8.1 String slicing

Strings are a sequence type, having characters ordered by position from left to right. An individual character is read using brackets. For example, my_str[5] reads the character in position 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 positions 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 8.1.1: String slicing.

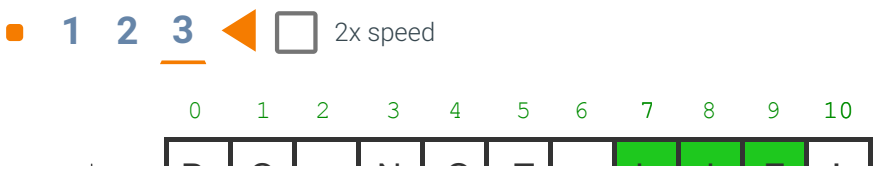
```
url = 'http://en.wikipedia.org/wiki/Turing'
domain = url[7:23]  # Read 'en.wikipedia.org' from url
print(domain)
```

```
en.wikipedia.org
```

Feedback?

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 position 0 (J), 1 (o), 2 (h), and 3 (n), but *not* 4, thus yielding 'John'. The space character in position 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 position greater than the length of the string can be used, e.g., my_str[5:8] or my_str[5:10] both yield the string 'Doe'.

PARTICIPATION ACTIVITY 8.1.1: Slicing.




```
my_str[0:2]  :  'DO'
my_str[0:6]  :  'DO NOT'
my_str[7:10] :  'LIE'
```

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

Feedback?

Negative indices can be used to specify start or end positions relative to the end of the string. For example, if the variable my_str is 'Jane Doe!?', then my_str[0:-2] yields 'Jane Doe' because the end position -2 refers to the character '!', which is not included in the slice.

PARTICIPATION ACTIVITY 8.1.2: Slicing basics.

Determine the output of the following code:

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

The

Check Show answer

Correct

The

The slice contains characters in positions 0, 1, and 2: 'The'.

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

cat

Check Show answer

Correct

cat

The slice includes the character in the start position, 3, as well as 4, 5, and 6. Those four characters are space, c, a, and t.

Feedback?

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 8.1.2: 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
```

Feedback?

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

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

PARTICIPATION ACTIVITY 8.1.3: String slicing tool.

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

Output: 'Hey folks!'

H	e	y		f	o	l	k	s	!
↑									↑

start end

Feedback?

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

zyDE 8.1.1: Slicing example: omitting start, end positions.

Run the program below.

Load default template...

```
1 usr_text = input('Enter a string:\n')
2
3 first_half = usr_text[:len(usr_text)//2]
4 last_half = usr_text[len(usr_text)//2:]
5
6 print('The first half of the string is "%s
7 print('The second half of the string is "%
8
```

Hello there. Nice to meet you

Run

```
Enter a string:
The first half of th
"Hello there. Ni"
The second half of t
"ce to meet you!"
```

Feedback?

A string slice operation creates a new string; an identical copy of a string can thus be created via `str1 = str2[:]`. In contrast, the assignment `str1 = str2` binds both str1 and str2 to the same string object.

Specifying a start position beyond the end of the string, or beyond the end position (like 3:2),

yields an empty string, e.g., my_str[2:1] is ''. Specifying an end position 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 8.1.1: 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 positions 10-18.
my_str[10:-5]	wikipedia.org/wiki/	Gets the characters in positions 10-28.
my_str[8:]	n.wikipedia.org/wiki/Nasa/	All characters from position 8 until the end of the string.
my_str[:23]	http://en.wikipedia.org	Every character up to position 23, but not including my_str[23].
my_str[:-1]	http://en.wikipedia.org/wiki/Nasa	All but the last character.
my_str[:]	http://en.wikipedia.org/wiki/Nasa/	A new copy of the my_str object.

Feedback?

PARTICIPATION ACTIVITY 8.1.4: Slicing.

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

```
my_str =
```

Correct

/r/python

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

/r/python

Check Show answer

The start position is the third / character in my_str. Since the end position is omitted, the slice reads the rest of the characters in the string.

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):]
```

reddit.com/r/python

Check Show answer

Correct

reddit.com/r/python

len(protocol) is 7, thus the slicing operation is my_str[7:], which reads the entire string excluding the first 7 characters.

3) Write a statement that assigns str2 with a copy of str1.

str2 = str1[:]

Check Show answer

Correct

str2 = str1[:]

Evaluating str1[:] creates a copy of str1.

Feedback?

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 8.1.3: 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'
```

```
All numbers: 0123456789
Every other number: 02468
Every third number between
```

```
print('Every third number between 1 and 8: %s
% numbers[1:9:3])
```

```
1 and 8: 147
```

Feedback?

PARTICIPATION ACTIVITY 8.1.5: Slice stride.

Assume the variable my_str is 'Agt2t3afc2kjMhagrds!'.

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

Agt2t

Check Show answer

Correct

Agt2t

Reads the first 5 characters, adding a stride of 1 to the index to find each new element to read

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

AttackMars

Check Show answer

Correct

AttackMars

Reads every other character because the stride is 2.

Feedback?

CHALLENGE ACTIVITY 8.1.1: Slice a rhyme.

Assign sub_lyric with 'cow' by slicing rhyme_lyric from start_index to end_index.
Sample output from given program:

```
COW
```

```
1 start_index = 4
```

```
2 end_index = 7
3 rhyme_lyric = 'The cow jumped over the moon.'
4 sub_lyric = rhyme_lyric[start_index:end_index]
5 print(sub_lyric)
```

Run ✓ All tests passed

✓ Testing start_index 4, end_index 7

Your output `cow`

✓ Testing start_index 8, end_index 14

Your output `jumped`

Feedback?

8.2 Advanced string formatting

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 programmer may include a ***conversion specifier*** as a placeholder for a 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. The % character outside of the string literal is an operator, similar to + or /, known as the ***conversion operator***. The conversion operator inserts values from the tuple on the right into conversion specifiers on the left.

Note that if only a single conversion specifier is used, the lone value can be used in place of a tuple, as in `'age is %d' % user_age`.

Construct 8.2.1: Conversion operators.

```
'string literal with conversion specifiers'  % (values_tuple)
```

Feedback?

A conversion specifier not only is a value placeholder, but also includes a character (like the d in %d above), known as the ***conversion type***, to indicate how to convert the value to a string. The d means integer (d stands for decimal integer), whereas an f stands for float. For example, `'%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).

PARTICIPATION ACTIVITY 8.2.1: String formatting using conversion specifiers.

1 2 2x speed

```
'Tokyo'  35.600000   140.800000
  '%s: %f North, %f East' % ('Tokyo', 35.6, 140.8)
```

'Tokyo: 35.600000 North, 140.800000 East'

```
'Tokyo'   35          140
  '%s: %d North, %d East' % ('Tokyo', 35.6, 140.8)
```

'Tokyo: 35 North, 140 East'

Because 35.6 is assigned to %d, a truncated integer value prints.

Feedback?

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 8.2.1: 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
```

Feedback?

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 8.2.2: 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
```

Feedback?

zyDE 8.2.1: Setting minimum field width of conversion specifiers.

Complete the program to print out nicely formatted football player statistics. M
following output as closely as possible -- the ordering of players is not importa
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

```
13      print('    %-15s: %4s' % (qb, comp)
14      # Hint: Use the conversion flag '-'
15
16  print('  Passing yards:')
17  for qb in quarterback_stats:
18      yards = quarterback_stats[qb]['YDS'
19      #print('    QB: yards')
20      print('    %-15s: %4s' % (qb, yards
21
22  print('  Touchdown / interception ratio
23  for qb in quarterback_stats:
24      td = quarterback_stats[qb]['TD']
25      int = quarterback_stats[qb]['INT']
26      ti_ratio = float(td) / float(int)
27      print('    %-15s: %.2f' % (qb, ti_r
28  # Hint: Convert TD/INTs to float before
29
```

```
    Peyton Manning :
    Aaron Rodgers  :
    Matt Leinart   :
  Passing yards:
    Greg McElroy   :
    Peyton Manning :
    Aaron Rodgers  :
    Matt Leinart   :
  Touchdown / interc
ratio:
    Greg McElroy   :
    Peyton Manning :
    Aaron Rodgers  :
```

Feedback?

PARTICIPATION ACTIVITY 8.2.2: Conversion specifiers.

What is the output of each print?

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

00150

Check Show answer

Correct

00150

Both the minimum field width and 0 conversion flag is set, thus 150 is expanded to the minimum field width of 5 by adding two leading 0s.

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

00075

Check Show answer

Correct

00075

The 'd' conversion type truncates the floating point value to 75. Three leading 0s are added to meet the minimum field width of 5.

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

075.5

Check Show answer

Correct

075.5

The precision is 1, thus the result will contain 75.5. The '.' character counts towards the minimum width requirement, thus one leading 0 must be added.

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

ID: 00860552

Check Show answer

Correct

ID: 00860552

The first conversion specifier pads 'ID:' with a single space character. The second conversion specifier adds two leading zeroes to the given integer value.

Feedback?

Mapping keys

Sometimes a string contains many conversion specifiers. Such strings can be hard to read and understand. Furthermore, the programmer must be careful with the ordering of the tuple values, lest items are mistakenly swapped. A dictionary may be used in place of a tuple to enhance clarity at the expense of brevity. If a dictionary is used, then all conversion specifiers must include a ***mapping key*** component. A mapping key is specified by indicating the key of the relevant value in the dictionary within parentheses.

PARTICIPATION ACTIVITY 8.2.3: Using a dictionary and conversion specifiers with mapping keys.

2x speed

11 02 15.002000

```
'time: %(hour)d:%(min)02d:%(sec)f' % {'hour':11, 'min':2, 'sec':15.002}
```

'time: 11:02:15.002000'

A mapping key is specified by indicating the key of the relevant value in the dict within parentheses.

Feedback?

Figure 8.2.3: Comparing conversion operations using tuples and dicts.

```
import time
gmt = time.gmtime()  # Get current Greenwich Mean Time

print('Time is: %02d/%02d/%04d  %02d:%02d %02d sec' % \
      (gmt.tm_mon, gmt.tm_mday, gmt.tm_year, gmt.tm_hour, gmt.tm_min, gmt.tm_sec))
```

```
Time is: 06/07/2013  20:16 24 sec
...
Time is: 06/07/2013  20:16 28 sec
```

```
import time
gmt = time.gmtime()  # Get current Greenwich Mean Time

print('Time is: %(month)02d/%(day)02d/%(year)04d  %(hour)02d:%(min)02d %(sec)02d
sec' %  \
      {
        'year': gmt.tm_year, 'month': gmt.tm_mon, 'day': gmt.tm_mday,
        'hour': gmt.tm_hour, 'min': gmt.tm_min, 'sec': gmt.tm_sec
      }
)
```

```
Time is: 06/07/2013  20:16 24 sec
...
Time is: 06/07/2013  20:16 28 sec
```

Feedback?

PARTICIPATION ACTIVITY 8.2.4: Advanced output.

Match the given terms with the definitions.

Conversion specifier	A placeholder for a value in a string.	Correct
Conversion type	Determines how to display a value assigned to a conversion specifier	Correct
Minimum field width	Optional component that determines the number of characters a conversion specifier must insert	Correct

'0' conversion flag	Optional component that indicates numeric conversion specifiers should add leading 0s if the minimum field width is also specified	Correct
Precision	Optional component that determines the number of digits to include to the right of a floating point value	Correct
Mapping key	Optional component that describes the key of the conversion specifiers value in a dict	Correct

Reset

Feedback?

CHALLENGE ACTIVITY 8.2.1: Format temperature output.

Print air_temperature with 1 decimal point followed by C. Sample output from given program:

```
36.4C
```

```
1 air_temperature = 36.4158102
2
3 print('%.1f%s' % (air_temperature,'C'))
4
```

Run ✓ All tests passed

✓ Testing air_temperature 36.4158102

Your output `36.4C`

✓ Testing air_temperature 24.590815

Your output `24.6C`

Feedback?

8.3 String methods

String objects come with several useful features. The features 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 features for an object type, including several useful methods.

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.

zyDE 8.3.1: String methods example: replace().

The following example uses the above function, asking the user to spell out numbers, and translating those numbers to another language.

Try the program below; enter the text 'one plus two is three':

Load default template...

