7.1 String slicing

String slicing basics

Strings are a sequence type, having characters ordered by index from left to right. An individual character is read using brackets. Ex: `my_str[5]` reads the character at index 5 of the string `my_str`.

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 mirrors 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:

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 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('The animal is a %s' % 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
```

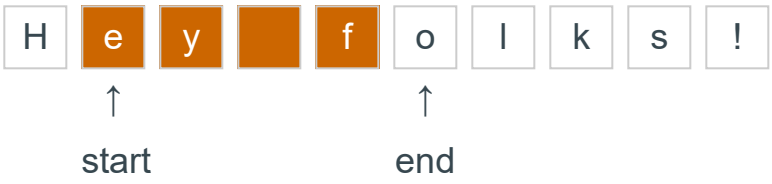
A programmer often wants to read all characters that occur before or after some index in the string. Omitting a start index, such as in `my_str[:end]` yields the characters from indices 0 to end-1. Ex: `my_str[:5]` reads indices 0-4. Similarly, omitting the end index yields the characters from the start 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 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 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 "%s"' %
8 print('The second half of the string is "%s"'
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	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 7.1.4: Slicing.

1) What is the value of `my_str` after the following statements are evaluated:

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

Check Show answer

2) What is the value of `my_str` after the following statements are evaluated:

```
my_str = 'http://reddit.com/r/python'
protocol = 'http://'
my_str = 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('All numbers: %s' % numbers[::])
print('Every other number: %s' % numbers[::2])
print('Every third number between 1 and 8: %s' %
numbers[1:9:3])
```

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

PARTICIPATION ACTIVITY 7.1.5: Slice stride.

Assume the variable `my_str` is 'Agt2t3afc2kjMhagrds!'.

1) What is the result of the expression `my_str[0:5:1]`?

Check Show answer

2) What is the result of the expression `my_str[::2]`

Check Show answer

CHALLENGE ACTIVITY 7.1.1: 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
```

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

Run

7.2 Advanced string formatting

Conversion specifiers

A program commonly needs to display nicely formatted output beyond the ability of basic print usage (i.e., `print(x)`). Consider a program that requires the following output:

```
Student ID: 00422332
Grade percentage: 94.20%
```

Notice that the student ID has two leading 0s, and the grade percentage uses exactly four significant digits. Such output is not possible using simple print statements without an overly complicated scheme, such as calculating the number of digits of the student's id number and manually prepending the appropriate number of 0s.

A ***conversion specifier*** is a placeholder for some value in a string literal. Ex: The expression `'age is %d' % user_age` uses the `%d` conversion specifier as a placeholder for the value of `user_age`. If the value of `user_age` was 22, the result of the expression is `age is 22`. The % character outside of the string literal is an operator, similar to + or /, known as the ***conversion operator***. The conversion operator converts the value from the right side into a string and replaces the conversion specifier on the left with that value. The result of the expression is a new string with the replacements.

Note that if only a single conversion specifier is used, a lone value can be supplied. If multiple conversion specifiers are present in the string, then the values should be placed in a tuple, as in `'age of the %s is %d' % ('cat', user_age)`.

A conversion specifier includes a character (like the d in %d above), known as the ***conversion type***, to indicate how to convert the value into the string. The d means integer (d stands for decimal integer), whereas an f stands for float. Ex: `'%f' % 5.2` yields the string '5.200000', while `'%d' % 5.2` yields the string '5' because the float is truncated when converted to an integer. Common conversion specifiers are integer (%d), floating-point (%f), and string (%s).

Table 7.2.1: Common conversion specifiers.

Conversion	Description
%s	Convert to string using `str()`
%d	Convert to integer
%f	Convert to floating point
%e	Floating point exponential format (Ex: 1.7e3)
%%	Convert to an actual percentage sign (%)

PARTICIPATION ACTIVITY 7.2.1: String formatting using conversion specifiers.

Animation captions:

1. Conversion specifiers convert the object into the desired type.
2. Because 35.6 is assigned to %d, a truncated integer value prints.

Text alignment and float precision

A conversion specifier may include optional components that provide advanced formatting capabilities. One such component is a ***minimum field width***, placed immediately before the conversion type, that describes the minimum number of characters to be inserted in the string. If a string value assigned to a conversion specifier is smaller in size than the specified minimum field width, then the left side of the string is padded with space characters.The statement `print('Student name (%5s)' % 'Bob')` produces the output:

```
Student name (  Bob)
```

The output contains an additional two spaces prior to 'Bob' because the minimum field width is five and Bob is only three characters (5 - 3 is 2).

An additional component known as ***conversion flags*** alter the output of conversion specifiers. If the '0' conversion flag is included, numeric conversion types (%d, %f) add the leading 0s prescribed by the minimum field width in place of spaces. Other conversion flags exist, such as '-', which left-justifies the formatted string, adding padding characters to the right instead of the left. Multiple conversion flags may be included in the same conversion specifier. The conversion flags are always placed before the minimum field width:

Figure 7.2.1: String formatting example: Add leading 0s by setting the minimum field width and 0 conversion flag.

```
student_id = int(input('Enter student ID: '))

print('The user entered %d' % student_id)
print('Full 8-character student ID: %08d' %
student_id)
```

```
Enter student ID: 1234
The user entered 1234
Full 8-character student ID:
00001234
```


A programmer commonly wants to set how many digits to the right of a float-point decimal number to print. The optional ***precision*** component of a conversion specifier indicates how many digits to the right of the decimal should be included. The precision must follow the minimum field width component in a conversion specifier, and starts with a period character: e.g., `'%.1f' % 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: e.g., `'%.5f' % 1.5` results in the string '1.50000'.

Figure 7.2.2: 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  # Approximate pi to 2 decimal places

print('pi is %f.' % real_pi)
print('22/7 is %f.' % approximate_pi)
print('22/7 is accurate for 2 decimal places: %.2f' % approximate_pi)
```

```
pi is 3.141593.
22/7 is 3.142857.
22/7 is accurate for 2 decimal places: 3.14
```

zyDE 7.2.1: Setting minimum field width of conversion specifiers.

Complete the program to print out nicely formatted football player statistics. Matc
following output as closely as possible -- the ordering of players is not important f
example.

```
2012 quarterback statistics:
  Passes completed:
    Greg McElroy   :  19
    Aaron Rodgers  : 371
    Peyton Manning : 400
    Matt Leinart   :  16
  Passing yards:
    ...
  Touchdowns / Interception ratio:
    Greg McElroy   : 1.00
    Aaron Rodgers  : 4.88
    Peyton Manning : 3.36
    Matt Leinart   : 0.00
```

Load default template... **Run**


```
1  quarterback_stats = {
2      'Aaron Rodgers': {'COMP': 371, 'YDS': 4
3      'Peyton Manning': {'COMP': 400, 'YDS': 
4      'Greg McElroy': {'COMP': 19, 'YDS': 214
5      'Matt Leinart': {'COMP': 16, 'YDS': 115
6  }
7
8  print('2012 quarterback statistics:')
9
10 print('  Passes completed:')
11 for qb in quarterback_stats:
12     comp = quarterback_stats[qb]['COMP']
13     #print('    %?: %?' % (qb, comp))  # Re
```

```
    # Hint: Use the conversion flag '-' to
print('  Passing yards:')
for qb in quarterback_stats:
    print('    QB: yards')
```

PARTICIPATION ACTIVITY 7.2.2: Conversion specifiers.

What is the output of each print?

1) `print('%05d' % 150)`

Check Show answer

2) `print('%05d' % 75.55)`

Check Show answer

3) `print('%05.1f' % 75.55)`

Check Show answer

4) `print('%4s %08d' % ('ID:', 860552))`

Check Show answer

PARTICIPATION ACTIVITY 7.2.3: Advanced output.

Match the given terms with the definitions.

Conversion specifier | Minimum field width | Conversion type

'0' conversion flag | Precision

0 conversion flag | Precision

	A placeholder for a value in a string.
	Determines how to display a value assigned to a conversion specifier
	Optional component that determines the number of characters a conversion specifier must insert
	Optional component that indicates numeric conversion specifiers should add leading 0s if the minimum field width is also specified
	Optional component that determines the number of digits to include to the right of a floating point value

Reset

CHALLENGE ACTIVITY 7.2.1: Format temperature output.

Print air_temperature with 1 decimal point followed by C.

Sample output with input: 36.4158102

```
36.4C
```



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

Run

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 only replaces the first count occurrences of old.

PARTICIPATION ACTIVITY 7.3.1: 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 position 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 first position is number 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 position start:
  - `my_str.find('oo', 2)  # Returns 5`
- ***find(x, start, end)*** -- Same as find(x, start), but stops the search at position end:
  - `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, than 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 num guesses to fill out the complete word.

Load default

```
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 #%d): ' % 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 actual cha
        position = -1
        for occurrence in range(num_occurrences):
            position = word.find(user_input, position+1) # Find the position of
            hidden_word = hidden_word[:position] + user_input + hidden_word[posi
```



```
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

'Yankee Sierra' > 'Amy Wise'	True	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 7.3.2: 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 7.3.2: 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 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 7.3.3: 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 all 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 sto passenger names and travel destinations in a database. The use of `strip()`, `lowe` `upper()` standardize user-input for easy comparison.

