# 7.1 String slicing

## String slicing basics

Strings are a sequence type, having characters ordered by index from left to right. An ***index*** is an integer matching a specific position in a string's sequence of characters. An individual character is read using an index surrounded by brackets. Ex: `my_str[5]` reads the character at index 5 of the string `my_str`. Indices start at 0, so index 5 is a reference to the 6th character in the string.

A programmer often needs to read more than one character at a time. Multiple consecutive characters can be read using slice notation. ***Slice notation*** has the form `my_str[start:end]`, which creates a new string whose value contains the characters of `my_str` from indices start to end -1. If `my_str` is 'Boggle', then `my_str[0:3]` yields string 'Bog'. Other sequence types like lists and tuples also support slice notation.

### Figure 7.1.1: String slicing.

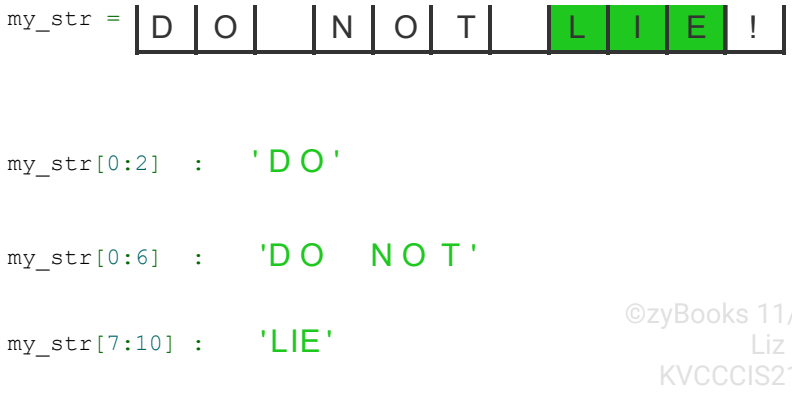
```
url = 'http://en.wikipedia.org/wiki/Turing'
domain = url[7:23]  # Read 'en.wikipedia.org' from
url
print(domain)
```

```
en.wikipedia.org
```

The last character of the slice is one location *before* the specified end. Consider the string `my_str = 'John Doe'`. The slice `my_str[0:4]` includes the element at index 0 (J), 1 (o), 2 (h), and 3 (n), but *not* 4, thus yielding 'John'. The space character at index 4 is not included. Similarly, `my_str[4:7]` would yield ' Do', including the space character this time. To retrieve the last character, an end index greater than the length of the string can be used. Ex: `my_str[5:8]` or `my_str[5:10]` both yield the string 'Doe'.

Negative numbers can be used to specify an index relative to the end of the string. Ex: If the variable `my_str` is 'Jane Doe!?', then `my_str[0:-2]` yields 'Jane Doe' because the -2 refers to the second-to-last character '!', and the character at the end index is not included in the result string.

**PARTICIPATION ACTIVITY** 7.1.1: Slicing.






### Animation content:

Static Figure: A string variable my_str is shown, with the contents of the string being "DO NOT LIE!". Each of the characters in the string are labeled with indexes 0 to 10.
'D' is located at index 0. 'O' is located at index 1. Index 2 is a blank space. 'N' is located at index 3. 'O' is located at index 4. 'T' is located at index 5. Index 6 is a blank space. 'L' is located at index 7. 'I' is located at index 8. 'E' is located at index 9. '!' is located at index 10.

The indexes 7, 8, and 9 are highlighted. Below my_str, 3 substrings of my_str are shown.
my_str[0:2] : 'DO'
my_str[0:6] : 'DO NOT'
my_str[7:10] : 'LIE'

Step 1: my_str[0:2] returns a substring of my_str starting at index 0 up to, but not including, index 2. The characters that start at index 0 up to, but don't include index 2 are 'D' and 'O', so the substring returns "DO".

Step 2: my_str[0:6] returns a substring of my_str starting at index 0 up to, but not including, index 6. The characters that start at index 0 up to , but don't include index 6 are 'D', 'O', ' ', 'N', 'O', and 'T', with a blank space at index 2. The substring returns "DO NOT".

Step 3: my_str[7:10] returns a substring of my_str starting at index 7 up to, but not including, index 10. The characters that start at index 7 up to, but not including index 10 are 'L', 'I', and 'E'. The substring returns "LIE".



### Animation captions:

1. my_str[0:2] returns a substring of my_str starting at index 0 up to, but not including, index 2.
2. my_str[0:6] returns a substring of my_str starting at index 0 up to, but not including, index 6.
3. my_str[7:10] returns a substring of my_str starting at index 7 up to, but not including, index 10.

**PARTICIPATION ACTIVITY** 7.1.2: Slicing basics.

Determine the output of the following code:

1)
```
my_str = 'The cat in the hat'
print(my_str[0:3])
```

Check Show answer



2)
```
my_str = 'The cat in the hat'
print(my_str[3:7])
```

Check Show answer

## Slicing and slicing operations

The Python interpreter creates a new string object for the slice. Thus, creating a slice of the string variable `my_str`, and then changing the value of `my_str`, does not also change the value of the slice.

Figure 7.1.2: A slice creates a new object.

```
my_str = "The cat jumped the brown cow"
animal = my_str[4:7]
print(f'The animal is a {animal}')

my_str = 'The fox jumped the brown llama'
print(f'The animal is still a {animal}')# animal
variable remains unchanged.
```

```
The animal is a
cat
The animal is
still a cat
```



A programmer often wants to read all characters that occur before or after some index in the string. Omitting a start index, such as in `my_str[:end]` yields the characters from indices 0 to end -1. Ex: `my_str[:5]` reads indices 0-4. Similarly, omitting the end index yields the characters from the start index to the end of the string. Ex: `my_str[5:]` yields all characters at and after index 5.

Use the below tool to experiment with slice notation. After using positive values only, try entering negative start or end indices. Then try omitting either the start or end index.

**PARTICIPATION ACTIVITY** 7.1.3: String slicing tool.

```
string_var = ' Hey folks! '
print(string_var[ 1 : 5 ])
```

Output: 'ey f'




Variables can also be used in place of literals to specify slice notation start and end indices. Ex: `my_str[x:y]`.

## zyDE 7.1.1: Slicing example: omitting start, end indices.

Run the program below.

