**An Experimental Exercise**

**Due: Thursday March 29, 11:55 p.m.**

**Objective:** This exercise is designed to help you learn more about strings through experimentation.

**The Lab:** Strings are actually a special type of variable in Python called an ***object***. First, let’s consider a variable that holds a simple type of information, such as an integer. There are operations that you can apply to integers, like addition, subtraction etc., and you can use an integer value/variables when you call a function, like when you call the randint function to generate a random number.

Objects such as Strings share many of these same characteristics, but they also have some extra features. Like a variable that holds an integer, for instance, a variable that holds a String object can be used in an assignment statement (we have been assigning values to strings and printing out those values, for instance), and can be used in functions (for instance, you give a string parameter to print). However, in addition, there are special pieces of code (like functions) that are ONLY associated with objects of a certain type. These pieces of code are called **methods**. Thus, there are methods associated with Strings that can only be used with Strings. The use of these methods is specified by the variable name, followed by a period, followed by the method name and any parameters it might take. The general formant is

object\_variable.method\_name( optional parameters)

If the method takes no parameters, the parenthesis still need to be there, but there will be nothing inside of them. It’s just the rules of the language.

Actually, we have already been using objects and calling their methods. Surprise! For instance, a file variable is actually an object. So, if we had the statement

students = open (‘class\_list.txt’,’r’)

And then

students.close()

You can see that we have the method *close* (which can only be used with file variables) invoked by specifying the file variable (students), followed by a period, followed by the method name (close), and empty (), since there are not parameters for the close method. So its not hard…we’ve already been doing it.

Now let’s get back to strings. A string is a sequence of characters, and that is how it is represented in the computer. Although we have not done it before, each character in the string can be referenced by its ***index***, that is, it’s location in the string. Python starts this numbering the first location/character with zero and then counts up from there. For instance, here is how I would trace the statement below (here showing the index values)

Name = ‘Rose’

As stated above, individual characters of a String can be accessed by indexing their location. The format for accessing individual characters in the string is by having the index value placed between square brackets after the String variable name:

String\_Variable[index]

For example, the following code will generate the following output:

name = ‘Rose’

print name[0]

ind = 2

print name[2]

Output:

R

S

**Be careful, the one thing you can not do with a String using indexing is to use the index to assign a new value to an individual location of a String. For example, you can NOT do the following**

**name[1] = ‘a’**

**The correct term is to say the Strings are immutable (they can’t be changed).**

**The len Function**

The length function will take a string as a parameter and return the length of the string as an integer. The format for a call is:

len(String\_variable)

A sample use would be:

name = “Rose”

namelength = len(name)

But what exactly is meant by the length? Write and run a small Python program to see for yourself.

*Write a small Python program that will use the len function to display the next to the last letter of a String. What would you do if you wanted to make sure that the string had a “next to the last letter”, before trying to output it?*

**For loop Variation**

Python provides a for loop variation that will access every character of a string in turn. Its format is

for loop\_variable in String\_variable:

this will place each character in turn into the loop\_variable. This loop has the same general concept as the for loop variation we looked at in class to read each line of a file in turn.

***Write a small python program to print out each letter of a String on its own line.***

**Slicing**

Python provides a way to specify a portion of a string variable. This technique is called *slicing* and it is accomplished by specifying a start and end indices as follows:

String\_variable[start:end]

But what do the start and end really mean?

***Write a small Python program starting with the assignment***

***Letters = ‘abcdefgh’***

***Try different values for start and end to see how they really work. What would happens if start is the same as end? What about greater than end? What about if you specify an end that is longer than the string?***

Python also allows some “short hand” notation with slicing. If the start index is omitted, it is assumed the start point is the beginning of the String:

String\_variable[:end]

For example, Letters[:2]. If the end index is omitted, it is assumed that the end point is the end of the String:

String\_variable[start:]

For example, Letters[3:].

If a third parameter is added, it acts as a “step value” and will move ahead by the step value, skipping over parts of the string:

String\_variable[start:end:step\_value]

For example, letters[0:7:2].

***Write a Python program to test each of these variations. Text values to see what would happen in what you may consider to be error choices of indices. How does Python handle these cases?***

Finally, if a negative value is specified as a start or end value, it indicates a location relative to the back of the string (positive values indicating a location relative to the front of the string).

For example, Letters[-3:]

***Write a small Python program to test this variation.***

***Write a small Python program that will ask the user to input the name of a file, with an extension, and your program should output “This is a Python file” or “This is not a Python file” based on the extension of the file name. Be sure that your program will not think that a file named “happy” is a Python program.***

**in and not in Operators**

We are already familiar with the concatenation operator (+) which is used with Strings. Two additional operators are the in and not in operators. These are boolean operators and they take two Strings as operands. The format is

String1 in String 2

*in* will return *true* if String1 is contained in String2, and *false* otherwise. Conversely, *not in* will return *true* if String1 is not contained in String2, and *false* otherwise.

