CptS 111 — Lab #11

The String Methods `.split()` and `.join()`, Slicing, and More on Lists

Learning Objectives:

- Gain some experience with string and list slicing
- Use the string methods `.split()` and `.join()` to do some useful things
- Learn when a counting `for`-loop is required to change a list

Prerequisites:

- Exposure to the `.split()` the `.join()` string methods
- Exposure to string and list slicing
- Ability to code iterating and counting `for`-loops

Task 1: Practicing with `.split()` and `.join`

Recall that the `.split()` method splits a string into a list and the `.join()` method joins a list into a string. Let's practice a litle with these two very useful methods.

```
>>> # Let's start with a simple string.
>>> tongue_twister = 'Peter Piper picked a peck of pickled peppers'
>>> tt_list = tongue_twister.split()
>>> tt_list
>>>
>>> # When we use .split() with no argument, the string is split at each
>>> # space to create the elements of the resulting list.  Next try:
>>> tt_list.sort()
>>> tt_list
>>>
>>> # We sorted the list to create a new list.
>>> new_tongue_twister = ' '.join(tt_list)
>>> new_tongue_twister
>>>
>>> # Then we used the .join() method to join the elements of the sorted
>>> # list into a new string!  Note that whatever we split on is removed.
>>> tt_list = tongue_twister.split(' p')
>>> tt_list
>>>
>>> # When we joined the previous list into a string, we used a blank
>>> # space between each element, but we can use anything we'd like.
>>> new_tongue_twister = ' * '.join(tt_list)
>>> new_tongue_twister
>>>
>>> # Okay, so perhaps what we just did wasn't very useful.  Let's
>>> # consider a more complex example.
>>> names = 'Lawrence, Jennifer; Hanks, Tom; Smith, Will; Streep, Meryl'
>>> name_list = names.split('; ')
>>> name_list
>>> name_list.sort()
>>> name_list
>>>
>>> # So first we created a string of names.  Then we split this string
>>> # on '; ' to create a list of names (note that we split on a semi-
>>> # colon AND a blank space).  Finally we alphabetized this list by
>>> # sorting it.  Now we want to switch first and last names which we
>>> # accomplish using a for-loop.
>>> for i in range(len(name_list)):
        name = name_list[i].split(', ')      # name is a list!
                                             # name_list[i] is a string
        name[0], name[1] = name[1], name[0]  # swap list elements!
        name_list[i] = ' '.join(name)        # replace original string

>>> name_list
>>> new_names = '; '.join(name_list)
>>> new_names
>>> names
>>>
>>> # What exactly did we do above?  Well, in the for-loop, we split
>>> # each string element of the list on ', ', and then we used
>>> # simultaneous assignment to switch first and last names.  Finally,
>>> # we overwrote the original string element with a new one.  After
>>> # we completed the for-loop, we created a new string using the
>>> # .join() method, and we compared the new string with the original
>>> # string.  ***TAKE SOME TIME GOING OVER THIS EXAMPLE TO ENSURE YOU
>>> # UNDERSTAND IT!***
>>>
>>> # Recall that we can use a for-loop to insert a character or
>>> # characters between words from a list to create a string, e.g.,
>>> words = ['stick', 'in', 'the', 'mud']
>>> string = ''
>>> for i in range(len(words) - 1):
        string += words[i] + '-'
>>> string += words[-1]
>>> string
>>>
>>> # We can use the .join() method to accomplish the same objective
>>> # much more tersely, i.e., with only one statement rather than four.
>>> string = '-'.join(words)
>>> string
>>>
>>> # As shown in class, another use of .split() and .join() is to
>>> # determine the length of a string.  This can be accomplished with
>>> # a single statement.
>>> line1 = 'mfvflvllpl vssqcvnltt rtqlppaytn sftrgvyypd kvfrssvlhs tqdlflpffs'
>>> len_line1 = len(''.join(line1.split()))
>>> len_line1
```

```
>>>
>>> # line1 gives the first of >21 lines of the protein sequence for
>>> # the SARS-CoV-2 spike protein, the protein that attaches to the
>>> # ACE2 receptor in a person and causes infection.  To determine how
>>> # many amino acids are in the first line, we first split the string
>>> # on the blank spaces to create a list.  We join the elements of
>>> # the list by empty strings to form a new string without blank spaces,
>>> # and finally we use the len() function to determine the number of
>>> # characters in the new string.
```

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: Odds and ends with strings and lists

