CptS 111 — Lab #3

Practicing with Iterables

Learning Objectives:

- Use of some advanced string formatting concepts
- Use of some string functions, string indexing, concatenating, and more
- Use of lists and some list methods and functions
- Basic use of dictionaries
- Understand the difference between mutability and immutability and how they relate to strings, lists, dictionaries, and tuples

Prerequisites:

- Know how to use the IDLE Editor window to write Python programs and how to run them in the IDLE Shell window
- Know about f-string string formatting, replacement fields, and format specifiers
- Know the basics of strings, lists, sets, tuples, and dictionaries

---

Task 1: Working with strings, tuples, and lists in the IDLE Shell window

As I said in class, strings, lists, tuples, dictionaries, and sets are all types of iterables. We'll understand why when we get to the chapter on loops, but for now it's enough to know that, for the most part, they're all a similar type of data structure. We aren't going to work with sets in CptS 111. We'll work the most with strings and lists, but we'll also be using a dictionary in one of our programming assignments. We actually used tuples in our last lab, but we'll be more specific about them here. So let's get started now by first opening an IDLE Shell window and copying the text from the first line below into your window (copying and pasting can be problematic and best to avoid with multiple lines, but it should work for a single line—at least it worked for me). Then follow the instructions given as comments (be sure to read all comments; they're important!). Type everything given in boldface font into the Shell window. Note that I've removed the results so you can see them for yourself.

```
>>> word = 'supercalifragilisticexpialidocious'
>>>
>>> # We can index strings using [ ] with an index value inside, e.g., [4].
>>> # word[4] gives the fifth character because indexing starts with 0.
>>> word[4]
>>>
>>> # You can also use negative indexing, which may seem strange, but it can
>>> # be useful.  Suppose you don't know how long a word is, but you want to
>>> # know its last letter.  Try the following:
>>> word[-1]
>>>
>>> # An index of [-1] will always give you the last element in an indexed
>>> # iterable (dictionaries and sets aren't indexed).  But speaking of
>>> # length, we can determine the length of a string (or list) using the
>>> # function len().
>>> len(word)
>>>
>>> # Wow.  That's a long word!  Now let's count the number of i's in word.
>>> # We use the .count() method.
>>> word.count('i')
>>>
>>> # iiiiiii, so many i's!  How about if we try replacing the i's by y's.
>>> word.replace('i', 'y')
>>>
>>> # But have we really changed word?  Let's check.
>>> word
>>>
>>> # Nope.  We can't change elements (characters) in a string because strings
>>> # are immutable, which means they can't be changed.  We can, however,
>>> # define a new string, even calling it by the same name.  We just have to
>>> # add an lvalue.
>>> word = word.replace('i', 'y')
>>> word
>>>
>>> # We've replaced i by y in word and then have assigned word to the
>>> # same lvalue word!  Let's revert to our original word and try a bit
>>> # of concatenation (+) using operator overloading.
>>> word = word.replace('y', 'i')    # Switch y back to i
>>> word
>>> word2 = word.replace('i', 'y')
>>> word3 = word + word2
>>> len(word3)
>>> word3
>>>
>>> # Wowee.  That's quite the word.  But let's try something different now.
>>> # Let's initialize an empty string and then add to it using augmented
>>> # assignment and operator overloading.
>>> ode = ''
>>> ode
>>> ode += 'a'
>>> ode += ' '
>>> ode += 'p'
>>> ode           # See what we have so far
>>> ode += 'oe'
>>> ode += 'm'
>>> ode           # See what we end up with
>>>
>>> # Grins.  I just wrote a poem.  Thus, you can see that by using
>>> # augmented assignment (+=) together with operator overloading
>>> # (concatenation) we can add letters to a string.
>>>
>>> # Now let's turn to tuples and lists.  Consider the following tuple.
>>> tup1 = (1, 2)
```

```
>>>
>>> # As with strings, tuples can be indexed:
>>> tup1[1]
>>>
>>> # Let’s replace this value with a 3.
>>> tup1[1] = 3
>>>
>>> # Oops.  We can’t change an element in a tuple because a tuple is also
>>> # immutable.  Let’s try this with a list.  Note that as with strings
>>> # and tuples, we index lists using [] because they’re also sequences.
>>> list1 = [1, 2]
>>> list1[1]
>>> list1[1] = 3
>>> list1
>>>
>>> # We can change elements in a list because a list is mutable.  This is
>>> # very important and very useful.  Of course we can always define a new
>>> # tuple.
>>> tup2 = (1, 3)
>>> tup2
>>>
>>> # But suppose we want to replace an element in a tuple, and the tuple is
>>> # really long--e.g., it might have 68 elements, and we don’t want to
>>> # type so much?  Then we can do the following:
>>> list2 = list(tup1)     # Creates a list from a tuple
>>> list2
>>> list2[1] = 3
>>> list2
>>> tup1 = tuple(list2)    # Creates a tuple from a list
>>> tup1
>>>
>>> # Clever, eh?  Tuples are often packed or unpacked by Python ‘‘underneath
>>> # the hood.’’  For example, let’s consider the divmod() function we used
>>> # in our last lab.
>>> divmod(11, 5)
>>>
>>> # We see that divmod() results in a tuple with two elements, the //
>>> # part and the % part.  Let’s assign the result of divmod() to an lvalue.
>>> result = divmod(11, 5)
>>> result          # result is a tuple with two elements
>>>
>>> # We see that result is a tuple, but we can unpack it as follows:
>>> whole, remainder = result
>>> whole          # whole is assigned the value of result[0]
>>> remainder      # remainder is assigned the value of result[1]
>>>
>>> # Of course we can also just index the tuple:
>>> result[0]
>>> result[1]
>>>
```

```
>>> # But often it's better to give more descriptive names and avoid indexing.
>>> # I prefer to use the following and let Python unpack the divmod() tuple
>>> # result.
>>> whole, remainder = divmod(11, 5)
>>> whole
>>> remainder
```

