CSci 1133
Lab Exercise 7

Fun With Strings

In this lab we explore operations and methods using *string* objects. Strings are non-scalar, *immutable* sequence objects consisting of an ordered collection of individual characters. Strings are primarily employed to represent text (words, sentences, names, paragraphs, etc.) but are often useful as data structures for other problems as well (genetic sequences, card games, state spaces, etc.). The primitive string operations and methods in Python provide a rich set of powerful constructs for manipulating these data in creative ways. The Python3 documentation is a great resource for learning about all the available (built-in) string methods:

https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

Word Games

A *palindrome* is a word or phrase in which the letters "read" the same way forward and backward. For example, "deed", "kayak" and "Madam, I'm Adam" are all palindromes. An interesting computational problem is to decide if any given string is a palindrome: you provide a string containing a possible palindrome to the process, and it returns "yes" or "no". Although this is fairly straightforward for humans to accomplish, doing so on a computer requires some skill in manipulating strings. Start Python in command-line mode and begin by entering a possible palindrome:

```
>>> p = "A man, a plan, a canal… Panama!" <ENTER>
```

When testing palindromic *phrases*, we need to first eliminate punctuation and spaces. One of the string methods we discussed is quite useful in this regard: `.replace(`*oldstring*,*newstring*`)`. Let's use it to remove all the commas in the string. First, create a copy of the string:

```
>>> s = p <ENTER>
```

Now let's replace all the instances of the substring "," (comma) with "" which is a way of representing a NULL string containing no characters at all:

```
>>> s.replace(",","") <ENTER>
```

That's a good start. The commas are gone. But are they really?

```
>>> s <ENTER>
```

The string method does not alter the string object because it's *immutable*. `.replace` returns a *new* string object. To effect the change, we need to bind the returned object with a variable name. Since we are ultimately trying to simplify the entire string we can just bind it with the same name:

```
>>> s = s.replace(",","") <ENTER>
>>> s <ENTER>
```

OK, let's get rid of the periods in the same fashion:

```
    >>> s=s.replace(".","") <ENTER>
    >>> s <ENTER>
```

…and the exclamation point:

```
    >>> s = s.replace("!","") <ENTER>
    >>> s <ENTER>
```

This is starting to look pretty good. We've managed to get rid of all the punctuation. But in order to determine if this is a palindrome, we have to compare the 1st letter to the last letter, the 2nd letter to the penultimate letter and so on. In order for this to work, the spaces will also have to go:

```
    >>> s = s.replace(" ","") <ENTER>
    >>> s <ENTER>
```

It all seems to work, but this is starting to become very tedious! We also have another problem: we've selected *specific* punctuation based only on what was in *our* string. In order to make this a more general procedure, we should anticipate and remove *all possible* punctuation characters. Here's a less tedious way to do everything we've accomplished so far using a single statement. First, restore the palindrome string to its original condition:

```
    >>> s = p  <ENTER>
    >>> s  <ENTER>
```

Now let's replace the punctuation and spaces all at once:

```
    >>> for ch in ",.! ":
            s = s.replace(ch,"") <ENTER><ENTER>
    >>> s <ENTER>
```

Voila! You simply define a string with all the punctuation you want to remove and iterate through the punctuation string, removing all occurrences of each. You might want to define the punctuation string as a variable to make it easier to change things later on.

We're getting pretty close. All we need to do is construct some sort of loop that will do the necessary comparisons and determine if the string reads the same way forward as it does backward. But the comparisons will be case sensitive, and the first character ('A') is currently not the same as the last one ('a'). This is easy to fix:

```
    >>> s = s.lower() <ENTER>
    >>> s
```

… and we now have a string that is suitable for automatic comparison. Note that we could have used `s.upper()` just as well.

