# Lab 4 – Strings

## Objective

The objective of this exercise is to learn how Python deals with strings

### Starting the IDLE Environment for Python

Start the IDLE environment. You should see some messages showing the version of Python and the version of IDLE, followed by the >>> prompt. This means that IDLE is waiting for you to type Python commands.

### Strings as a Sequence

Strings objects are defined as a sequence of characters. Therefore, "*Hello World*" is a sequence of 11 characters – remember that a space is a character. Because a sequence is in order, we can number the characters by their position in the sequence. The position is called the index of the character within the string.

Create a new string in Python by typing the following at the >>> prompt:

helloString = "Hello World"

Now type each of the following at the prompt and see what gets printed.

>>> helloString[0]

>>> helloString[5]

>>> helloString[-1]

>>> helloString[10]

>>> helloString[11]

>>> ord('A')

>>> ord('a')

>>> chr(97)

>>> chr(65)

**More Indexing and Slicing**

Indexing in Python is more powerful than in many languages. Not only can you index a character at a single position, you can also select sub-sequences of the string when you index.

>>> helloString[6:10]

The colon (:) within the index operator brackets indicates that instead of a single position being selected, a range of indices is being selected. Python uses half-open range.

Every sequence can be specified with two values: the beginning index of the sequence and end index of the sequence, separated by a colon (:). If a value on either side of the colon is missing, a *default* is used in place of the missing value. If the first index is missing, the index before the colon (:), Python assumes the start index is the beginning of the string, i.e. index 0. If the last index is missing, the value after the colon (:), Python assumes the end of the sequence *including* the last element.

>>> helloString[6:11]

>>> helloString[-1]

>>> helloString[6:]

>>> helloString[:5]

>>> helloString[0:5]

>>> helloString[3:-2]

### Extended Slicing

Slicing allows a third parameter that specifies the step in the slice. This means that you can have as many as three numbers in the index operator brackets separated by two colon characters: the first number is the beginning of the sequence, the second number specifies the end of the sequence, and the third is the step to take along the sequence. As with the first two arguments, the step number has a default if not indicated: a step of 1. The step value indicates how much to add to the present index to get the next element of the subsequence.

>>>helloString = "Hello World"

>>>helloString[::2]

>>>helloString[::3]

>>>helloString[::-1]

>>>helloString[::-2]

>>>digits = "0123456789"

>>>digits[::2]

>>>digits[1::2]

>>>digits[::-1]

>>>digits[::-2]

>>>digits[-2::-2]

### Copy Slice

### The final piece of slicing is the copy slice. If the programmer provides neither a beginning nor an end – that is, there is only a colon character in the square brackets – a copy of the string is made. Could you find the reason for the following?

>>> nameOne = "Monty"

>>> nameTwo = nameOne[:]

>>> nameTwo

### String Operations

Strings can utilize some of the same binary operators that integers and floating point numbers use, though there are some interesting differences.

The + and the \* operators can be used with string objects.

+: concatenate: The + requires two string objects and creates a new string object. The new string object is formed by joining the two string objects together: the

first string is joined at its end to the beginning of the second string.

\*: repeat. The \* takes a string object and an integer and creates a new string object. The new string object has as many copies of the string as is indicated by the

integer. Try to guess the values of each of the following expressions before trying them on the Python shell.

>>> myString = "Hello"

>>> yourString = "World"

>>> myString + yourString

>>> yourString + myString

>>> myString + " " + yourString

>>> myString \* 3

>>> 3 \* myString

>>> (myString + " ") \* 3

>>> myString + " " \* 3

>>> myString

>>> yourString

>>> "hello" + 3

### String Comparison

>>> 'abc' < 'cde'

>>> 'abc' < 'abd'

>>> '' < 'a'

The ***in*** operator is useful for checking membership in a collection.

>>> vowels = "aeiou"

>>> 'a' in vowels

>>> 'x' in vowels

>>> if 'e' in vowels:

print (" is a vowel")

### String Collections are immutable (just like in Java)

>>> myString = "hello"

>>> myString[0] = 'J'

>>> myString = "Hello"

>>> myString = 'J' + myString[1:]

>>> myString

**Preview of functions and methods**

A function is a small program that performs a specific task. Instead of writing the same code over and over again, we encapsulate that code in a function, making it easier to use.

>>> myStr = "Hello World"

>>> len(myStr)

>>> strLength = len(myStr)

>>> print (strLength)

>>> len()

A method is a variation on a function. It looks very similar. It has a name, and it has a list of arguments in parentheses. It differs, however, in the way it is invoked. Every method is called in conjunction with a particular object. The kind of methods that can be used in conjunction with an object depends on the object’s type. String objects have particular methods suited for strings.

>>> myString = "Python rules"

>>> myString.upper()

>>>

The calling object may be just an object, such as "A string", or a variable associated with an object such as myString.

An example string method with arguments would be the find method. The find method’s task is to locate a substring within the calling string. The find method returns the index of the substring in the string where the substring first occurs (if there are multiple occurrences), but returns -1 if the substring is not found.

