Lecture 7: More Strings and Sequences*

Today we'll talk more about **strings** and **sequences** and how to use Python's built-in functions on them to derive new sequences. We will also cover the **sorted function** and **format printing** today.

Operations with sequences

Recall that in Python, a sequence, is a series of items for which the relative order to one-another matters. A sequence is the parent class for strings, lists, and ranges. This way, all of these classes share their behavior (in terms of what operations can be applied to them), but they also have differences, which we will be explore in the coming lectures.

[1]:
```
word = "Williams"  # our string word
digits = [1, 2, 3, 4]  # our list digits
digRange = range(1, 5)  # our range object digRange
```

[2]:
```
word
```

[2]:
```
'Williams'
```

[3]:
```
digits
```

[3]:
```
[1, 2, 3, 4]
```

[4]:
```
digRange
```

[4]:
```
range(1, 5)
```

Indexing. We can access an element by using the `[ ]` operator and a number that is the index of the element in the sequence.

[5]:
```
word[2]
```

[5]:
```
'l'
```

[6]:
```
digits[2]
```

[6]:
```
3
```

[7]:
```
digRange[2]
```

*Acknowledgement. This notebook has been adapted from the Wellesley CS111 Spring 2019 course materials (http://cs111.wellesley.edu/spring19).

[7]: 3

Finding length. Because sequences consist of zero or more items, we can use `len` to find how many items they contain.

[8]:
```
len(word)
```

[8]: 8

[9]:
```
len(digits)
```

[9]: 4

[10]:
```
len(digRange)
```

[10]: 4

Concatenation: Sequences can be concatenated by using the operators '+' or '*'

[11]:
```
word + " Globe"
```

[11]: 'Williams Globe'

[12]:
```
digits + [4]
```

[12]: [1, 2, 3, 4, 4]

Note. Concatenation is not supported for range objects.

