CptS 111 — Lab #9

Nested Loops & Lists, Files, and `with`

Learning Objectives:

- Code nested loops and lists
- Open, read, write or print, and close files
- Use the `with` command

Prerequisites:

- Basics of nested loops and lists
- File basics
- Exposure to `with`

Task 1: Nested lists (AKA lists of lists) and nested loops

As mentioned in class, lists can contain any kind of elements including other lists. Nested lists, or lists of lists, can be very useful and are often used in conjunction with nested `for`-loops. **Study the examples below because you'll need to understand them for Task 3!**

```
>>> # Let's start with indexing lists of lists.  Recall that a single
>>> # list is indexed using two square brackets.
>>> coins = ['quarters', 'dimes', 'nickels', 'pennies']
>>> coins[0]
'quarters'
>>> len(coins)
4
>>>
>>> # Next let's consider a list of lists (an array or table, this one
>>> # with four rows and two columns, i.e, a 4x2 array):
>>> coins = [['quarters', 0], ['dimes', 0], ['nickels', 0], ['pennies', 0]]
>>>
>>> # How long is this list?  Let's check?
>>> len(coins)
4
>>>
>>> # It's the same length as before, but now each element is a list which
>>> # we index as before:
>>> coins[0]
['quarters', 0]
>>>
>>> # However, if we want to index an element within an inner list, we
>>> # must use double indexing, the first index for the outer list and
>>> # the second index for the inner list:
>>> coins[0][0]  # Make sure you understand this!
'quarters'
>>> coins[0][1]  # Make sure you understand this!
```

```
0
>>>
>>> # Next, let's figure out how to print everything in this list.  This
>>> # requires the use of nested for-loops.
>>> for i in range(len(coins)):          # i = row loop variable
        print(f'coins[{i}]: {coins[i]}')
        for j in range(len(coins[i])):   # j = col loop variable
            print(f'\tcoins[{i}][{j}]: {coins[i][j]}')
>>>
>>> # The outer loop handles the rows, and the inner loop handles the
>>> # columns.  Examine the output and the code until you understand
>>> # how nested counting for-loops work with a 2D array.
>>>
>>> # Next let's see how we can change the second element of each inner
>>> # list to represent the number of coins we have.
>>> coins
[['quarters', 0], ['dimes', 0], ['nickels', 0], ['pennies', 0]]
>>>
>>> for i in range(len(coins)):
        num = int(input(f'Enter number of {coins[i][0]}: '))
        coins[i][1] = num  # Note the indexing!

Enter number of quarters: 25   # quarters: coins[0][0]
Enter number of dimes: 14      # dimes:    coins[1][0]
Enter number of nickels: 18    # nickels:  coins[2][0]
Enter number of pennies: 132   # pennies:  coins[3][0]
>>>
>>> coins
[['quarters', 25], ['dimes', 14], ['nickels', 18], ['pennies', 132]]
>>>
>>> # We see from the example above that we can change the number of coins
>>> # in each inner list using a for-loop and the correct indexing.  Be
>>> # sure to look at the string formatting used to create the prompt!
>>>
>>> # Next let's come up with a way to total the amount of money we have.
>>> tot_cents = 0
>>> tot_cents += coins[0][1] * 25  # 25 cents in a quarter
>>> tot_cents += coins[1][1] * 10  # 10 cents in a dime
>>> tot_cents += coins[2][1] * 5   # 5 cents in a nickel
>>> tot_cents += coins[3][1]       # 1 cent in a penny
>>> dollars, cents = divmod(tot_cents, 100)
>>> print(f'I have a total of ${dollars}.{cents} in my piggybank!')
>>>
>>> # Finally, let's come up with a report on all our wealth using an
>>> # iterating for-loop!
>>> print('*** Piggybank Report ***')
*** Piggybank Report ***
>>> for coin, num in coins:  # Understand this for Task 3!
        print(f''{coin+':':9}{num:>4}'')
```

```
quarters:  25
dimes:     14
nickels:   18
pennies:  132

>>> print(f'I have a total of ${dollars}.{cents} in my piggybank!')
>>>
>>> # Of course our report would look better if we had coded a program
>>> # and not used an IDLE Shell window, but we can see that if the
>>> # commands weren't between the output, it would look okay.
>>>
>>> # Note the header of the iterating for-loop because you'll need
>>> # to know this for Task 3!  Each element in the coins list is itself
>>> # a list.  In fact, it's a two-element list with the first element
>>> # the type of coin and the second element the number of that type
>>> # of coin.  As it iterates through the list, the for-loop takes each
>>> # list element in the outer list and assigns coin to the first
>>> # element in the inner list and num to the second element in the
>>> # inner list.
>>>
>>> # Let's try one more thing before moving on to the next task.
>>> # Suppose we have a 3 x 4 table of characters.  Recall that the
>>> # first value 3 gives us the number of rows and the second value 4
>>> # gives us the number of columns.  Thus, our table has 3 rows and
>>> # 4 columns:
>>> table = [
            ['a', 'b', 'c', 'd'],
            ['e', 'f', 'g', 'h'],
            ['i', 'j', 'k', 'l']
            ]
>>> table
>>>
>>> # How can we print this list of lists so it will look like a table?
>>> # We use nested, iterating for-loops!
>>> for row in table:
        for col in row:
            print(col, end=' ')
        print()
>>>
>>> # In the outer loop, we consider a row in the table.  In the inner loop,
>>> # we consider a column in the row.  We loop through all the columns in
>>> # a row, ending with a space so that letters don't appear on different
>>> # lines.  After we've looped through the columns in the row, we print a
>>> # blank line so the next row of letters is on a new line.  Then we
>>> # return to the outer loop and consider the next row in the table, and
>>> # in the inner loop, we consider a column in this row.  We continue
>>> # until we've considered each row followed by all the columns in each
>>> # row, and we're done!
```