Load default template...

```
1 usr_text = input('Enter a str
2 print()
3
4 first_half = usr_text[:len(us
5 last_half = usr_text[len(usr_
6
7 print(f'The first half of the
8 print(f'The second half of th
9
```

Hello there. Nice to meet you!

Run



Specifying a start index beyond the end of the string, or beyond the end index (like 3:2), yields an empty string. Ex: `my_str[2:1]` is `''`. Specifying an end index beyond the end of the string is equivalent to specifying the end of the string, so if a string's end is 5, then 1:7 or 1:99 are the same as 1:6.

## Table 7.1.1: Common slicing operations.

A list of common slicing operations a programmer might use.
Assume the value of my_str is 'http://en.wikipedia.org/wiki/Nasa/'

| Syntax | Result | Description |
|---|---|---|
| `my_str[10:19]` | wikipedia | Returns the characters in indices 10-18. |
| `my_str[10:-5]` | wikipedia.org/wiki/ | Returns the characters in indices 10-28. |
| `my_str[8:]` | n.wikipedia.org/wiki/Nasa/ | Returns all characters from index 8 until the end of the string. |
| `my_str[:23]` | http://en.wikipedia.org | Returns every character up to index 23, but not including my_str[23]. |
| `my_str[:-1]` | http://en.wikipedia.org/wiki/Nasa | Returns all but the last character. |



**PARTICIPATION ACTIVITY** 7.1.4: Slicing.

1) What is the output?

```
my_str =
'http://reddit.com/r/python'
print(my_str[17:])
```

Check Show answer

2) What is the output?

```
my_str =
'http://reddit.com/r/python'
protocol = 'http://'
print(my_str[len(protocol):])
```

Check Show answer

## The slice stride

Slice notation also provides for a third argument known as the stride. The ***stride*** determines how much to increment the index after reading each element. For example, `my_str[0:10:2]` reads every other element between 0 and 10. The stride defaults to 1 if not specified.

Figure 7.1.3: Slice stride.



```
numbers = '0123456789'

print(f'All numbers: {numbers[::]}')
print(f'Every even number: {numbers[::2]}')
print(f'Every third number between 1 and 8: {numbers[1:9:3]}')
```

```
All numbers: 0123456789
Every even number: 02468
Every third number between 1 and 8: 147
```

**PARTICIPATION ACTIVITY** 7.1.5: Slice stride.

1) What is the output?

```
my_str = 'Agt2t3afc2kjMhagrds!'
print(my_str[0:5:1])
```

Check Show answer

2) What is the output?

```
my_str = 'Agt2t3afc2kjMhagrds!'
print(my_str[::2])
```

Check Show answer



**CHALLENGE ACTIVITY** 7.1.1: Enter the output of the sliced string.

566436.4601014.qx3zqy7

Start

Type the program's output

```
city = 'Dublin'
city_slice = city[0:5]
print(city_slice)
```

Dubli

1 2 3 4 5



Check Next

**CHALLENGE ACTIVITY** 7.1.2: String slicing.

566436.4601014.qx3zqy7

Start

Integers start_index and end_index are read from input. Assign sub_saying with the result of slicing full_ end_index.

▶ **Click here to show example**

```
start_index = int(input())
end_index = int(input())
full_saying = "The best is yet to come, and won't that be fine"

''' Your code goes here '''

print('Full saying:', full_saying)
print('Sliced saying:', sub_saying)
```

1 2 3



Check Next level

# 7.2 Advanced string formatting

## Field width

A program must display formatted output beyond the ability of basic print usage such as `print(x)`. Consider a program that displays a table of soccer player statistics:



Figure 7.2.1: A formatted table of soccer statistics.

```
Player Name      Goals    Games Played     Goals Per
Game
-----------------------------------------------------
----
Sadio Mane         22          36               0.61
Mohamed Salah      22          38               0.58
Sergio Aguero      21          33               0.64
Jamie Vardy        18          34               0.53
Gabriel Jesus       7          29               0.24
```

Note in the above example how the text is formatted into columns with the contents of each column (except the leftmost column) centered under the column header. A programmer could achieve this careful formatting by placing spaces into their output strings, but each row would require different numbers of spaces depending on the player name (longer names require fewer spaces between the first and second columns).

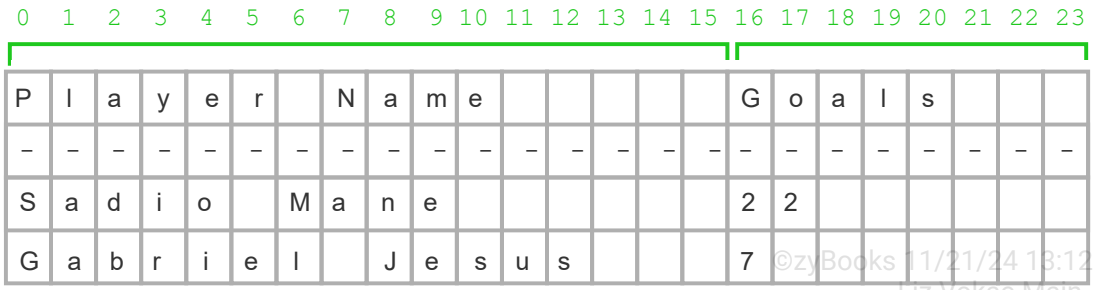
A format specification may include a ***field width*** that defines the minimum number of characters that must be inserted into the string. If the replacement value is smaller in size than the given field width, then the string is padded with space characters. Field widths set on each column in the example above cause the output to be formatted. A field width is defined in a format specification by including an integer after the colon, as in `{name:16}` to specify a width of 16 characters. Numbers will be right-aligned within the width by default, whereas most other types like strings will be left-aligned.

**PARTICIPATION ACTIVITY** 7.2.1: Field width.



```
print(f'{"Player Name":16}{"Goals":8}')
print('-' * 24)

print(f'{"Sadio Mane":16}{"22":8}')
print(f'{"Gabriel Jesus":16}{"7":8}')
```






## Animation content:

Static Figure:
Begin python code:
print(f'{"Player Name":16}{"Goals":8}')
print('-' * 24)
print(f'{"Sadio Mane":16}{"22":8}')
print(f'{"Gabriel Jesus":16}{"7":8}')
End python code.

