figure. This accounts for the visual distortion we experience when looking at objects from different points of view — perspectives, if you will (see? it all comes together somehow!).

The take away from this is that, in projective space, we can ignore certain geometric features (i.e. distances and angles) while maintaining others (i.e. which two points are connected by a line segment). Some fundamental principles of projective geometry are as follows:

1. If **A** and **B** are distinct points on a plane, there is at least one line containing both **A** and **B**.
2. If **A** and **B** are distinct points on a plane, there is not more than one line containing both **A** and **B**.
3. Any two lines in a plane have at least one point of the plane (which, in the case of "parallel" lines, may be the point at infinity) in common.
4. There is at least one line on a plane.
5. Every line contains at least three points of the plane.
6. All the points of the plane do not belong to the same line.

For your project, you need to be familiar with the Fano Plane. Check here for a nice article discussing this plane — the smallest instance of projective space.

If you need some extra help (or are just deeply interested in the topic), I recommend njwildberger's YouTube videos explaining projective geometry. He goes *way* more in depth than you'll need to know, but he's the best at explaining it that I've found on YouTube so far.

6 Module 3

In this module, we're going to discuss list comprehensions, revisit the aforementioned dictionaries (as well as the slightly modified default dictionary), randomness, enumerate, zip and touch upon markov chains (these are incredibly cool, powerful tools that you likely won't encounter in detail until later.)

6.1 List Comprehensions

It may be easiest to think about list comprehensions like syntactically modified *for loop*s. In general, the structure of list comprehensions in Python appear like so:

```
[operation | for item in list | if conditional]
```

I have the different pieces of a traditional *for loop* separated here by a "|". Note that the "|" would not be used in practice — it is only to help you visualize where the different pieces of the loop go. The above list comprehension is equivalent to:

```
for item in list:
 if conditional == True:
     operation
```

You can think of this conditional statement as a sort of filter on the input data, the same way the conditional statement in the traditional *for loop* acts on the input data structure.

6.2 Randomness

Randomness has its own subdivision of research and extensive testing to determine whether or not a generator is *truly* random, but for the purposes of this course we shall refer to the "pseudo-random" numbers generated by the python *random* library simply as "random".

As you probably already learned in your video lectures, the idea of randomness is rooted in probability, which is a mathematical way of expressing the likelihood of an event. We can refer to all the possible events that may occur in a given scenario as that scenario's event space. The sum of the probabilities of the events occurring across the event space should be 1, since one of the "events" should occur. As would logically follow, P(0) — a probability of an event taking place being valued at 0 — would indicate no chance of an event occurring, while P(1) would indicate an event is definitely going to occur, no way around it. Typically, event probabilities are scaled between these two values.

In COMP 140, you will be using the following random library functions. From the Python documentation, their purposes are as follows:

1. To return a random floating point number in the range [0.0, 1.0), all with equal probability, use:

   ```
   random.random(a)
   ```

2. To return a randomly selected integer with equal probability either from 0 to a stopping number, or from a specified range, use:

   ```
   random.randrange(a,b)
   ```

where *a* represents a starting number of the range, and *b* represents the number up to (but not including) the range covers. If only one number is given as input, it is inherently assumed to be *b*, and *a* shall be assumed to be 0.

3. To return a random element from a non-empty sequence with equal probability, use:

```
random.choice(sequence)
```

6.3 Dictionaries

Although we've already touched on dictionaries, they are worth discussing in a bit more detail. The idea of keys and values were briefly mentioned above in section 2, and are integral to your understanding of a dictionary. Dictionaries map keys to values, and not the other way around. In this course, you should limit the *key*s of your dictionaries to numbers and strings, while the *value*s can be any Python data structure. Therefore, if we refer back to the college dictionary we defined in section 2.2, the college names would be the keys, and the comments would be the values. As shown earlier, you can index the dictionary by calling:

```
sampleDict[''Rice'']
```

to return the corresponding value. However, if you were to attempt to index a dictionary by, say, a value (or any other string/variable/object not represented as a key in said dictionary):

```
sampleDict[''unconventional'']
```

you would encounter an error. That is, you would encounter an error *unless* you were using a default dictionary. Default dictionaries allow you to do exactly what they sound like — set a default value for any key not already explicitly represented in the dictionary. To implement a default dictionary, you must first consider the functionality of the dictionary in order to determine what you set the default value to be. Say you want to count the occurrence of something — you would implement a default dictionary in the following manner:

```
countDict = defaultdict(int)
```

Thus, when countDict is indexed by some object that is not already present in the keys, the corresponding value has already been initialized to 0. This allows us to directly manipulated the mapping without having to explicitly set values for new keys we are adding, although that functionality is also preserved from the standard Python dictionary data structure. If we wanted to use a default dictionary to represent the function collegeComments, we would need to use a lambda — so let's hold off on setting specific default values (rather than just types).

To iterate over either all the keys or values in a dictionary, you may call (respectively):

```
for key in sampleDict.keys():
 print key
for value in sampleDict.values():
 print value
```

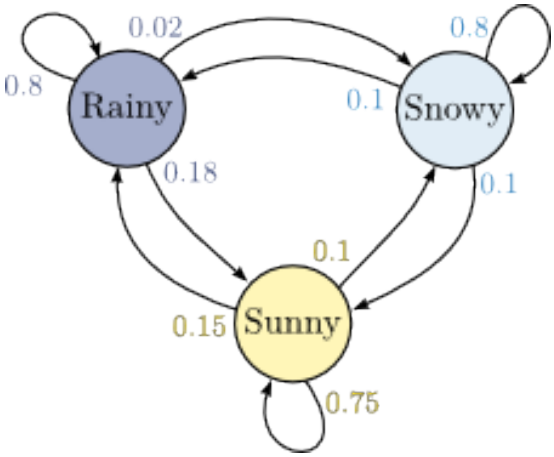
If you were to call just:

```
sampleDict.keys()
```

you would receive a sequence of the keys in *sampleDict* in no particular order — remember, dictionaries are an *unordered* data structure.

6.4 Markov Chains

Markov models are especially cool methods of investigating hidden patterns/processes. I encourage you to check them out in your free time, like this eLife paper authored by a friend of mine. However, for the purposes of this course we shall refrain from going to deep into detail of Markov models, restricting ourselves to Markov chains with and without memory. It may be easiest to understand if we begin discussing a simple markov chain ourselves.



Imagine the chain above represents the weather forecast for Chicago, since it rarely snows in Houston, and I can confirm Chicago weather adheres to a similar pattern. This chain exhibits the Markov property since the future states rely only on the current state the city is in. Say it is sunny in Chicago. Regardless of whether it was raining or snowing yesterday, the probability of whether it will rain or snow tomorrow remains unchanged. If say, the probabilities were different based on *how* we got to be in the nice state — if it had a better chance of being nice again tomorrow if it rained yesterday than if it had snowed — then the chain does not exhibit the Markov property.

Maybe you're looking at this chain and are struggling to read it. Assuming the weather in Chicago is sunny, there is a 75% chance that it will be sunny again tomorrow. Otherwise, the chance of rain is 15%, and the chance of snow is 10%.