In this task, we'll cover several different concepts including a little simultaneous assignment with lists, string and list slicing, and modifying lists in `for`-loops.

```
>>> # The simplest form of simultaneous assignment is when we assign
>>> # two values on the right to two lvalues on the left as we did in
>>> # our first task with first and last names.  However, we can also
>>> # use simultaneous assignment with lists.  Consider the following
>>> # example:
>>> core = [
    ['Yr 1', ['121', '122']],
    ['Yr 2', ['223', '260', '317', '322', '355']],
    ['Yr 3', ['302', '327', '350', '360']],
    ['Yr 4', ['421', '423']]
    ]
>>>
>>> # These are the 13 core courses that all computing students must
>>> # take regardless of their major (except for the last year, and
>>> # cybersecurity students have a choice between 355 and two other
>>> # courses).  We can use a for-loop to create a list of these:
    for cs_courses in core:
        yr, courses = cs_courses
        print(f'{yr} Core Courses:')
        for course in courses:
            print(f'   - CptS {course}')
>>>
>>> # Recall that for both string and list slicing, we use [ : ] to
>>> # indicate the range of elements we want to include.  Let's start
>>> # with a string and try a bunch of different slices.
>>> string1 = 'Sponteneity has its time and place.'
>>> string1[ : ]     # Default start=0 and stop=end
>>> string1[1 : -1]  # start=1, stop=one_before_end
>>> string1[4 : ]    # start=4, default stop=end
>>> string1[ : 1000] # Default start=0, stop=large_number
>>>
```

```
>>> # In the examples above, we note that [ : ] gives the entire string,
>>> # negative indexing can be used, and when stop > len(string), Python
>>> # ignores the actual number and stops at the end of the string.
>>>
>>> # Next, let's consider a list and try a bunch of different slices
>>> # with it.
>>> list1 = ['bill', 'dill', 'fill', 'gill', 'hill', 'kill', 'mill']
>>> list1[ : ]        # Default start=0 and stop=end
>>> list1[1 : -1]     # start=1, stop=one_before_end
>>> list1[4 : ]       # start=4, default stop=end
>>> list1[ : 1000]    # Default start=0, stop=large_number
>>>
>>> # So list slicing works exactly like string slicing.  There's
>>> # another important use for list slicing.  Type the following
>>> # commands, think about them, and note the results.
>>> list2 = [0, 1, 2, 3, 4, 5]
>>> list3 = list2
>>> list3
>>> list2[-1] = 500
>>> list2
>>> list3
>>>
>>> # We made a copy of list2 which we called list3.  Then we changed a
>>> # value in list2, but list3 was also changed.  This is because list3
>>> # wasn't really a copy of list2 but rather an alias for list2.  Both
>>> # list2 and list3 point to the same memory location.  So how do we
>>> # make a copy of a list?  Yep.  You guessed it.  We use list
>>> # slicing to make a real copy of a list.
>>> list2 = [0, 1, 2, 3, 4, 5]
>>> list3 = list2[ : ]
>>> list3
>>> list2[-1] = 500
>>> list2
>>> list3
>>>
>>> # We see in the series of commands above that by using slicing, we
>>> # can make what we call a deep copy of a list.
>>>
>>> # Note that we can combine list indexing and string slicing.
>>> list1            # If you killed this list, then retype it as
>>> list1[5][ : 3]   # given several examples above.
>>>
>>> # Try a few more of examples of indexing and slicing until you can
>>> # slice exactly what you want.  Note that if you have nested
>>> # lists, you can also combine indexing and list slicing.
>>> list4 = [[0, 1, 2], [3, 4, 5], [6, 7]]
>>> list4[1][1 : ]
>>>
>>> # For the final exercise in this task, let's consider changing the
>>> # elements of a list using a for-loop.
>>> tally = [0] * 10     # Easy way to create list of 0's
>>> tally
>>> for num in tally:
        num += 1
        print(num, end=' ')

>>> tally
>>>
>>> # What went wrong?  From the print statement we saw that 'num'
>>> # really was increased by one.  The problem is that 'num' is
>>> # just a loop variable in the iterative for-loop.  It's assigned
>>> # each value in the list 'tally' for each iteration, but the
>>> # list itself isn't changed.  This is a case when we're required
>>> # to use a counting for-loop (we actually discussed this when we
>>> # covered Ch. 6).
>>> for i in range(len(tally)):
        tally[i] += 1
        print(tally[i], end=' ')

>>> tally
>>>
>>> # By using a counting for-loop, we actually change each value in
>>> # the list itself!  We won't do an example like the one shown in
>>> # class involving comparing two lists of names, but you're
>>> # encouraged to review the Jupyter notes posted on our website to
>>> # ensure you understand what was done.
```