```
user_input = input("Enter sentence:\n")

translation = user_input[:]  # Make a copy of the str

# Replace English with Spanish.
translation = translation.replace('one', 'uno')
translation  = translation.replace('two', 'dos')
translation = translation.replace('three', 'tres')
translation = translation.replace('four', 'quatro')
translation = translation.replace('five', 'cinco')
```

one day, I'll have three horse

Run

```
translation = translation.replace('six', 'seis')
translation  = translation.replace('seven', 'siete')
translation  = translation.replace('eight', 'ocho')
translation = translation.replace('nine', 'nueve')

print('Translation:', translation)
```

Feedback?

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 8.3.1: Use 'in' to check if a character or substring is contained by another string

contained by another string.

```
if 'b' in my_str:
    # Statements to execute if my_str contains a 'b' character.
```

Feedback?

zyDE 8.3.2: 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...

```
14          num_occurrences = word.count(user_input)
15
16          # Replace the appropriate position(s) in hidden_word with the actual chara
17          position = -1
18          for occurrence in range(num_occurrences):
19              position = word.find(user_input, position+1)  # Find the position of t
20              hidden_word = hidden_word[:position] + user_input + hidden_word[positi
21
22          guess += 1
23
24  if not '-' in hidden_word:
25      print('Winner!', end=' ')
26  else:
27      print('Loser!', end=' ')
28
29  print('The word was %s.' % word)
30
```

```
a
s
|
```

Run

```
------------
Enter a character (guess #1): ------------
Enter a character (guess #2): ---m--------
Enter a character (guess #3): -n-m--------
Enter a character (guess #4): -n-ma------a
```

```
Enter a character (guess #5): -n-ma--p---a
Enter a character (guess #6): -n-mat-p---a
Enter a character (guess #7): -n-mat-p-e-a
Enter a character (guess #8): -n-mat-p-e-a
Enter a character (guess #9): -n-mat-p-e-a
Enter a character (guess #10): Loser! The word was onomatopoeia.
```

Feedback?

Comparing strings

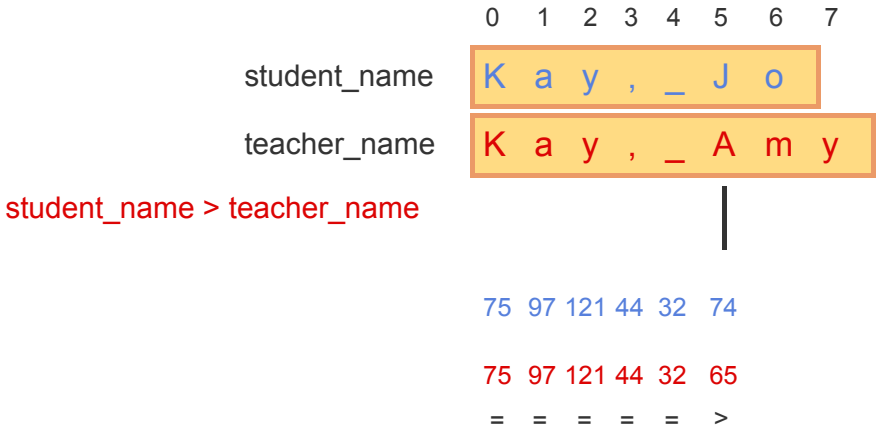
String objects may be compared using relational operators (<, <=, >, >=), equality operators (==, !=), membership operators (in, not in), and identity operators (is, is not). The following provides examples, given my_str is 'Hello', student_name is 'Kay, Jo', and teacher_name is 'Kay, Amy':

- my_str == 'Hello' *# Evaluates to True*
- my_str == 'Hello.' *# Evaluates to False*
- student_name == teacher_name *# Evaluates to False*
- student_name > teacher_name *# Evaluates to True because 'J' > 'A' is True*
- student_name >= teacher_name *# Evaluates to True because ">"*
- student_name >= 'Kay, Jo' *# Evaluates to True because "=="*
- student_name != 'Kay, jo' *# Evaluates to True because 'J' != 'j'*
- 'Jo' in student_name *# Evaluates to True*
- student_name in 'Jo' *# Evaluates to False*
- student_name is 'Kay, Jo' *# Evaluates to False, because 'Kay, Jo' and student name are bound to different objects.*

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, as illustrated in the following animation.

PARTICIPATION ACTIVITY 8.3.1: String comparison.

1 2 3 2x speed



'J' is greater than 'A', so student_name is greater than teacher_name.

Feedback?

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. For example, a programmer may write `student_name is 'Kay, Jo'`, intending to compare the value of student_name to 'Kay, Jo'. Instead, the interpreter creates a new string object from the string literal and compares the identity (typically the memory location) of the new string to the object bound to student_name, returning False.

Figure 8.3.2: Identity vs. equality operators.

'

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

