Modify the double\_word function so that it returns the same word repeated twice, followed by the length of the new doubled word. For example, double\_word("hello") should return hellohello10.

def double\_word(word):

    result=word\*2

    result=result+str(len(result))

    return result

print(double\_word("hello")) # Should return hellohello10

print(double\_word("abc"))   # Should return abcabc6

print(double\_word(""))      # Should return 0

Here is your output:

hellohello10

abcabc6

0

Great work! You're getting comfortable using some of the

basic string operations. You'll soon be working on even more

interesting tasks!

Want to give it a go yourself? Be my guest! Modify the first\_and\_last function so that it returns True if the first letter of the string is the same as the last letter of the string, False if they’re different. Remember that you can access characters using message[0] or message[-1]. Be careful how you handle the empty string, which should return True since nothing is equal to nothing.

def first\_and\_last(message):

    if len(message)==0:

        return True

    elif message[0]==message[-1]:

        return True

    return False

print(first\_and\_last("else"))

print(first\_and\_last("tree"))

print(first\_and\_last(""))

Here is your output:

True

False

True

Nice job! You've specified the correct condition for empty

strings, and used appropriate string indexing to get the

work done.

String Indexing and Slicing

String indexing allows you to access individual characters in a string. You can do this by using square brackets and the location, or index, of the character you want to access. It's important to remember that Python starts indexes at 0. So to access the first character in a string, you would use the index [0]. If you try to access an index that’s larger than the length of your string, you’ll get an **IndexError**. This is because you’re trying to access something that doesn't exist! You can also access indexes from the end of the string going towards the start of the string by using negative values. The index [-1] would access the last character of the string, and the index [-2] would access the second-to-last character.

You can also access a portion of a string, called a slice or a substring. This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5]. This range is similar to the range() function we saw previously. It includes the first number, but goes to one less than the last number. You can also easily reference the start or end of the string by leaving one value blank. For example [:5] would give us all characters from the start to the fourth character in the string. We can also use [5:] to get the characters from the fourth character to the end of the string.

**try using the index method yourself now**!

Using the index method, find out the position of "x" in "supercalifragilisticexpialidocious".

word = "supercalifragilisticexpialidocious"

print(word.index('x'))

RunReset

Here is your output:

21

Right on! The index method is a very useful way of working

with strings, and not having to do the hard work manually.

Basic String Methods

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method *index* to locate it and return the index. Let's imagine we have the string **"lions tigers and bears"** in the variable **animals**. We can locate the index that contains the letter **g** using *animals.index("g")*, which will return the index; in this case 8. We can also use substrings to locate the index where the substring begins. *animals.index("bears")* would return 17, since that’s the start of the substring. If there’s more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we’ll receive a **ValueError** explaining that the substring was not found.

We can avoid a ValueError by first checking if the substring exists in the string. This can be done using the ***in*** keyword. We saw this keyword earlier when we covered *for* loops. In this case, it's a conditional that will be either True or False. If the substring is found in the string, it will be True. If the substring is not found in the string, it will be False. Using our previous variable **animals**, we can do **"horses"** **in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren’t included in our example string. If we did **"tigers" in animals**, we'd get True, since this substring is contained in our string.

Want to try some string methods yourself? Give it a go!

Fill in the gaps in the initials function so that it returns the initials of the words contained in the phrase received, in upper case. For example: "Universal Serial Bus" should return "USB"; "local area network" should return "LAN”.

def initials(phrase):

    words = phrase.split()

    result = ""

    for word in words:

        result += word[0]

    return result.upper()

print(initials("Universal Serial Bus")) # Should be: USB

print(initials("local area network")) # Should be: LAN

print(initials("Operating system")) # Should be: OS

RunReset

Here is your output:

USB

LAN

OS

Awesome! You've filled in the correct string methods and

remembered how to use them! We’re getting through all this

stuff really fast now -- great job!