Each character of each print statement is shown on a table. Each print statement occupies a row in the table, with there being 4 total rows. Each column of the table is labeled with an index starting from 0 and ending with 23. Indexes 0 to 15 are grouped together, and indexes 16 to 23 are also grouped together.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P | l | a | y | e | r | | N | a | m | e | | | | | | G | o | a | l | s | | | |
| - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| S | a | d | i | o | | M | a | n | e | | | | | | | 2 | 2 | | | | | | |
| G | a | b | r | i | e | l | | J | e | s | u | s | | | | 7 | | | | | | | |



Below the table is an output console containing the print statement outputs. The output console contains:

```
Player Name     Goals
------------------------
Sadio Mane      22
Gabriel Jesus   7
```

Step 1: "Player Name" is inserted into the leftmost part of the first 16-character wide field. "Goals" is inserted into the leftmost part of the second 8-character wide field. The lines of code "print(f'{"Player Name":16}{"Goals":8}')" and "print('-' * 24)" are highlighted, and the contents of the print statements are shown in the table. Each of the print statements contents from the table are then displayed on the output console below, printing in groups based on the print statements. "Player Name " is printed (index 0 to 15), and on the same line, "Goals " is printed afterwards (index 16 to 23). The next row of the table is then printed (index 0 to 23). The output console starts empty, and eventually shows:

```
Player Name     Goals
------------------------
```

Step 2: The inserted values align themselves automatically according to the field width. Note: The numbers output here are strings (Ex: "22") and are left-aligned. Numerical types, like integers, would be right-aligned within the width by default. The lines of code, "print(f'{"Sadio Mane":16}{"22":8}')" and "print(f'{"Gabriel Jesus":16}{"7":8}')" are highlighted, and the contents of the print statements are shown in the table. Each of the print statements contents from the table are then displayed on the output console below, printing in groups based on the print statements. "Sadio Mane " is printed (index 0 to 15), and on the same line, "22 " is printed afterwards (index 16 to 23). "Gabriel Jesus " is printed on a new line (index 0 to 15), and on the same line, "7 " is printed afterwards (index 16 to 23). Finally, the output console shows:

```
Player Name     Goals
------------------------
Sadio Mane      22
Gabriel Jesus   7
```

## Animation captions:

1. "Player Name" is inserted into the leftmost part of the first 16-character wide field. "Goals" is inserted into the leftmost part of the second 8-character wide field.
2. The inserted values align themselves automatically according to the field width. Note: The numbers output here are strings (Ex: "22") and are left-aligned. Numerical types, like integers, would be right-aligned within the width by default.



**PARTICIPATION ACTIVITY** 7.2.2: Format specification field widths.

1) Complete the format specification to assign a field width of 10 characters.

{name:}

Check Show answer

2) Write a complete replacement field that assigns a field with named value "count" and a field width of 5.



Check Show answer

**CHALLENGE ACTIVITY** 7.2.1: Field widths.

Notes:

- print() ends with a newline, so **don't forget to include a newline** by pressing Enter or Return.
- Strings are left-aligned by default.
- Numbers are right-aligned by default.

566436.4601014.qx3zqy7

Start

Type the program's output

```
name = 'Joseph'
print(f'{name:8}')
```

Joseph

1 2

Check Next



## Aligning text

A format specification can include an ***alignment character*** that determines how a value should be aligned within the width of the field. Alignment is set in a format specification by adding a special character before the field width integer. The basic set of possible alignment options include left-aligned (<), right-aligned (>) and centered (^). Numbers will be right-aligned within the width by default, whereas most other types like strings will be left-aligned.

## Figure 7.2.2: Aligning strings within a field.

Consider the following code that prints a table, and how changing the alignment impacts the column organization.

```
names = ['Sadio Mane', 'Gabriel Jesus']
goals = [22, 7]

print(<f-string 1>)          #Replaced in table below
print('-' * 24)
for i in range(2):
    print(<f-string 2>)      #Replaced in table below
```



| Alignment type | <f-string 1><br><f-string 2> | Output |
|---|---|---|
| Left-aligned | `f'{"Player Name":<16}{"Goals":<8}'`<br>`f'{names[i]:<16}{goals[i]:<8}'` | `Player Name Goals`<br>`------------------------`<br>`-`<br>`Sadio Mane 22`<br>`Gabriel Jesus 7` |
| Right-aligned | `f'{"Player Name":>16}{"Goals":>8}'`<br>`f'{names[i]:>16}{goals[i]:>8}'` | `Player Name`<br>`Goals`<br>`------------------------`<br>`-`<br>`Sadio Mane`<br>`22`<br>`Gabriel Jesus`<br>`7` |
| Centered | `f'{"Player Name":^16}{"Goals":^8}'`<br>`f'{names[i]:^16}{goals[i]:^8}'` | `Player Name Goals`<br>`------------------------`<br>`-`<br>`Sadio Mane 22`<br>`Gabriel Jesus 7` |

**PARTICIPATION ACTIVITY** 7.2.3: Aligning text in fields.



For each question, determine the value of the given expression.

1) `f'{"Bob":<5}'`

Check Show answer

2) `f'{"Bob":>5}'`

Check Show answer

3) `f'{"Bob":^5}'`



Check Show answer

4) `f'{"Bob":<5}{1:<2}'`

Check Show answer

5) `f'{"Bob":<5}{1:>2}'`

Check Show answer

## Fill

The ***fill character*** is used to pad a replacement field when the inserted string is smaller than the field width. The default fill character is an empty space `' '`. A programmer may define a different fill character in a format specification by placing the different fill character before the alignment character. Ex: `{score:0>4}` generates "0009" if score is 9 or "0250" if score is 250.

Table 7.2.1: Using fill characters to pad tables.

| Format specification | Value of score | Output |
|---|---|---|
| `{score:}` | 9 | `9` |
| `{score:4}` | 9 | `   9` |
| `{score:0>4}` | 9 | `0009` |

| | | |
|---|---|---|
| {score:0>4} | 18 | 0018 |
| {score:0^4} | 18 | 0180 |

A programmer can set different alignments, widths, and fills on each field to construct neatly formatted output, as demonstrated below.