>>> myString = "mellow yello"

>>> myString.find("m")

>>> myString.find("ll")

>>> myString.find("z")

**Chaining of Methods**

A powerful feature of Python is that methods and functions can be chained, meaning there are a series of “dot notation” invocation.

>>> myStr = "Python rules! "

>>> myStr.upper()

>>> myStr.upper().find("O")

**Optional Arguments**

Some methods have optional arguments. If the argument is not provided, a *default* for that argument is assumed. The default value depends on the method. However, you can choose to provide that argument and override that default. The find method is one with default arguments. You can start the find process from an index other than zero, the leftmost index. By default, find starts at index 0, but if you provide a second argument, that is the index where the find process begins.

>>> aStr = "He had the bat. "

>>> aStr .find("t")

>>> aStr.find("t", 8)

>>> aStr.find("t", aStr.find("t") + 1)

**Formatted output for Strings**

Using the default print function is easy, but it provides no control of what is called the *format* of the output. By format, we mean a low-level kind of typesetting to better control how the output looks on the console. Python provides a finer level of control that gives us the option to provide “prettier”, more readable output. Conveniently, the control of console typesetting is done through the use of the string format method. The basic form of the format method is shown below:

"*format string*"*.format (data1, data2, …)*

The process of creating formatted output can be a bit complicated. In fact, there is enough detail to the formatting process that the Python documentation refers to it as a “mini language”.

As with all strings, use of the format method creates a new string. The *format string*, the string that is used to call the format method, provides a normal, everyday string that is the “source” for the new string. Everything in that format string will be reproduced exactly in the new string (same spacing, same character), with one important exception. The programmer can insert special character sequences, enclosed in braces ({}), in the format string that indicate a kind substitution that should occur at that position in the new string. The substitution is driven by the arguments provided in the format method. The objects in the arguments will be placed in the new string at the indicated position, as well as *how* it will be placed in the string at that position. After substituting the formatted data into the new string, the new string is returned.

In its simplest form, the formatting commands are just empty braces. The objects that will be substituted for each brace are determined by the order of both the braces and the arguments. The first brace will be replaced by the first argument, the second brace by the second argument and so on. Here are some examples.

>>> "{} is {} years old".format("Bill", 25)

'Bill is 25 years old'

>>>import math

>>> "{} is divine but {} is more divine".format(1, math.pi)

'1 is divine but 3.141592653589793 is more divine'

>>> print("Sorry, is this the {} minute {}?".format(5, 'Argument'))

Sorry, is this the 5 minute Argument?

>>>

The way each object is formatted in the string is done by default based on its type as shown by the above examples. However, each brace can include *formatting commands* that provide directives about how a particular object is to be printed. The four pieces of information that one can provide for a particular object are descriptor code, an alignment number, a width number, and a precision descriptor.

The general structure of the most commonly used parts of the format command is:

{:[align] [minimum\_width] [.precision] [descriptor]}

where the square brackets, [], indicate optional arguments. It is important to note the placement of the colon. All the optional information comes after a colon in the braces.

The different types are described in the following table. There are actually 10 possible format commands.

|  |  |
| --- | --- |
| s | string |
| d | decimal integer |
| f | floating-point decimal |
| e | floating-point exponential |
| % | Floating-point as percent |

**Width and Alignment Descriptors**

A field width can be specified for each data item. It specifies a printing-field width, counted as the number of spaces the object occupies. By default, formatted strings are left justified and formatted numbers are right justified. If the specification includes a less than (<), the data are placed left justified within the indicated width: a greater than (>) forces right justification. Centering can be done using "^". Here is an example. Note that the string "Bill" is right justified in a width of 10 spaces, and the number 25 s left justified in 10 spaces.

>>> print("{:>10s} is {:<10d} years old.".format("Bill", 25))

>>> for i in range(5):

print("{:10d} ---> {:4d}".format(i, i \*\* 2))

>>> import math

>>> print (math.pi)

>>> print ("Pi is {:.4f}".format(math.pi))

>>> print ("Pi is {:8.4f}".format(math.pi))

>>> print ("Pi is {:8.2f}".format(math.pi))

Finally, there is a % floating point descriptor that converts from a decimal to a percent, including the insertion of the % character.

>>> 2/3

>>> print ("{:8.2%}".format(2/3))

>>>

Let us use the find method using some examples

>>> river = "Mississippi"

>>> river.find("p")

>>> len(river)

>>> for index in range(len(river)):

print (index, end = ' ')

>>> for index in range(len(river)):

print (river[index], end = ' ')

# Our implementation of the find function. Prints the index where

# the target is found: a failure message, if it isn't found.

# This version only searches for a single character

river = "Mississippi"

target = input("Input a character to find: ")

for index in range(len(river)): # for each index

if (river[index] == target): #check if the target is found

print (target, " found at index: ", index) # if so, print the index

break

else:

print ("Letter ", target, " not found in ", river)

We frequently look for both an index and the character, so Python provides the enumerate iterator, which provides both the index of the character and the character itself as its steps through the string. Let us try it:

>>> for index, letter in enumerate(river):

print (index, letter)

Interestingly, notice that the above ***for*** statement has two variables. This is because the enumerate statement yields two values each time: the index and the character. As in double assignment, we can capture both of those values by providing those two iterator variables. If we provide only one variable, what would we get? Let’s see:

>>> for value in enumerate(river):

print (value)

Now that we know how enumerate works, we can refactor our code.