The following code example, shows how in could be used:

Line = ‘catch the dog’

if ‘cat’ in Line:

print ‘cat appears ’

else:

print ‘cat does not appear’

**String Methods**

Strings have a number of methods that can be used to:

* Test their values
* Perform modifications
* Search for substring/replace characters

As specified above, these methods are invoked follows:

String\_variable.method\_name(optional parameters)

Please consult <http://python.about.com/od/pythonstandardlibrary/a/string-methods_2.htm> for a list of methods. Please note that if the description says it “returns” some value, such returns a Boolean, you would use the method as you would any expression of that value, such as:

trythis = ‘abc123’

if trythis.isdigit():

print ‘string only contains all digits’

else:

print ‘string does not contain only digits’

If the description indicates that the method “returns a copy of a string”, it does returns a string as a result, which you can then use as any other string. Be careful, it is not changing the actual string that the method was invoked with. That string will remain unchanged unless you assign the result back to the original string.

For example:

Letters = ‘abc’

Nletters = Letters.upper()

This will assign ‘ABC’ to Nletters, but Letters will remain unchanged, still containing ‘abc’. If you wanted to change the value of Letters to all upper case, you would use the statement(s):

Letters = Letters.upper()

or, using two steps:

Temp = Letters.upper()

Letters=Temp

***Write a python program that will prompt the user for a file name. Make sure that you have already created a file by this name in the same directory as your program with several lines of text in it. After reading in the file name from the user, create a new file with the same name, but extension .out. Your program should to write each line of the specified input to the new output file, but all of the contents of the file should be in uppercase letters.***

For example, if the contents of the original file, named **labtest**, were:

See the dog

Run Down the Street

The file named **labtest.out** should contain

SEE THE DOG

RUN DOWN THE STREET

Make sure to drop this final program in moodle.

Python's built-in string methods are incredibly powerful. As the name implies, each of the following methods are available through class **String**. Every string object is an instance of that class and has these methods available. In the following summary, the string object is referenced as **stringObject** for pedagogical reasons. Where character cases are involved, the treatment of 8-bit strings is locale-dependent.

This is the first of a two-page reference, in which you may find methods of use.

* **stringObject.capitalize():** Returns a copy of the string with its first character capitalised.
* **stringObject.center(width[, fillchar]):** Returns the string centered according to the length of the string indicated as width. By default, padding is done using an empty space, but the exact character may be set by specifying the optional fillchar argument.
* **stringObject.count(sub[, start[, end]]):** Returns the number of occurrences of substring sub in the string which begins at position start and ends at position end.
* **stringObject.decode([encoding[, errors]]):** Decodes the string using the codec registered for encoding. If no encoding is declared, the default string encoding is used. If no error handling scheme is set with errors, Python uses 'strict', and encoding errors thus raise UnicodeErrors. Other error handling schemes are 'ignore', 'replace' and any other name registered via codecs.register\_error (to find out what these are for your Python installation, import codec and call for a directory of the codecs.register\_error function using 'dir(codecs.register\_error)').
* **stringObject.encode([encoding[,errors]]):** Returns an encoded version of the string. As with decode(), above, the default encoding is 'strict' and others are available. These include 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' and any other scheme listed in codecs.register\_error (to find out what these are for your Python installation, import codec and call for a directory of the codecs.register\_error function using 'dir(codecs.register\_error)').
* **stringObject.endswith(suffix[, start[, end]]):** Returns "True" if the string ends with suffix, otherwise it returns "False". The value of suffix may also be a tuple of suffixes to match. The optional start and end values denote the beginning and end points of the match within the string.
* **stringObject.expandtabs([tabsize]):** Returns a copy of the string where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.
* **stringObject.find(sub[, start[, end]]):** Returns the lowest index in the string where the substring sub is found within the slice range of start and end. Returns -1 if sub is not found.
* **stringObject.index(sub[, start[, end]]):** Functions Like find() but raises ValueError if the substring is not found. Start and end positions are interpreted as in slice notation.
* **stringObject.isalnum():** Returns "true" if all characters in the string are alphanumeric and there is at least one character, false otherwise.
* **stringObject.isalpha():** Returns "true" if all characters in the string are alphabetic and there is at least one character, false otherwise.
* **stringObject.isdigit():** Returns true if all the characters in the string are digits. There must be at least one character, otherwise it returns "false".
* **stringObject.islower():** Returns true if all cased characters in the string are lowercase. There must be at least one cased character. It returns false otherwise.
* **stringObject.isspace():** Returns true if there are only whitespace characters in the string. There must be at least one character. It returns false otherwise.
* **stringObject.istitle():** Returns true if the string is a titlecased string and there is at least one character. For example, uppercase characters may only follow uncased characters (i.e., characters that do not have uppercase and lowercase forms). Lowercase characters may follow only cased ones. Returns "false" otherwise.
* **stringObject.isupper():** Tests whether all cased characters in the string are uppercase and requires that there be at least one cased character. Returns "true" if so and "false" otherwise.
* **stringObject.join(seq):** Returns a string which is the concatenation of the strings in the sequence seq. The delimiter of the sequence is the string providing this method.
* **stringObject.ljust(width[, fillchar]):** Returns the string left justified in a string of length width. Padding is done using the specified fillchar (default is a space). The original string is returned if width is less than len(stringObject).
* **stringObject.lower():** Returns a copy of the string converted to lowercase.
* **stringObject.lstrip([chars]):** Returns a copy of the string with leading characters removed. One can use the optional chars argument to specify a set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a prefix; rather, all combinations of its values are stripped when they lead the string:
* >>> ' big string '.lstrip()
* 'big string '
* >>>
* >>> 'Doremifasolatido'.lstrip('dmo')
* 'Doremifasolatido'
* >>>
* >>> 'Doremifasolatido'.lstrip('Dmo')
* 'remifasolatido'