[13]:
```
digRange + range(4)
```

```
TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

[14]:
```
(word + ' ') * 3
```

[14]: 'Williams Williams Williams '

Membership operator `in`**:** this operator returns True when an item is part of the sequence, and False otherwise

[15]:
```
'w' in word  #case sentitive
```

[15]: False

[16]:
```
'W' in word
```

[16]: True

[17]:
```
'iams' in word  # can be used for substrings
```

[17]:
```
True
```

[18]:
```
1 in digits
```

[18]:
```
True
```

[19]:
```
5 in digRange
```

[19]:
```
False
```

Slicing. This operation uses two indices (a start and a stop) to create a subsequence of items between the two indices.

[20]:
```
word[1:4]
```

[20]:
```
'ill'
```

[21]:
```
digits[1:4]
```

[21]:
```
[2, 3, 4]
```

[22]:
```
digRange[1:4]
```

[22]:
```
range(2, 5)
```

Default start and end. If the first index is omitted, the start index is by default 0. If the stop index is omitted it returns the sequence until the end.

Important. If the stop index is greater than the length of the sequence, Python doesn't return an error, it returns the sequence until the end.

[23]:
```
word[:3]  # substring starting at index 0 and ending at 2
```

[23]:
```
'Wil'
```

[24]:
```
word[2:]   # substring starting at index 2 until the end
```

[24]:
```
'lliams'
```

[25]:
```
word[4:100]  # substring starting at index 4 until the end in this case
```

[25]:
```
'iams'
```

[26]:
```
digits[:3]
```

[26]:
```
[1, 2, 3]
```

[27]:
```
digRange[:3]
```

[27]:
```
range(1, 4)
```

Optinal Step Parameter We can use a third parameter, step, with the slicing operator. Step argument tells Python how many characters (or items) to skip over within the range. By default the step is set to 0 and no items are skipped over.

[28]:
```
word[0:6:2]
```

[28]:
```
'Wli'
```

[29]:
```
digits[0:5:2]
```

[29]:
```
[1, 3]
```

[30]:
```
digRange[0:5:2]
```

[30]:
```
range(1, 5, 2)
```

We can omit the stop argument as before, and Python automatically will look until the end of the sequence.

[31]:
```
word[0::2]
```

[31]:
```
'Wlim'
```

Reversing through slicing. Because Python allows negative indexing, by using step -1, we can reverse a sequence!

[32]:
```
word[::-1]  #reverse string
```

[32]:
```
'smailliW'
```

[33]:
```
digits[::-1]
```

[33]:
```
[4, 3, 2, 1]
```

[34]:
```
digRange[::-1]
```

[34]:
```
range(4, 0, -1)
```

Question. How would I generate the string `mail` from the word `Williams`? How about `small`?

[35]:
```
word[-2:-6:-1]      # expression for "mail"
```

[35]:
```
'mail'
```

[36]:
```
word[-1:-4:-1] + word[2:4]           # expression for "small"
```

[36]:
```
'small'
```

String specific methods

The following functions are specific to strings.

- `myString.replace('str1', 'str2')`: returns a new string that has occurence of `str1` in `myString` replaced with `str2`
- `myString.upper()`: returns a new string which is `myString` converted to uppercase
- `myString.lower()`: returns a new string which is `myString` converted to lowercase
- `myString.isalpha()`: returns true if all characters in the string are alphabet, false otherwise
- `myString.isspace()`: returns true if there are only whitespace characters in the string, false otherwise

[37]:
```
myString = 'Williams College'
```

[38]:
```
myString.replace('iams', 'eslley')
```

[38]:
```
'Willeslley College'
```

[39]:
```
myString.replace('tent', 'eslley')  #what if `str1` is not a substring?
```

[39]:
```
'Williams College'
```

[40]:
```
upperCase = myString.upper()
upperCase
```

[40]:
```
'WILLIAMS COLLEGE'
```

[41]:
```
lowerCase = myString.lower()
lowerCase
```

[41]:
```
'williams college'
```

[42]:
```
myString # notice myString does not change with these operations
```

[42]:
```
'Williams College'
```

[43]:
```
myEmail = "shikha@cs.williams.edu"
```

[44]:
```
myEmail.isalpha()
```

[44]:
```
False
```

[45]:
```
hspace = ' '
vspace = '\n'
```

[46]:
```
hspace.isspace()
```

[46]:
```
True
```

[47]:
```
vspace.isspace()
```

[47]:
```
True
```

Converting strings into lists

We can create a list from a string in several different ways.

- Using the `list` function on a string returns a list of all its characters
- Invoking the .split() function on a string creates a list of words (which were separated by spaces in the string)
- Sorting a string using the sorted function converts it into a list of the sorted sequence

Let us take examples of each of the above one by one.

List function on a string. Converts the string into a list of all its characters.

[48]:
```
word = 'Williams'
```

[49]:
```
list(word)
```

[49]:
```
['W', 'i', 'l', 'l', 'i', 'a', 'm', 's']
```

[50]:
```
list('Commas, and spaces. ')  # string with punctuations and spaces
```

[50]:
```
['C',
 'o',
 'm',
 'm',
 'a',
 's',
 ',',
 ' ',
 'a',
 'n',
 'd',
 ' ',
 's',
 'p',
 'a',
 'c',
 'e',
 's',
 '.',
 ' ']
```

Split function. Splits a string (default at the space characters) and returns a list of those words. This is commonly use to split a sentence into a list of words.

[51]:
```
phrase = "New England's weather is unpredictable."
phrase.split()
```

[51]:
```
['New', "England's", 'weather', 'is', 'unpredictable.']
```

Optional arguments given to split. When the `split` method doesn't take arguments, it splits by default at the white space. If needed, you can split at some other character.

[52]:
```
names = "Shikha, Hanna, Chris, Lauren, Jacob, Aamir"
listOfNames = names.split(',')
```

Notice how the character "," was removed and the string was split exactly where the "," was.

[53]:
```
listOfNames  # notice the spaces around the names
```

[53]:
```
['Shikha', ' Hanna', ' Chris', ' Lauren', ' Jacob', ' Aamir']
```

Question. How would we remove the spaces? Which function is useful for that?

[54]:
```
newNameList = []
for name in listOfNames:
    newNameList.append(name.strip())
```

[55]:
```
newNameList = [name.strip() for name in listOfNames]  # can write it very␣
↪succintly (List Comprehensions)
```

Strip function. `myString.strip()` removes all leading and trailing spaces from `myString`.

[56]:
```
newNameList  # new name list with no spaces
```

[56]:
```
['Shikha', 'Hanna', 'Chris', 'Lauren', 'Jacob', 'Aamir']
```

Question. Given an email address find the domain name or username

[57]:
```
myEmail = "shikha@cs.williams.edu"  # find the orgnanization name
```

[58]:
```
myEmail.split('@')[-1]  # domain name
```

[58]:
```
'cs.williams.edu'
```

[59]:
```
myEmail.split('@')[0]   # user name
```

[59]:
```
'shikha'
```

Converting lists of strings to strings

If you have a list of strings, you can "join" them together in a string using Python's join method. It works as follows.

[60]:
```
' '.join(newNameList)  # join with a space
```

[60]:
```
'Shikha Hanna Chris Lauren Jacob Aamir'
```

[61]:
```
', '.join(newNameList)  # join with a comma and a space
```

[61]:
```
'Shikha, Hanna, Chris, Lauren, Jacob, Aamir'
```

[62]:
```
'*'.join(newNameList) # join with a *
```

[62]:
```
'Shikha*Hanna*Chris*Lauren*Jacob*Aamir'
```

Sorting Sequences with the `sorted` function

The built-in function `sorted` can be applied to sequences and always returns a new list.

[63]:
```
numbers = [35, -2, 17, -9, 0, 12, 19]
sorted(numbers)
```

[63]:
```
[-9, -2, 0, 12, 17, 19, 35]
```

Notice that `numbers` has not changed.

[64]:
```
numbers
```

[64]:
```
[35, -2, 17, -9, 0, 12, 19]
```

By default the list is sorted in the **ascending** order, but we can easily reverse the order:

[65]:
```
sorted(numbers, reverse=True)
```

[65]:
```
[35, 19, 17, 12, 0, -2, -9]
```

Sorting other sequences

Strings can also be sorted in the same way. The result is **always** going to be a new list.

[66]:
```
phrase = 'Red Code 1'
sorted(phrase)
```

[66]:
```
[' ', ' ', '1', 'C', 'R', 'd', 'd', 'e', 'e', 'o']
```

Question: Why do we see a space as the first element in the sorted list?

Recall. How does this comparison of string values work? Because the computer doesn't know anything about letters, it converts everything into numbers. Each character has a numerical code that is summarized in this http://www.asciitable.com/.

We can also look up the ASCII code via the Python built-in function `ord`:

[67]:
```
ord(' ')
```

[67]: 32

[68]:
```
ord('A')
```

[68]: 65

ASCII value to char: You can also use the `chr` function to find out which character is given a particular ASCII value.

[69]:
```
chr(55)
```

[69]: '7'

[70]:
```
chr(100)
```

[70]: 'd'

Format printing. To print the ascii code for every character in a `for`, we use the `format` string method below.

- `s.format(*args)` method: `*args` means: zero or more arguments: format method takes zero or more arguments

[71]:
```
for item in sorted(phrase):
    print("'{}' has ASCII code {}".format(item, ord(item)))  # format printing
```