```
if student_name is 'Kay, Jo':
    print('Identity operator: True')
else:
    print('Identity operator: False')

if student_name == 'Kay, Jo':
    print('Equality operator: True')
else:
    print('Equality operator: False')
```

```
Enter student name: Kay, Jo
Identity operator: False
Equality operator: True
```

Feedback?

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. For example, the string 'abc?'.islower() returns True, ignoring the question mark.

PARTICIPATION ACTIVITY 8.3.2: String methods: Boolean string comparisons.

Determine whether the given expression evaluates to True or False.

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

   ○ True

Correct

The forward slash and period characters are

- False

not alpha numeric.

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

- True
- False

Correct

The string starts with 'HTTP'.

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

- True
- False

Correct

newlines and spaces count as whitespace.

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

- True
- False

Correct

A space character is not a digit.

5) 'LINCOLN, ABRAHAM'.isupper()

- True
- False

Correct

All of the cased characters are uppercase. Commas, spaces, and periods are not cased.

Feedback?

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 8.3.3: 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 some passengers into the database. Add a duplicate passenger name, using different capitalization, and print the list again.

Load default template...

```
            print('Unknown destination.\n')
        else:
            passengers[name] = destination

    elif user_input == 'del':
        name = input('Enter passenger name:\n').str
        if name in passengers:
            del passengers[name]

    elif user_input == 'print':
        for passenger in passengers:
            print('%s --> %s' % (passenger.title(),
    else:
        print('Unrecognized command.')

    user_input = input('Enter command:\n').strip().
```

```
add
DISTY Baker
PHX
```

Run

```
   (print) Print pass
   (exit) Exit the pr
Enter command:
Enter passenger name
Available destinatic
(PHX) Phoenix
(AUS) Austin
(LAS) Las Vegas
Enter destination:
Enter command:
Disty Baker --> PHX
Enter command:
```

Feedback?

CHALLENGE ACTIVITY 8.3.1: Find abbreviation.

Complete the if-else statement to print 'LOL means laughing out loud' if user_tweet contains 'LOL'. Sample output from given program:

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

```
1 user_tweet = 'I was LOL during the whole movie!'
2
3 if 'LOL' in user_tweet:
4
5     print('LOL means laughing out loud.')
6 else:
7     print('No abbreviation.')
```

Run ✓ All tests passed

✓ Testing user_tweet 'I was LOL during the whole movie!'

Your output `LOL means laughing out loud.`

✓ Testing user_tweet 'Be serious.'

Your output `No abbreviation.`

✓ Testing user_tweet 'I am LOL...'

Your output `LOL means laughing out loud.`

Feedback?

CHALLENGE ACTIVITY 8.3.2: Replace abbreviation.

Assign decoded_tweet with user_tweet, replacing any occurrence of 'TTYL' with 'talk to you later'. Sample output from given program:

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

```
1 user_tweet = 'Gotta go. I will TTYL.'
2
3 decoded_tweet = user_tweet.replace('TTYL', 'talk to you later')
4 print(decoded_tweet)
```

Run ✓ All tests passed

✓ Testing user_tweet 'Gotta go. I will TTYL.'

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

✓ Testing user_tweet 'I will be right back.'

Your output `I will be right back.`

Feedback?

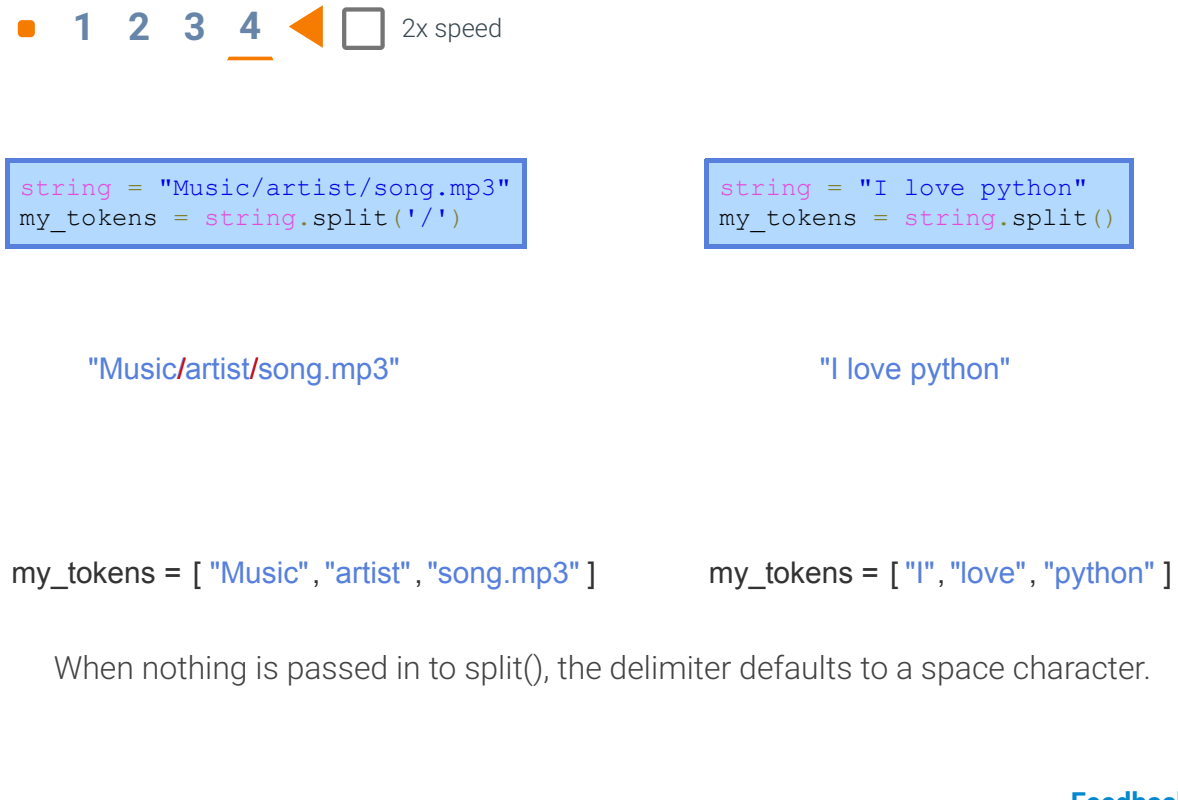
8.4 Splitting and joining strings

A common programming task is to break a large string down into the comprising substrings. The string method ***split()*** can be used to split up a string into a list of tokens. Each ***token*** is a sequence of characters 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.

An example of splitting a string is `'Martin Luther King Jr.'.split()`. The split method is applied to the string literal "Martin Luther King Jr." using any whitespace character as the default separator. The result of the method is the list of tokens ['Martin', 'Luther', 'King', 'Jr.'].

The separator can be changed by specifying a string within the parentheses of the split() method. For example, `'a#b#c'.split('#')` uses the "#" separator to split the string "a#b#c" into three tokens and produce the list ['a', 'b', 'c']. The following example demonstrates splitting strings:

PARTICIPATION ACTIVITY 8.4.1: Splitting a string into tokens.



Feedback?

Figure 8.4.1: 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',
'']
```

Feedback?

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 object 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 there are two consecutive separators, then the resulting list will contain an empty string element for each such occurrence. As an example, 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 8.4.1: 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 Xavier//Documents//report.doc
2
3 separator = '/'
4 results = file.split(separator)
```

```
Separator ( ): ['C:/
'Xavier//Documents//
```

```
5 print('Separator (%s):' % separator, results)
6
```

Feedback?

PARTICIPATION ACTIVITY 8.4.2: String split method.

Assume that the variable song has the value:
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.']

Correct

Omitting the arguments to split() incurs a split on every whitespace, including newlines and spaces.

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

Correct

The separator is the newline character \n. The string

- ['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']

The separator is the newline character '\n'. The string ends with a separator character, thus an empty string element is created in the tokens list.

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']

Correct

The separator is "scream". Whitespace and newlines are treated like any other text.

Feedback?

The ***join()*** method performs the inverse operation of split() by joining a list of strings together to create a single string. The statement `my_str = '@'.join(['billgates', 'microsoft'])` binds the name my_str to a new string object with the value '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 -- the separator is placed between each pair of list elements. The separator can be any string, including multiple characters or an empty string.

The following animation illustrates the use of join():

PARTICIPATION ACTIVITY 8.4.3: String join() method.

1 2 3 2x speed

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

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

Then join() concatenates the list of strings with the separator "/".

Feedback?

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. Thus, a statement such as `''.join(['http://', 'www.', 'ebay', '.com'])` produces the string 'http://www.ebay.com'. An alternative method is to concatenate a string within a loop, however the join() method accomplishes the same task in a single line of code, is easier to read, and is performed faster by the interpreter.

Figure 8.4.2: String join example: Comparing join vs. loops.

The following programs are equivalent, however join() is a simpler approach, using less code and being 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.
```

Feedback?

PARTICIPATION ACTIVITY 8.4.4: 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 =
```

'.'.join(x)

Check Show answer

Correct

'.'.join(x)

join() builds a new string from the list elements, adding the '.' separator between each pair of elements.

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

my_str=''.join(x)

Check Show answer

Correct

my_str = ''.join(x)

The empty string join() method is a useful way to build a new string.

Feedback?

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 following example illustrates splitting and joining methods.

Figure 8.4.3: 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
```

Feedback?

