Exploration: String Manipulation

in, not in

There are operators called **in** and **not in** that can be used to check whether or not a string contains another string. For example:

> "ow" in "chowder"  
> "ow" not in "chowder"  
> animal = "anteater"  
> "ant" in animal  
> bug = "aphid"  
> bug in animal

find()

If you need to know where a string occurs inside another string, you can use the find() method, which returns the leftmost index of the substring within the other string:

> animal = "elephant"  
> index = animal.find("ant") #index would now equal 5

If the string does not appear as a substring of the other word, then find() returns -1. You can also search within a specified section of the string:

> animal = "elephan"  
> index = animal.find("ant", 2, 7) #index now equals -1, since "ant" doesn't occur in "ephan"

The end index works the same as for ranges - the section being searched extends up to, but does not include, the end index.

len()

There is a function called len that returns the length of a string:

> len("hokey pokey")  
> city = "Charleston"  
> len(city)

upper() and lower()

You can use the upper() and lower() methods to get an upper-case or lower-case version of a string.

> "slithy".upper()  
> var = "TOVES"  
> var.lower()

Operators vs. functions vs. methods

Notice the different syntax we've used for *in* and *not in*, *len*, and *upper* and *lower*. The *in* and *not in* **operators** are used between two values (the operands) in a format similar to arithmetic operators, like \* and /. These arithmetic operators each have two numeric operands. However in the examples for *in* and *not in*, we used strings for both operands.

The *len* **function** is called in the same way we've called other functions - we first give the name of the function, and then list inside parentheses any values we're passing to the function. In this case we're passing a string to the function.

The *find*, *upper,* and *lower* **methods** are called using the dot notation syntax that we use with objects. That's because strings (like everything else in Python) are objects that have certain methods defined for them. In this case, we don't have to pass the string as an argument. You can think of it as politely asking the string if it would be so kind as to carry out the requested operation. (The string is implicitly passed to the method as the *self* argument.)

Indexing

We can get the character at a certain position, or **index**, in a string by indexing into the string. Imagine numbering the letters of a string in order, **starting with zero**. So in the string "onyx", the "o" is at index zero, the "n" is at index 1, the "y" is at index 2, and the "x" is at index 3. **Notice that the index of the last character is the length of the string minus 1.** We index into a string by following it with square brackets containing the index that we want to access.

> gem[2]  
> gem[0]  
> gem[len(gem)-1]

We can also use negative indices, in which case we count from the back of the string. The character at index -1 would be the last character in the string, the one at -2 would be the next-to-last character, etc.

> dino[-1]  
> dino[-12]

Slicing

"Slicing" a string gives you back part of that string, based on the indices for the start and end of the part you want.

> fruit[2:5]

Notice that the substring you get back starts at the first index, and goes up to, **but does not include** the second index (similar to making a range). If you omit the first index, the substring starts at the first character. If you omit the second index, the substring ends at the last character.

> hero[:6]  
> hero[6:]

The character at index 6 (the seventh character) doesn't get included in the first slice, but does in the second slice. If you omit both indices:

> hero[:]

then you get a copy of the whole string.

We can also specify the "stride" for a slice, which allows us to get a slice that has every other character (a stride of 2), every third character (a stride of 3), etc.

> poem[::2]

Because the first and second index were omitted, they default to the start and end of the string, respectively. The stride of 2 makes it so only every 2nd letter is included. The stride can also be negative, which makes us count backwards in the string, in which case the first index needs to be larger than the second. We can reverse a string like this (when the stride is negative, the first index will default to the end of the string and the second index will default to the beginning):

> poem[::-1]

Comparison operators

We can use the normal comparison operators with strings, where if it's true that A < B, then that means A comes before B in lexicographic order (dictionary order).

> "zebra" < "aardvark"  
> "zebra" > "aardvark"  
> word = "catharsis"  
> "cat" == word[:3]

However, there's an important caveat: because characters are encoded as numbers, and because upper-case letters have lower numbers than lower-case letters, Python will say that upper-case letters come before lower-case letters.

> "Zebra" > "aardvark"

One way to handle this is to convert the strings to upper-case (or lower-case) before comparing them.

> "Zebra".upper() > "aardvark".upper()

Exercises

(See the module overview for a link to example solutions.)

