STRING METHODS

Introduction

Do you have a gigantic string that you need to parse for information? Do you need to sanitize a users input to work in a function? Do you need to be able to generate outputs with variable values? All of these things can be accomplished with *string methods*!

Python comes with built-in *string methods* that gives you the power to perform complicated tasks on strings very quickly and efficiently. These string methods allow you to change the case of a string, split a string into many smaller strings, join many small strings together into a larger string, and allow you to neatly combine changing variables with string outputs.

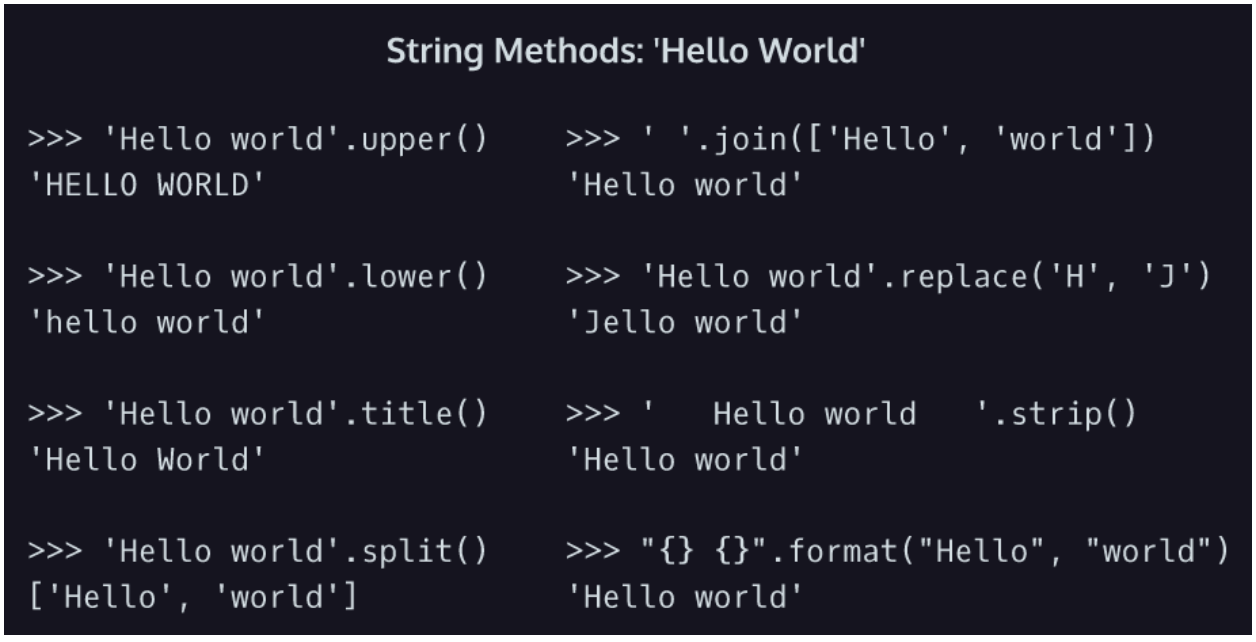
In the previous lesson, you worked with `len()`, which was a *function* that determined the number of characters in a string. This, while similar, was NOT a string method. String methods all have the same syntax:

```
string_name.string_method(arguments)
```

Unlike `len()`, which is called with a string as its argument, a string method is called at the end of a string and each one has its own method specific arguments.

Instructions

The diagram shows all of the string methods you can expect to learn in this lesson. Take a quick look at them and then let's get started!



Formatting Methods

There are three string methods that can change the casing of a string. These are `.lower()`, `.upper()`, and `.title()`.

- `.lower()` returns the string with all lowercase characters.
- `.upper()` returns the string with all uppercase characters.
- `.title()` returns the string in title case, which means the first letter of each word is capitalized.

Here's an example of `.lower()` in action:

```
favorite_song = 'SmOoTH'
favorite_song_lowercase = favorite_song.lower()
print(favorite_song_lowercase)
# => 'smooth'
```

Every character was changed to lowercase! It's important to remember that string methods can only **create** new strings, they do not change the original string.

```
print(favorite_song)
# => 'SmOoTH'
```

See, it's still the same! These string methods are great for sanitizing user input and standardizing the formatting of your strings.

Instructions

1.

You're a programmer working for an organization that is trying to digitize and store poetry called *Preserve the Verse*.

You've been given two strings, the title of a poem and its author, and have been asked to reformat them slightly to fit the conventions of the organization's database.

Make `poem_title` have title case and save it to `poem_title_fixed`.

2.

Print `poem_title` and `poem_title_fixed`.

How did the string change?

3.

The organization's database also needs the author's name to be uppercase only.

Make `poem_author` uppercase and save it to `poem_author_fixed`.

4.
Print `poem_author` and `poem_author_fixed`.

Again, how did the string change?

script.py

```
poem_title = "spring storm"
poem_author = "William Carlos Williams"

poem_title_fixed = poem_title.title()
print(poem_title)
print(poem_title_fixed)

poem_author_fixed = poem_author.upper()
print(poem_author)
print(poem_author_fixed)
```

```
spring storm
Spring Storm
William Carlos Williams
WILLIAM CARLOS WILLIAMS
```

Splitting Strings

`.upper()`, `.lower()`, and `.title()` all are performed on an existing string and produce a string in return. Let's take a look at a string method that returns a different object entirely!

`.split()` is performed on a string, takes one argument, and returns a list of substrings found between the given argument (which in the case of `.split()` is known as the delimiter). The following syntax should be used:

```
string_name.split(delimiter)
```

If you do not provide an argument for `.split()` it will default to splitting at spaces.

For example, consider the following strings:

```
man_its_a_hot_one = "Like seven inches from the midday sun"
print(man_its_a_hot_one.split())
# => ['Like', 'seven', 'inches', 'from', 'the', 'midday', 'sun']
```

`.split` returned a list with each word in the string. Important to note: if we run `.split()` on a string with no spaces, we will get the same string in return.

Instructions

1.

In the code editor is a string of the first line of the poem *Spring Storm* by William Carlos Williams.

Use `.split()` to create a list called `line_one_words` that contains each word in this line of poetry.

script.py

```
line_one = "The sky has given over"


line_one_words = line_one.split()
print(line_one_words)
```

```
['The', 'sky', 'has', 'given', 'over']
```

Splitting Strings II

If we provide an argument for `.split()` we can dictate the character we want our string to be split on. This argument should be provided as a string itself.

Consider the following example:

```
greatest_guitarist = "santana"
print(greatest_guitarist.split('n'))
# => ['sa', 'ta', 'a']
```

We provided `'n'` as the argument for `.split()` so our string “santana” got split at each `'n'` character into a list of three strings.

What do you think happens if we split the same string at `'a'`?

```
print(greatest_guitarist.split('a'))
# => ['s', 'nt', 'n', '']
```

Notice that there is an unexpected extra `''` string in this list. When you split a string on a character that it also ends with, you'll end up with an empty string at the end of the list.

You can use *any* string as the argument for `.split()`, making it a versatile and powerful tool.

Instructions

1.
Your boss at the Poetry organization sent over a bunch of author names that he wants you to prepare for importing into the database. Annoyingly, he sent them over as a long string with the names separated by commas.

Using `.split()` and the provided string, create a list called `author_names` containing each individual author name as it's own string.

2.
Great work, but now it turns out they didn't want poet's first names (why didn't they just say that the first time!?)

Create another list called `author_last_names` that only contains the last names of the poets in the provided string.

Hint
There are several ways to do this, but one way is to iterate through the list you created in part one and use `.split()`, negative indexing, and `.append()` to construct the new list.

script.py

```
authors = "Audre Lorde,Gabriela Mistral,Jean Toomer,An Qi,Walt Whitman,Shel Silve
rstein,Carmen Boullosa,Kamala Suraiyya,Langston Hughes,Adrienne Rich,Nikki Giovan
ni"

author_names = authors.split(',')
print(author_names)
subresult = []
author_last_names = []

for author in author_names:
  subresult.append(author.split(' '))
```

```
for author in subresult:
  author_last_names.append(author[1])

print(author_last_names)
```

```
['Audre Lorde', 'Gabriela Mistral', 'Jean Toomer', 'An
Qi', 'Walt Whitman', 'Shel Silverstein', 'Carmen
Boullosa', 'Kamala Suraiyya', 'Langston Hughes',
'Adrienne Rich', 'Nikki Giovanni']
['Lorde', 'Mistral', 'Toomer', 'Qi', 'Whitman',
'Silverstein', 'Boullosa', 'Suraiyya', 'Hughes',
'Rich', 'Giovanni']
```

Splitting Strings III

We can also split strings using *escape sequences*. Escape sequences are used to indicate that we want to split by something in a string that is not necessarily a character. The two escape sequences we will cover here are

- `\n` Newline
- `\t` Horizontal Tab

Newline or `\n` will allow us to split a multi-line string by line breaks and `\t` will allow us to split a string by tabs. `\t` is particularly useful when dealing with certain datasets because it is not uncommon for data points to be separated by tabs.

Let's take a look at an example of splitting by an escape sequence:

```
smooth_chorus = \
"""And if you said, "This life ain't good enough."
I would give my world to lift you up
I could change my life to better suit your mood
Because you're so smooth"""

chorus_lines = smooth_chorus.split('\n')

print(chorus_lines)
```

This code is splitting the multi-line string at the newlines (`\n`) which exist at the end of each line and saving it to a new list called `chorus_lines`. Then it prints `chorus_lines` which will produce the output

```
['And if you said, "This life ain\'t good enough."', 'I would give my world to lift you up', 'I could change my life to better suit your mood', "Because you're so smooth"]
```

The new list contains each line of the original string as its own smaller string. Also, notice that Python automatically escaped the `'` character in the first line and adjusted to double quotation marks to allow the apostrophe on last line when it created the new list.

Instructions

1.

The organization has sent you over the full text for William Carlos Williams poem *Spring Storm*. They want you to break the poem up into its individual lines.

Create a list called `spring_storm_lines` that contains a string for each line of *Spring Storm*.

Hint

You will have to use `.split()` and the escape character for a newline, `\n`.

script.py

```
spring_storm_text = \
"""The sky has given over
its bitterness.
Out of the dark change
all day long
rain falls and falls
as if it would never end.
Still the snow keeps
its hold on the ground.
But water, water
from a thousand runnels!
It collects swiftly,
dappled with black
cuts a way for itself
through green ice in the gutters.
Drop after drop it falls
from the withered grass-stems
of the overhanging embankment."""

spring_storm_lines = spring_storm_text.split('\n')
```

```
print(spring_storm_lines)
```

```
['The sky has given over ', 'its bitterness. ', 'Out of
the dark change ', 'all day long ', 'rain falls and
falls ', 'as if it would never end. ', 'Still the snow
keeps ', 'its hold on the ground. ', 'But water, water
', 'from a thousand runnels! ', 'It collects swiftly,
', 'dappled with black ', 'cuts a way for itself ',
'through green ice in the gutters. ', 'Drop after drop
it falls ', 'from the withered grass-stems ', 'of the
overhanging embankment.']
```

Joining Strings

Now that you've learned to break strings apart using `.split()`, let's learn to put them back together using `.join()`. `.join()` is essentially the opposite of `.split()`, it *joins* a list of strings together with a given delimiter. The syntax of `.join()` is:

```
'delimiter'.join(list_you_want_to_join)
```

Now this may seem a little weird, because with `.split()` the argument was the delimiter, but now the argument is the list. This is because *join* is still a string method, which means it has to act on a string. The string `.join()` acts on is the delimiter you want to join with, therefore the list you want to join has to be the argument.

This can be a bit confusing, so let's take a look at an example.

```
my_munequita = ['My', 'Spanish', 'Harlem', 'Mona', 'Lisa']
print(' '.join(my_munequita))
# => 'My Spanish Harlem Mona Lisa'
```

We take the list of strings, `my_munequita`, and we joined it together with our delimiter, `' '`, which is a space. The space is important if you are trying to build a sentence from words, otherwise, we would have ended up with:

```
print(''.join(my_munequita))
# => 'MySpanishHarlemMonaLisa'
```

Instructions

1.

You've been provided with a list of words from the first line of Jean Toomer's poem *Reapers*.

Use `.join()` to combine these words into a sentence and save that sentence as the string `reapers_line_one`.

Hint

Make sure that you are running join on a space, `' '`, otherwise you'll mash the words together.

script.py

```
reapers_line_one_words = ["Black", "reapers", "with", "the", "sound", "of", "steel", "on", "stones"]

reapers_line_one = ' '.join(reapers_line_one_words)
print(reapers_line_one)
```

```
Black reapers with the sound of steel on stones
```

Joining Strings II

In the last exercise, we joined together a list of words using a space as the delimiter to create a sentence. In fact, you can use any string as a delimiter to join together a list of strings. For example, if we have the list

```
santana_songs = ['Oye Como Va', 'Smooth', 'Black Magic Woman', 'Samba Pa Ti', 'Maria Maria']
```

We could join this list together with ANY string. One often used string is a comma `,` because then we can create a string of *comma separated variables*, or CSV.

```
santana_songs_csv = ','.join(santana_songs)
print(santana_songs_csv)
# => 'Oye Como Va,Smooth,Black Magic Woman,Samba Pa Ti,Maria Maria'
```

You'll often find data stored in CSVs because it is an efficient, simple file type used by popular programs like Excel or Google Spreadsheets.

You can also join using *escape sequences* as the delimiter. Consider the following example:

```
smooth_fifth_verse_lines = ['Well I\'m from the barrio', 'You hear my rhythm on your radio', 'You feel the turning of the world so soft and slow', 'Turning you \'round and \'round']

smooth_fifth_verse = '\n'.join(smooth_fifth_verse_lines)

print(smooth_fifth_verse)
```

This code is taking the list of strings and joining them using a newline `\n` as the delimiter. Then it prints the result and produces the output:

```
Well I'm from the barrio
You hear my rhythm on your radio
You feel the turning of the world so soft and slow
Turning you 'round and 'round
```

Instructions

1.

You've been given a list, `winter_trees_lines`, that contains all the lines to William Carlos Williams poem, *Winter Trees*. You've been asked to join together the strings in the list together into a single string that can be used to display the full poem. Name this string `winter_trees_full`.

Print your result to the terminal. Make sure that each line of the poem appears on a new line in your string.

Hint

Use `\n`, the escape character for a line break.

script.py

```
winter_trees_lines = ['All the complicated details', 'of the attiring and', 'the disattiring are completed!', 'A liquid moon', 'moves gently among', 'the long branches.', 'Thus having prepared their buds', 'against a sure winter', 'the wise trees', 'stand sleeping in the cold.']


winter_trees_full = '\n'.join(winter_trees_lines)


print(winter_trees_full)
```

```
All the complicated details
of the attiring and
the disattiring are completed!
A liquid moon
moves gently among
the long branches.
Thus having prepared their buds
against a sure winter
the wise trees
stand sleeping in the cold.
```

.strip()

When working with strings that come from real data, you will often find that the strings aren't super clean. You'll find lots of extra whitespace, unnecessary linebreaks, and rogue tabs.

Python provides a great method for cleaning strings: `.strip()`. Stripping a string removes all whitespace characters from the beginning and end. Consider the following example:

```
featuring = "           rob thomas                  "
print(featuring.strip())
# => 'rob thomas'
```

All the whitespace on either side of the string has been stripped, but the whitespace in the middle has been preserved.

You can also use `.strip()` with a character argument, which will strip that character from either end of the string.

```
featuring = "!!!rob thomas       !!!!!"
print(featuring.strip('!'))
# => 'rob thomas       '
```

By including the argument `'!'` we are able to strip all of the `!` characters from either side of the string. Notice that now that we've included an argument we are no longer stripping whitespace, we are ONLY stripping the argument.

Instructions

1.

They sent over another list containing all the lines to the Audre Lorde poem, *Love, Maybe*. They want you to join together all of the lines into a single string that can be used to display the poem again, but this time, you've noticed that the list contains a ton of unnecessary whitespace that doesn't appear in the actual poem.

First, use `.strip()` on each line in the list to remove the unnecessary whitespace and save it as a new list `love_maybe_lines_stripped`.

Hint

Use a `for` loop to iterate through each line in the list and strip them. Be sure to use `.append()` to add the stripped lines to a new list.

2.

`.join()` the lines in `love_maybe_lines_stripped` together into one large multi-line string, `love_maybe_full`, that can be printed to display the poem.

Each line of the poem should show up on its own line.

3.

Print `love_maybe_full`.

script.py

```
love_maybe_lines = ['Always     ', '     in the middle of our bloodiest battles  '
, 'you lay down your arms', '           like flowering mines    ','\n' ,'   to co
nquer me home.    ']

love_maybe_lines_stripped = []
for sentence in love_maybe_lines:
  love_maybe_lines_stripped.append(sentence.strip())

print(love_maybe_lines_stripped)

love_maybe_full = '\n'.join(love_maybe_lines_stripped)

print(love_maybe_full)
```

```
['Always', 'in the middle of our bloodiest battles',
'you lay down your arms', 'like flowering mines', '',
'to conquer me home.']
Always
in the middle of our bloodiest battles
you lay down your arms
like flowering mines

to conquer me home.
```

Replace

The next string method we will cover is `.replace()`. Replace takes two arguments and replaces all instances of the first argument in a string with the second argument. The syntax is as follows

```
string_name.replace(substring_being_replaced, new_substring)
```

Great! Let's put it in context and look at an example.

```
with_spaces = "You got the kind of loving that can be so smooth"
with_underscores = with_spaces.replace(' ', '_')
print(with_underscores)
# 'You_got_the_kind_of_loving_that_can_be_so_smooth'
```

Here we used `.replace()` to change every instance of a space in the string above to be an underscore instead.

Note that in this example, we used a single character, but these substrings can be multiple characters long!

Instructions

1.

The poetry organization has sent over the bio for Jean Toomer as it currently exists on their site. Notice that there was a mistake with his last name and all instances of *Toomer* are lacking one `o`.

Use `.replace()` to change all instances of `Tomer` in the bio to `Toomer`. Save the updated bio to the string `toomer_bio_fixed`.

script.py

```
toomer_bio = \
"""
Nathan Pinchback Tomer, who adopted the name Jean Tomer early in his literary car
eer, was born in Washington, D.C. in 1894. Jean is the son of Nathan Tomer was a
mixed-
race freedman, born into slavery in 1839 in Chatham County, North Carolina. Jean
Tomer is most well known for his first book Cane, which vividly portrays the life
 of African-Americans in southern farmlands.
"""

toomer_bio_fixed = toomer_bio.replace("Tomer", "Toomer")
print(toomer_bio_fixed)
```

```
Nathan Pinchback Toomer, who adopted the name Jean
Toomer early in his literary career, was born in
Washington, D.C. in 1894. Jean is the son of Nathan
Toomer was a mixed-race freedman, born into slavery in
1839 in Chatham County, North Carolina. Jean Toomer is
most well known for his first book Cane, which vividly
portrays the life of African-Americans in southern
farmlands.
```

.find()

Another interesting string method is `.find()`. `.find()` takes a string as an argument and searching the string it was run on for that string. It then returns the first *index value* where that string is located.

Here's an example:

```
print('smooth'.find('t'))
# => '4'
```

We searched the string `'smooth'` for the string `'t'` and found that it was at the fourth index spot, so `.find()` returned `4`.

You can also search for larger strings, and `.find()` will return the index value of the first character of that string.

```
print("smooth".find('oo'))
# => '2'
```

Notice here that `2` is the index of the *first* o.

Instructions

1.

In the code editor is the first line of Gabriela Mistral's poem *God Wills It*.

At what index place does the word "disown" appear? Save that index place to the variable `disown_placement`.

script.py

```
god_wills_it_line_one = "The very earth will disown you"

disown_placement = god_wills_it_line_one.find("disown")
print(disown_placement)
```

```
20
```

.format()

Python also provides a handy string method for including variables in strings. This method is `.format()`. `.format()` takes variables as an argument and includes them in the string that it is run on. You include `{}` marks as placeholders for where those variables will be imported.

Consider the following function:

```
def favorite_song_statement(song, artist):
  return "My favorite song is {} by {}.".format(song, artist)
```

The function `favorite_song_statement` takes two arguments, song and artist, then returns a string that includes both of the arguments and prints a sentence. Note: `.format()` can take as many arguments as there are `{}` in the string it is run on, which in this case is two.

Here's an example of the function being run:

```
print(favorite_song_statement("Smooth", "Santana"))
# => "My favorite song is Smooth by Santana."
```

Now you may be asking yourself, I could have written this function using string concatenation instead of `.format()`, why is this method better? The answer is legibility and reusability. It is much easier to picture the end result `.format()` than it is to picture the end result of string concatenation and legibility is everything. You can also reuse the same base string with different variables, allowing you to cut down on unnecessary, hard to interpret code.

Instructions

1.

Write a function called `poem_title_card` that takes two inputs: the first input should be `title` and the second `poet`. The function should use `.format()` to return the following string:

```
 The poem "[TITLE]" is written by [POET].
```

For example, if the function is given the inputs

```
poem_title_card("I Hear America Singing", "Walt Whitman")
```

It should return the string

```
The poem "I Hear America Singing" is written by Walt Whitman.
```

Hint

Remember to escape the `"` characters!

script.py

```
def poem_title_card(title, poet):
  return "The poem \"{}\" is written by {}.".format(title, poet)

print(poem_title_card("I Hear America Singing", "Walt Whitman"))
```

```
The poem "I Hear America Singing" is written by Walt Whitman.
```

.format() II

`.format()` can be made even more legible for other people reading your code by including *keywords*. Previously, with `.format()`, you had to make sure that your variables appeared as arguments in the same order that you wanted them to appear in the string, which added unnecessary complications when writing code.

By including keywords in the string, and in the arguments, you can remove that ambiguity. Let's look at an example.

```
def favorite_song_statement(song, artist):
  return "My favorite song is {song} by {artist}.".format(song=song,
artist=artist)
```

Now it is clear to anyone reading the string what it is supposed to return, they don't even need to look at the arguments of `.format()` in order to get a clear understanding of what is supposed to happen. You can even reverse the order of `artist` and `song` in the code above and it will work the same way.

For example, if the arguments of `.format()` are in a different order, the code will still work since the keywords are present:

```
def favorite_song_statement(song, artist):
  # this will have the same output as the above example
  return "My favorite song is {song} by
{artist}.".format(artist=artist, song=song)
```

This makes writing AND reading the code much easier.

Instructions

1.

The function `poem_description` is supposed to use `.format()` to print out some quick information about a poem, but it seems to be causing some errors currently.

Fix the function by using keywords in the `.format()` method.

2.

Run `poem_description` with the following arguments and save the results to the variable `my_beard_description`:

```
author = "Shel Silverstein"
title = "My Beard"
original_work = "Where the Sidewalk Ends"
publishing_date = "1974"
```

script.py

```
def poem_description(publishing_date, author, title, original_work):
  poem_desc = "The poem {title} by {author} was originally published in {original_work} in {publishing_date}.".format(publishing_date=publishing_date, author=author, title=title, original_work=original_work)
  return poem_desc

my_beard_description = poem_description(author = "Shel Silverstein",
title = "My Beard",
original_work = "Where the Sidewalk Ends",
publishing_date = "1974")

print(my_beard_description)
```

```
The poem My Beard by Shel Silverstein was originally published in Where the Sidewalk Ends in 1974.
```

Review

Excellent work! This lesson has shown you the vast variety of string methods and their power. Whatever the problem you are trying to solve, if you are working with strings then string methods are likely going to be part of the solution.

Over this lesson you've learned:

- `.upper()`, `.title()`, and `.lower()` adjust the casing of your string.
- `.split()` takes a string and creates a list of substrings.
- `.join()` takes a list of strings and creates a string.
- `.strip()` cleans off whitespace, or other noise from the beginning and end of a string.
- `.replace()` replaces all instances of a character/string in a string with another character/string.
- `.find()` searches a string for a character/string and returns the index value that character/string is found at.
- `.format()` allows you to interpolate a string with variables.

Well I've been stringing you along for long enough, let's get some more practice in!

Instructions

1.
Preserve the Verse has one final task for you. They have delivered you a string that contains a list of poems, titled `highlighted_poems`, they want to highlight on the site, but they need your help to parse the string into something that can display the name, title, and publication date of the highlighted poems on the site.

First, start by printing `highlighted_poems` to the terminal and see how it displays.

2.
The information for each poem is separated by commas, and within this information is the title of the poem, the author, and the date of publication.

Start by splitting `highlighted_poems` at the commas and saving it to `highlighted_poems_list`.

Hint
Recall that the syntax for splitting a string into a list is:

```
py my_string.split(delimiter)
```

3.
Print `highlighted_poems_list`, how does the structure of the data look now?

4.
Notice that there is inconsistent whitespace in `highlighted_poems_list`. Let's clean that up.

Start by creating a new empty list, `highlighted_poems_stripped`.

Then, iterate through `highlighted_poems_list` using a for loop and for each poem strip away the whitespace and append it to your new list, `highlighted_poems_stripped`.

Stuck? Get a hint

5.
Print `highlighted_poems_stripped`.

Looks good! All the whitespace is cleaned up.

6.
Next we want to break up all the information for each poem into it's own list, so we end up with a list of lists.

Create a new empty list called `highlighted_poems_details`.

7.

Iterate through `highlighted_poems_stripped` and split each string around the `:` characters and append the new lists into `highlighted_poems_details`.

8.

Great! Now we want to separate out all of the titles, the poets, and the publication dates into their own lists.

Create three new empty lists, `titles`, `poets`, and `dates`.

9.

Iterate through `highlighted_poems_details` and for each list in the list append the appropriate elements into the lists `titles`, `poets`, and `dates`.

For example, for the poem *The Shadow* (1915) by William Carlos Williams your code should be adding `"The Shadow"` to `titles`, `"William Carlos Williams"` to `poets`, and `"1915"` to `dates`.

10.

Finally, write a for loop that uses `.format()` to print out the following string for each poem:

```
The poem TITLE was published by POET in DATE.
```

script.py

```
highlighted_poems = "Afterimages:Audre Lorde:1997,  The Shadow:William Carlos Wil
liams:1915, Ecstasy:Gabriela Mistral:1925,   Georgia Dusk:Jean Toomer:1923,   Par
ting Before Daybreak:An Qi:2014, The Untold Want:Walt Whitman:1871, Mr. Grumpledu
mp's Song:Shel Silverstein:2004, Angel Sound Mexico City:Carmen Boullosa:2013, In
 Love:Kamala Suraiyya:1965, Dream Variations:Langston Hughes:1994, Dreamwood:Adri
enne Rich:1987"

print(highlighted_poems)

highlighted_poems_list = highlighted_poems.split(',')
print(highlighted_poems_list)

highlighted_poems_stripped = []

for poem in highlighted_poems_list:
  highlighted_poems_stripped.append(poem.strip())

print(highlighted_poems_stripped)
```

```
highlighted_poems_details = []

for poem in highlighted_poems_stripped:
  highlighted_poems_details.append(poem.split(':'))

titles = []
poets = []
dates = []

for list in highlighted_poems_details:
  titles.append(list[0])
  poets.append(list[1])
  dates.append(list[2])

print(titles)
print(poets)
print(dates)

for i in range(len(titles)):
  print("The poem " + titles[i] + " was published by " + poets[i] + " in " + date
s[i] + ".")
```

```
Afterimages:Audre Lorde:1997,  The Shadow:William Carlos Williams:1915, Ecstasy:Gabriela Mistral:1925, Georgia Dusk:Jean Toomer:1923,   Parting Before Daybreak:An Qi:2014, The Untold Want:Walt Whitman:1871, Mr. Grumpledump's Song:Shel Silverstein:2004, Angel Sound Mexico City:Carmen Boullosa:2013, In Love:Kamala Suraiyya:1965, Dream Variations:Langston Hughes:1994, Dreamwood:Adrienne Rich:1987
['Afterimages:Audre Lorde:1997', '  The Shadow:William Carlos Williams:1915', ' Ecstasy:Gabriela Mistral:1925', '   Georgia Dusk:Jean Toomer:1923', ' Parting Before Daybreak:An Qi:2014', ' The Untold Want:Walt Whitman:1871', " Mr. Grumpledump's Song:Shel Silverstein:2004", ' Angel Sound Mexico City:Carmen Boullosa:2013', ' In Love:Kamala Suraiyya:1965', ' Dream Variations:Langston Hughes:1994', ' Dreamwood:Adrienne Rich:1987']
['Afterimages:Audre Lorde:1997', 'The Shadow:William Carlos Williams:1915', 'Ecstasy:Gabriela Mistral:1925', 'Georgia Dusk:Jean Toomer:1923', 'Parting Before Daybreak:An Qi:2014', 'The Untold Want:Walt Whitman:1871', "Mr. Grumpledump's Song:Shel Silverstein:2004", 'Angel Sound Mexico City:Carmen
```