Another recurring "pattern" that often occurs when processing text is the isolation of *words* in a sentence or paragraph. In this case, spaces serve as *delimiters*. "Words" are those elements that occur between "white space". In order to obtain only the words, we need to remove the extra white space and extraneous punctuation. However, we cannot simply remove *all* the spaces because we need them to determine where words start and end! Let's try to isolate and count all the words in a sentence:

```
>>> s = "    Now is    the   time  for … what exactly?" <ENTER>
>>> s  <ENTER>
```

This sentence has some issues with leading (extraneous) space, punctuation, etc. If we're going to count just the words, we need to figure out where words start and end. First let's get rid of the punctuation:

```
>>> s = s.replace("?","").replace(".","") <ENTER>
>>> s <ENTER>
```

Initially we need to determine exactly where the first word starts. We could write a loop of some sort, but there is a string method that can help us:

```
>>> s = s.lstrip() <ENTER>
>>> s <ENTER>
```

`.lstrip()` is a method that *strips* all the leading white space from the left end of the string. Since words are delimited by spaces, the resulting string will (by definition) start with a word. After we strip the leading white space, the word will always start at index `[0]`. But where does the word *end*? Well, here's another method that proves very useful:

```
>>> idx = s.find(' ') <ENTER>
```

Recall that the `.find(`*substring*`)` method will return the index of the first occurrence of the *substring* (or -1 if its not found). So `idx` should now contain the index of the space that immediately follows (delimits) the word. Let's see:

```
>>> idx <ENTER>
```

If we are counting words, we might increment our count at this point, but for now let's figure out how to find the *next* word in the string. Recall the *slicing* operator `[`*start* : *end* `]`, and also recall that if we omit the *start* or *end* of the slice, we get "everything from the beginning", or "everything to the end" respectively. This provides a powerful capability to *split* the string at the index into two separate strings… Let's try it:

```
>>> s[ : idx] <ENTER>
>>> s[ idx: ] <ENTER>
```

The first "slice" contains all of the string up to, *but not including* the space and the second "slice" is the remainder of the string with the space at the beginning. Since we are finished with the first word, we can essentially get rid of it by just keeping the remainder of the string. And since we know that the remaining string starts with a space we may as well remove it at the same time:

```
>>> s = s[ idx: ].lstrip() <ENTER>
>>> s <ENTER>
```

Very nice! Now we just continue this process (in a loop would be more useful!) and count the words in the sentence. Note that if you continue on in this way, the `.find(' ')` method for the last word will fail (and return a -1). Why? How might you deal with this?

Understanding the basic function of string operations and methods is important, but perhaps more important is the skill (developed through practice!) of using these methods to transform strings into forms that are suitable for the task at hand. As always, you should explore on your own and experiment with various ways of manipulating strings.

Mystery-Box Challenge

The "Mystery-Box Challenge" is designed to help you develop a better understanding of how programs work. The Mystery-Box Challenge presents a short code fragment that you must interpret to determine what computation it is *intending* to accomplish at a "high" (semantic) level. For example, the following line of code:

```
3.14159 * radius * radius
```

is intended to "*compute the area of a circle whose radius is contained in a variable named 'radius'*". An answer such as "*multiply 3.14159 by radius and then multiply it by radius again*", although technically accurate, would not be considered a correct answer!

Here is your first *mystery-box challenge*: Determine what the following Python function does and describe it to one of your Lab TAs :

```
def foo(istring):
    p = '0123456789'
    os = ''
    for iterator in istring:
        if iterator not in p:
            os += iterator
    return os
```

Warm-up

Use the command-line Python interface to accomplish the following:

1) Using the string class `.count` method, display the number of occurrences of the character '`s`' in the string "`mississippi`".

2) Replace all occurrences of the substring '`iss`' with '`ox`' in the string "`mississippi`".

3) Find the index of the first occurrence of '`p`' in the string "`mississippi`".

4) Construct a 20-character string that contains the word '`python`' in all capital letters in the exact center. The remaining characters should all be spaces.

Stretch

1) **Letter Count**

Write a function named `ltrcount` that will take a single string argument containing any legal character and return the number of characters in the string that are *alphabetic* letters (lower-case a-z or upper-case A-Z). Test your function using the command-line interface with the following strings:

```
"abcdeF4562_*"
""
"1234567"
"ABCDE"
```

2) **Reverse a String**

Write a function named `reverse` that will take a single string argument and return a string that is the reverse (mirror-image) of the argument. Do not convert the string to a list and .reverse() it! Test your function using the command-line interface.

