CSCI 141

Lecture 19

String Methods

String Comparisons and Ordering
Announcements
Announcements

• A4 due next Wednesday.
Announcements

• A4 due next Wednesday.
• I have office hours 2-3:30 today.
Announcements

• A4 due next Wednesday.

• I have office hours 2-3:30 today.

• QOTD explanations continue to be linked from the last question.
Announcements

• A4 due next Wednesday.

• I have office hours 2-3:30 today.

• QOTD explanations continue to be linked from the last question.

• No class monday! No labs next week!
• The green corner should have 255 green.

• The green corner does not need to have 0 red and 0 blue.

• My color calculations are based on distance from the corner, irrespective of direction.

• Other approaches are ok too!
Goals

• Know how to use a few of the basic methods of string objects:
  • upper, lower, find, replace

• Understand the behavior of the following operators on strings:
  • <, >, ==, !=, in, and not in

• Understand how Python orders strings using lexicographic ordering
\textbf{QOTD}

\[ s = \text{"blockade"} \]

\begin{tabular}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
-8 & -7 & -6 & -5 & -4 & -3 & -2 & -1 \\
\end{tabular}

\begin{verbatim}
print(s[  4  ])
print(s[ 4:6  ])
print(s[ -5:5 ])
print(s[  :4  ])
print(s[ -4:  ])
\end{verbatim}
QOTD

\[ s = "blockade" \]

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>b</td>
<td>l</td>
<td>o</td>
<td>c</td>
<td>k</td>
<td>a</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

\textcolor{green}{\text{print}(s[\ 4\ ])} \quad \textcolor{red}{k}
\textcolor{green}{\text{print}(s[\ 4:6\ ])}
\textcolor{green}{\text{print}(s[\ -5:5\ ])}
\textcolor{green}{\text{print}(s[\ :4\ ])}
\textcolor{green}{\text{print}(s[\ -4:\ ])}
QOTD

\[ s = "blockade" \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>l</td>
<td>o</td>
<td>c</td>
<td>k</td>
<td>a</td>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

-8 -7 -6 -5 -4 -3 -2 -1

\[
\text{print}(s[ 4 ]) \quad \text{ka}
\]
\[
\text{print}(s[ 4:6 ]) \quad \text{ka}
\]
\[
\text{print}(s[ -5:5 ]) \quad \text{ka}
\]
\[
\text{print}(s[ :4 ]) \quad \text{ka}
\]
\[
\text{print}(s[ -4: ]) \quad \text{ka}
\]
QOTD

s = "blockade"

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>l</td>
<td>o</td>
<td>c</td>
<td>k</td>
<td>a</td>
<td>d</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

print(s[  4  ])  k
print(s[  4:6  ])  ka
print(s[ -5:5  ])  ck
print(s[  :4  ])
print(s[ -4:  ])
QOTD

s = "blockade"

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>l</td>
<td>o</td>
<td>c</td>
<td>k</td>
<td>a</td>
<td>d</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

-8 -7 -6 -5 -4 -3 -2 -1

print(s[ 4 ])
print(s[ 4:6 ])
print(s[ -5:5 ])
print(s[  :4 ])
print(s[ -4: ])
QOTD

s = "blockade"

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>b</strong></td>
<td><strong>l</strong></td>
<td><strong>o</strong></td>
<td><strong>c</strong></td>
<td><strong>k</strong></td>
<td><strong>a</strong></td>
<td><strong>d</strong></td>
<td><strong>e</strong></td>
<td></td>
</tr>
<tr>
<td><strong>-8</strong></td>
<td><strong>-7</strong></td>
<td><strong>-6</strong></td>
<td><strong>-5</strong></td>
<td><strong>-4</strong></td>
<td><strong>-3</strong></td>
<td><strong>-2</strong></td>
<td><strong>-1</strong></td>
<td></td>
</tr>
</tbody>
</table>

print(s[  4  ]): k
print(s[  4:6  ]): ka
print(s[ -5:5  ]): ck
print(s[ :4  ]): bloc
print(s[ -4:  ]): kade
def uun_letters(first_name, last_name):
    """ Return the letters in a student's WWU Universal Username given the student's first_name and last_name. The username begins with the first 6 characters of the last name, followed by the first letter of the first name. Return the username in all lower case. Example: uun_letters("Scott", "Wehrwein") => "wehrwes" """

return (last_name[1:6] + first_name[0]).lower()

return (last_name[0:6] + first_name[0]).lower()

return last_name[0:6].lower() + first_name[0:0].lower()