# Our implementation of the find function. Prints the index where

# the target is found: a failure message, if it isn't found.

# This version only searches for a single character

river = "Mississippi"

target = input("Input a character to find: ")

for index, letter in enumerate(river): # for each index

if (river[index] == target): #check if the target is found

print (target, " found at index: ", index) # if so, print the index

break

else:

print ("Letter ", target, " not found in ", river)

**Working with Strings**

Let us use a few string methods and try to do something useful. Here is our problem.

Transform a name from the order of "First Middle Last" to the order of "Last, First Middle"

For example, if the name provided is "John E. Doe" it would be transformed as "Doe, John E. "

The string split method is very useful for this problem. The split method takes the calling string and creates substrings of that string, where substring creation occurs at a particular character(s). The invocation to split on spaces is .split(' '); at every comma is .split(', '). For example

>>> 'The Spanish Inquisition'.split()

['The', 'Spanish', 'Inquisition']

The default is to split on whitespace, so commonly we simply use split() with no arguments.

>>> name = "John Fitzgerald Kennedy"

>>> first, middle, last = name.split()

>>> transformed = last + ", " + first + " " + middle

>>> print (transformed)

>>> print (name)

>>> print (last)

>>> print (first)

>>> print (middle)

>>> first, middle = name.split() # error, not enough pieces

>>> line = "bob, carol, ted, alice"

>>> first, second, third, fourth = line.split(",")

>>> print (first, second, third, fourth)

>>> op1, op2 = "A+B".split("+")

>>> print (op1, op2)

**Palindromes**

>>> palindrome1 = "Madam, I'm Adam"

>>> palindrome2 = "A man, a plan, a canal, Panama"

>>> print("Forward: {} \nBackward: {}".format(palindrome1, palindrome1[::-1]))

>>> print("Forward: {} \nBackward: {}".format(palindrome2, palindrome2[::-1]))

We did a few things in the above session. We had to use double quotes for the first palindrome so we could capture the apostrophe in the word “I'm”, and we did a little string formatting so that we could easily compare the two strings.

Suppose we want to check whether a string is a palindrome. Usually we consider only letters and digits and ignore the punctuation marks. To check membership in a collection we can use the in operator and query whether certain characters are in the string. Conveniently, Python provides another module that stores various groups of characters as a string for exactly this purpose. If we import the string module, you will see that it contains a number of predefined strings, including:

string.punctuation: '!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~'

string.digits: "0123456789"

string.ascii.lowercase: "abcdefghijklmnopqrstuvwxyz"

string.whitespace: '\t\n\x0b\x0c\r '

The last one looks a bit odd, but it contains a representation using the backslash prefix for the control characters, such as tab, carriage return, and space (which we couldn’t see otherwise). We can use these predefined strings to do membership tests. For our palindromes we want to test only lowercase letters or digits and exclude punctuation and whitespace. Once we find a character we do not want, we can use Python’s string method replace. The replace method takes two arguments: the first is the string we are looking for, and the second is the replacement string if the first string is found. For example, abbc.replace("bv", "r") would return the new string arrc. One way to remove a character is to replace it with empty string(""). Here is our full solution:

# Palindrome tester

import string

originalStr = input('Input a string: ')

modifiedStr = originalStr.lower()

bad\_chars = string.whitespace + string.punctuation

for char in modifiedStr:

if char in bad\_chars: # remove bad characters

modifiedStr = modifiedStr.replace(char, '')

if modifiedStr == modifiedStr[::-1]: #it is a palindrome

print(\

'The original string is: {}\n\

The modified string is: {}\n\

The reversal is: {}\n\

String is a palindrome'. format(originalStr, modifiedStr, modifiedStr[::-1]))

else:

print(\

'The original string is: {}\n\

The modified string is: {}\n\

The reversal is: {}\n\

String is not a palindrome'. format(originalStr, modifiedStr, modifiedStr[::-1]))

Lab Problems:

1. Suppose you have a string ab\_string = 'abababababab' Write an expression to remove all the b's and create a string a\_string = 'aaaaaa'
2. Given a string 'abcdefghij' write a single line of code that will print the following:

'jihgfedcba'

'adgj'

'igeca'

1. In the palindrome program we used replace to remove bad characters. Refactor that program to **keep** the good characters rather than remove the bad characters.
2. Pig Latin is a game of alterations played on words. To make the Pig Latin form of an English word the initial consonant sound is transposed to the end of the word and "ay" is affixed. Specifically, there are two rules:
3. If a word begins with a vowel, append "yay" to the end of the word
4. If a word begins with a consonant, remove all the consonants from the beginning up to first vowel and append them to the end of the word. Finally, append "ay" to the end of the word.

For example,

* dog 🡺 ogday
* scratch🡺 atchscray
* is 🡺 isyay
* apple 🡺 appleyay

Write a program that repeatedly prompts for an English word to translate into Pig Latin and prints the translated word. If the user enters a period, halt the program.