Advanced String Methods

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The string method **lower** will return the string with all characters changed to lowercase. The inverse of this is the **upper** method, which will return the string all in uppercase. Just like with previous methods, we call these on a string using dot notation, like **"this is a string".upper()**. This would return the string **"THIS IS A STRING"**. This can be super handy when checking user input, since someone might type in all lowercase, all uppercase, or even a mixture of cases.

You can use the **strip** method to remove surrounding whitespace from a string. Whitespace includes spaces, tabs, and newline characters. You can also use the methods  **lstrip** and **rstrip**to remove whitespace only from the left or the right side of the string, respectively.

The method **count** can be used to return the number of times a substring appears in a string. This can be handy for finding out how many characters appear in a string, or counting the number of times a certain word appears in a sentence or paragraph.

If you wanted to check if a string ends with a given substring, you can use the method **endswith**. This will return True if the substring is found at the end of the string, and False if not.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, **" ".join(["This","is","a","sentence"])** would return the string **"This is a sentence"**.

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

Modify the student\_grade function using the format method, so that it returns the phrase "X received Y% on the exam". For example, student\_grade("Reed", 80) should return "Reed received 80% on the exam".

def student\_grade(name, grade):

    return "{} received {}% on the exam".format(name,grade)

print(student\_grade("Reed", 80))

print(student\_grade("Paige", 92))

print(student\_grade("Jesse", 85))

RunReset

Here is your output:

Reed received 80% on the exam

Paige received 92% on the exam

Jesse received 85% on the exam

You got it! Your string formatting skills are coming along

nicely!

String Formatting

You can use the **format** method on strings to concatenate and format strings in all kinds of powerful ways. To do this, create a string containing curly brackets, **{}**, as a placeholder, to be replaced. Then call the format method on the string using *.format()* and pass variables as parameters. The variables passed to the method will then be used to replace the curly bracket placeholders. This method automatically handles any conversion between data types for us.

If the curly brackets are empty, they’ll be populated with the variables passed in the order in which they're passed. However, you can put certain expressions inside the curly brackets to do even more powerful string formatting operations. You can put the name of a variable into the curly brackets, then use the names in the parameters. This allows for more easily readable code, and for more flexibility with the order of variables.

You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example, the formatting expression **{:.2f}** means that you’d format this as a float number, with two digits after the decimal dot. The colon acts as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: **>**. For example, the expression **{:>3.2f}** would align the text three spaces to the right, as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

## String Reference Cheat Sheet

## String Reference Cheat Sheet

In Python, there are a lot of things you can do with strings. In this cheat sheet, you’ll find the most common string operations and string methods.

### **String operations**

* len(string) Returns the length of the string
* for character in string Iterates over each character in the string
* if substring in string Checks whether the substring is part of the string
* string[i] Accesses the character at index i of the string, starting at zero
* string[i:j] Accesses the substring starting at index i, ending at index j-1. If i is omitted, it's 0 by default. If j is omitted, it's len(string) by default.

### **String methods**

* string.lower() / string.upper() Returns a copy of the string with all lower / upper case characters
* string.lstrip() / string.rstrip() / string.strip() Returns a copy of the string without left / right / left or right whitespace
* string.count(substring) Returns the number of times substring is present in the string
* string.isnumeric() Returns True if there are only numeric characters in the string. If not, returns False.
* string.isalpha() Returns True if there are only alphabetic characters in the string. If not, returns False.
* string.split() / string.split(delimiter) Returns a list of substrings that were separated by whitespace / delimiter
* string.replace(old, new) Returns a new string where all occurrences of old have been replaced by new.
* delimiter.join(list of strings) Returns a new string with all the strings joined by the delimiter

Check out the official documentation for [all available String methods](https://docs.python.org/3/library/stdtypes.html#string-methods).

## Formatting Strings Cheat Sheet

## Formatting Strings Cheat Sheet

Python offers different ways to format strings. In the video, we explained the format() method. In this reading, we'll highlight three different ways of formatting strings. For this course you only need to know the format() method. But on the internet, you might find any of the three, so it's a good idea to know that the others exist.