## PythonTutor: Fill characters in strings.

```
1  name = 'Wayne Rooney'
2  goals = 36
3
4  # Use default empty space fill character
5  print(f'{name:<16}{goals:>6}')
6
7  # Use '0' as a fill character for score
8  print(f'{name:<16}{goals:0>6}')
9
10 # Use '_' as fill character for name
11 print(f'{name:_<16}{goals:0>6}')
```

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 5

Program output



**PARTICIPATION ACTIVITY** 7.2.4: Fill characters.

1) What's the fill character in the following format specification?

```
{score:*>4}
```

- score
- *
- :
- 4



2) What's the fill character in the following format specification?

```
{score:>4}
```

- >
- *
- 4
- space character

3) If `name = 'Sally'`, what is the result of: `{name:@>8}`?

- Sally@@@
- Sally
- @Sally@@
- @@@Sally
- Sally>>>

## Floating-point precision

A programmer sets the number of digits to the right of a floating-point number to print. The optional ***precision*** component of a format specification indicates how many digits should be included in the output of floating types. The precision follows the field width component in the format specification, if a width is specified at all, and starts with a period character. Ex: `f'{1.725:.1f}'` indicates a precision of 1, thus the resulting string would be '1.7'.



If the specified precision is greater than the number of digits available, trailing 0s are appended. Ex: `f'{1.5:.3f}'` results in the string '1.500'. If the specified precision is less than the existing precision in the given number, then the number is rounded. Ex: `f'{1.666:.2f}'` results in the string '1.67'.

Figure 7.2.3: String formatting example: Setting precision of floating-point values.

```
import math
real_pi = math.pi  # math library provides close approximation of pi
approximate_pi = 22.0 / 7.0  # Approximately correct pi to within 2 decimal places

print(f'pi is {real_pi}')
print(f'22/7 is {approximate_pi}')
print(f'22/7 looks better like {approximate_pi:.2f}')
```

```
pi is 3.141592653589793
22/7 is 3.142857142857143
22/7 looks better like 3.14
```



**PARTICIPATION ACTIVITY** 7.2.5: Floating-point precision in formatted strings.

Fill in the string that results from evaluating the given expression.

1) `f'{5:.1f}'`

'5.'

Check Show answer

2) `f'{5:.3f}'`

'5.'

Check Show answer

3) `f'{5.25:.3f}'`

''

Check Show answer

4) `f'{5.2589:.3f}'`

''

Check Show answer



5) `f'{5:4.1f}'`

''

Check Show answer

**CHALLENGE ACTIVITY** 7.2.2: Advanced string formatting.

566436.4601014.qx3zqy7

Start



String color_name is read from input. Use two print(f' ') statements to output the following two lines:

- color_name with a width of 10 characters, left-aligned, and with the fill character '+'.
- color_name with a width of 10 characters, right-aligned, and with the fill character '+'.

▶ **Click here for example**

```
color_name = input()

''' Your code goes here '''

```

1 2

Check Next level



# 7.3 String methods

String objects have many useful methods to do things like replacing characters, converting to lowercase, capitalizing the first character, etc. The methods are made possible due to a string's

implementation as a *class*, which for purposes here can just be thought of as a mechanism supporting a set of methods for a particular type of object.

## Finding and replacing

A common task for a programmer is to edit the contents of a string. Recall that string objects are immutable - once created, strings can not be changed. To update a string variable, a new string object must be created and bound to the variable name, replacing the old object. The *replace* string method provides a simple way to create a new string by replacing all occurrences of a substring with a new substring.



- ***replace(old, new)*** - Returns a copy of the string with all occurrences of the substring old replaced by the string new. The old and new arguments may be string variables or string literals.
- ***replace(old, new, count)*** - Same as above, except replace(old, new, count) only replaces the first count occurrences of old.

### PythonTutor: replace() string method.

```
1  phrase = 'Someday I will have three goats, six horses, and nir
2
3  # Replace English with Spanish.
4  phrase = phrase.replace('one', 'uno')
5  phrase = phrase.replace('two', 'dos')
6  phrase = phrase.replace('three', 'tres')
7  phrase = phrase.replace('four', 'cuatro')
8  phrase = phrase.replace('five', 'cinco')
9  phrase = phrase.replace('six', 'seis')
10 phrase = phrase.replace('seven', 'siete')
11 phrase = phrase.replace('eight', 'ocho')
12 phrase = phrase.replace('nine', 'nueve')
13
14 print('Translation:', phrase)
```

line that just executed
next line to execute



<< First | < Prev | Next > | Last >>

Step 1 of 11

Program output

Frames Objects

Some methods are useful for finding the position of where a character or substring is located in a string:

- ***find(x)*** -- Returns the index of the first occurrence of item x in the string, otherwise, find(x) returns -1. x may be a string variable or string literal. Recall that in a string, the index of the first character is 0, not 1. If `my_str` is 'Boo Hoo!':
  - `my_str.find('!')  # Returns 7`
  - `my_str.find('Boo')  # Returns 0`
  - `my_str.find('oo')  # Returns 1 (first occurrence only)`
- ***find(x, start)*** - Same as find(x), but begins the search at index start:
  - `my_str.find('oo', 2)  # Returns 5`
- ***find(x, start, end)*** -- Same as find(x, start), but stops the search at index end - 1:
  - `my_str.find('oo', 2, 4)  # Returns -1 (not found)`
- ***rfind(x)*** -- Same as find(x) but searches the string in reverse, returning the last occurrence in the string.



Another useful function is count, which counts the number of times a substring occurs in the string:

- ***count(x)*** -- Returns the number of times x occurs in the string.
  - `my_str.count('oo')  # Returns 2`

Note that methods such as `find()` and `rfind()` are useful only for cases where a programmer needs to know the exact location of the character or substring in the string. If the exact position is not important, then the `in` membership operator should be used to check if a character or substring is contained in the string:

### Figure 7.3.1: Use 'in' to check if a character or substring is contained by another string.

```
if 'batman' in superhero_name:
    # Statements to execute if superhero_name contains 'batman' in any
position.
```

### zyDE 7.3.1: String searching example: Hangman.



The following example carries out a simple guessing game, allowing a user a number of guesses to fill out the complete word.