After you've completed this task, show your work to your TA to get credit. Keep your IDLE session open for the next task.

Task 2: Opening, reading from and writing to, and closing files (and also `with` and the `os` module)

Using the `input()` and `print()` functions to read input from stdin and to write output to stdout is very limiting. Often we want to access data contained in files or write results to a file. In this task, we're going to practice working with files. To do this task, you need to download the two text files from our class website under the Labs link into the same folder/directory as your IDLE session.

Of course you can check your hard disk to ensure your files are all in the same folder, also called a directory, as you may have for PA 4 or PA 5. However, you can also do this by importing the `os` module which is the operating system module. Recall,

```
>>> import os
>>> os.getcwd()  # Tells you the current working directory (cwd)
>>> os.chdir()   # Use to change to another directory; need to include
>>>              # entire directory name in quotes (ask your TA)
>>> os.listdir() # This will print all the files in the directory
```

You don't have to use this now, but you may if you'd like.

```
>>> # Let's start by opening a file and then reading the first line of
>>> # the file using the .readline() method which reads a single line.
>>> f_in = open('genome.fna')
>>> line1 = f_in.readline()
>>> line1
>>>
>>> # What data type does .readline() result it?
>>> type(line1)
>>>
>>> # The first line of a genome, gene, or protein file is the header;
>>> # it gives information about the rest of the file.  This header tells
>>> # us that the file contains the complete sequence (genome) of a
>>> # plasmid (no need to know what a plasmid is) from the bacterium
>>> # Yersinia pestis which causes bubonic plague.  This disease wiped
>>> # out half the European population in the 14th century, and Y. pestis
>>> # is still alive and well and causing bubonic plague.  Fortunately,
>>> # if bubonic plague is caught early enough, it can be successfully
>>> # treated with antibiotics.
>>>
>>> # Next let's read the genome using the .read() method (which reads
>>> # an entire file as a single string).
>>> genome = f_in.read()
>>> type(genome)
>>>
```

```
>>> # We don't want to print the genome because it's very long!  How long?
>>> len(genome)
9611
>>>
>>> # So you can see why we didn't want to print it.  That's a lot of
>>> # characters!  Genomes are composed of DNA, and there are just four
>>> # different types of DNA abbreviated by A, T, G, and C.  Let's see
>>> # what percentage of the plasmid's genome is made up of G and C
>>> # which is often of interest to life scientists.
>>> dna_g = genome.lower().count('g')
>>> dna_g
>>> dna_c = genome.lower().count('c')
>>> dna_c
>>> percent_gc = (dna_g + dna_c) / len(genome) * 100
>>> print(f'The plasmid of Y. pestis is {percent_gc:.2f}% GC.')
The plasmid of Y. pestis is 45.27% GC.
>>>
>>> # We used the .lower() method above to ensure our count is correct
>>> # regardless of the case of the letters
>>>
>>> # It's necessary to close output files, but it's good practice to close
>>> # all files.  Also, you can't reread an input file; you must close
>>> # it and then reopen it.
>>> f_in.close()
>>>
>>> # Let's now move on to a different way of reading the contents of a file.
>>> f_in = open('poem.txt')
>>> for line in f_in:          # Note what we've done here!
        print(line, end='')
>>>
>>> # We see that we can use an input file as an iterable!  This is useful
>>> # if we know that we have distinct lines in a file, and we want to
>>> # operate on them in some way.  The reason we used end='' is because
>>> # the file itself has newline characters to indicate new lines.
>>>
>>> # Let's close the input file and consider yet another way of reading a
>>> # file.
>>> f_in.close()
>>>
>>> with open('poem.txt') as f_in:
        poem = f_in.readlines()
>>> poem  # .readlines() results in a list of string elements
>>>
>>> # In this example, we have used the .readlines() method which reads
>>> # in each line as an element in a list!  The with command is useful
>>> # because it closes the file after all the statements in its body
>>> # have been run.  You can also use with to write to a file.  In
>>> # fact, let's give this a try!
>>> with open('new_poem.txt', 'w') as f_out:
```

```
        for item in poem:
            print(item, end='', file=f_out)

>>> # We've had to use a for-loop in the example above because we need
>>> # to write each element of the list, poem, to the output file.
>>> # Again, we use end='' so we don't end up with blank lines between
>>> # lines of text.  Look for the file 'new_poem.txt' in your
>>> # directory/folder and see what's inside.  As you should see, your
>>> # file has the contents you wrote to it.
```