A programmer may also want to add, remove, or replace specific token(s) from the string. The following program 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 8.4.4: 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('%s not a valid address.' % url)
    print('Redirecting to %s' % new_url)
else:
    print('Loading %s' % 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 not a valid address.
Redirecting to http://en.wikipedia.org/wiki/Rome
```

Feedback?

PARTICIPATION

ACTIVITY 8.4.5: Joining strings.

1) Write a statement that joins all the elements of my_list together, without any separator character between elements.

".join(my_list)

Check Show answer

Correct

".join(my_list)

Using the join() method of an empty string object just appends the elements of the list together without any separator character.

Feedback?

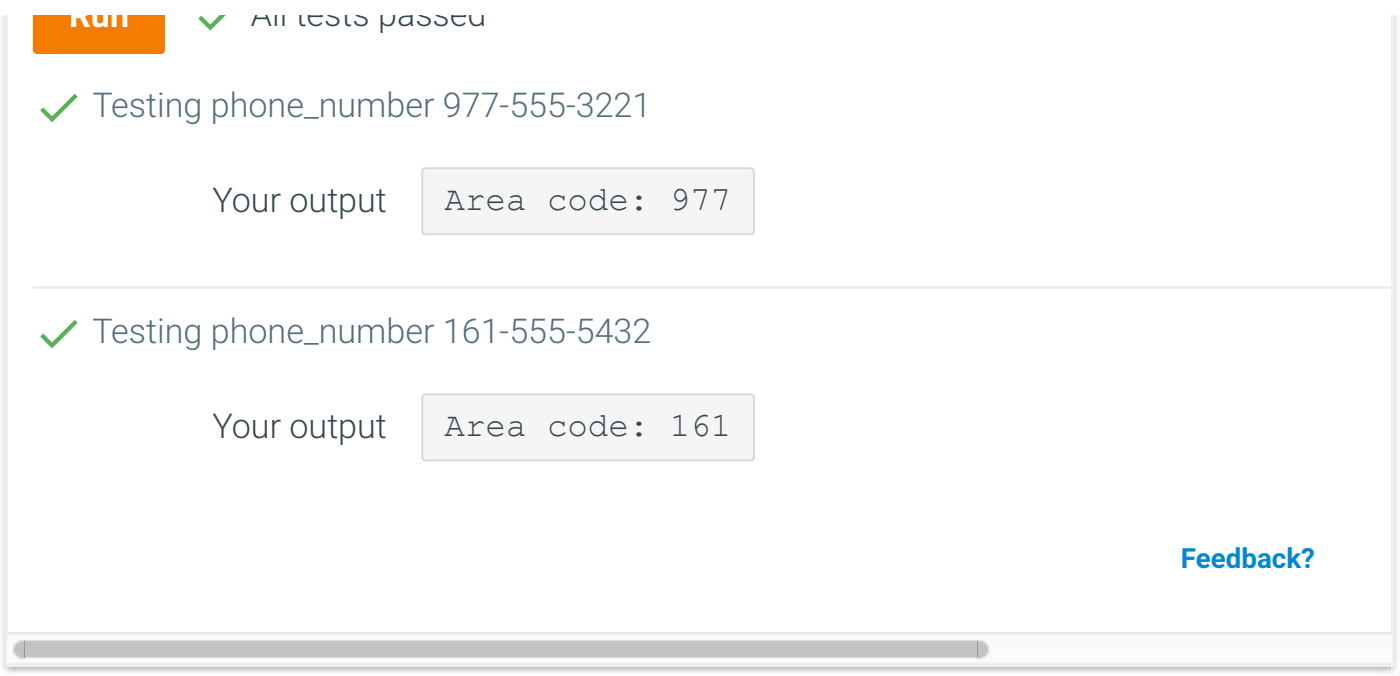
CHALLENGE ACTIVITY 8.4.1: Extract area code.

Assign number_segments with phone_number split by the hyphens. Sample output from given program:

```
Area code: 977
```

```
1 phone_number = '977-555-3221'
2 number_segments = phone_number.split('-')
3 area_code = number_segments[0]
4 print('Area code:', area_code)
```

Run ✓ All tests passed


Run
All tests passed
Testing phone_number 977-555-3221
Your output
Area code: 977
Testing phone_number 161-555-5432
Your output
Area code: 161
Feedback?

8.5 The string format() method

The string ***format()*** method can be used to format text, similar in behavior 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 below figure demonstrates basic usage of the string format() method:

Figure 8.5.1: string.format() example.

```
print('{0} is {1} years old'.format('Johnny', 18))
```

```
Johnny is 18 years old
```

Feedback?

The string 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 within the format() parentheses will be inserted into the string depending on the contents of a replacement field. There are three ways to fill in a replacement field:

1. *Positional*: An integer that describes the position of the value.

```
'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 ways to fill in a replacement field 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. Note that either all replacement fields are inferred using {}, or none of them do -- statements such as `'{} + {1} is {2}'.format(2, 2, 4)` are not allowed.

The third way allows a programmer to create a ***keyword argument*** in which a name is assigned to a value and the name and associated value is specified in the format() argument list. Good practice is to use the naming approach when formatting strings having many replacement fields, to make the code is more readable.

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

PARTICIPATION ACTIVITY 8.5.1: 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))
```

error

Check Show answer

Correct

error

This code would produce an error because 43=minute is backwards; should be minute=43.

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

Hi {Bilbo}!

Check Show answer

Correct

Hi {Bilbo}!

A single brace is printed by using the double brace 2-character sequence. Thus, the outside two braces of {{{0}}} print single brace characters around the replacement field {0}.

3)
```
month = 'April'
day = 22
print('Today is {month}
{0}'.format(day,
month=month))
```

Today is April 22

Check Show answer

Correct

Today is April 22

The keyword argument month is assigned the same value as the variable month, 'April'. Mixing positional and named replacement fields is also OK.

Feedback?

(*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.

8.6 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 PyFiddle, Repl.It, or PythonFiddle 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

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'))
```