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

Load default

```
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')
user_input = input(menu_prompt).strip().lower()

while user_input != 'exit':
```

```
add
Dusty Baker
PHX
```

Run

CHALLENGE ACTIVITY 7.3.1: 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.
```



```
user_tweet = input()

''' Your solution goes here '''

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

Run

CHALLENGE ACTIVITY 7.3.2: 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.
```



```
user_tweet = input()

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

Run

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:

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 3 tokens are assigned to the variable my_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 t
method separator to "//" and " " and observe the output.

Load default template...

Run


```
1 file = 'C:/Users/Charles Xavier//Documents//re
2
3 separator = '/'
4 results = file.split(separator)
5 print('Separator (%s):' % separator, results)
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 the string 'billgates@microsoft' to my_str. 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.

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 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']

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 if the third token 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 position 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('{} 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 7.4.5: Splitting and joining strings.

1) What line of code is needed to change separators in a string from hyphens (-) to colons (:) after the following code block?

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


Check Show answer

CHALLENGE ACTIVITY 7.4.1: Extract area code.

ACTIVITY

Assign number_segments with phone_number split by the hyphens.

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

```
Area code: 977
```



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

Run

7.5 The string format() method

The format() method

The string ***format()*** method formats text, similar to the string-formatting % operator. The format() method was initially introduced in Python 2.6 and was meant to eventually replace the % syntax completely. However, the % syntax is so pervasive in Python programs and libraries that the language still supports both techniques.[1]

The format() method and % string-formatting syntax behave very similarly, each using value placeholders within a string literal. Each pair of braces {} is a placeholder known as a

replacement field. A value from the format() arguments is inserted into the string depending on the contents of a replacement field. Three methods exist to fill in a replacement field:

1. *Positional*: An integer that corresponds to the value's position.

```
'The {1} in the {0}'.format('hat', 'cat')
```
→ `The cat in the hat`

2. *Inferred positional*: Empty {} assumes ordering of replacement fields is {0}, {1}, {2}, etc.

```
'The {} in the {}'.format('cat', 'hat')
```
→ `The cat in the hat`

3. *Named*: A name matching a keyword argument.

```
'The {animal} in the {headwear}'.format(animal='cat',
headwear='hat')
```
→ `The cat in the hat`

The first two methods are based on the ordering of the values within the format() argument list. Placing a number inside of a replacement field automatically treats the number as the position of the desired value. Empty braces {} indicate that all replacement fields are positional, and values are assigned in order from left-to-right as {0}, {1}, {2}, etc. The positional and inferred positional methods cannot be combined. Ex: `'{} + {1} is {2}'.format(2, 2, 4)` is not allowed.

The third method allows a programmer to create a ***keyword argument*** in which a name is assigned to a value from the format() keyword argument list. Ex: `animal='cat'` is a keyword argument that assigns 'cat' to {animal}. Good practice is to use the named method when formatting strings with many replacement fields to make the code more readable.

A programmer can include a brace character { or } in the resulting string by using a double brace {{ or }}. Ex: `'{0} {{x}}'.format('val')` produces the string 'val {x}'.

PARTICIPATION ACTIVITY 7.5.1: string.format() usage.

Determine the output of the following code snippets. If an error would occur, type "error".

1)
```
print('{hour}:
{minute}'.format(hour=9, 43=minute))
```

Check Show answer

2)
```
print('Hi {{{0}}}!'.format('Bilbo'))
```

Check Show answer

3)

```
month = 'April'
day = 22
print('Today is {month}
{0}'.format(day, month=month))
```

Check Show answer

(*1) This material primarily uses the more popular % conversion operator syntax. The discussion of the format() string method here is to give the reader a brief introduction, should the reader come across such code in the wild.

©zyBooks 11/11/19 20:23 569464
Diep Vu
UNIONCSC103OrhanFall2019