After you've completed this task, show your work to your TA to get credit.

Task 3: Flower shop order

This task uses a lot of the commands and constructs demonstrated in Task 1 and the `with` command from Task 2. It requires coding four functions as described below in the order in which they should appear. The program first prompts a florist for the number of different types of flowers they want to order (see example below). It then calls a function to prompt the florist for the names of the different flowers which returns a list of lists. Next it passes this list to another function which prompts the florist for the numbers of the flowers to order, and finally it passes this list to another function which opens a file and prints out the order. An example of an output file is shown below. Save your program as `lab9_t3.py` in your Lab9 folder under CS111.

Before beginning, please recall the difference between a counting `for`-loop and an iterating `for`-loop as shown below. We also used counting and interating `for`-loops in Task 1, and it might help you to go back and look at the loops we used there.

Suppose we have a list called `nums`. If we want to use a counting `for`-loop to loop through all the elements in this list, we would use:

```
for i in range(len(nums)):
    <loop_body that uses nums[i] for each element in nums>
```

Note the use of the `range()` function in the counting `for`-loop. For an iterating `for`-loop, we would use a header with an iterable!

```
for num in nums:
    <loop_body that uses num for each element in nums>
```

Note the use of the list iterable `num` in the iterating `for`-loop. For the following functions, code `main()` first, and develop your program incrementally using the function stub `pass`. Also, don't forget to call `main`!

- **`get_flowers()`**: This non-void function has a single integer parameter, the number of different types of flowers. It initializes an empty list and then uses a counting `for`-loop to prompt the florist for the name of the flower (with color) as shown in the example below. It appends a list with the flower name as the first element and `0` as the second, e.g., `[name, 0]`, to the list that was initialized, and returns this list of lists to `main()`.
- **`get_nums()`**: This void function has one parameter, the list of lists generated in `get_flowers()`. It uses a counting `for`-loop to prompt the florist for the integer number of each type of flower

they want to order; string formatting is needed for the prompt (see example). The integer is used to update the list of lists from `0` flowers to the number of flowers entered by the florist. Note that the reason you don't have to return the list you've changed is because lists are mutable and because when you pass the name of the list to `get_nums()`, the parameter points to the same memory location as the argument in the function call.

- **`order()`**: This void function has one parameter, the updated list of lists. It uses the `with` command to open an output file called `flower_order.txt`. In the body of `with`, it prints `*** Flower Order ***` and then a blank line to the output file. Finally, it uses an iterating `for`-loop with two variables (see Task 1) to print the flower name and number of each flower to the output file (see example output file below).
- **`main()`**: This void function prompts the florist for the integer number of different types of flowers they want to order, prints a blank line, and calls the non-void function `get_flowers()` with the number of flower types as its argument. Choose an lvalue to which to assign the list of lists returned by `get_flowers()`. Next, `main()` should print another blank line, call `get_nums()` with the list of lists as the argument, and finally call `order()`, again with the list of lists as the argument.

Example of prompts and input.

```
Enter number of flower types: 5

Enter flower name [with color]: pink roses
Enter flower name [with color]: white carnations
Enter flower name [with color]: white stargazer lilies
Enter flower name [with color]: pink alstromeria
Enter flower name [with color]: lavender tulips

Enter number of pink roses: 12
Enter number of white carnations: 24
Enter number of white stargazer lilies: 12
Enter number of pink alstromeria: 36
Enter number of lavender tulips: 24
```

Example of output file `flower_order.txt`.

```
*** Flower Order ***

pink roses:  12
white carnations:  24
white stargazer lilies:  12
pink alstromeria:  36
lavender tulips:  24
```

When your program is working properly, demonstrate it to your TA to get credit, and show them your `flower_order.txt` file which should be in your current directory/folder.

As with last week, there are only three tasks in this week's lab because all three tasks took considerable concentration. You can now look back at Lab #1 and see how far you've come and how much you've learned to do!