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

Task 3: Imitating a Unix command

A command available on Unix systems (including Macs and Linux machines) called `wc` returns the number of lines, words, and characters in a file. For example, `poem.txt`, which you used in Lab #9 (if you can't find it on your computer, please download it and make sure it's in your current working directory) contains the following text (excluding the numbers on the right which give the number of characters for that line):

```
One, two,           9
Buckle my shoe;    15
Three, four,       12
Shut the door;     14
Five, six,         10
Pick up sticks;    15
Seven, eight,      13
Lay them straight; 18
Nine, ten,         10
A big, fat hen.    15
```

When `wc` is called to analyze this file, it gives the following output:

```
sweetie> wc poem.txt
      10      26     141 poem.txt
```

The first line starts with the prompt `sweetie>` (`sweetie` is the name of the computer) followed by the command `wc` to run the program on the file `poem.txt`. The subsequent output in the second line shows there are 10 lines, 26 words, and 141 characters in the file where a word is any collection of characters bounded by whitespace. The number of printed characters and blank spaces, shown on the right for each line, totals 131, so why does `wc` report 141? It's because newlines (`'\n'`) are characters and, thus, are included in the total. The `wc` command is actually quite useful, e.g., we might want to know the total of number of lines in a program or in a data file.

Open an IDLE Editor window and save it as `lab11_t3.py` under your `CS111` folder in a folder called `Lab11`. Then write the following two functions which will give you a program that imitates the `wc` command in Unix. Don't forget your header, a call to `main()`, and function docstrings.

- **`wc()`**: This void function has one parameter, the name of the file input by the user. It should do what the `wc` Unix command does which is to print the number of lines, words, and characters in the file as well as the file name as shown in the example below. You'll need to open and read the file using the `.read()` method which, as you'll recall, reads the file as a single string. Next, think about this string. You need to figure out how many lines are in it. What creates a line (hint: newline)? What if the last line doesn't end with a newline (hint: conditional)? What string method will help you figure out how many words are in the string (hint: create a list from the string)? Finally, what string function will give you the number of characters in the string (hint: join the list elements)? An easy way to create the whitespace for the output between numbers is to include tab characters `'\t'`.
- **`main()`**: This void function prompts the user for a text file name, prints `<computer_name> wc <filename>` as shown in the example below, and then calls `wc()` with the filename as the argument. Feel free to create a computer name of your own.

The proper behavior of the code is shown below:

```
Enter text file name: poem.txt

sweetie> wc poem.txt
        10      26      141 poem.txt
```

After your program is working correctly, demonstrate it to your TA to get credit.

Task 4: Using county census data to determine a state's population

For this task, you're going to write a program that will use county census data for the state of Washington to determine its population in 2020. Download the census data file from the class website, and be sure it's located in the same folder as your IDLE session. **Also, it's very important to get a feeling for your data, so first use it as the input file for the program you wrote in Task 3.** You should find there are 39 lines because there are 39 counties in Washington, 20 in Eastern Washington and 19 in Western Washington. Now look at the first few lines of this data file using

TextEdit (macOS) or Notepad (Windows) (these text editors won't corrupt the file contents the way a word processing app such as Word will). Determine where the county population is located. Next, open an IDLE Editor window and save it as `lab11_t4.py` in your `Lab11` folder under `CS111`. Finally, write the two functions described below. Don't forget your header, a call to `main()`, and function docstrings.

- **`calc_pop()`**: This non-void function has one parameter, a file name. First you'll need to initialize an accumulator for the population. Next use an **iterating** `for`-loop to read the lines of the file. In the `for`-loop body, break the line into list elements, choose the element with the population value, strip this element of any whitespace, convert it to an integer, and add it to the accumulator. You can combine these actions in whatever way you wish, but strive for readability (I used 3 statements in the body). Finally, return the population to `main()`.
- **`main()`**: This void function opens the census data file for reading (`wa_county_pop.dat`), calls `calc_pop()` with the file name assigned to the file as the argument, and prints the result as shown below. Note that you can use an f-string as follows: `print(f'<text> {<variable>:,}.')` to add commas in the number.

The proper behavior of the code is shown below:

```
The population of Washington state in 2020 was 7,705,281.
```

After your program is working correctly, demonstrate it to your TA to get credit.

Just one more lab to go so hang in there. You're doing great!