```
' ' has ASCII code 32
' ' has ASCII code 32
'1' has ASCII code 49
'C' has ASCII code 67
'R' has ASCII code 82
'd' has ASCII code 100
'd' has ASCII code 100
'e' has ASCII code 101
'e' has ASCII code 101
'o' has ASCII code 111
```

Just as in the case of the list `numbers` in the above example, the string value of `phrase` hasn't changed:

[72]:
```
phrase
```

[72]:
```
'Red Code 1'
```

Getting a sorted string. When we use the sorted method, it sorts the string but returns a list. How do we turn the sorted list back to a string to get a sorted string?

[73]:
```
sortedPhraseList = sorted(phrase)
```

[74]:
```
sortedPhraseList
```

[74]:
```
[' ', ' ', '1', 'C', 'R', 'd', 'd', 'e', 'e', 'o']
```

[75]:
```
''.join(sortedPhraseList)
```

[75]:
```
'  1CRddeeo'
```

Question. What if we wanted to remove the spaces from the sorted string?

[76]:
```
''.join(sortedPhraseList).strip()  # use the strip method!
```

[76]:
```
'1CRddeeo'
```

Sorting a list of sequences

We can sort list of sequences such as list of strings and list of lists.

[77]:
```
# a long string that we will split into a list of words
phrase = "99 red balloons *floating* in the Summer sky"
words = phrase.split()
words
```

[77]:
```
['99', 'red', 'balloons', '*floating*', 'in', 'the', 'Summer', 'sky']
```

[78]:
```
sorted(words)
```

[78]:
```
['*floating*', '99', 'Summer', 'balloons', 'in', 'red', 'sky', 'the']
```

Question: Can you explain the results of sorting here? What rules are in place?

[79]:
```
sorted(words, reverse=True)
```

[79]:
```
['the', 'sky', 'red', 'in', 'balloons', 'Summer', '99', '*floating*']
```

Remember, the original list is unchanged:

[80]:
```
words
```

[80]:
```
['99', 'red', 'balloons', '*floating*', 'in', 'the', 'Summer', 'sky']
```

Sorting lists in place

Python list objects have their methods to sort a list **in place** (i.e., by **mutating the existing list**, not returning a new list):

[81]:
```
numbers = [35, -2, 17, -9, 0, 12, 19]
```

[82]:
```
numbers.sort()
```

Notice that nothing was returned and `numbers` list has now changed.

[83]:
```
numbers
```

[83]:
```
[-9, -2, 0, 12, 17, 19, 35]
```

Formatting Strings and Format Printing

We can also print elements of a list using format printing.

- Given list, `myList`, then `*myList` means put the elements of `myList` in as arguments
- For every pair of braces ({}), format consumes one argument.
- The argument is converted to a string (with `str`) and catenated with the remaining parts of the format string.
- If, in the braces, we include a position, that indicates which argument you wish to use

[84]:
```
"Hello, you {} world{}".format("silly",'!')  # creates a new string
```

[84]:
```
'Hello, you silly world!'
```

[85]:
```
print("Hello, {}.".format("you silly world!"))
```

```
Hello, you silly world!.
```

[86]:
```
 myList = ['you', 'silly', 'world!']
```

[87]:
```
print(*myList)  # note the resulting spaces
```

```
you silly world!
```

[88]:
```
print('Hello, {} {} {}'.format(*myList))
```

```
Hello, you silly world!
```

[89]:
```
print("Hello, {1} {2} {0}".format('you','silly','world!'))
# notice the indices in {}
```

```
Hello, silly world! you
```

Summary. Format printing allows us a lot of flexibility in printing and works well with lists as well.