return last_name[0:6].lower() + first_name[0].lower()
def uun_letters(first_name, last_name):
    """ Return the letters in a student's WWU Universal Username given the student's first_name and last_name. The username begins with the first 6 characters of the last name, followed by the first letter of the first name. Return the username in all lower case. Example: uun_letters("Scott", "Wehrwein") => "wehrwes" """
    return (last_name[1:6] + first_name[0]).lower()

    return (last_name[:6] + first_name[0]).lower()

    return last_name[:6].lower() + first_name[:0].lower()

    return last_name[0:6].lower() + first_name[0].lower()
def uun_letters(first_name, last_name):
    """ Return the letters in a student's WWU Universal Username given the student's first_name and last_name. The username begins with the first 6 characters of the last name, followed by the first letter of the first name. Return the username in all lower case. Example: uun_letters("Scott", "Wehrwein") => "wehrwes" """
    return (last_name[1:6] + first_name[0]).lower()
    return (last_name[:6] + first_name[0]).lower()
    return last_name[:6].lower() + first_name[:0].lower()
def uun_letters(first_name, last_name):
    """ Return the letters in a student's WWU Universal Username given the student's first_name and last_name. The username begins with the first 6 characters of the last name, followed by the first letter of the first name. Return the username in all lower case. Example: uun_letters("Scott", "Wehrwein") => "wehrwes" """

    return (last_name[1:6] + first_name[0]).lower()

    return (last_name[:6] + first_name[0]).lower()

    return last_name[:6].lower() + first_name[:0].lower()

    return last_name[0:6].lower() + first_name[0].lower()
def uun_letters(first_name, last_name):
    """ Return the letters in a student's WWU Universal Username given the student's first_name and last_name. The username begins with the first 6 characters of the last name, followed by the first letter of the first name. Return the username in all lower case. Example: uun_letters("Scott", "Wehrwein") => "wehrwes" """

    return (last_name[1:6] + first_name[0]).lower()

    return (last_name[:6] + first_name[0]).lower()

    return last_name[:6].lower() + first_name[:0].lower()

    return last_name[0:6].lower() + first_name[0].lower()
Strings have methods!

Strings are objects - they also have methods.

last_name = "Wehrwein"
Strings have methods!

Strings are objects - they also have methods.

variable that refers to a string object

last_name = "Wehrwein"
Strings have methods!

Strings are objects - they also have methods.

last_name

variable that refers to a string object

a string literal

last_name = "Wehrwein"
Strings have methods!

Strings are objects - they also have methods.

```
last_name = "Wehrwein"
last_name.upper()
```
Strings have methods!

Strings are objects - they also have methods.

```
last_name = "Wehrwein"
last_name.upper()
```
Strings have methods!

```

<table>
<thead>
<tr>
<th>Ind</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;Wehrwein&quot;</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
```

Strings are objects - they also have methods.

```
last_name = "Wehrwein"
```

Methods can be called directly on the literal string, too:

```
"Wehrwein".upper()
```
Strings have many methods

here are a few of them:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper</td>
<td>none</td>
<td>Returns a string in all uppercase</td>
</tr>
<tr>
<td>lower</td>
<td>none</td>
<td>Returns a string in all lowercase</td>
</tr>
<tr>
<td>strip</td>
<td>none</td>
<td>Returns a string with the leading and trailing whitespace removed</td>
</tr>
<tr>
<td>count</td>
<td>item</td>
<td>Returns the number of occurrences of item</td>
</tr>
<tr>
<td>replace</td>
<td>old, new</td>
<td>Replaces all occurrences of old substring with new</td>
</tr>
<tr>
<td>find</td>
<td>item</td>
<td>Returns the leftmost index where the substring item is found, or -1 if not found</td>
</tr>
</tbody>
</table>
String methods: demo
upper, lower, count, replace, find, strip
String methods: demo

upper, lower, count, replace, find, strip

```python
word = "Banana"
word.upper()
word.lower()
word.count("a")
word.replace("a", "A")

line = " snails are out "
line.find("s")
line.find("snails")
line.find("banana")
line.strip()
line.strip().upper()

word = "Bellingham"
word = word[:9] + word[9].upper()
```
String Methods: More

The textbook (Section 9.5) has a more complete listing of string methods:
http://interactivepython.org/runestone/static/thinkcspy/Strings/StringMethods.html

The Python documentation has full details of the str type and all its methods:
https://docs.python.org/3/library/stdtypes.html#str