1. Write a function called *last\_char*that takes a string parameter and returns the last letter of that string.

Sample input: "Understand well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand"  
Expected output: "d"

2. Write a function called *midstring* that takes a string parameter and returns a copy of that string minus its first and last letters. If the string passed in has only has 1 or 2 letters, the function should return the empty string "".

Sample input: "Understand well as I may my comprehension can only be an infinitesimal fraction of all I want to understand"  
Expected output: "nderstand well as I may, my comprehension can only be an infinitesimal fraction of all I want to understan"

3. Write a function called *sort\_two\_strings* that takes two string parameters and returns a single string of both of them in dictionary order, ignoring case. For example, if the strings "aardvark" and "Zebra" are passed, it should return "aardvark Zebra".

Sample input: "Lovelace", "Byron"  
Expected output: "Byron Lovelace"

4. A palindrome is a string that reads the same forward or backward. Write a function called *is\_pal* that takes a string parameter and returns True if that string is a palindrome, but returns False otherwise.

Sample input: tacocat  
Expected output: True

Exploration: Lists

Introduction

A list is an ordered collection of objects of any type. You can have lists of floats, strings, objects made from classes you've defined, etc. You can even have lists contain other lists. You're allowed to mix the types of objects in a list, for example you can have a list that contains both integers and strings.

Most lists contain multiple values, but we can have lists of one or zero values, which can be useful. Here are a few lists for you to interact with.

len

With lists, len() returns the number of elements in the list:

> len(["Mary","had","a","little","lamb"])  
> count = [1,2,3,4,5]  
> len(count)

Indexing and slicing

Indexing and slicing work the same with lists as they do with strings. For example, try entering these commands by typing them to the right of the little red arrow above. (Enter one line at a time.)

> some\_primes[0]  
> some\_primes[0:10:2]  
> some\_names[::-2]

When you index into a nested list to get a sublist, you can then index into that list. Try entering the following to get the list within some\_stuff, then the item 'Tollbooth' within that sublist, and then its first character:

> some\_stuff[3]  
> some\_stuff[3][1]  
> some\_stuff[3][1][0]

in, not in

Also like strings, we can use *in* and *not in*. With these two operators, the second operand can be of any iterable type, which includes both strings and lists, and the first operand can be of any type at all, including iterable types. Try these examples to see for yourself:

> 13 in some\_primes  
> 13 not in some\_primes  
> "Fido" in some\_stuff  
> "Phantom" in some\_stuff  
> "Phantom" in some\_stuff[3]

What happened with those last two examples? The string "Phantom" is not in *some\_stuff* - it's in a list that's in *some\_stuff*. That list is at index 3, so we were able to find it there.

Let's look at some more things we can do, with new list examples.

Min and max, sort, sorted

There are min and max functions we can use. Try these:

> min(odds)  
> max(palindromes)

The min and max functions wouldn't make sense for things like *some\_stuff* in the first set of list examples, since Python doesn't know how to compare the different types in that list. There's also a sort function we can use.

> evens.sort()  
> palindromes.sort()

You'll notice that nothing prints out when you try those. But now look at the lists again:

> evens  
> palindromes

The sort function also wouldn't make sense for *some\_stuff* - again because Python doesn't know how to compare the different types in that list. (Try it and see!)

There's also a function called sorted(). Unlike the sort() method, the sorted() function **returns** a sorted copy without changing the original list:

> sorted\_odds = sorted(odds)

Concatenation

We can concatenate lists with the + operator.

> evens + odds

Here's one more list example for us to practice on.

