9.9. Non-mutating Methods on Strings

There are a wide variety of methods for string objects. Try the following program.

Save & Run Original - 1 of 1 Show in CodeLens

```
ss = "Hello, World"
print(ss.upper())

tt = ss.lower()
print(tt)
print(ss)

```

```
HELLO, WORLD
hello, world
Hello, World
```

Activity: 1 -- ActiveCode (ac8_8_1)

In this example, `upper` is a method that can be invoked on any string object to create a new string in which all the characters are in uppercase. `lower` works in a similar fashion changing all characters in the string to lowercase. (The original string `ss` remains unchanged. A new string `tt` is created.)

You've already seen a few methods, such as `count` and `index`, that work with strings and are non-mutating. In addition to those and `upper` and `lower`, the following table provides a summary of some other useful string methods. There are a few activecode examples that follow so that you can try them out.

Method	Parameters	Description
upper	none	Returns a string in all uppercase
lower	none	Returns a string in all lowercase
count	item	Returns the number of occurrences of item
index	item	Returns the leftmost index where the substring item is found and causes a runtime error if item is not found
strip	none	Returns a string with the leading and trailing whitespace removed
replace	old, new	Replaces all occurrences of old substring with new
format	substitutions	Involved! See String Format Method, below

You should experiment with these methods so that you understand what they do. Note once again that the methods that return strings do not change the original. You can also consult the Python documentation for strings.

Save & Run Original - 1 of 1 Show in CodeLens

```
ss = "    Hello, World    "

els = ss.count("l")
print(els)

print("***"+ss.strip()+"***")

news = ss.replace("o", "***")
print(news)

```

```
3
***Hello, World***
    Hell***, W***rld
```

Activity: 2 -- ActiveCode (ac8_8_2)

Save & Run Original - 1 of 1 Show in CodeLens

```
food = "banana bread"
print(food.upper())

```

```
BANANA BREAD
```

Activity: 3 -- ActiveCode (ac8_8_3)

Check your understanding

seqmut-8-1: What is printed by the following statements?

```
s = "python rocks"
print(s.count("o") + s.count("p"))
```

- A. 0
- B. 2
- C. 3

Check me | Compare me

✔ Yes, add the number of o characters and the number of p characters.

Activity: 4 -- Multiple Choice (question8_8_1)

seqmut-8-2: What is printed by the following statements?

```
s = "python rocks"
print(s[1]*s.index("n"))
```

- A. yyyyy
- B. 55555
- C. n
- D. Error, you cannot combine all those things together.

Check me | Compare me

✔ Yes, s[1] is y and the index of n is 5, so 5 y characters. It is important to realize that the index method has precedence over the repetition operator. Repetition is done last.

Activity: 5 -- Multiple Choice (question8_8_2)

9.9.1. String Format Method

Until now, we have created strings with variable content using the + operator to concatenate partial strings together. That works, but it's very hard for people to read or debug a code line that includes variable names and strings and complex expressions. Consider the following:

Save & Run | Original - 1 of 1 | Show in CodeLens

```
1 name = "Rodney Dangerfield"
2 score = -1  # No respect!
3 print("Hello " + name + ". Your score is " + str(score))
4
```

```
Hello Rodney Dangerfield. Your score is -1
```

Activity: 6 -- ActiveCode (ac8_8_4)

Or perhaps more realistically:

Save & Run | Original - 1 of 1 | Show in CodeLens

```
scores = [("Rodney Dangerfield", -1), ("Marlon Brando", 1), ("You", 100)]
for person in scores:
    name = person[0]
    score = person[1]
    print("Hello " + name + ". Your score is " + str(score))

```

```
Hello Rodney Dangerfield. Your score is -1
Hello Marlon Brando. Your score is 1
Hello You. Your score is 100
```

Activity: 7 -- ActiveCode (ac8_8_5)

In this section, you will learn to write that in a more readable way:

Save & Run Original - 1 of 1 Show in CodeLens

```
scores = [("Rodney Dangerfield", -1), ("Marlon Brando", 1), ("You", 100)]
for person in scores:
    name = person[0]
    score = person[1]
    print("Hello {}. Your score is {}.".format(name, score))

```

```
Hello Rodney Dangerfield. Your score is -1.
Hello Marlon Brando. Your score is 1.
Hello You. Your score is 100.
```

Activity: 8 -- ActiveCode (ac8_8_6)

In grade school quizzes a common convention is to use fill-in-the blanks. For instance,

> Hello _____!

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. *We use this as an analogy:* Python has a similar construction, better called fill-in-the-braces. The string method `format`, makes substitutions into places in a string enclosed in braces. Run this code:

Save & Run Original - 1 of 1 Show in CodeLens

```
person = input('Your name: ')
greeting = 'Hello {}!'.format(person)
print(greeting)

```

```
Hello Tom!
```

Activity: 9 -- ActiveCode (ac8_8_7)

There are several new ideas here!