You should know how to use upper, lower, replace, and find.
String Methods: Evaluation

Problem: write an expression to determine if a string `user_input` contains the word "yes", with any capitalization and with any amount of spaces.
Problem: write an expression to determine if a string `user_input` contains the word "yes", with any capitalization and with any amount of spaces.

```python
user_input

=> " Y eS "
```
String Methods

**Problem:** write an expression to determine if a string `user_input` contains the word "yes", with any capitalization and with any amount of spaces.

```python
user_input.replace(" ", ")
```

=> "YeS"
String Methods

**Problem:** write an expression to determine if a string `user_input` contains the word "yes", with any capitalization and with any amount of spaces.

```python
user_input.replace(" ", ").lower()
```

=> "yes"
String Methods

**Problem:** write an expression to determine if a string `user_input` contains the word "yes", with any capitalization and with any amount of spaces.

```python
user_input.replace(" ", "").lower()
```

=> "yes"

dot (method call) operators are evaluated left-to-right!
String Methods

Problem: write an expression to determine if a string user_input contains the word "yes", with any capitalization and with any amount of spaces.

```python
user_input.replace(" ", ").lower() == "yes"
```

=> " Y eS ".replace(" ", ").lower() == "yes"

=> "YeS".lower() == "yes"

=> "yes" == "yes"

dot (method call) operators are evaluated left-to-right!
String Methods

**Problem:** write an *expression* to determine if a string `user_input` contains the word "yes", with any capitalization and with any amount of spaces.

```
user_input.replace(" ", "").lower() == "yes"
```

=> " Y   eS " .replace(" ", "").lower() == "yes"
=> "YeS".lower() == "yes"
=> "yes" == "yes"
=> True

dot (method call) operators are evaluated left-to-right!
Effects vs Return Values, again.

Most turtle methods **change the state** of the turtle object they're called on:

```python
t.forward(100)  # actually moves t forward
```

Most string methods return a **new** string with the given modifications:
Effects vs Return Values, again.

Most turtle methods **change the state** of the turtle object they're called on:

t.forward(100) # actually moves t forward

Most string methods return a **new** string with the given modifications:

s = "BOO"
Effects vs Return Values, again.

Most turtle methods **change the state** of the turtle object they're called on:

```python
t.forward(100)  # actually moves t forward
```

Most string methods return a **new** string with the given modifications:

```python
s = "BOO"
s.lower()  # => "boo"
```
Effects vs Return Values, again.

Most turtle methods **change the state** of the turtle object they're called on:

```python
t.forward(100)  # actually moves t forward
```

Most string methods return a **new** string with the given modifications:

```python
s = "BOO"
s.lower()   # => "boo"
print(s)    # prints BOO
```
Effects vs Return Values, again.

Most turtle methods **change the state** of the turtle object they're called on:

```python
t.forward(100) # actually moves t forward
```

Most string methods return a **new** string with the given modifications:

```python
s = "BOO"
s.lower() # => "boo"
print(s) # prints BOO
t = s.lower() # if you want "boo", save it
```
Effects vs Return Values, again.

Most turtle methods **change the state** of the turtle object they're called on:

```python
t.forward(100)  # actually moves t forward
```

Most string methods return a **new** string with the given modifications:

```python
s = "BOO"
s.lower()  # => "boo"
print(s)  # prints BOO
t = s.lower()  # if you want "boo", save it
```

Why is this? Because strings can't be modified. Try this:
Effects vs Return Values, again.

Most turtle methods change the state of the turtle object they're called on:

```python
t.forward(100)  # actually moves t forward
```

Most string methods return a new string with the given modifications:

```python
s = "BOO"
s.lower()  # => "boo"
print(s)  # prints BOO
t = s.lower()  # if you want "boo", save it
```

Why is this? Because strings can't be modified. Try this:

```python
s = "Scott"
s[3] = "o"  # error
```
String Methods

• What does this expression evaluate to?