Well we covered a lot of territory in this task, but the ideas were all related, and it probably didn't take you long. We have more to cover, but we'll start it in a new task. However, before starting, first show your TA the work you've done to get credit.

Task 2: More work with lists and some work with dictionaries

In this task we're going to cover more on lists and also do some work with dictionaries. Most of our focus will be on lists which are one of the most useful data structures in Python. Again, we'll work in the Python Shell window.

```
>>> # Let's begin by initializing an empty list.
>>> list1 = []
>>> list1
>>>
>>> # Now let's add items to our list using the append method.
>>> list1.append('apples')
>>> list1.append('oranges')
>>> list1.append('cherries')
>>> list1
>>>
>>> # Here's another way to create a list.
>>> list2 = ['bananas', 'grapes', 'nectarines']
>>>
>>> # We can combine lists using operator overloading as follows:
>>> fruits = list1 + list2
>>> fruits
>>>
>>> # Simple, eh?  Now let's replace 'apples' with 'plums'.  We do this
>>> # using list indexing, and we can do it because lists are mutable.
>>> fruits[0] = 'plums'
>>> fruits
>>>
>>> # Next let's remove 'oranges.'  We do this using the .remove() method.
>>> fruits.remove('oranges')
>>> fruits
>>>
>>> # We can determine the length of a list using the len() function. The
>>> # length of a list is the number of elements in it, but a single
>>> # element can be another list--or a tuple or a dictionary!
>>> len(fruits)
>>>
>>> # Next let's create a list of integers.
>>> primes = [5, 2, 17, 7, 13, 11, 3]
>>> primes
>>>
>>> # We can use the .sort() method to sort these integers from low to high.
>>> primes.sort()
>>> primes
>>>
>>> # Or from high to low.
>>> primes.sort(reverse=True)
>>> primes
>>>
>>> # Let’s add another 2 to this list.
>>> primes.append(2)
>>> primes
>>>
>>> # Now we can count the number of 2’s in the list using the .count()
>>> # method.
>>> primes.count(2)
>>>
>>> # We used the .sort() method to sort the list, but we can also use
>>> # the sorted() function.  The difference between the two is that
>>> # .sort() is a method which sorts a list ‘‘in place,’’ changing the
>>> # list itself.  The sorted() function doesn’t change the original
>>> # list.  Instead you have to use an lvalue which is assigned to the
>>> # new sorted list.
>>> integers = sorted(primes)
>>> integers                    # New sorted list
>>> primes                      # Original list
>>>
>>> # We can also reverse the sort order in the sorted() function.
>>> reversed = sorted(primes, reverse=True)
>>> reversed
>>>
>>> # Next let’s sum the values in integers.
>>> total = sum(integers)
>>> total
>>>
>>> # The functions sum() and sorted() as well as the .sort() method
>>> # can be used with both floats and integers or a combination of the
>>> # two.  Perhaps surprisingly, lists of strings can also be sorted,
>>> # but a list with both numbers and strings can’t.  Consider the
>>> # following.
>>> names = [’Jin’, ’saana’, ’janet’, ’Zach’, ’mohammed’, ’josiah’, ’abe’]
>>> names.sort()
>>> names
>>>
>>> # The sorted list is kind of what we might expect based on how we
>>> # alphabetize names, but the names starting with an uppercase
>>> # letter appear first.  This is because Python sorts a list of
>>> # strings by their binary values which can be represented by