Iterating through a list

Lists are iterable, so we can use a for loop to access each element. For example, this loop prints each element in the list fun\_floats:

for number in fun\_floats:  
 print(number)

The following loop prints out the total of the values in the list:

total = 0  
for number in fun\_floats:  
 total += number  
print(total)

**Common mistake:** some beginning students might write the above loop like this:

total = 0  
for index in range(len(fun\_floats)):  
 total += fun\_floats[index]  
print(total)

This will work correctly, but it's more complicated than necessary. Don't use a range to loop over the indices unless you actually need the indices. If you do need the indices, another possible way to get them is to use **enumerate()**. This allows us to have two loop variables - one of which takes the value of each index, while the other one takes the values of each element. For example:

for index, number in enumerate(fun\_floats):  
 print(index, number)

Iterating through multiple lists in parallel

If you have multiple lists of the same size and want to iterate through them at once, in parallel, then you do need to use indices. In this case, we could either use a range or enumerate(). This example uses a range to print out each food item with its corresponding unit price:

for index in range(len(students))  
 print(food\_items[index], unit\_prices[index])

Splitting a string

You can split a string into a list of tokens using the string split() method. For example:

> sentence = "Not the comfy chair!"  
> word\_list = sentence.split()

word\_list is now this list: ["Not", "the", "comfy", "chair!"]. The default is that white space (spaces, tabs, and newlines) is the separator, but you can specify other separators:

> sentence = "The cat, a stray tabby, climbed in the window, tail twitching."  
> word\_list = sentence.split(", ")

Now word\_list is: ["The cat", "a stray tabby", "climbed in the window", "tail twitching."]

Lists of objects

We can make a list of BankAccount objects (which we defined in Module 5) like this:

account\_1 = BankAccount("235349", 730.29)  
account\_2 = BankAccount("783848", 240.89)  
account\_3 = BankAccount("732005", 1390.20)  
account\_list = [account\_1,account\_2,account\_3]

What if we want to access the balance of the first account in the list?  We can do that like this:

account\_list[0].get\_balance()

Where "account\_list[0]" gives us a BankAccount object and ".get\_balance()" returns the balance of that object.

List comprehensions

List comprehensions are a concise way to construct a new list by applying some transformation to an existing list (or other iterable type). For example, the following code creates a new list whose elements are double the elements in *fun\_floats*:

> fun\_floats\_doubled = [2 \* n for n in fun\_floats]

Here's a similar example that works from a range instead of a list:

> [2\*x for x in range(1,11)]

We can optionally filter out certain values from the original list (or other iterable).

> [2\*x for x in range(1,11) if x % 2 == 1]

In this example, the original iterable was a range. If a value in that range is odd (the remainder of dividing by 2 is 1), then we apply the transformation (multiplying by 2). Note that values are filtered out **before** the transformation is applied. If we had doubled the numbers and then filtered out the even ones, then the new list would have been empty.

We can also use a list comprehension to filter without applying a transformation:

> nums = [1,2,3,4,5,6,7,8,9]  
> [x for x in nums if x % 3 == 0]

Don't let this new use of the **for** and **if** keywords confuse you. List comprehensions are a separate thing from for loops and if statements.

Exercises

(See the module overview for a link to example solutions.)

1. Write a function named *every\_other* that takes as a parameter a list and returns a list that only contains every other element starting with the first one. For example, if the original list is [7, "joe", "apple", 9.81, False], then the new list should be [7, "apple", False]. Use slicing.

Sample input: ["parsley", "sage", "rosemary", "thyme"]  
Expected output: ["parsley", "rosemary"]

2. Write a function named *array\_sum* that takes as a parameter a list of strings and returns the total number of characters in all the strings.

Sample input: ["Wayne", "Katie", "Daryl", "Dan"]  
Expected output: 18

3. Write a function named rev\_string\_list that takes as a parameter a list of strings and returns a list that contains the reverse of each of those strings. Use a list comprehension.

Sample input: ["Wayne", "Katie", "Daryl", "Dan"]  
Expected output: ['enyaW', 'eitaK', 'lyraD', 'naD']

4. Write a function named *contain\_string* that takes as a parameter a list of strings and the target string, and returns a list of the strings from the original list that contain the target string. Use a list comprehension.

