Strings in Python

Use of String Variables

We have already seen strings, but since we've been introduced to loops and index variables, we can learn a bit more about manipulating strings. The most natural way to initialize a string variable is through the input statement:

```
name = input("Please enter your name.\n")
```

In this example, whatever the user enters will be stored in name.

Once we have a string stored in a variable, there are a number of ways to access parts of the string, check the string for letters or manipulate the string.

Checking to see if a letter is in a string

Python allows for a very simple method to check to see if an letter or any other character for that matter is in the string, using the in operator:

```
name = input("Please enter your name.\n")

if 't' in name:
    print("You have an t in your name.")
```

This operation works as follows:

```
<character> in <string>
```

This expression is a Boolean expression that will evaluate to True if the specified character is in the string, and false otherwise. Not only will this work to check for a character, but it will also work to check for a substring, or a consecutive sequence of characters in a larger string. For example, the expression `'put' in 'computer'` evaluates to true since the fourth, fifth and sixth letters of `'computer'` are `'put'`.

Indexing into a String – Non-negative Indexes

A common operation with a string is to access one character in a string. This can be done with square brackets. If name is a string, then `name[0]` represents the first character in the string, `name[1]` represents the second character in the string name, and so on. In addition, to find the length of a string, we use the `len` function, which will return the total number of characters in the string. Thus, a method to determine if a character is in a string from first principles is as follows:

```
name = input("Please enter your name.\n")

flag = False
for i in range(len(name)):
    if name[i] == 't':
        flag = True

if flag:
    print("You have an t in your name.")
```

We can easily use indexing to count the number of times a particular character appears in a string. Here is the previous segment of code edited to print out the number of times 't' appears in the string entered by the user:

```
name = input("Please enter your name.\n")

count = 0
for i in range(len(name)):
    if name[i] == 't':
        count = count + 1

print("You have an t in your name", count, "times.")
```

Indexing into a String – Negative Indexes

Python also allows negative indexes into a string, which is a feature many other languages do not support. If you give a negative integer as an index to a string, Python will start counting from the end of the string. For example, here are the corresponding indexes for the string hello:

index	-5	-4	-3	-2	-1
string	'h'	'e'	'l'	'l'	'o'

Though you are not required to use negative indexes, they can come in handy sometimes, if you want to look for a character a specified number of positions from the end of a string. Without negative indexing, you would have to do the math on your own, using the len function and subtracting.

Here is a simple example where negative indexing simplifies a program. In the following program we will ask the user to enter a string with only uppercase letters and we will determine whether or not the string is a palindrome. A palindrome is a string that reads the same forwards and backwards.

The key strategy here will be to maintain two indexes: one from the front, counting from 0 and one from the back counting backwards from -1. We want to check to see if corresponding characters from the front and back match. If we find a mismatch, we immediately know our string is not a palindrome. We can stop halfway through the string. Remember we must use integer division to determine the point since the range function only takes in integers.

```
def main():

    word = input("Please enter a string, uppercase letters only.\n")

    back = -1
    isPal = True
    for i in range(len(word)//2):
        if word[i] != word[back]:
            isPal = False
            break
        back = back - 1

    if isPal:
        print(word,"is a palindrome.")
    else:
        print(word,"is not a palindrome.")

main()
```

Slicing a String

Slicing refers to obtaining a substring of a given string. An explicit way to denote a substring is to give both its starting index and ending index. In Python, just as we saw with the range function, the ending value is not included in the set of values described in a slice. Thus, the starting index is inclusive, while the ending index is exclusive. Given that the string word was set to “hello”, the slice word[2:4] would be “ll” and the slice word[1:2] would simply be “e”.

We can slice strings with negative indexes as well. This IDLE transcript should clarify the rules for slicing where both indexes are specified:

```
>>> word = "PYTHONISCOOL"
>>> print(word[2:7])
THONI
>>> print(word[6:6])

>>> print(word[4:2])

>>> print(word[5:11])
NISCOO
>>> print(word[8:12])
COOL
>>>
>>> print(word[-7:-2])
NISCO
>>> print(word[-9:8])
HONIS
>>> print(word[-3:-7])

>>>
```

Notice if you attempt to slice a string where the starting index refers to a position that occurs at or after the ending index, the slice is the empty string, containing no characters.