### **Using the format() method**

The format method returns a copy of the string where the {} placeholders have been replaced with the values of the variables. These variables are converted to strings if they weren't strings already. Empty placeholders are replaced by the variables passed to format in the same order.

1

2

3

4

5

6

7

8

9

10

11

# "base string with {} placeholders".format(variables)

example = "format() method"

formatted\_string = "this is an example of using the {} on a string".format(example)

print(formatted\_string)

"""Outputs:

this is an example of using the format() method on a string

"""





If the placeholders indicate a number, they’re replaced by the variable corresponding to that order (starting at zero).

1

2

3

4

5

6

7

8

9

10

11

12

13

# "{0} {1}".format(first, second)

first = "apple"

second = "banana"

third = "carrot"

formatted\_string = "{0} {2} {1}".format(first, second, third)

print(formatted\_string)

"""Outputs:

apple carrot banana

"""





If the placeholders indicate a field name, they’re replaced by the variable corresponding to that field name. This means that parameters to format need to be passed indicating the field name.

1

# "{var1} {var2}".format(var1=value1, var2=value2)





1

"{:exp1} {:exp2}".format(value1, value2)





If the placeholders include a colon, what comes after the colon is a formatting expression. See below for the expression reference.

Official documentation for [the format string syntax](https://docs.python.org/3/library/string.html#formatstrings)

1

2

# {:d} integer value

'{:d}'.format(10.5) → '10'





### **Formatting expressions**

| **Expr** | **Meaning** | **Example** |
| --- | --- | --- |
| {:d} | integer value | '{:d}'.format(10.5) → '10' |
| {:.2f} | floating point with that many decimals | '{:.2f}'.format(0.5) → '0.50' |
| {:.2s} | string with that many characters | '{:.2s}'.format('Python') → 'Py' |
| {:<6s} | string aligned to the left that many spaces | '{:<6s}'.format('Py') → 'Py    ' |
| {:>6s} | string aligned to the right that many spaces | '{:>6s}'.format('Py') → '    Py' |
| {:^6s} | string centered in that many spaces | '{:^6s}'.format('Py') → '  Py ' |

Check out the official documentation for [all available expressions](https://docs.python.org/3/library/string.html#format-specification-mini-language).

### **Old string formatting (Optional)**

The format() method was introduced in Python 2.6. Before that, the % (modulo) operator could be used to get a similar result. While this method is **no longer recommended** for new code, you might come across it in someone else's code. This is what it looks like:

"base string with %s placeholder" % variable

The % (modulo) operator returns a copy of the string where the placeholders indicated by %  followed by a formatting expression are replaced by the variables after the operator.

"base string with %d and %d placeholders" % (value1, value2)

To replace more than one value, the values need to be written between parentheses. The formatting expression needs to match the value type.

"%(var1) %(var2)" % {var1:value1, var2:value2}

Variables can be replaced by name using a dictionary syntax (we’ll learn about dictionaries in an upcoming video).

"Item: %s - Amount: %d - Price: %.2f" % (item, amount, price)

The formatting expressions are mostly the same as those of the format() method.

Check out the official documentation for [old string formatting](https://docs.python.org/3/library/stdtypes.html#old-string-formatting).

### **Formatted string literals (Optional)**

This feature was added in Python 3.6 and isn’t used a lot yet. Again, it's included here in case you run into it in the future, but it's not needed for this or any upcoming courses.

A formatted string literal or f-string is a string that starts with 'f' or 'F' before the quotes. These strings might contain {} placeholders using expressions like the ones used for format method strings.

The important difference with the format method is that it takes the value of the variables from the current context, instead of taking the values from parameters.

Examples:

>>> name = "Micah"

>>> print(f'Hello {name}')

Hello Micah

>>> item = "Purple Cup"

>>> amount = 5

>>> price = amount \* 3.25

>>> print(f'Item: {item} - Amount: {amount} - Price: {price:.2f}')

Item: Purple Cup - Amount: 5 - Price: 16.25

Check out the official documentation for [f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings).