10.1 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 to 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 10.1.1: 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 10.1.1: Slicing.

Animation content:

```
undefined
```

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 | 10.1.2: 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 10.1.2: A slice creates a new object.

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

my_str = 'The fox jumped the brown llama'
print('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 10.1.3: String slicing tool.

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

Output: 'ey f'

H	e	y		f	o	l	k	s	!
	↑				↑				
	start				end				

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

zyDE 10.1.1: Slicing example: omitting start, end indices.

Run the program below.

Load default template...

```
1 usr_text = input('Enter a string: ')
2 print()
3
4 first_half = usr_text[:len(usr_text)//2
5 last_half = usr_text[len(usr_text)//2:]
6
7 print('The first half of the string is '
8 print('The second half of the string is
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 10.1.1: 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	Gets the characters in indices 10-18.
`my_str[10:-5]`	wikipedia.org/wiki/	Gets the characters in indices 10-28.
`my_str[8:]`	n.wikipedia.org/wiki/Nasa/	All characters from index 8 until the end of the string.
`my_str[:23]`	http://en.wikipedia.org	Every character up to index 23, but not including my_str[23].
`my_str[:-1]`	http://en.wikipedia.org /wiki/Nasa	All but the last character.

PARTICIPATION ACTIVITY 10.1.4: 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 10.1.3: Slice stride.


```
numbers = '0123456789'

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

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

PARTICIPATION ACTIVITY 10.1.5: 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 10.1.1: String slicing.

334598.939404.qx3zqy7

Start

Type the program's output

```
state = 'Maine'
state_slice = state[0:4]
print(state_slice)
```

1 2 3 4 5

Check Next

CHALLENGE ACTIVITY 10.1.2: Slice a rhyme.

Assign sub_lyric by slicing rhyme_lyric from start_index to end_index which are given as inputs.

Sample output with inputs: 4 7

```
COW
```

334598.939404.qx3zqy7

```
1 start_index = int(input())
2 end_index = int(input())
3 rhyme_lyric = 'The cow jumped over the moon.'
4 sub_lyric = rhyme_lyric[''' Your solution goes here '''] 
5 print(sub_lyric)
```


Run

View your last submission

10.2 Advanced string formatting

Field width

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

Figure 10.2.1: 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) 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 requires less 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's left side is padded with space characters. Field widths set on each column in the example above cause the output to be formatted nicely.

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 10.2.1: Field width.

Animation content:

undefined

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.

PARTICIPATION ACTIVITY | 10.2.2: 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" a field width of 5.

Check Show answer

CHALLENGE ACTIVITY | 10.2.1: Field widths.

334598.939404.qx3zqy7

Start

Type the program's output

```
format_string = '{name:10}'
print(format_string.format(name='Henry'))
```


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 '^'.

Figure 10.2.2: Aligning strings within a field.

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

```
format_string = <format_string>  # Replaced in table below

print(format_string.format(name='Player Name', goals='Goals'))
print('-' * 24)

print(format_string.format(name='Sadio Mane', goals=22))
print(format_string.format(name='Gabriel Jesus', goals=7))
```

Alignment type	<format_string>	Output

Left-aligned	`'{name:<16}{goals:<8}'`	`Player Name     Goals`
		`------------------------`
		`Sadio Mane      22`
		`Gabriel Jesus   7`

Right-aligned	`'{name:>16}{goals:>8}'`	`     Player Name   Goals`
		`------------------------`
		`      Sadio Mane      22`
		`   Gabriel Jesus       7`

Centered	`'{name:^16}{goals:^8}'`	`  Player Name    Goals`
		`------------------------`
		`   Sadio Mane      22`
		`Gabriel Jesus     7`

PARTICIPATION ACTIVITY 10.2.3: Aligning text in fields.

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

1) `'{name:<5}'.format(name='Bob')`

Check Show answer

2) `'{name:>5}'.format(name='Bob')`

Check Show answer

3) `'{name:^5}'.format(name='Bob')`

Check Show answer

4)
```
x = '{name:<5}{score:
<2}'.format(name='Bob', score=1)
```

Check Show answer

5)
```
'{name:
<5}{score:>2}'.format(name='Bob',
score=1)
```

Check Show answer

Fill

The ***fill character*** is used to pad a replacement field when the string being inserted 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 10.2.1: 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 nicely formatted output, as demonstrated below.

PARTICIPATION ACTIVITY 10.2.4: Fill characters in strings.

PARTICIPATION ACTIVITY 10.2.5: 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

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

- space character
- Sally@@@
- Sally
- @Sally@@
- @@@Sally
- Sally>>>

Floating-point precision

A programmer commonly wants to set how many digits to the right of a floating-point number to print. The optional ***precision*** component of a format specification indicates how many digits to the right of the decimal 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: `'{:.1f}'.format(1.725)` 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: `'{:.3f}'.format(1.5)` 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: `'{:.2f}'.format(1.666)` results in the string '1.67'.

Figure 10.2.3: 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('pi is {pi}'.format(pi=real_pi))
print('22/7 is {pi}'.format(pi=approximate_pi))
print('22/7 looks better like
{pi:.2f}'.format(pi=approximate_pi))
```

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

PARTICIPATION ACTIVITY 10.2.6: Floating-point precision in formatted strings.

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

1) `'{x:.1f}'.format(x=5)`

'5.' '

Check Show answer

2) `'{x:.3f}'.format(x=5)`

'5.' '

Check Show answer

3) `'{x:.3f}'.format(x=5.25)`

' '

Check Show answer

4) `'{x:.3f}'.format(x=5.2589)`

' '

Check Show answer

5) `'{x:4.1f}'.format(x=5)`

' '

Check Show answer

CHALLENGE ACTIVITY 10.2.2: Format temperature output.

Print `air_temperature` with 1 decimal point followed by C.

Sample output with input: 36.4158102

```
36.4C
```


334598.939404.qx3zqy7

```
1 air_temperature = float(input())
2
3 ''' Your solution goes here '''
4
```

Run

10.3 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 only replaces the first count occurrences of old.

PARTICIPATION ACTIVITY 10.3.1: replace() string method.

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, else 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 10.3.1: 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 10.3.1: String searching example: Hangman.

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

Load default ter


```
word = 'onomatopoeia'
num_guesses = 10

hidden_word = '-' * len(word)

guess = 1

while guess <= num_guesses and '-' in hidden_word:
    print(hidden_word)
    user_input = input('Enter a character (guess #{}): '.format(guess))

    if len(user_input) == 1:
        # Count the number of times the character occurs in the word
        num_occurrences = word.count(user_input)

        # Replace the appropriate position(s) in hidden_word with the a
        position = -1
```

```
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 10.3.1: 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 10.3.2: String comparison.

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 10.3.2: Identity vs. equality operators.

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

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()*** -- Return True if all cased characters are uppercase letters.
- ***isspace()*** -- Return True if all characters are whitespace.
- ***startswith(x)*** -- Return True if the string starts with x.
- ***endswith(x)*** -- Return 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 10.3.3: 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 10.3.2: 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()` `upper()` standardize user-input for easy comparison.

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

Load default ter

```
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 = {}

```

```
add
Dusty Baker
PHX
```

Run

CHALLENGE ACTIVITY 10.3.1: Find abbreviation.

Complete the if-else statement to print 'LOL means laughing out loud' if user_tweet contains 'LOL'.

Sample output with input: 'I was LOL during the whole movie!'

```
LOL means laughing out loud.
```

334598.939404.qx3zqy7

```
user_tweet = input()

''' Your solution goes here '''

    print('LOL means laughing out loud.')
else:
    print('No abbreviation.')
```


Run

CHALLENGE ACTIVITY 10.3.2: Replace abbreviation.

Assign decoded_tweet with user_tweet, replacing any occurrence of 'TTYL' with 'talk to you later'.

Sample output with input: 'Gotta go. I will TTYL.'

```
Gotta go. I will talk to you later.
```

334598.939404.qx3zqy7

```
user_tweet = input()

decoded_tweet = '' Your solution goes here '''
print(decoded_tweet)
```

Run

10.4 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 10.4.1: Splitting a string into tokens.

Animation content:

`undefined`

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 10.4.1: 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 10.4.1: More string splitting.

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

Load default template...

Run


```
1 file = 'C:/Users/Charles Xavier//Documer
2
3 separator = '/'
4 results = file.split(separator)
5 print('Separator ({}):'.format(separato
6
```

PARTICIPATION ACTIVITY 10.4.2: 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']`
- `['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']`

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

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 10.4.3: String join() method.

Animation content:

undefined

Animation captions:

1. web_path is a list of strings that form the path of the webpage.
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 10.4.2: 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 10.4.4: 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 10.4.3: 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() (explained further elsewhere) to correct the URL by adding 'wiki' before index 3.