A string can also be sliced using only one index. If only one index is given, Python must know if it's the start or end of the slice. It assumes that the omitted index must refer to the beginning or end of the string, accordingly. These examples should clarify slicing using only one index:

```
>>> print(word[8:])
COOL
>>> print(word[-6:])
ISCOOL
>>> print(word[:6])
PYTHON
>>> print(word[:-4])
PYTHONIS
>>>
```

String Concatenation

We have briefly seen string concatenation before. It is accomplished via the plus sign (+). If two strings are "added", the result is sticking the first string followed by the second string. This is helpful in printing and especially in the input statement, which only takes a single string as a parameter. Here is a short example that utilizes string concatenation:

```
first = input("Please enter your first name.\n")
last = input("Please enter your last name.\n")
full = first+" "+last
print(full)
```

In order for python to recognize that you want string concatenation, BOTH operands must be strings. For example, if you try to do "hello" + 7, you will get a syntax error since strings and integers are not allowed to be added or concatenated.

Pig Latin Example

Pig Latin is a common children's code used to (sort of) hide the meaning of what is being said. There are many variations, but the variation implemented here will use the following rules[1]:

1) For words that start with a consonant and have a vowel, take the first consonant cluster, cut and paste it to the end of the word and add "ay" after it.

2) For words that start with a vowel, add "way" after it.

3) For words with no vowels, keep them as is, since they are probably difficult enough to understand![2]

Our first task will be to find the index of the first vowel, if it exists. We must be careful to not commit an index out of bounds error, where we attempt to index the string at an invalid index. We do this by first checking to see if the index is within the appropriate bounds before indexing the string with it. Short-circuiting allows us to do this in one check. Namely, if the first part of a Boolean expression with an and is False, then Python will NOT evaluate the second portion of the expression at all.

Once we identify this index, we split our work into the three cases outlined above, using slicing and string concatenation appropriately.

[1] The first two rules are a simplification of what is posted on Wikipedia.
[2] This rule is my own incarnation so that our program distinguishes between words with and without vowels.

Here is the program in its entirety:

```
def main():

    VOWELS = "AEIOU"
    ans = "yes"

    # Allow multiple cases.
    while ans == "yes":

        mystr = input("Enter a word, all uppercase letters.\n")

        # Pig Latin Rule - Find first vowel
        index = 0
        while index < len(mystr) and (not mystr[index] in VOWELS):
            index = index+1

        # Just add "way" to words that start with a vowel.
        if index == 0:
            print(mystr+"WAY")

        # Move first consonant cluster to end and add "ay"
        elif index < len(mystr):
            print(mystr[index:]+mystr[:index]+"AY")

        # If there are no vowels, just keep it as is!!!
        else:
            print(mystr)

        ans = input("Would you like to translate another word?\n")

main()
```

The outer loop allows the user to test multiple words. The inner loop finds the first index storing a vowel. Note that we could have created a string with consonants to see if mystr[index] was in it, but since it was shorter to type out the vowels, this route was chosen and the not operator was used. In each of the three cases, the appropriate slicing and concatenating is done to produce the output string. Notice that in the second case, the slicing works out nicely, since we are able to use the same index in specifying both slices to “reorganize” the word, so to speak.

Build in String Methods

A list of built in string methods can be found here:

https://www.tutorialspoint.com/python/python_strings.htm

This list is fairly extensive. Here is a short example that deals with various ways to capitalize (or not) a string:

```
word = input("Enter a word.")
capVersion = word.capitalize()
upperVersion = word.upper()
lowerVersion = word.lower()
print(word,capVersion,upperVersion,lowerVersion)
```

If we enter "hello", then this is what gets printed:

```
hello Hello HELLO hello
```

Each of the function calls creates a new string, leaving the string referenced by word unchanged. The three lines of code after reading in the string assign the references capVersion, upperVersion and lowerVersion to the three new strings that are created.

Another useful built in function is the find function. This function finds the first index (if it exists) that a substring exists in another string. For example, if

```
phrase = "mississippi"
search = "iss"
```

then

```
loc = phrase.find(search)
```

would set loc to 1, since the first occurrence of "iss" in "mississippi" starts at index 1.

In addition, if you want to start searching for the substring from a particular point in the first string, then you can add a second parameter indicating where you want the search to start. For example,

```
loc = phrase.find(search, 2)
```

would set loc to 4, the first index 2 or greater with the substring "iss" in "mississippi".

The best way to learn these functions is to try them out in the IDLE interpreter window!