Load default template...

```
word = 'onomatopoeia'
num_guesses = 10

hidden_word = '-' * len(word)
```

```
4 hidden_word = '-' * len(word)
5
6 guess = 1
7
8 while guess <= num_guesses and '-' in hidden_word:
9     print(hidden_word)
10    user_input = input(f'Enter a character (guess #{guess}
11
12    if len(user_input) == 1:
13        # Count the number of times the character occurs i
14        num_occurrences = word.count(user_input)
15
16        # Replace the appropriate position(s) in hidden_wo
17        position = -1
18
```



```
y
m
n
```

Run

## Comparing strings

String objects may be compared using relational operators (<, <=, >, >=), equality operators (==, !=), membership operators (`in`, `not in`), and identity operators (`is`, `is not`).

Evaluation of relational and equality operator comparisons occurs by first comparing the corresponding characters at element 0, then at element 1, etc., stopping as soon as a determination can be made. For an equality (==) comparison, the two strings must have the same length and every corresponding character pair must be the same. For a relational comparison (<, >, etc.), the result will be the result of comparing the ASCII/Unicode values of the first differing character pair.

### Table 7.3.1: String comparisons.

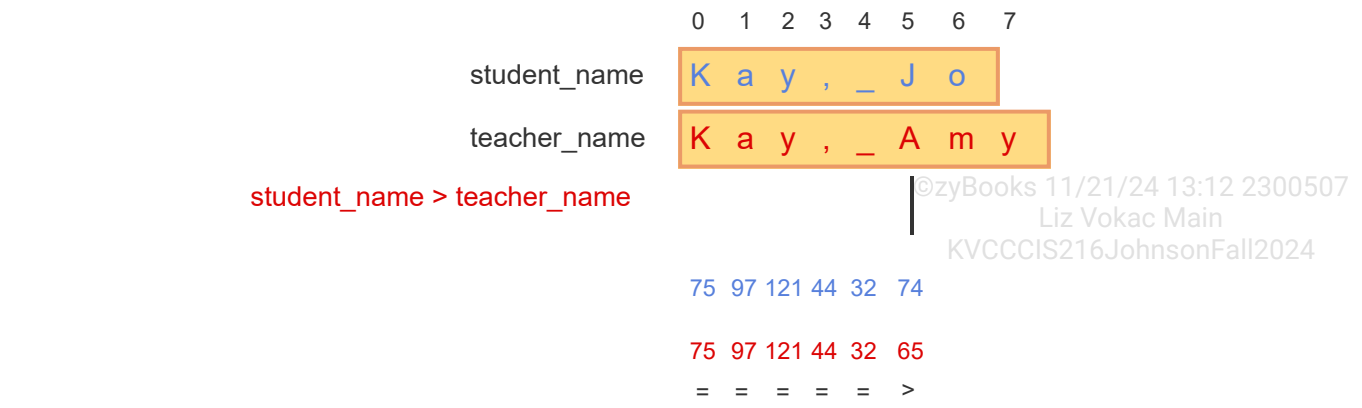
| Example | Expression result | Why? |
|---|---|---|
| `'Hello' == 'Hello'` | True | The strings are exactly identical values |
| `'Hello' == 'Hello!'` | False | The left hand string does not end with '!'. |

| | | |
|---|---|---|
| `'Yankee Sierra' > 'Amy Wise'` | True | The first character of the left side 'Y' is "greater than" (in ASCII value) the first character of the right side 'A'. |
| `'Yankee Sierra' > 'Yankee Zulu'` | False | The characters of both sides match until the second word. The first character of the second word on the left 'S' is not "greater than" (in ASCII value) the first character on the right side 'Z'. |
| `'seph' in 'Joseph'` | True | The substring 'seph' can be found starting at the 3rd position of 'Joseph'. |
| `'jo' in 'Joseph'` | False | 'jo' (with a lowercase 'j') is not in 'Joseph' (with an uppercase 'J'). |



The following animation shows the process of comparing two string variables character by character using their ASCII values. Recall that ASCII values are an integer value representation of a character. 'A' is represented by the integer value 65, 'B' by 66, 'C' by 67, and so on. An ***ASCII table*** provides a quick lookup of ASCII values. There are many ASCII tables available online, for example www.asciitable.com.

**PARTICIPATION ACTIVITY** 7.3.1: String comparison.






## Animation content:

Static Figure: 2 string variables are displayed, called "student_name" and "teacher_name". The variable "student_name" contains the string "Kay,_Jo". The variable "teacher_name" contains the string "Kay,_Amy".

Below the strings is a comparison call: "student_name > teacher_name", along with a walkthrough of the string comparison character by character. The string comparison uses ASCII values to check equality. The first 5 characters (characters 'K', 'a', 'y', ',', and '_' at indexes 0 to 4) are the same, and are represented by the ASCII values 75, 97, 121, 44, 32. A cursor showing where we are in comparing the strings stops at index 5. In the first string, the character is 'J' while in the second string, the character is 'A'. The ASCII value of 'J' is 74 and the ASCII value of 'A' is 65, therefore "student_name" is greater than "teacher_name".



Step 1: Each comparison uses ASCII values. A string comparison is called: "student_name > teacher_name". The first characters of each variable are compared using their ASCII values. Both variables start with the character 'K' with an ASCII value of 75, so index 0 for both variables are equal.

Step 2: Values at indexes 0-4 are the same for both student_name and teacher_name. The characters at indexes 1 to 4 are compared. The characters for both variables are the same, so the comparison finds them to be equal in value. All characters have been equal in ASCII value for both variables so far.

Step 3: 'J' is greater than 'A', so student_name is greater than teacher_name. The cursor stops at index 5 to compare values. "J" is converted to 74 in ASCII, and "A" is converted to 65. The comparison finds that 74 is greater than 65, therefore "student_name" is greater than "teacher_name".

### Animation captions:

1. Each comparison uses ASCII values.
2. Values at indexes 0-4 are the same for both student_name and teacher_name.
3. 'J' is greater than 'A', so student_name is greater than teacher_name.

If one string is shorter than the other with all corresponding characters equal, then the shorter string is considered less than the longer string.

The membership operators (`in`, `not in`) provide a simple method for detecting whether a specific substring exists in the string. The argument to the right of the operator is examined for the existence of the argument on the left. Note that reversing the arguments does not work, as 'Jo' is a substring of 'Kay, Jo', but 'Kay, Jo' is not a substring of 'Jo'.



