1. TL;DR

Pseudocode:
Set \textit{var} to \textit{val}
if \textit{var} equals \textit{val}, then
execute this block
for each number, \textit{CN}, in the list of numbers
execute this block, where \textit{CN} is the current number
Set \textit{i} to 1
while \textit{i} is less than or equal to \textit{N},
execute indented block
increment \textit{i}

Python:
\textit{var} = \textit{val}
if \textit{var} == \textit{val}:
    #execute this block
for \textit{CN} in \textit{numList}:
    # execute this block, where \textit{CN} is the current number
\textit{i} = 1
while \textit{i} <= \textit{N}:
    # execute indented block
    \textit{i} += 1
for \textit{i} in \textit{range}(1, \textit{N} + 1):
    # execute indented block (this and while loop are the same)

2. Variables

Variables can be instantiated using = as shown in the translation from pseudocode to python below:

Pseudocode:
(I may comment my pseudocode however I like)
Set \textit{var} to \textit{val}

Python:
# Comments are indicated with a pound sign. They continue for the rest of the current line.
\textit{var} = \textit{val}

In python, there are three basic data types for variables: \textit{int}, \textit{float}, and \textit{string}. \textit{int} represents an integer value, so 1, 0, 5, and −12 are all \textit{int} values. \textit{float} represents a decimal value, so 3.2, −1.3, 2.0, and 0.0 are all of type \textit{float}. Note that in python the values 1 and 1.0 are treated differently since they are of different types. The data type \textit{string} is used for what are called string literals, which are just sequences of characters. “\textit{Hello}”, “\textit{Bob}”, and “\textit{world}!” are examples of strings. They are called literals because if you say \textit{print(myStringVal)} where \textit{myStringVal} = “\textit{Hello, World}!” it will print out all characters in “\textit{Hello, World}!” without trying to treat any words inside as other variables.

Typecasting is converting a value of one type to another by calling the functions \textit{int()}, \textit{float()}, or \textit{str()}. So 5 == \textit{int}(5.3), 5.0 == \textit{float}(5), 3 == \textit{int} (“3”), and “52” == \textit{str}(52). This is useful when
getting input from a user since the function `input(“prompt”)` always returns type `string`. So if you want an input of type `int` or `float`, you need to use typecasting to change the `string` to the desired type.

3. Conditionals

Like in pseudocode, conditionals are blocks of code that are executed at most once if a given condition is satisfied. We use conditionals with the keywords `if` and `else`. The following translation shows the difference between pseudocode and python for conditionals.

Pseudocode:

```
if conditon1 is true, then
    execute this block
else if conditon2 is true, then
    execute this other block
else, then
    execute the final block
```

Python:

```
if conditon1:
    # execute this block
elif conditon2:
    # execute this other block
else:
    # execute the final block
```

The format in python is very similar to what we did in pseudocode. Now, instead of saying `then`, we use a colon, `:`, to indicate we want to start a new indented block. Forgetting the colon is a very common bug that will be a syntactic error in python. Like in pseudocode, not every `if` statement needs to be followed by an `else` or an `elif` statement. Although this is helpful when you want to concisely state a string of conditions where you always want exactly one condition to hold (like for the pizza example).

**Conditions** are expressions that evaluate to `true` or `false`. A very common condition is checking for the equality between to variables. Since we used `=` for assignment, we cannot use the same symbol for equality. Thus, in python we use `==` to test equality in a condition. Other symbols used are `! =` for inequality, `< =` for less than or equal to, `> =` for greater than or equal to, `>`, and `<`. We can also use **logical connectives** like `and`, `or`, and `not`, which are keywords in python syntax, to create more complex conditions in python. So `(cond1 and cond2)` is true if and only if both `cond1` is true and `cond2` is true. Similarly, `(cond1 or cond2)` is true if and only if either `cond1` is true or `cond2` is true. Lastly, `not cond1` is true if and only if `cond1` is not true, i.e. `cond1` is false. Conditions, or boolean expressions, have a deep theory that is studied called propositional logic that can take entire semester long courses to fully explain.

Examples of conditions in python:

```
if a > b:
    # execute this block
elif a == b:
    # execute this other block
else:
    # execute the final block
```
if \( a \geq 1 \) and \( a \leq 5 \):
    # execute this block
if not \((a \neq b \text{ or } a \neq c)\):
    # execute this block

4. Iteration

Just like in pseudocode, we used for loops and while loops, in python we have the same constructs. The following code shows how for loops and while loops can be translated to python:

Pseudocode:

for each number, \( CN \), in the list of numbers
    execute this block, where \( CN \) is the current number
while \( condition_1 \) is true,
    execute indented block

Python:

for \( CN \) in \( numList \):
    # execute this block, where \( CN \) is the current number
while \( condition_1 \):
    # execute indented block

Notice that we uses colons, :, when indicating an indented block with loops just as we did for conditionals. This again will be another source of bugs in the python code that you will write. The for loop in python is the same idea as in pseudocode. Every time the block of code repeats, the variable specified will implicitly be set to the next value in the list until there is no value for the variable to be set to. For while loops, the format is nearly identical to the pseudocode.

We can use either for loops or while loops to iterate over a range of indices. The translation is as follows:

Pseudocode:

Set \( i \) to 1
while \( i \) is less than or equal to \( N \),
    execute indented block
    increment \( i \)
for \( i = 1 \) to \( N \):
    # execute indented block

Python:

\( i = 1 \)
while \( i \leq N \):
    # execute indented block
    \( i+ = 1 \)
for \( i \) in \( range(1, N + 1) \):
    # execute indented block (this and while loop are the same)

In the while loop, instead of saying increment \( i \), I instead use the assignment \( i+ = 1 \). \( i+ = n \) is interpreted by python as \( i = i + n \), which is the same as saying increment \( i \) by \( n \) in pseudocode. I
also use the built-in python function `range()` in the for loop. `range(low, high)` creates a list of values 
\[low, ..., high - 1\], so it includes the lower term but does not include the highest term. `range(high)`
is the same as `range(0, high)`. We will talk about why this is in more detail later. The basic reason
is that lists in python are indexed starting with 0 instead of 1. After the list has been created with
the `range()` function, the for loop works exactly as described before. This is a convenience instead of
writing a while loop for going over a range of indices.

5. Input

We did not really talk about how you got your input in pseudocode. We generally assumed that the
algorithm had certain values that are provided rather than asking the user for input values. When
we write functions in the future, this will mostly be the case. However, until then, we will get out
input from the user or a file mostly. I will give a short example of how to get input from a user in the
following python code:

```python
myStringVar = input("prompt")
myIntVar = int(input("prompt"))
myFloatVar = float(input("prompt"))
```

In each example, the code will print the prompt to the user of the program. It will wait for the user to
type something (or possibly nothing) and then hit the enter key. It will then return the string literal
before the enter key was pressed to the user. The only difference for types `int` or `float` is that you
are typecasting the resulting string into your desired type. We will go into more detail about reading
input from a file in weeks to come.

6. Output

In pseudocode, we used the words `output` and `print` interchangeably. In python, there are more specific
uses for two specific keywords, `return` and `print`, that correspond, respectively. I will briefly talk about
`return` since we will learn more about it later when we talk about functions. `return` is used inside
of function to specify the value you want to communicate with the program that called the function.
`return` also completely stops the execution of the function, so if we want to communicate multiple
values, we may have to put them in a list or other data type.

`print` differs from return in that it can be used any time in any piece of your code. `print` specifies a
built-in python function that takes in multiple arguments for what to print and how to print them.
Since `print` is a function, it follows the following format used to specify arguments inside of parentheses:

```python
name = "Cody"
print("My name is", name)
```

The way python interprets this function is that it looks at each argument one at a time. If the argument
is of type `string`, the `print` function displays that string to the user (the standard output) or possibly
a file if specified. If the argument is of type `int` or `float`, the function first converts the argument to
type `string` by typecasting it, then it displays it to the user. If the argument is a variable, it looks
up the value associated with the variable and displays that value as a string. The `print` function does
this for each (standard) argument given, printing a specified value `sep` (for separation) in between each
argument. After all arguments have been printed, a special value `end` is printed out. By default, `sep`
is just a space, and `end` is a new line character. As a result, the string printed out from this example
is, “My name is Cody” followed by a new line character. However, we can change these values to be
whatever we like to format what we want to print out. The following is an example of this:

```python
name = "Cody"
print("My name is", name, sep = ", ", end = "!")
```

Now this function prints the following message: “My name is, Cody!” with no new line character at
the end. The `print` function gives us power to specify exactly how we want our computer to display
certain values.
7. Strings

Doing operations on numbers is fairly straightforward. However, it is not well known what we often mean by adding two strings or multiplying a string by anything. We also have been using double quotation marks around text to specify that it is of type string, but this creates confusion if we want to use quotation marks inside of a string. We explore how python 3.4 answers these questions.

Adding two or more strings together in python (or most programming languages) performs what is called concatenation. **Concatenation** is a big word that just means appending or putting strings together. The following code exhibits string concatenation:

```python
myString = "Hello" + "World"
print(myString)
```

This prints out "HelloWorld", formed by putting the strings together to make a new string. Just like for incrementing a number by a value, we can append a string or variable of type string to a current variable of type string with the += operator. Again var1 += var2 is the same as var1 = var1 + var2, where var2 can be a variable or a string value.

```python
myString = "Hello"
myString += "World"
print(myString)
```

This prints out the same string as the previous example. Multiplication of a string type by an int is defined in python to be repeated addition. So the following code displays two different ways to print the same value, given that myString1 and myString2 start off as the same string value:

```python
for i in range(k - 1): # repeat (k - 1) times since there is already one copy
    myString1 += myString1
print(myString1)
myString2 * = k # the same as myString2 = myString2 * k
print(myString2)
```

Now we visit the question of printing something with quotation marks. There are two solutions python provides for doing this. The first, you can escape the quotation marks inside the string by using a backslash to tell python that you want to print out the quotation marks exactly as they are rather than thinking of them as part of the code. The second way is to use single quotes to specify your string to reduce confusion. Python treats single quotes and double quotes almost identically. Personally, I find it confusing to use double quotes some times and single quotes other times, so I usually stick to using escape characters to print quotation marks. The following code snippet exhibits these two methods:

```python
print("Shesaid,\"Hello!\"")
print(′Shesaid,"Hello!"′)
```

We can do a similar trick as the second solution if we want to have single quote inside of double quotes. However, if we want to use both double and single quote inside of a string, we must use backslash to escape either the single or double quotation marks. To reduce confusion, again I suggest you just always get in the habit of using a backslash to escape the characters if you want to use quotation marks inside of a string. I will now talk a little more about escape characters in general.

**Escape characters** are signified by a backslash (\) followed by another character. Escape characters are used for two reasons. Either we want to represent a character that cannot be represented on a standard keyboard, or we want to represent a character that is already special by itself. Examples of the first kind include: tab (\t), new line (\n), and a bell (\a). The tab will advance the current line to the next tabbed area on the line. This is helpful if you want to format your output in some sort of
tabular form. The new line character is important since you likely will want to print things on multiple lines. We have already mentioned that implicitly, the `print()` function prints out a new line character after printing all the arguments. The bell is pretty pointless, but it causes your computer to make a sound. Examples of when you would see escape characters to escape a special characters include: single or double quotation marks (`', `"`) or a backslash (`\`). We have already seen why you would escape quotation marks. The backslash is just since we need something to signify escape characters, so that character can no be used on its own. So now to print a backslash, you must type two backslashes. You can test all of these out in Idle. Here is how you may use these characters:

```python
print(1, 2, 3, 4, 5, sep = "\t") # prints tabs between each number
print(1, 2, 3, 4, 5, sep = "\n") # prints new lines between each number
print("What is your favorite color?", end = "\n") # two new lines at end
print("What is your favorite color?\n") # same as previous line
print("\\\") # prints three backslashes, and odd number would cause an error
```

We will now move the discussion to the other basic data types, Numerics.

8. Numerics

I will briefly discuss how we can do various operations on variables of type `int` and `float`. Addition (+), subtraction (−), and multiplication (∗) work mostly how you would expect. For these operations (⋆, in general), if your two operands, a and b, are type `int`, then a ⋆ b is also of type `int`. If either operand is of type `float`, then a ⋆ b becomes the type `float`. The following code exhibits these equivalences:

```python
3 + 4 == 7
3 + 4.0 == 7.0
3 − 4 == −1
3.0 − 4 == −1.0
3 * 4 == 12
3.0 * 4.0 == 12.0
```

Division is treated slightly differently in Python 3. A single division sign (/) is standard division. No matter the type of a or b, a / b is always of type `float`. If you want to perform integer division, i.e. divide two integers and get an integer back, you must use the double division sign (//). So a // b is type `int` if a and b are also of type `int`. Integer division is technically allowed with floating point operands, but the behavior is not clearly defined. I would not try to use this at all. If we were to represent a fraction a / b as a mixed number, we would get some equivalent value c + r / b, where 0 ≤ r < b. Thus, integer division tells us what c is in the mixed number representation. Sometimes, it is also important to compute the remainder term r. The modulus operator (%) allows us to compute remainders. So if a and b are integers, a % b is some value between 0 and b − 1 (inclusive), representing the r term for a / b. I illustrate these concepts with the following code:

```python
3 / 4 == 0.75
3 // 4 == 0
3 % 4 == 3
7 / 3 == 2.333...
7 // 3 == 2
7 % 3 == 1
(−5) / 5 == −1.0
```
Other mathematical operations built into python 3 that you might want to use include: negation, absolute values, and powers. Negation is very straightforward. $-\text{var}$ has the value of the negation of var. Absolute values can be computed with the abs() function. So abs(var) returns the absolute value of the variable var. Powers can be computed in two ways. pow($x, y$) and $x \ast \ast y$ both compute $x^y$ in python 3. Here are some examples:

\[
\begin{align*}
x &= 5 \\
-x &= -5 \\
3 &= \text{abs}(-3) \\
3.4 &= \text{abs}(-3.4) \\
2 &= \text{abs}(2) \\
9 &= 3 \ast \ast 2 \\
9 &= \text{pow}(3, 2) \\
2.0 &= 4 \ast \ast 0.5 \\
2 &= \text{int}(4 \ast \ast 0.5) \\
1.414213... &= 2 \ast \ast 0.5
\end{align*}
\]

These string and numeric operations pretty much sum up the basics for manipulating basic data types (int, float, string). We will next discuss a complex data type (or data structure), list, which is much less intuitive but gives us much more power.

9. Lists

Lists are a complex data type also known as a data structure. They are under the class of data structures known as containers. What this means is that the list type is used to hold other types, including lists. Other data structures in python include tuples, sets, and dictionaries, but we will likely not talk about these in this course. I will first talk about the basics of lists and how to use them. Then I will talk about some more advanced operations you can perform on lists.

a) Basics

I will just jump into the code and explain line by line after:

```
1. myList = []
2. myList.append(5)
3. myList.append(9)
4. myList.append(-1)
5. print(myList)
```

The previous code prints out the following: [5, 9, -1]. You can see that this resembles a list with each of the numbers we appended to the list. In python, we use the square brackets ([ ]) to denote a list. If we want to use a list data structure, we must first instantiate our list as I did in step 1. You can also instantiate a list with values already in it using the syntax like what was printed out. This tells python, the variable myList is going to be used as type list. This allows us to call functions on the list like in steps 2-4. I have alluded to functions many times without explicitly defining them, and I will probably keep doing this until we define our own functions. However, it is important to know that if you want to perform an operation on a list to modify it, you will call a function or use some other python syntax to do so. In step 2, append is a function be called on myList with a single argument 5. This says, I want to change myList such that the last value in the list is a 5. This is repeated again in steps 3 and 4 to add multiple items to the list. append() is probably the most common function we will call on a list. Lastly, in step 5, we print out the list.
This implicitly calls `str(myList)` and prints that result to the console. The list data structure comes with a built-in way of converting it to a string to be read by the user. The format is what we saw for the output. I will now continue the previous example in the following code:

6. `print(myList[0])`
7. `print(myList[1])`
8. `myList[0] = 2`
9. `print(myList[0])`
10. `print(myList)`

Let’s go through what this code does step by step. The brackets are used to access specific elements in the list. In step 6, 5 is printed to the console. This is the first weird thing you will notice about python. For some reason, the programming gods have decided that lists shall be indexed beginning with 0 instead of 1. The most intuitive way to explain this is you want to find how far the item is from the first element. So the value 5 happens to be 0 elements away from the first element in the list (since it is the first element in the list), so `myList[0]` is the value 5. Then, following this (rather annoying) format, step 7 prints out a 9 to the console. This is because 9 is the second element in the list (or 1 element away from the first element in the list).

Then some magic happens in step 8. We now modify our list by assigning the value 2 to the first element in the list. We use the same syntax with the brackets to access a specific element when modifying the list as well. Now, step 9 prints out a 2 to the console instead of a 5. Step 10 then prints out the following representation of `myList`: [2, 9, -1]. We see that our list is now different than it was before we made the modification in step 8. Because we can make this modification, we say that the type list is mutable, i.e. it can be changed (as opposed to immutable, which we may see later).

I want to go back to the concept of the for loop now that we understand the basics of a list. Say I want to print out the squares of each number in my list from the above example. I can do this in two ways as follows:

11. for `num` in `myList`:
11.1 `square = num ** 2`
11.2 `print(square, end = " ")`
12. `print("\n")`
13. for `i` in `range(len(myList))`:
13.1 `myList[i] = myList[i] ** 2`
13.2 `print(myList[i], end = " ")`

In the first way, I use the typical syntax for the for loop. `num` is updated to the current number in the list each iteration. I compute the square, then I print the value (with a space instead of a new line at the end). After step 11 has completed, the following will be the output: 4, 81, 1. This is the square of each number in the list, so this is what we wanted. (Line 12 just prints a new line character as to not confuse the outputs from steps 11 and 13).

Step 13 now uses the trick we learned to go through the index values from 0 up to the length of `myList` – 1. Python defines the `len()` function on lists to return the number of items in the list. Thus, at every iteration of the loop, `myList[i]` is the same as `num` was in the previous for loop. `myList[i]` is just the current number in the list. However, with this structure, we can do more than just know what each number is. Here I actually modify the list such that the list now contains the squares of each number. The output is the same, but now `myList` has changed from [2, 9, -1] to [4, 81, 1]. Using the method with indices can give us more power since we now know what position each current number is at. Then we can look at other numbers relative to the current number or modify that current position in the list.

That covers the basics that you will need to know about lists for this class. I will introduce some more advanced things that you can do with lists in the next section.
b) Advanced

One advanced operation you can do with lists is splice lists. Splicing is somewhat fancy way of saying cut up into its smaller pieces. The syntax can be seen as follows:

1. `myList = [6, 2, 3, 9, 2]`
2. `print(myList[1 : 3])`
3. `print(myList[2 :])`
4. `print(myList[: 2])`

In the first line, I instantiate my list with the values shown. The second line prints out the sublist from element 1 (the second element in the list) up to before element 3 (so it does not include the element at the second defined position). The results is [2, 3]. In the next line, it prints out the sublist starting at element 2 and continuing until the end of the list. This prints out [3, 9, 2] (since element 2 is the third element in the list). The last line prints the sublist starting at the beginning and going up to before element 2. The result is [6, 2]. This is useful if you only want to look at a subsection of the list.

I can also access elements in a list starting from the end of the list. In this way, the element at position −1 is the last element on the list. Then the element in position −2 is the element preceding it on the list. The pattern continues. However, if you go too far positive or negative, you will get an out of index error from python, so be careful!

I will briefly mention the other important functions defined on a list. `myList.extend(myOtherList)` will append every item in `myOtherList` to `myList`. `myList.sort()` (uses no arguments, but still requires the parentheses since it is a function) sorts all elements in the list. You can use this function as long as what we are telling you to do is actually sort the list yourself. `myList.reverse()` (which also has no arguments) reverses the elements in the list, so the last becomes the first element, etc. `myList.copy()` creates a copy of `myList`. This is equivalent to creating a new list and extending it by the current list. `myList.clear()` removes all items from the list. These function are helpful but not necessary. I will give examples of all of these in the following code (the comments show the output):

```python
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers[-1]) # 9
print(numbers[-2]) # 8
numbers.extend([10, 11])
print(numbers) # [0,1,2,3,4,5,6,7,8,9,10,11] (changes the list)
numbers.sort()
print(numbers) # [0,1,2,3,4,5,6,7,8,9,10,11] (list already sorted)
numbers.reverse()
print(numbers) # [11,10,9,8,7,6,5,4,3,2,1,0]
a = numbers.copy()
print(a) # [11,10,9,8,7,6,5,4,3,2,1,0]
numbers.clear()
print(numbers) # [] (the list is now empty)
```

10. File I/O

In this section we discuss reading to and writing from a file. In general, this is called I/O for Input and Output. Up until now, all of our input has been from what is called Standard Input and Standard Output. This form of I/O is helpful when you directly want to interact with a user. You may ask them to enter data and you will tell them something about that data immediately. However, sometimes we want to work with larger data sets that we might get from an outside source. The easiest way to do this is to read data from a file.
A file is its own complex data type in python. To instantiate a file we use the `open` function. Open takes in two arguments. First, the file name is provided so python knows what file to use in your filesystem. By default, python will look in the same directory for the file as the .py file you are working in. You can also give an absolute or relative path if you want to look for a file in a different directory, but we will not concern ourselves with that. The second argument to the `open` function is a flag specifying whether the file is meant to be read from or written to. Not surprisingly, "r" corresponds to read, and "w" corresponds to write. You also can specify the flag to be "a", which stands for append, to open an existing file and write to the end of the file. This flag determines what operations you can call on the file you are creating.

```python
infile = open("my_input_file.txt","r")
outfile = open("new_output_file.txt","w")
afile = open("old_output_file.txt","a")
```

Once we have opened a file, we can read and write to it. I will start by talking about reading from the file. To read from a file we can use a few different methods. `infile.readline()` reads and returns the current line in the file, and then advances what the current line is. In this way, the current line is a pointer into the file that says where in the file we are currently reading from. Thus, if we call `infile.readLine()` twice in a row, instead of reading the same line twice, it will read two consecutive lines in the file. Alternatively, we can call `infile.readlines()`, which will read the rest of the file from the current line and return a list of all lines. `infile.readlines()` essentially calls `infile.readline()` repeatedly and stores the lines into a list. Thus, after `readlines` is called, you cannot read from the file again without reopening the file into a new file object.

It is also important to note that when reading from a file, the text is read exactly as it is. The most obvious example of this is that at the end of each line there will be a new line character since that is what is actually stored in the file. Usually, we do not care about extra new line or white space in a file, so we can call `line.rstrip()`, where line is any string type, to remove leading an ending white space including tabs, new lines, and spaces. This is important if you are going to do any comparisons on the lines read in from the file. The following example shows what `infile.readlines()` does by using `infile.readline()`, although I will always remove whitespace from the lines. Note that I know the file has finished when the value returned from `readline()` is essentially false, meaning the current pointer in the file is the end of the file.

```python
allLines = []
current = infile.readline().rstrip()
while current:
    allLines.append(current)
    current = infile.readline().rstrip()
```

When writing to a file, if we are in `w` mode, a new file will be created or a current file will be replaced. In `a` mode, a current file will be used, and anything written will be added to the end of the file. Actually writing to the file is done via the `outfile.write(arg)` function. This writes the string, `arg`, to the file in the specified mode. Unlike `print`, `write` takes only a single string value as an argument. So if we want to write multiple things, we must either call `write` multiple times or build up a string with multiple values. It also does not add a new line or any spaces to the string. So if you want to print a new line to the file, you must explicitly include the new line character at the end of your string. By default, `write` always writes to the end of the current file object. This can be modified, but we will not consider this for this class. The following is an example of how write could be used:

```python
outfile.write("Hello, World!\n")
for i in range(10):
    afile.write(str(i) +"\t")
```