Sample input: ['cats', 'tacks', 'scat', 'stack'], 'cat'  
Expected output: ['cats', 'scat']

Code samples

* [List comprehension with strings and numbers and comparison with using a for loopLinks to an external site.](https://pythontutor.com/visualize.html#code=%23Create%20a%20new%20list%20using%20list%20comprehension%20from%20an%20existing%20list%20that%20stores%0A%23the%20squares%20of%20the%20numbers%20from%20the%20original%20list%0Anums%20%3D%20%5B1,2,3,4,5,6,7,8,9%5D%0Anew_nums%20%3D%20%5Bx**2%20for%20x%20in%20nums%5D%0A%0A%23The%20above%20list%20comprehension%20is%20similar%20to%20writing%20this%20for%20loop%0A%23new_list%20%3D%20%5B%5D%0A%23for%20n%20in%20nums%3A%0A%20%20%20%20%23new_list.append%28n**2%29%0A%0A%23...same%20as%20above%20but%20only%20only%20if%20those%20numbers%20are%20not%201%20or%209%0Aanother_new_nums%20%3D%20%5Bx**2%20for%20x%20in%20nums%20if%20x%20!%3D%201%20and%20x%20!%3D%209%5D%0A%0A%23The%20above%20list%20comprehension%20is%20similar%20to%20writing%20this%20for%20loop%0A%23new_list%20%3D%20%5B%5D%0A%23for%20n%20in%20nums%3A%0A%20%20%20%20%23if%20n%20!%3D%201%20and%20n!%3D9%3A%0A%20%20%20%20%20%20%20%20%23new_list.append%28n**2%29%0A%20%20%20%20%0A%23Create%20a%20new%20list%20using%20list%20comprehension%20that%20%0A%23has%20each%20character%20from%20the%20original%20list%20doubled%0Acharacters%20%3D%20%5B'a','b','c','d','e','f'%5D%0Anew_characters%20%3D%20%5Bchar*2%20for%20char%20in%20characters%5D%0A%0A%23The%20above%20list%20comprehension%20is%20similar%20to%20writing%20this%20for%20loop%0A%23new_list%20%3D%20%5B%5D%0A%23for%20char%20in%20characters%3A%0A%20%20%20%20%23new_list.append%28char*2%29%0A&cumulative=false&heapPrimitives=nevernest&mode=edit&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)
* [len() with a list and using len with ifLinks to an external site.](https://pythontutor.com/visualize.html#code=l%20%3D%20%5B'Mary','had','a','little','lamb'%5D%0Ax%20%3D%20len%28l%29%0Aif%20x%20%3E%203%3A%0A%20%20%20%20print%28%22No%22%29%0Aelse%3A%0A%20%20%20%20print%28%22Yo%22%29%0A&cumulative=false&heapPrimitives=nevernest&mode=edit&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)
* [How a function can call a method on a passed objectLinks to an external site.](https://pythontutor.com/visualize.html#code=class%20Teapot%3A%0A%20%20%20%20def%20__init__%28self%29%3A%0A%20%20%20%20%20%20%20%20pass%0A%20%20%20%20%0A%20%20%20%20def%20boil_water%28self%29%3A%0A%20%20%20%20%20%20%20%20print%28%22Boiling%20water...%22%29%0A%0Adef%20tea_preparation%28my_teapot%29%3A%0A%20%20%20%20my_teapot.boil_water%28%29%0A%20%20%20%20%0A%0At1%20%3D%20Teapot%28%29%0At2%20%3D%20Teapot%28%29%0A%0A%23t1.boil_water%28%29%0A%23t2.boil_water%28%29%0Atea_preparation%28t1%29%0Atea_preparation%28t2%29%0A&cumulative=false&heapPrimitives=nevernest&mode=edit&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)
* [How a function can call methods on each object from a list passed to it OR using for loop with a list of objects passed to a functionLinks to an external site.](https://pythontutor.com/visualize.html#code=class%20Teapot%3A%0A%20%20%20%20def%20__init__%28self%29%3A%0A%20%20%20%20%20%20%20%20pass%0A%20%20%20%20%0A%20%20%20%20def%20boil_water%28self%29%3A%0A%20%20%20%20%20%20%20%20print%28%22Boiling%20water...%22%29%0A%0Adef%20tea_preparation%28my_teapots%29%3A%0A%20%20%20%20for%20teapot%20in%20my_teapots%3A%0A%20%20%20%20%20%20%20%20teapot.boil_water%28%29%0A%20%20%20%20%0A%20%20%20%20%0A%0At1%20%3D%20Teapot%28%29%0At2%20%3D%20Teapot%28%29%0A%0Amy_teapots%20%3D%20%5Bt1,%20t2%5D%0A%23t1.boil_water%28%29%0A%23t2.boil_water%28%29%0Atea_preparation%28my_teapots%29%0A%0A&cumulative=false&heapPrimitives=nevernest&mode=edit&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)
* [Visualization of simplified version of Code sample 1 from Tinker Lab 2Links to an external site.](https://pythontutor.com/visualize.html#code=%23Simplification%20of%20Code%20Sample%201%20from%20https%3A//canvas.oregonstate.edu/courses/1928696/pages/additional-examples-for-module-6-part-2%3Fmodule_item_id%3D22739843%0Adef%20translate_single_word%28word%29%3A%0A%20%20%20%20%22%22%22translates%20a%20single%20English%20word%20to%20Pig%20Latin%22%22%22%0A%20%20%20%20vowels%20%3D%20'aeiou'%0A%20%20%20%20if%20word%5B0%5D%20in%20vowels%3A%0A%20%20%20%20%20%20%20%20return%20word%20%2B%20'ay'%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20index%20%3D%200%0A%20%20%20%20%20%20%20%20length_of_word%20%3D%20len%28word%29%0A%20%20%20%20%20%20%20%20word_index%20%3D%20word%5Bindex%5D%0A%20%20%20%20%20%20%20%20while%20index%20%3C%20length_of_word%20and%20word_index%20not%20in%20vowels%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20index%20%2B%3D%201%0A%20%20%20%20%20%20%20%20word_index_colon%20%3D%20word%5Bindex%3A%5D%0A%20%20%20%20%20%20%20%20word_zero_colon_index%20%3D%20word%5B0%3Aindex%5D%0A%20%20%20%20%20%20%20%20x%20%3D%20word_index_colon%20%0A%20%20%20%20%20%20%20%20x%20%2B%3D%20word_zero_colon_index%20%0A%20%20%20%20%20%20%20%20x%20%2B%3D%20'ay'%0A%20%20%20%20%20%20%20%20return%20x%0Adef%20translate%28text%29%3A%0A%20%20%20%20%22%22%22translates%20English%20text%20to%20Pig%20Latin%22%22%22%0A%20%20%20%20new_text%20%3D%20''%0A%20%20%20%20word_list%20%3D%20text.split%28%29%0A%20%20%20%20for%20word%20in%20word_list%3A%0A%20%20%20%20%20%20%20%20new_text%20%2B%3D%20translate_single_word%28word%29%20%2B%20'%20'%0A%20%20%20%20return%20new_text%0A%0Aprint%28translate%28%22look%20at%20me%20still%20talking%20when%20there's%20science%20to%20do%22%29%29&cumulative=false&heapPrimitives=nevernest&mode=edit&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)