>>> # decimal values.  Recall, that uppercase letters start at 65, but
>>> # lowercase letters start at 97.  Thus, 'J' is ``smaller'' than 'a'.
>>>
>>> # Let's now turn to dictionaries.  AS MENTIONED, PA #5 REQUIRES THE
>>> # USE OF A DICTIONARY, SO REMEMBER TO LOOK BACK AT THIS LAB WHEN PA #5
>>> # IS ASSIGNED.  We'll start by initializing an empty dictionary.
>>> my_info = {}     # Use a set of curly braces to initialize
>>> my_info          # an empty dictionary
>>>
>>> # Next we'll add key-value pairs to this dictionary.  Note that keys
>>> # always have to be immutable, i.e, they can't be lists or dictionaries,
>>> # but values can be anything, including other dictionaries!
>>> # Dictionaries themselves are mutable.
>>>
>>> # Note how we add key-value pairs!  We use the name of the dictionary
>>> # followed by the key in square brackets.  If the key is a string,
>>> # it must be in quotes.  The value is to the right of the assignment
>>> # operator.  Remember this!
>>> my_info['age'] = 50
>>> my_info['height'] = '6 ft 5 in'
>>> my_info['weight'] = 260
>>> my_info['occupation'] = ['athlete', 'actor', 'businessman']
>>> my_info
>>>
>>> # Can you guess who this is?  Notice that we can check the value of a
>>> # dictionary simply by using its key.
>>> my_info['occupation']
>>>
>>> # We can also change dictionarie entries.  Think about the second item!
>>> my_info['height'] = '77 in'
>>> my_info['occupation'].append('producer')
>>> my_info
>>>
>>> # Notice that because my_info['occupation'] is a list, we can simply
>>> # append another list element using the dictionary key.  We can add
>>> # key-value pairs to dictionaries and also remove them.
>>> my_info['nickname'] = 'The Rock'
>>> my_info
>>> del my_info['nickname']
>>> my_info
>>>
>>> # We can also delete all key-value pairs from a dictionary using the
>>> # .clear() method.
>>> my_info.clear()
>>> my_info
>>>
>>> # Be careful with the .clear() method because you don't want to
>>> # accidentally delete the contents of a large dictionary!
```

When you have finished this task, show your work to your TA to get credit.

Task 3: Practicing a little with advanced string formatting

For this task, you're going to have a little fun while practicing a few advanced string formatting techniques. **You'll be using the same or similar techniques in PA #6.** We'll use only f-strings in this lab, but in the future we'll learn two other approaches to string formatting.

In IDLE, open a new Editor window (C-N), write your program header, and then write a program that prompts the user and displays the results as shown below in the examples. You'll need to use the fill and alignment characters for string formatting (Sec. 3.4 in your zyBook). Each sentence in the short story should be a single print statement that uses string formatting with the input entered by the user. The examples should help you figure out the details. Again recall that the user's input is given in boldface font except for the boldface used in the story (boldface there is just to show you where replacement fields {}'s are used). Also, recall that `{:0<5}` tells Python to use left justification and enough **`0`**'s to result in a 5-character string with zero padding, e.g., **`00007`**. However, if the variable being printed is larger than 5, Python will ignore the 5 and will print, e.g., **`3141592653`** with no zeros.

Examples:

```
Enter a name: Sam
Enter a 1- or 2-digit integer: 12
Enter another 1- or 2-digit integer: 8
Enter a political, e.g., senator, dictator: tsar
Enter another name: Horace
Enter an integer: 99
Enter a noun: donut

There was once a secret service agent named Sam, also known as Agent 0012.
Agent 0012 was sent on a mission to free Agent 0008 who was imprisoned by
the cruel tsar Horace.
Agent 0012 had to fight off 990000 opponents, but Sam won the day and
was declared a donut!!!!
```

Notice that the number of zeros (0's) and exclamation marks (!'s) depends on what is entered by the user. You'll probably have to play around a bit to get the output correct, but the next example should help.

```
Enter a name: Dora the Explorer
Enter a 1- or 2-digit integer: 2
Enter another 1- or 2-digit integer: 13
Enter a political position, e.g., senator, dictator: mayor
Enter another name: SpongeBob
Enter an integer: 111
Enter a noun: valentine

There was once a secret service agent named Dora the Explorer, also known
as Agent 0002.
Agent 0002 was sent on a mission to free Agent 0013 who was imprisoned by
the cruel mayor SpongeBob.
Agent 0002 had to fight off 111000 opponents, but Dora the Explorer won
the day and was declared a valentine!
```

Save your program in your `Lab3` directory as `lab3_t3.py`. When it works, demonstrate it to your TA for credit.

Task 4: Using **`chr()`** and **`ord()`** to create a simple encryption program.

A Caesar cipher is a simple cryptography technique used by Julius Caesar. It offsets letters by a specified amount. Thus, for example, if we assign 1-26 to the English alphabet letters a-z and we want to encrypt the word ‘cat’, we can offset each letter by 3 letters so that ‘cat’ becomes ‘fdw’.

Recall that you can use the `chr()` function to determine the character represented by an ASCII decimal value, e.g., `chr(65)` returns `'A'`. Conversely, you can use the `ord()` function to determine the ASCII decimal value representing a character, e.g., `ord('A')` returns `65`. Let’s check out these two functions.

```
>>> # First let's try the two examples given above.
>>> chr(65)
>>> ord('A')
>>>
>>> # chr() and ord() are reciprocals as shown below.
>>> chr(ord('A'))
>>> ord(chr(65))
>>>
>>> # Now see what happens when we add or subtract an offset from the
>>> # integer given by the ord() function.
>>> e_chr = chr(ord('a') + 3)
>>> e_chr
>>> d_chr = chr(ord(e_chr) - 3)
>>> d_chr
```

Now you’re going to write a simple cryptography program. Follow the steps below to create this program using your IDLE Editor window. Save your program in your `Lab3` directory as `lab3_t4.py`.

1. Initialize two empty strings, one for encryption and the other for decryption. Use good names for your strings!

2. Prompt the user for a small integer offset value (see example below) and then prompt the user to enter five letters, one at a time, that spell a word. After each letter has been entered, it should be encrypted using addition of the offset and added to the encryption string using augmented assignment **+=** with operator overloading. Note that augmented assignment isn’t really necessary, but I want you to practice its use!

3. Next, the five letters that were entered as input, **i.e., the unencrypted letters**, should be concatenated using **+** to obtain your five-letter word.

4. To decrypt the word, subtract the offset from the encrypted value of each encrypted letter and again use augmented assignment with operator overloading to add each decrypted letter to the decryption string.

5. Print your encrypted and decrypted words as shown in the last example. Add a blank line between your input and output!

```
Enter offset (a small integer): 4
Enter first letter to encrypt: s
Enter second letter to encrypt: c
Enter third letter to encrypt: o
Enter fourth letter to encrypt: n
Enter last letter to encrypt: e

scone encrypted is wgsri.
wgsri decrypted is scone.
```

When your program is working properly, show it to your TA and demonstrate that it works.

Another lab completed! Good job!