* **stringObject.partition(sep):** Splits the string at the first occurrence of delimiter sep, and returns a tripartite tuple that contains the part before the separator, the separator itself, and the part after the separator. If the separator is not found, this method returns a tripartite tuple that contains the string itself, followed by two empty strings. This method first shipped with Python 2.5.
* **stringObject.replace(old, new[, count]):** Returns a copy of the string with all occurrences of substring old replaced by substitute new. The optional argument count indicates the maximum number of times for the replacement to be affected.
* **stringObject.rfind(sub [,start [,end]]):** Returns the index of the last place in the string where the substring sub is found. The field to be searched may be restricted by using the optional arguments start and end. Optional arguments start and end are interpreted as in slice notation. On failure, this method returns "-1".
* **stringObject.rindex(sub[, start[, end]]):** Like rfind(), but this method raises a ValueError when the substring sub is not found.
* **stringObject.rjust(width[, fillchar]):** Returns the string right justified in a string of length width. As with ljust(), padding is done using the specified fillchar; if undeclared or left blank, the fillchar defaults to a space). The original string is returned if width is less than len(stringObject).
* **stringObject.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. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself. New in version 2.5.
* **stringObject.rsplit([sep [,maxsplit]]):** Returns a list of the words in the string using sep as the delimiter. It does this by processing from the right and splitting on whitespace. If the optional arguement maxsplit is used, it indicates the maximum number of splits that should be done. If the delimiter sep is not specified or "None" is used, any whitespace string is considered a delimiter. Except for splitting from the right, rsplit() behaves much like split().
* **stringObject.rstrip([chars]):** Returns a copy of the string with trailing characters removed. The chars argument is a string specifying the set of characters to be removed. If omitted or None, the chars argument defaults to removing whitespace. The chars argument is not a suffix; rather, all combinations of its values are stripped:
* >>> ' big string '.rstrip()
* ' big string'
* >>> 'Doremifasolatido'.rstrip('dmo')
* 'Doremifasolati'
* >>> 'Doremifasolatido'.rstrip('Dmo')
* 'Doremifasolatid'

* **stringObject.split([sep [,maxsplit]]):** Returns a list of the words in the string using sep as the delimiter string. If the option maxsplit is given, at most that number of splits are done. The list will consequently have at most *maxsplit*+1 parts. Note: If the maxsplit option is not specified, then all possible splits are made. If the delimiter sep is not specified or is "None", several different splitting algorithms are applied. First, whitespace characters (spaces, tabs, newlines, returns, and formfeeds) are stripped from both ends. Then, words are separated by arbitrary length strings of whitespace characters. Consecutive whitespace delimiters are treated as a single delimiter.
* **stringObject.splitlines([keepends]):** Returns a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.
* **stringObject.startswith(prefix[, start[, end]]):** Returns "True" if the string starts with the given prefix, otherwise returns "False". "prefix" can also be a tuple of matches to find. Start and end delineate the beginning and end points within the string for the matching process.
* **stringObject.strip([chars]):** Returns a copy of the string with the leading and trailing characters removed. When the chars argument is used, it represents a set of characters to be removed. If omitted or "None", the chars argument indicates that whitespace is to be removed by default. Once again, the chars argument is not a prefix or suffix, all combinations of its values are stripped.
* >>> ' big string '.strip()
* 'big string'
* >>> 'Doremifasolatido'.strip('dmo')
* 'Doremifasolati'
* >>> 'Doremifasolatido'.strip('Dmo')
* 'remifasolatid'

* **stringObject.swapcase():** Returns a copy of the string with uppercase characters converted to lowercase and vice versa.
* **stringObject.title():** Returns a titlecased version of the string. All words start with uppercase characters, and all remaining (cased) characters are lowercase.
* **stringObject.translate(table[, deletechars]):** Returns a copy of the string where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256 (i.e., cover an entire keymap). Note: For Unicode objects, the translate() method does not accept the optional deletechars argument. Instead, it returns a copy of the string where all characters have been mapped through the given translation table, mapping Unicode ordinals to Unicode ordinals, Unicode strings or "None". Unmapped characters are left untouched. Characters mapped to "None" are deleted.
* **stringObject.upper():** Returns a copy of the string converted to uppercase.
* **stringObject.zfill(width):** Returns the numeric string left filled with zeros in a string of length width. The original string is returned if width is less than len(stringObject).