The identity operators (`is`, `is not`) determine whether the two arguments are bound to the same object. *A common error is to use an identity operator in place of an equality operator.* Ex: A programmer may write `name is 'Amy Adams'`, intending to check if the value of `name` is the same as the literal 'Amy Adams'. Instead, the Python interpreter creates a new string object from the string literal on the right, and compares the identity of the new object to the `name` object, which returns False. *Good practice is to always use the equality operator== when comparing values.*

Figure 7.3.2: Identity vs. equality operators.

```
student_name = input('Enter student name:
')

if student_name is 'Amy Adams':
    print('Identity operator: True')
else:
    print('Identity operator: False')

if student_name == 'Amy Adams':
    print('Equality operator: True')
else:
    print('Equality operator: False')
```

```
Enter student name: Amy
Adams
Identity operator: False
Equality operator: True
```



Because comparison uses the encoded values of characters (ASCII/Unicode), comparison may not behave intuitively for some situations. Comparisons are case-sensitive, so 'Apple' does not equal 'apple'. In particular, because the encoded value for 'A' is 65, and for 'a' is 97, then 'Apple' is less-than 'apple'. Furthermore, 'Banana' is less than 'apple', because 'B' is 66 while 'a' is 97.

A number of methods are available to help manage string comparisons. The list below describes the most commonly used methods; a full list is available at docs.python.org.

- Methods to check a string value that returns a True or False Boolean value:
  - ***isalnum()*** -- Returns True if all characters in the string are lowercase or uppercase letters, or the numbers 0-9.
  - ***isdigit()*** -- Returns True if all characters are the numbers 0-9.
  - ***islower()*** -- Returns True if all cased characters are lowercase letters.
  - ***isupper()*** -- Returns True if all cased characters are uppercase letters.
  - ***isspace()*** -- Returns True if all characters are whitespace.
  - ***startswith(x)*** -- Returns True if the string starts with x.
  - ***endswith(x)*** -- Returns True if the string ends with x.

Note that the methods `islower()` and `isupper()` ignore non-cased characters. Ex: `'abc?'.islower()` returns True, ignoring the question mark.



**PARTICIPATION ACTIVITY** 7.3.2: String methods: Boolean string comparisons.

Determine whether the given expression evaluates to True or False.

1) `'HTTPS://google.com'.isalnum()`

   ○ True

- False

2) `'HTTPS://google.com'.startswith('HTTP')`

- True
- False

3) `'\n \n'.isspace()`

- True
- False

4) `'1 2 3 4 5'.isdigit()`

- True
- False

5) `'LINCOLN, ABRAHAM'.isupper()`

- True
- False



## Creating new strings from a string

A programmer often needs to transform two strings into similar formats to perform a comparison. The list below shows some of the more common string methods that create string copies, altering the case or amount of whitespace of the original string:

- Methods to create new strings:
  - ***capitalize()*** -- Returns a copy of the string with the first character capitalized and the rest lowercased.
  - ***lower()*** -- Returns a copy of the string with all characters lowercased.
  - ***upper()*** -- Returns a copy of the string with all characters uppercased.
  - ***strip()*** -- Returns a copy of the string with leading and trailing whitespace removed.
  - ***title()*** -- Returns a copy of the string as a title, with first letters of words capitalized.

A user may enter any one of the non-equivalent values 'Bob' , 'BOB ', or 'bob' into a program that reads in names. The statement `name = input().strip().lower()` reads in the user input, strips the leading and trailing whitespace, and changes all the characters to lowercase. Thus, user input of 'Bob', 'BOB ', or 'bob' would each result in name having just the value 'bob'.



*Good practice when reading user-entered strings is to apply transformations when reading in data (such as input), as opposed to later in the program.* Applying transformations immediately limits the likelihood of introducing bugs because the user entered an unexpected string value. Of course, there are many examples of programs in which capitalization or whitespace should indicate a unique string - the programmer should use discretion depending on the program being implemented.

## zyDE 7.3.2: String methods example: Passenger database.

The example program below shows how the above methods might be used to store passenger names and travel destinations in a database. The use of `strip()`, `lower()`, and `upper()` standardize user input for easy comparison.

Run the program below and add passengers into the database. Add a duplicate passenger name, using different capitalization, and print the list again.



Load default template...

```
menu_prompt = ('Available commands:\n'
               '  (add) Add passenger\n'
               '  (del) Delete passenger\n'
               '  (print) Print passenger list\n'
               '  (exit) Exit the program\n'
               'Enter command:\n')

destinations = ['PHX', 'AUS', 'LAS']

destination_prompt = ('Available destinations:\n'
                      '(PHX) Phoenix\n'
                      '(AUS) Austin\n'
                      '(LAS) Las Vegas\n'
                      'Enter destination:\n')

passengers = {}

print('Welcome to Mohawk Airlines!\n')
```

```
add
Dusty Baker
PHX
```

Run



**CHALLENGE ACTIVITY** 7.3.1: String methods.

566436.4601014.qx3zqy7

Start

String vowel_letters is read from input. If 'u' is in vowel_letters, then:

- Output 'At position: ' followed by the index of the first occurrence.
- Create a string from vowel_letters with all occurrences of 'u' replaced by '+' and output vowel_lette

Otherwise, output 'Not found'.

▶ **Click here for example**

```
vowel_letters = input()

''' Your code goes here '''
```



1 2 3

Check Next level

# 7.4 Splitting and joining strings

## The split() method

A common programming task is to break a large string down into the comprising substrings. The string method ***split()*** splits a string into a list of tokens. Each ***token*** is a substring that forms a part of a larger string. A ***separator*** is a character or sequence of characters that indicates where to split the string into tokens.



Ex: `'Martin Luther King Jr.'.split()` splits the string literal "Martin Luther King Jr." using any whitespace character as the default separator and returns the list of tokens ['Martin', 'Luther', 'King', 'Jr.'].

The separator can be changed by calling split() with a string argument. Ex: `'a#b#c'.split('#')` uses the "#" separator to split the string "a#b#c" into the three tokens ['a', 'b', 'c'].

**PARTICIPATION ACTIVITY** 7.4.1: Splitting a string into tokens.




## Animation content:

Static figure: A code block is displayed.
Begin Python code:
string = 'Music/artist/song.mp3'
my_tokens = string.split('/')
End Python code.