The string for the `format` method has a special form, with braces embedded. Such a string is called a *format string*. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the `format` method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier `greeting`, and then the string is printed.

The identifier `greeting` was introduced to break the operations into a clearer sequence of steps. However, since the value of `greeting` is only referenced once, it can be eliminated with the more concise version:

Save & Run | Original - 1 of 1 | Show in CodeLens

```
person = input('Enter your name: ')
print('Hello {}!'.format(person))

```

```
Hello Tom!
```

Activity: 10 -- ActiveCode (ac8_8_8)

There can be multiple substitutions, with data of any type. Next we use floats. Try original price $2.50 with a 7% discount:

Save & Run | Original - 1 of 1 | Show in CodeLens

```
origPrice = float(input('Enter the original price: $'))
discount = float(input('Enter discount percentage: '))
newPrice = (1 - discount/100)*origPrice
calculation = '${} discounted by {}% is ${}.'.format(origPrice, discount, newPrice)
print(calculation)

```

```
$100.0 discounted by 25.0% is $75.0.
```

Activity: 11 -- ActiveCode (ac8_8_9)

It is important to pass arguments to the `format` method in the correct order, because they are matched *positionally* into the `{}` places for interpolation where there is more than one.

If you used the data suggested, this result is not satisfying. Prices should appear with exactly two places beyond the decimal point, but that is not the default way to display floats.

Format strings can give further information inside the braces showing how to specially format data. In particular floats can be shown with a specific number of decimal places. For two decimal places, put `:.2f` inside the braces for the monetary values:

Save & Run | Original - 1 of 1 | Show in CodeLens

```
origPrice = float(input('Enter the original price: $'))
discount = float(input('Enter discount percentage: '))
newPrice = (1 - discount/100)*origPrice
calculation = '${:.2f} discounted by {}% is ${:.2f}.'.format(origPrice, discount, r
print(calculation)

```

```
$100.00 discounted by 25.0% is $75.00.
```

Activity: 12 -- ActiveCode (ac8_8_10)

The 2 in the format modifier can be replaced by another integer to round to that specified number of digits.

This kind of format string depends directly on the order of the parameters to the format method. There are

other approaches that we will skip here, such as explicitly numbering substitutions.

It is also important that you give `format` the same amount of arguments as there are `{}` waiting for interpolation in the string. If you have a `{}` in a string that you do not pass arguments for, you may not get an error, but you will see a weird `undefined` value you probably did not intend suddenly inserted into your string. You can see an example below.

For example,

Save & Run | Original - 1 of 1 | Show in CodeLens

```
name = "Sally"
greeting = "Nice to meet you"
s = "Hello, {}. {}."

print(s.format(name,greeting)) # will print Hello, Sally. Nice to meet you.

print(s.format(greeting,name)) # will print Hello, Nice to meet you. Sally.

print(s.format(name)) # 2 {}s, only one interpolation item! Not ideal.

```

```
Hello, Sally. Nice to meet you.
Hello, Nice to meet you. Sally.
Hello, Sally. .
```

Activity: 13 -- ActiveCode (ac8_8_11)

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final *formatted* string. The rule is to double the braces: `\{\{` and `\}\}`. For example mathematical set notation uses braces. The initial and final doubled braces in the format string below generate literal braces in the formatted string:

```
a = 5
b = 9
setStr = 'The set is \{\{\{}, {}}}.'.format(a, b)
print(setStr)
```

Unfortunately, at the time of this writing, the ActiveCode format implementation has a bug, printing doubled braces, but standard Python prints `{5, 9}`.

seqmut-8-3: What is printed by the following statements?

```
x = 2
y = 6
print('sum of {} and {} is {}; product: {}.'.format( x, y, x+y, x*y))
```

- A. Nothing - it causes an error
- B. sum of {} and {} is {}; product: {}. 2 6 8 12
- C. sum of 2 and 6 is 8; product: 12.
- D. sum of {2} and {6} is {8}; product: {12}.

Check me | Compare me

✔ Yes, correct substitutions!

Activity: 14 -- Multiple Choice (question8_8_3)

seqmut-8-4: What is printed by the following statements?

```
v = 2.34567
print('{:.1f} {:.2f} {:.7f}'.format(v, v, v))
```

- A. 2.34567 2.34567 2.34567
- B. 2.3 2.34 2.34567
- C. 2.3 2.35 2.3456700

Check me | Compare me

✔ Yes, correct number of digits with rounding!

Activity: 15 -- Multiple Choice (question8_8_4)

You have attempted 16 of 15 activities on this page

9.8. Append versus Concatenate">
9.10. The Accumulator Pattern with Lists">
9.8. Append versus Concatenate">
✔ Completed. Well D
9.10. The Accumulator Pattern with Lists">Next Section - 9.10. The Accumulator Pattern with Lists

| Back to top