"Wow".replace("W", "t").upper()

A. tot
B. WOW
C. TOW
D. TOT
Operators on Strings

Familiar:

+    concatenation
*    repetition
[ ]   indexing, slicing
==   equals
!=   not equals
Operators on Strings

Familiar:

+    concatenation  "a" + "b" => "ab"

*    repetition

[ ]  indexing, slicing

==   equals

!=   not equals
Operators on Strings

Familiar:

+    concatenation    "a" + "b" => "ab"

*    repetition        "ha" * 3 => "hahaha"

[ ]  indexing, slicing

==   equals

!=   not equals
Operators on Strings

Familiar:

+    concatenation    "a" + "b" => "ab"
*    repetition        "ha" * 3 => "hahaha"
[ ]  indexing, slicing "batman"[::3] => "bat"
==   equals
!=   not equals
Operators on Strings

Familiar:

+  concatenation  
   "a" + "b" => "ab"

*  repetition  
   "ha" * 3 => "hahaha"

[ ]  indexing, slicing  
   "batman"[:3] => "bat"

==  equals  
   "antman" == "natman" => False

!=  not equals
Operators on Strings

Familiar:

+    concatenation    "a" + "b" => "ab"

*    repetition        "ha" * 3 => "hahaha"

[]    indexing, slicing "batman"[:3] => "bat"

==   equals           "antman" == "natman" => False

!=   not equals       "antman" != natman" => True
String operators
String operators

Unfamiliar, but intuitive:
String operators

Unfamiliar, but intuitive:

in
String operators

Unfamiliar, but intuitive:

```
in  "a" in "abc". # => True
```
String operators

Unfamiliar, but intuitive:

```
in "a" in "abc".    # => True
"dab" in "abacadabra" # => True
```
String operators

Unfamiliar, but intuitive:

```
\texttt{in}  \quad \texttt{"a" in "abc".} \quad \# \Rightarrow \text{True}
\texttt{"dab" in "abacadabra"} \quad \# \Rightarrow \text{True}
\texttt{"A" in "abate"} \quad \# \Rightarrow \text{False}
```
String operators

Unfamiliar, but intuitive:

```
in           "a" in "abc".                      # => True
             "dab" in "abacadaabra"                # => True
             "A" in "abate"                      # => False
             "eye" in "team"                     # => False
```
String operators

Unfamiliar, but intuitive:

```
in  "a"  in  "abc".             # => True
    "dab"  in  "abacadabra"   # => True
    "A"  in  "abate"             # => False
    "eye"  in  "team"            # => False
```

not  in: exactly what you think (opposite of in)
String operators

Familiar, but (a little) unintuitive:

$$<, >$$

Inequality comparisons follow lexicographic ordering:

- Order based on the first character
- If tied, use the next character,
- and so on

These are all True:

```
"a" < "b"
"ab" < "ac"
"a" < "aa"
"" < "a"
```
String operators

Familiar, but (a little) unintuitive:

<, >

Caveat: lexicographic ordering is case-sensitive, and ALL upper-case characters come before ALL lower-case letters:

These are all True:

"A" < "a"
"Z" < "a"
"Jello" < "hello"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"

"Bellingham"
"Bellevue"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"

"Bellingham"
"Bellevue"

Tie - next character
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"

"Bellingham"
"Bellevue"

Tie - next character
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"

"Bellingham"
"Bellevue"

Tie - next character
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"

"Bellingham"
"Bellevue"

Tie - next character
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"

"Bellingham"
"Bellevue"

i > e, so "Bellingham" > "Bellevue"
Lexicographic Ordering

Example: "Bellingham" > "Bellevue"

"Bellingham"

"Bellevue"

i > e, so "Bellingham" > "Bellevue"

Aside:
"Bell" < "Bellingham" => True

When all letters are tied, the shorter word comes first.
Lexicographic Ordering: Aside

"?" < "!"  # => ???
Lexicographic Ordering: Aside

"?" < "!"  #  =>  ???

The `ord` function takes a character and returns its numerical (ASCII) code, which determines its ordering.
Lexicographic Ordering: Aside

"?" < "!"  # => ???

The `ord` function takes a character and returns its numerical (ASCII) code, which determines its ordering.

The `chr` function takes a numerical (ASCII) code and returns the corresponding character.
Lexicographic Ordering: Aside

"?" < "!" # => ???

The `ord` function takes a character and returns its numerical (ASCII) code, which determines its ordering.

The `chr` function takes a numerical (ASCII) code and returns the corresponding character.

```
ord("?") # => 63
ord("!") # => 33
```
Lexicographic Ordering: Aside

"?" < "!" # => False

The `ord` function takes a character and returns its numerical (ASCII) code, which determines its ordering.

The `chr` function takes a numerical (ASCII) code and returns the corresponding character.

```
ord("?") # => 63
ord("!") # => 33
"?" < "!" # => False
```
Lexicographic Ordering

ABCD: Which of the these evaluates to True?

A. "bat" > "rat"
B. "tap" < "bear"
C. "Jam" < "bet"
D. "STEAM" > "STEP!"