Step 1: Original string contains a pathname to an mp3 of your favorite song. The line of code, string = 'Music/artist/song.mp3', is highlighted, and the text, 'Music/artist/song.mp3', is displayed.
Step 2: The pathname is split using the delimiter ' / '. The line of code, my_tokens = string.split('/'), is highlighted, and the delimiters ,'/', are highlighted in the text, 'Music/artist/song.mp3'. The text, 'Music' 'artist' 'song.mp3', is displayed.
Step 3: The variable my_tokens is assigned with the 3 tokens as a list of strings. The text, 'Music' 'artist' 'song.mp3', is now, my_tokens = [ 'Music', 'artist', 'song.mp3' ].
Step 4: When split() is called with no argument, the delimiter defaults to a space character. A new code block is displayed:
Begin Python code:
string = 'I love python'
my_tokens = string.split()
End Python code.
The line of code, string = 'I love python', is highlighted, and the text, 'I love python', is displayed. The line of code, my_tokens = string.split(), is highlighted, and the text, my_tokens = [ 'I', 'love', 'python' ], is displayed.



## Animation captions:

1. Original string contains a pathname to an mp3 of your favorite song.
2. The pathname is split using the delimiter ' / '.
3. The variable my_tokens is assigned with the 3 tokens as a list of strings.
4. When split() is called with no argument, the delimiter defaults to a space character.

## Figure 7.4.1: String split example.



```
url = input('Enter URL:\n')

tokens = url.split('/')   # Uses '/' separator
print(tokens)
```

```
Enter URL:
http://en.wikipedia.org/wiki/Lucille_ball
['http:', '', 'en.wikipedia.org', 'wiki',
'Lucille_ball']
...
Enter URL: en.wikipedia.org/wiki/ethernet/
['en.wikipedia.org', 'wiki', 'ethernet', '']
```

The example above shows how split() might be used to find the elements of a path to a web page; the separator used is the forward slash character '/'. The split() method creates a new list, ordered from left to right, containing a new string for each sequence of characters located between '/' separators. Thus the URL http://en.wikipedia.org/wiki/Lucille_ball is split into ['http:', '', 'en.wikipedia.org', 'wiki', 'Lucille_ball']. The separator character is not included in the resulting strings.

If the split string starts or ends with the separator, or if two consecutive separators exist, then the resulting list will contain an empty string for each such occurrence. Ex: The consecutive forward slashes of 'http://' and the ending forward slash of '.../wiki/ethernet/' generate empty strings. If the separator argument is omitted from split(), thus splitting the string wherever whitespace occurs, then no empty strings are generated.

## zyDE 7.4.1: More string splitting.

Run the following program and observe the output. Edit the program by changing the split() method separator to "//" and " " and observe the output.

Load default template... Run



```
1 file = 'C:/Users/Charles Xavi
2
3 separator = '/'
4 results = file.split(separato
5 print(f'Separator ({separator
6
```

**PARTICIPATION ACTIVITY** 7.4.2: String split() method.

Use the variable song to answer the questions below.

```
song = "I scream; you scream; we all scream, for ice cream.\n"
```

1) What is the result of `song.split()`?

- `['I scream; you scream; we all scream, for ice cream.\n']`
- `['I scream;', 'you scream;', 'we all scream,', 'for ice cream.\n']`
- `['I', 'scream;', 'you', 'scream;', 'we', 'all', 'scream,', 'for', 'ice', 'cream.']`

2) What is the result of `song.split('\n')`?

- `['I scream; you scream; we all scream, for ice cream.', '']`
- `['I scream; you scream;\n', 'we all scream,\n', 'for ice cream.\n']`
- `['I scream; you scream; we all scream, for ice cream']`



3) What is the result of `song.split('scream')`?

- `['I ', '; you ', '; we all ', ', for ice cream.\n']`

```
['I scream; you scream; we
all scream, for ice
cream.\n']
```

```
['I', 'you', 'we all',
'for ice cream.\n']
```

## The join() method



The ***join()*** string method performs the inverse operation of split() by joining a list of strings together to create a single string. Ex: `my_str = '@'.join(['billgates', 'microsoft'])` assigns my_str with the string 'billgates@microsoft'. The separator '@' provides a join() method that accepts a single list argument. Each element in the list, from left to right, is concatenated to create a new string object with the separator placed between each list element. The separator can be any string, including multiple characters or an empty string.

**PARTICIPATION ACTIVITY** 7.4.3: String join() method.

```
web_path = [ 'www.website.com', 'profile', 'settings' ]
separator = '/'
url = separator.join(web_path)
```

```
url = 'www.website.com/profile/settings'
```

### Animation content:

Static figure: A code block is displayed.
Begin Python code:
web_path = ['www.website.com', 'profile', 'settings']
separator = '/'
url = separator.join(web_path)
End Python code.



Step 1: web_path is a list of strings that form the path of the web page. The line of code, web_path = ['www.website.com', 'profile', 'settings'], is highlighted and the text, www.website.com, profile, and, settings, are displayed.
Step 2: Create a string with the separator "/". The line of code, separator = '/', is highlighted, and the separator, /, is displayed.
Step 3: Then join() concatenates the list of strings with the separator "/". The line of code, url =

separator.join(web_path), is highlighted, and the text joins together and becomes, url = 'www.website.com/profile/settings'.

### Animation captions:

1. web_path is a list of strings that form the path of the web page.
2. Create a string with the separator "/".
3. Then join() concatenates the list of strings with the separator "/".



A useful application of the join() method is to build a new string without separators. The empty string ('') is a perfectly valid string object, just with a length of 0. So the statement `''.join(['http://', 'www.', 'ebay', '.com'])` produces the string 'http://www.ebay.com'.