Figure 10.4.4: 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('{} is not a valid address.'.format(url))
    print('Redirecting to {}'.format(new_url))
else:
    print('Loading {}'.format(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 10.4.5: 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 10.4.1: String split and join.

334598.939404.qx3zqy7

Start

Type the program's output

```
item_info = 'Bat 10 19'

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

print(item, 'stock:', quantity)
print('Price:', price)
```


1 2

Check Next

CHALLENGE ACTIVITY 10.4.2: Extract area code.

Assign number_segments with phone_number split by the hyphens.

Sample output with input: '977-555-3221'

```
Area code: 977
```

334598.939404.qx3zqy7

```
1 phone_number = input()
2 number_segments = ''' Your solution goes here '''
3 area_code = number_segments[0]
4 print('Area code:', area_code)
```


Run

View your last submission

10.5 LAB: Checker for integer string

Forms often allow a user to enter an integer. Write a program that takes in a string representing an integer as input, and outputs yes if every character is a digit 0-9.

Ex: If the input is:

```
1995
```

the output is:

```
yes
```

Ex: If the input is:

```
42,000
```

or any string with a non-integer character, the output is:

```
no
```

334598.939404.qx3zqy7

LAB ACTIVITY | 10.5.1: LAB: Checker for integer string 10 / 10

main.py Load default template...

```
1 user_string = input()
2
3 if user_string.isdigit() == True:
4     print('yes')
5 else:
6     print('no')
```

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

Program output displayed here

Coding trail of your work What is this?

```
9/5 M10 min:3
```

10.6 LAB: Name format

Many documents use a specific format for a person's name. Write a program whose input is:

firstName middleName lastName

and whose output is:

lastName, firstInitial.middleInitial.

Ex: If the input is:

```
Pat Silly Doe
```


the output is:

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

If the input has the form:

firstName lastName

the output is:

lastName, firstInitial.

Ex: If the input is:

```
Julia Clark
```

the output is:


```
Clark, J.
```

334598.939404.qx3zqy7

LAB ACTIVITY | 10.6.1: LAB: Name format | 10 / 10

main.py — Load default template...

```
1 fullname = input().strip().split(' ')
2
3 lastname = fullname[-1]
4 fullname.pop(-1)
5 print('{}, '.format(lastname.title()), end="")
6
7 for names in fullname:
8     print('{}.'.format(names[0].upper()), end="")
9 print()
```

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) → 0

Program output displayed here

Coding trail of your work What is this?

```
9/5 M---10 min:7
```


10.7 LAB: Count characters

Write a program whose input is a string which contains a character and a phrase, and whose output indicates the number of times the character appears in the phrase.

Ex: If the input is:

```
n Monday
```

the output is:

```
1
```

Ex: If the input is:

```
z Today is Monday
```

the output is:

```
0
```

Ex: If the input is:

```
n It's a sunny day
```

the output is:


```
2
```

Case matters.

Ex: If the input is:

```
n Nobody
```

the output is:

```
0
```

n is different than N.

334598.939404.qx3zqy7

LAB ACTIVITY | 10.7.1: LAB: Count characters | 10 / 10

main.py

```
1 Loading latest submission...
```

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) → 0

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

10.8 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 the doctor away.
Eating 2 shoes a day keeps the doctor away.
```

334598.939404.qx3zqy7

LAB ACTIVITY | 10.8.1: LAB: Mad Lib - loops 10 / 10

main.py Load default template...

```
1 libput = input().strip().split(' ')
2 while libput[0].lower() != 'quit':
3     if len(libput) < 2:
4         print('Not enough arguments.')
5         exit()
6     else:
7         print('Eating {} {} a day keeps the doctor away.'.format(libput[1], libput[
8         libput = input().strip().split(' ')
9
```

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) → 0

Program output displayed here

Coding trail of your work What is this?

```
9/5 M10 min:5
```

10.9 LAB: Remove all non-alpha characters

Write a program that removes all non-alpha characters from the given input.

Ex: If the input is:

```
-Hello, 1 world$!
```

the output is:


```
Helloworld
```

334598.939404.qx3zqy7

LAB ACTIVITY | 10.9.1: LAB: Remove all non-alpha characters 0 / 10

main.py

```
1 Loading latest submission...
```


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) → 0

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

10.10 Strings practice

Learning resources

Following is a list of recommended resources for use when completing these exercises.

- Strings in the Python Tutorial
- Standard Types: Strings in the Python Standard Library
- Common String Operations in the Python Standard Library
- Python Strings from W3Schools.com
- Strings and Character Data in Python from Real Python

Tasks

WGU is providing additional practice exercises for all students. The best method of learning Python is to practice. This will also help you prepare for the assessment, which requires you to write code.

You may use a web-based version of Python such as Repl.It to complete the exercises. Just make sure you are using version 3 or greater. If you use a web-based Python environment, you can easily share code with course instructors if you need help. You may also use a local installation of Python.

Below is a sample exercise. You should copy the entire code snippet into your Python editor. Your code should be placed in the area that says "student code goes here" highlighted in yellow. This is inside the function. When you run this entire code snippet, there are test cases below your code. Your output should match the expected output.

