**Name : Session:**

**Programming I**

**String Theory**

**Lab Exercise 10.4.2024**

For each of these problems, you are to submit your documented source code.

In this exercise, we will be learning to use a variety of string functions. Some of these we have already used but some will be new.

Text in our programs is represented by the string data type. You can think of a string as a sequence of characters. Python allows you to designate strings with either a single of double quote. Most languages require double quotes but in Python you can use either. The only restriction is that they must be a matched set. For example, “hello’ would be illegal.

Strings may be stored in variables just as you would numbers.

Now let’s look at some useful string functions (aka methods)

The first function is the *split*() function. Here are two examples that demonstrate the *split()* function returns a list of strings. The default parameter of *split()* is the space character but it can be others.

test = 'this is a test'

result = test.split()

print(result)

which returns the list ['this', 'is', 'a', 'test']

test2 = '3/21/2016'

result2 = test2.split('/')

print(result2)

which returns the list ['3', '21', '2016']

Next we have the join function which takes a list and rebuilds the string.

answer = ' '.join(result)

print(answer)

which returns the string ‘this is a test’

answer2 = '-'.join(result2)

print(answer2)

which returns the string ‘3-21-2016’

**Notice on the first example, we rebuilt the string with a single space between each string and in the second example, we rebuilt the string with a dash character between each string.**

The next technique that can be really useful is string slicing. Here are some examples:

test3 = 'abcdefg'

slice1 = test3[2:5] #extract element 2 to 4

print(slice1)

which will return the string ‘cde’

slice2 = test3[:5] #extract element 0 to 4

print(slice2)

which will return the string ‘abcde’

slice3 = test3[2:] #extract element 2 to end of string

print(slice3)

which will return the string ‘cdefg’

slice4 = test3[1:6:2] #extract elements 1, 3, and 5

print(slice4)

which will return the string ‘bdf’

slice5 = test3[:] #extract the entire string

print(slice5)

which will return the string ‘abcdefg’

Here is a way we can get the last element of a string without having to know it’s length.

test5 = 'abc'

last = test5[-1]

print(last)

which will return the string ‘c’

One last trick with string slicing will cause the reversing of a string the easy way

test4 = 'abcdefghijklmnopqrstuvwxyz'

slice6 = test4[::-1]

print(slice6)

which will return the string ‘zyxwvutsrqponmlkjihgfedcba’

Here is a summary of operators that can operate on strings

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| + | Concatenation |
| \* | Repetition |
| <string>[ ] | Indexing |
| <string>[:] | Slicing |
| len(<string>) | Length |
| for <var> in <string> | Iteration through characters |

Here is a summary of string methods that are available for you to use

|  |  |
| --- | --- |
| **Method** | **Meaning** |
| **capitalize**( ) | Return a copy of the string with only its first character capitalized |
| **center**(*width*[*, fillchar*]) | Return centered in a string of length *width* |
| |  |  | | --- | --- | |  |  |   **count**(*sub*[*, start*[*, end*]]) | Return the number of occurrences of substring *sub* in string S[*start*:*end*] |
| **endswith**(*suffix*[*, start*[*, end*]]) | Return True if the string ends with the specified *suffix*, otherwise return False |
| **expandtabs**([*tabsize*]) | Return a copy of the string where all tab characters are expanded using spaces |
| **find**(*sub*[*, start*[*, end*]]) | Return the lowest index in the string where substring *sub* is found |
| **index**(*sub*[*, start*[*, end*]]) | Return the lowest index in the string where substring *sub* is found |
| **isalnum**( ) | Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise |
| **isalpha**( ) | Return true if all characters in the string are alphabetic and there is at least one character, false otherwise |
| **isdigit**( ) | Return true if all characters in the string are digits and there is at least one character, false otherwise |
| **islower**( ) | Return true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise |
| **isspace**( ) | Return true if there are only whitespace characters in the string and there is at least one character, false otherwise |
| **istitle**( ) | Return true if the string is a titlecased string and there is at least one character |
| **isupper**( ) | Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise |
| **join**(*seq*) | Return a string which is the concatenation of the strings in the sequence *seq* |
| **ljust**(*width*[*, fillchar*]) | Return the string left justified in a string of length *width* |
| **lower**( ) | Return a copy of the string converted to lowercase |
| **lstrip**([*chars*]) | Return a copy of the string with leading characters removed |
| **partition**(*sep*) | Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator |
| **replace**(*old, new*[*, count*]) | Return a copy of the string with all occurrences of substring *old* replaced by *new* |
| **rfind**(*sub*[*,start*[*,end*]]) | Return the highest index in the string where substring *sub* is found, such that *sub* is contained within s[start,end] |
| **rindex**(*sub*[*, start*[*, end*]]) | Return the highest index in the string where substring *sub* is found, such that *sub* is contained within s[start,end] |
| **rjust**(*width*[*, fillchar*]) | Return the string right justified in a string of length *width* |
| **rpartition**(*sep*) | Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator |
| **rsplit**([*sep*[*,maxsplit*]]) | Return a list of the words in the string, using *sep* as the delimiter string |
| **rstrip**([*chars*]) | Return a copy of the string with trailing characters removed |
| **split**([*sep*[*,maxsplit*]]) | Return a list of the words in the string, using *sep* as the delimiter string |
| **splitlines**([*keepends*]) | Return a list of the lines in the string, breaking at line boundaries |
| **startswith**(*prefix*[*, start*[*, end*]]) | Return True if string starts with the *prefix*, otherwise return False |
| **strip**([*chars*]) | Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed |
| **swapcase**( ) | Return a copy of the string with uppercase characters converted to lowercase and vice versa. |
| **title**( ) | Return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase |
| **upper**( ) | Return a copy of the string converted to uppercase |
| **zfill**(*width*) | Return the numeric string left filled with zeros in a string of length *width* |

**Exercises**

1. In molecular biology and genetics, GC-content (or guanine-cytosine content) is the percentage of nitrogenous bases on a DNA molecule that are either guanine or cytosine (from a possibility of four different ones, also including adenine and thymine and possibly a non-encoded). Since we do not have a DNA sample to work with, we will generate our own. Create a DNA sequence of 1,000,000 bases (designate them C, G, A, T, or N). Parse your DNA sequence and count the G and C bases and report the percentage of GC content.
2. In molecular biology, we often need to find the reverse complement of a DNA sequence. This an operation that you might find in the field of bioinformatics. In order to solve this problem, we need to perform two operations on our DNA string. We must reverse the string and then build a new string that is the complement of the reversed string. You might consider using a dictionary for this process. Write a reverseComplent function that is passed a DNA string and returns a reverse complement string. To test it, use the string “ACGTAAGTCA” it should return the string “TGACTTACGT”.
3. In molecular biology, a reading frame is a way of dividing the sequence of nucleotides in a nucleic acid (DNA or RNA) molecule into a set of consecutive, non-overlapping triplets. Where these triplets equate to amino acids or stop signals during translation, they are called codons. In the genetic code, a stop codon (or termination codon) is a nucleotide triplet within messenger RNA that signals a termination of translation into proteins.

https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Reading_Frame.png/400px-Reading_Frame.png

In a DNA sequence, there are three stop condons (TAA, TAG, and TGA). Write a function that will test the presence of a stop codon in a DNA sequence given a DNA sequence and a reading frame of 0, 1, or 2.

def hasStopCodon(dna, frame):

#Add code here