3) **Palindrome Check**

Now write a Python program that will include the function from the previous problem and an additional user-defined function named `ispalindrome` that will take a single string argument and return `True` if the argument is a palindrome, `False` otherwise. The `ispalindrome` function must call the `reverse` function you created in Stretch Problem (2). Be sure to normalize the strings to uppercase or lowercase before the comparison!

Your program should input a string containing a palindrome or palindromic phrase, call the `ispalindrome` function, and display a message indicating if the string is indeed a palindrome. Include a loop to allow the user to repeat the process as many times as he/she desires.

Example:

```
Enter a string: Abba
Abba is a palindrome
Continue? (y/n): y

Enter a string: Telex
Telex is not a palindrome
Continue? (y/n): y

Enter a string: MadaM I'm Adam
MadaM I'm Adam is a palindrome
Continue? (y/n): n
```

Workout

1) **Split**

The Python language provides numerous powerful and concise object *methods* that facilitate the rapid construction of compact scripts. One of the most useful string methods is `.split(`*delimiter*`)` that will split a string into a list of substrings and will later prove quite useful when we begin to read data from files.

Assume for the moment that Python does not have such a method. Create a non-recursive function named `mysplit` that will take two arguments: a string object and a *delimiter* string, and return a list of all the substrings separated by the *delimiter*. Your function should exactly replicate the functionality of the `.split` string method:

```
     def mysplit(stringarg, delimiter):
```

Test your function using the command-line interface and compare the results to those obtained when you use the .split method. e.g.,:

```
     >>> s = "yada,yada,yada"
     >>> mysplit(s,',')
     [yada, yada, yada]
     >>> s.split(',')
     [yada, yada, yada]
     >>>
     >>> s = "my country tis of me but not tis of thou"
     >>> mysplit(s,'tis')
     ['my country ', ' of me but not ', ' of thou']
     >>> s.split('tis')
     ['my country ', ' of me but not ', ' of thou']
```

2) **DNA**

The DNA molecule employs a 4-element *code* consisting of sequences of the following nucleotides: *Adenine*, *Guanine*, *Cytosine* and *Thymine*. Biologists often use a letter sequence to represent the genome, in which each letter stands for one of the four nucleotides. For example:

. . . AGTCTATGTATCTCGTT . . .

Individual *genes* are substrings of a genome delineated by 3-element *start* and *stop codons*. The majority of genes begin with the start codon: ATG and end with one of the following 3 *stop* codons: TAG, TAA or TGA. Start codons can appear anywhere in the string, followed by a series of 3-element codons and ending with a stop codon. Note that genes are multiples of 3 in length and do not contain any of the triples ATG, TAG, TAA or TGA.

Write a Python program that will read in a genome as a string and display all the genes in the genome. Note that that finding a substring matching an end codon but which overlaps two codons would not be an actual end codon (i.e., the resulting gene string should be divisible by three).

Example:

```
     Enter a DNA sequence: TCATGTGCCCAAGCTGACTATGGCCCAATAGCG
     Gene 1 TGCCCAAGC
     Gene 2 GCCCAA
```

Challenge

1) **Check Amount**

Write a function named `dollarstring` that will accept a dollar value (float) and return a string representing the value (in words) as it should appear on a printed voucher. For example, the value 356.37 should be converted to the string: 'three hundred fifty-six and 37/100 dollars". Note the following:

- You may ssume the largest possible amount will be 99,999.99
- Do not report amounts of "zero" (e.g., 3,000.28 would be 'three thousand and 28/100 dollars')
- Report all cents using *exactly* two digits (e.g., 2 cents would be '02/100 dollars')
- Tens and ones values should be separated by a dash, but *only* if the ones value is not zero (e.g., 21.36 would be 'twenty-one and 36/100 dollars' but 20.36 would be 'twenty and 36/100'

Write a Python program that will input a dollar amount, call the `dollarstring` function and print the converted value:

Example:

```
Enter a dollar amount: 50300.45
fifty thousand three hundred and 45/100 dollars

Enter a dollar amount: 1256.32
one thousand two hundred fifty-six and 32/100 dollars

Enter a dollar amount: 5.99
five and 99/100 dollars

Enter a dollar amount: 801.22
eight hundred one and 22/100 dollars

Enter a dollar amount: 4000.07
four thousand and 07/100 dollars

Enter a dollar amount: .36
36/100 dollars
```