```
# Complete the function to print the first X number of characters in the given string
def printFirst(mystring, x):
    # Student code goes here

# expected output: WGU
printFirst('WGU College of IT', 3)

# expected output: WGU College
printFirst('WGU College of IT', 11)
```

Task 1

Complete the function to print the first X number of characters in the given string


```
# Complete the function to print the first X number of characters in the given string
def printFirst(mystring, x):
# Student code goes here

# expected output: WGU
printFirst('WGU College of IT', 3)

# expected output: WGU College
printFirst('WGU College of IT', 11)
```

Task 2

Complete the function to return the last X number of characters in the given string

```
# Complete the function to return the last X number of characters
# in the given string
def getLast(mystring, x):
# Student code goes here

# expected output: IT
print(getLast('WGU College of IT', 2))

# expected output: College of IT
print(getLast('WGU College of IT', 13))
```


Task 3

Complete the function to return True if the word WGU exists in the given string otherwise return False

```
# Complete the function to return True if the word WGU exists in the given string
# otherwise return False
def containsWGU(mystring):
# Student code goes here

# expected output: True
print(containsWGU('WGU College of IT'))

# expected output: False
print(containsWGU('Night Owls Rock'))
```

Task 4

Complete the function to print all of the words in the given string

```
# Complete the function to print all of the words in the given string
def printWords(mystring):
# Student code goes here

# expected output: ['WGU', 'College', 'of', 'IT']
printWords('WGU College of IT')

# expected output: ['Night', 'Owls', 'Rock']
printWords('Night Owls Rock')
```

Task 5

Complete the function to combine the words into a sentence and return a string

```
# Complete the function to combine the words into a sentence and return a string
def combineWords(words):
# Student code goes here

# expected output: WGU College of IT
print(combineWords(['WGU', 'College', 'of', 'IT']))

# expected output: Night Owls Rock
print(combineWords(['Night', 'Owls', 'Rock']))
```


Task 6

Complete the function to replace the word WGU with Western Governors University and print the new string

```
# Complete the function to replace the word WGU with Western Governors University
# and print the new string
def replaceWGU(mystring):
# Student code goes here

# expected output: Western Governors University Rocks
replaceWGU('WGU Rocks')

# expected output: Hello, Western Governors University
replaceWGU('Hello, WGU')
```

Task 7

Complete the function to remove the word WGU from the given string ONLY if it's not the first word and return the new string

```
# Complete the function to remove the word WGU from the given string
# ONLY if it's not the first word and return the new string
def removeWGU(mystring):
# Student code goes here

# expected output: WGU Rocks
print(removeWGU('WGU Rocks'))

# expected output: Hello, John
print(removeWGU('Hello, WGUJohn'))
```

Task 8

Complete the function to remove trailing spaces from the first string and leading spaces from the second string and return the combined strings


```
# Complete the function to remove trailing spaces from the first string
# and leading spaces from the second string and return the combined strings
def removeSpaces(string1, string2):
# Student code goes here

# expected output: WGU Rocks-You know it!
print(removeSpaces('WGU Rocks    ', '  -You know it!'))

# expected output: Welcome WGU-IT Students
print(removeSpaces('Welcome WGU ', ' -IT Students'))
```

Task 9

Complete the function to print the specified hourly rate with two decimals

```
# Complete the function to print the specified hourly rate with two decimals
def displayHourlyRate(rate):
# Student code goes here

# expected output: $34.79
displayHourlyRate(34.789123)

# expected output: $24.12
displayHourlyRate(24.123456)
```


Task 10

Complete the function to return the number of uppercase letters in the given string

```
# Complete the function to return the number of uppercase letters in the given string
def countUpper(mystring):
# Student code goes here

# expected output: 4
print(countUpper('Welcome to WGU'))

# expected output: 2
print(countUpper('Hello, Mary'))
```