## Figure 7.4.2: String join() example: Comparing join vs. loops.

The following programs are equivalent, but join() is a simpler approach that uses less code and is easier to read.

```
phrases = ['To be, ', 'or not to be.\n', 'That is the
question.']

sentence = ''
for phrase in phrases:
    sentence += phrase
print(sentence)
```

```
To be, or not to
be.
That is the
question.
```

```
phrases = ['To be, ', 'or not to be.\n', 'That is the
question.']

sentence = ''.join(phrases)
print(sentence)
```

```
To be, or not to
be.
That is the
question.
```

**PARTICIPATION ACTIVITY** 7.4.4: String join() method.



1) Write a statement that uses the join() method to set my_str to 'images.google.com', using the list x = ['images', 'google', 'com']

```
my_str =
```

Check Show answer

2) Write a statement that uses the join() method to set my_str to 'NewYork', using the list x = ['New', 'York']

```
my_str =
```

Check Show answer



## Using the split() and join() methods together

The split() and join() methods are commonly used together to replace or remove specific sections of a string. Ex: A programmer may want to change 'C:/Users/Brian/report.txt' to 'C:\\Users\\Brian\\report.txt', perhaps because a different operating system uses different separators to specify file locations. The example below illustrates how split() and join() are used together.

Figure 7.4.3: Splitting and joining: Replacing separators.

```
path = input('Enter file name: ')

new_separator = input('Enter new separator:
')
tokens = path.split('/')
print(new_separator.join(tokens))
```

```
Enter file name:
C:/Users/Wolfman/Documents/report.pdf
Enter new separator: \\
C:\\Users\\Wolfman\\Documents\\report.pdf
```

A programmer may also want to add, remove, or replace specific token(s) from a string. Ex: The program below reads in a URL and checks whether the fourth token (index 3) is 'wiki', as Wikipedia URLs follow the format of http://language.wikipedia.org/wiki/topic. If 'wiki' is missing from the URL, the program uses the list method insert() to correct the URL by adding 'wiki' before index 3.



Figure 7.4.4: Splitting and joining: Editing tokens.

```
url = input('Enter Wikipedia URL: ')

tokens = url.split('/')

if 'wiki' != tokens[3]:
    tokens.insert(3, 'wiki')
    new_url = '/'.join(tokens)

    print(f'{url} is not a valid address.')
    print(f'Redirecting to {new_url}')
else:
    print(f'Loading {url}')
```



```
Enter Wikipedia URL:
http://en.wikipedia.org/wiki/Rome
Loading http://en.wikipedia.org/wiki/Rome
...
Enter Wikipedia URL: http://en.wikipedia.org/Rome
http://en.wikipedia.org/Rome is not a valid
address.
Redirecting to http://en.wikipedia.org/wiki/Rome
```

**PARTICIPATION ACTIVITY** 7.4.5: Splitting and joining strings.

1) Write a statement that replaces the separators in the string variable title from hyphens (-) to colons (:)

```
title = 'Python-Lab-Warmup'
tokens = title.split('-')
title =
```

Check Show answer

**CHALLENGE ACTIVITY** 7.4.1: String split and join.



566436.4601014.qx3zqy7

Start

Type the program's output

```
item_info = 'Hat 12 15'

item_tokens = item_info.split()
item = item_tokens[0]
quantity = item_tokens[1]
price = item_tokens[2]

print(f'{item} stock: {quantity}')
print(f'Price: {price}')
```

```
Hat stock: 12
Price: 15
```

1 2



Check Next

**CHALLENGE ACTIVITY** 7.4.2: Splitting and joining strings.

566436.4601014.qx3zqy7

Start

String instructor_names is read from input. Split instructor_names into tokens using a period (".") as the patients_list with the result.

Ex: If the input is `Yulia.Sahar.Astrid`, then the output is:

```
['Yulia', 'Sahar', 'Astrid']
```

```
1 instructor_names = input()
2
3 """ Your code goes here """
4
5 print(patients_list)
```



1 2

Check Next level

# 7.5 LAB: Checker for integer string

This section's content is not available for print.



# 7.6 LAB: Name format

Many documents use a specific format for a person's name. Write a program that reads a person's name in the following format:

firstName middleName lastName (in one line)

and outputs the person's name in the following format:

lastName, firstInitial.middleInitial.

Ex: If the input is:

```
Pat Silly Doe
```

the output is:

```
Doe, P.S.
```

If the input has the following format:

firstName lastName (in one line)

the output is:

lastName, firstInitial.

Ex: If the input is:

```
Julia Clark
```



the output is:

```
Clark, J.
```

566436.4601014.qx3zqy7

LAB ACTIVITY | 7.6.1: LAB: Name format | 10 / 10

main.py — Load default template...

```
1  user_name = input().strip().lower()
2  tokens = user_name.split()
3
4  if len(tokens) == 3:
5      first_name = tokens[0]
6      middle_name = tokens[1]
7      last_name = tokens[2]
8
9      caps_first_name = first_name.capitalize()
10     caps_first_init = caps_first_name[0]
11
12     caps_middle_name = middle_name.capitalize()
13     caps_middle_init = caps_middle_name[0]
14
15     caps_last_name = last_name.capitalize()
16
17     print(f'{caps_last_name}, {caps_first_init}.{caps_middle_init}.')
18
```



Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → Output

Program output displayed here



Coding trail of your work What is this?

11/6 W-----0--10 min:10

# 7.7 LAB: Count characters

This section's content is not available for print.



# 7.8 LAB: Mad Lib - loops

Mad Libs are activities that have a person provide various words, which are then used to complete a short story in unexpected (and hopefully funny) ways.

Write a program that takes a string and an integer as input, and outputs a sentence using the input values as shown in the example below. The program repeats until the input string is `quit` and disregards the integer input that follows.

Ex: If the input is:

```
apples 5
shoes 2
quit 0
```

the output is:

```
Eating 5 apples a day keeps you happy and healthy.
Eating 2 shoes a day keeps you happy and healthy.
```

566436.4601014.qx3zqy7

**LAB ACTIVITY** 7.8.1: LAB: Mad Lib - loops 10 / 10

main.py Load default template...

```
1  user_input = input()
2  tokens = user_input.split()
3
4  user_word = tokens[0]
5  user_int = tokens[1]
6
7
8  while user_word != 'quit':
9
10     print(f'Eating {user_int} {user_word} a day keeps you happy and healthy.')
11     user_input = input()
12     tokens = user_input.split()
13     user_word = tokens[0]
14     user_int = tokens[1]
```

```
14      user_int = tokens[1]
15
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.



Enter program input (optional)

```
apples 5
grapes 23
```

Run program

Input (from above) → main.py (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

```
11/6 W---0------------ R-------0-0---10 min:34
```

# 7.9 LAB: Palindrome

This section's content is not available for print.



# 7.10 LAB: Remove all non-alpha characters

This section's content is not available for print.