Tools we used

The only tool we used during this Help Session for demo:

* [https://pythontutor.com/python-debugger.html#mode=editLinks to an external site.](https://pythontutor.com/python-debugger.html#mode=edit)
  + To learn how to use this, try and use the different buttons on the site :) You won't break the tool.

How to use these code samples (or pythontutor.com in general)?

* Click on "Visualize execution" and then use the "next" and "prev" button to run the program step by step.
* As you run through the program step by step, you will see variables from your program on the right hand side of the screen changing. Note how each operation affects the variable.
* You can enter input, when prompted to do so, below the code editor window.
* The output appears above the variables.
* Try changing the code using "Edit this code" link. Try different variations by asking questions like "What if I put a different initial value for this variable?", "What if I switch the variables around this operator?", "What if I change the order of lines in this code?"
* I promise you that you won't hurt your computer by doing anything with this tool ;) The code is executed on their server and not your machine.
* If at any point, the code is "broken" or you want to go back to the original version come back to this Canvas page and click on the code link.
* You can use PythonTutor.com for any code samples that you come across in this course or even your own assignment code to debug it.

Questions?

Do one of these:

1. Ask in the Ed Discussions thread for this Help Session.
2. Ask in General channel on Teams
3. Schedule an office hour with me using the info at https://tinyurl.com/hedaoos

Note

Help Sessions or Webinars in this course are neither mandatory nor graded based on attendance or watching. They are completely optional.

More info at  [https://edstem.org/us/courses/31713/discussion/2376398Links to an external site.](https://edstem.org/us/courses/31713/discussion/2376398)