CSCI 141

Lecture 18
Strings: Slicing, String Methods, Comparison and in operators
Announcements
Announcements

- QOTD is out of 5 points (whoops)
Announcements

• QOTD is out of 5 points (whoops)

• QOTD explanation is linked from the explanation of the question on Canvas
Announcements

• QOTD is out of 5 points (whoops)

• QOTD explanation is linked from the explanation of the question on Canvas

• A4 is due next Wednesday.
Announcements

• QOTD is out of 5 points (whoops)

• QOTD explanation is linked from the explanation of the question on Canvas

• A4 is due next Wednesday.
  • Monday's a holiday, so (I assume) no mentor hours that day!
Announcements

- QOTD is out of 5 points (whoops)
- QOTD explanation is linked from the explanation of the question on Canvas
- A4 is due next Wednesday.
  - Monday's a holiday, so (I assume) no mentor hours that day!
  - Monday's a holiday, so no labs next week!
Goals

• Know how to use slicing to get substrings

• 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

  • Know strings are compared using lexicographic ordering

• Understand the meaning and implications of strings being immutable objects.
Inclusive Learning Environment: Redux
Inclusive Learning Environment: Redux

- Remember Lecture 1?
Inclusive Learning Environment: Redux

• Remember Lecture 1?
Inclusive Learning Environment: Redux

• Remember Lecture 1?

• Anyone felt like this at any point in the course?
Inclusive Learning Environment: Redux

• Remember Lecture 1?

• Anyone felt like this at any point in the course? (I have...?)
My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.

• You learn by doing.
• This involves making mistakes and asking questions.
• Nobody writes perfect code on the first try, not even professionals.

Keep this in mind when:
Inclusive Learning Environment: Redux

• My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.
  • You learn by doing.
  • This involves making mistakes and asking questions.
  • Nobody writes perfect code on the first try, not even professionals.

• Keep this in mind when:

This is you.
Inclusive Learning Environment: Redux

• My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.

  • You learn by **doing**.
  • This involves **making mistakes** and **asking questions**.
  • **Nobody** writes perfect code on the first try, not even professionals.

• **Also** keep this in mind when:
Inclusive Learning Environment: Redux

• My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.
  • You learn by doing.
  • This involves making mistakes and asking questions.
  • Nobody writes perfect code on the first try, not even professionals.

• Also keep this in mind when:

This is you.
People from underrepresented groups face extra obstacles.
People from underrepresented groups face extra obstacles.

This claim is (heavily) backed by scientific research.
People from underrepresented groups face extra obstacles.

This claim is (heavily) backed by scientific research.

Disclaimer: I am not a psychologist.
People from underrepresented groups face extra obstacles.

This claim is (heavily) backed by scientific research.

Stereotype threat:
sterotypes become self-fulfilling when the subjects of the stereotype are conscious of them.

Disclaimer: I am not a psychologist
People from underrepresented groups face extra obstacles.

This claim is (heavily) backed by scientific research.

**Stereotype threat:**
stereotypes become self-fulfilling when the subjects of the stereotype are conscious of them.

**Implicit bias:**
well-intentioned people exhibit biases that they're not even aware they have.

Disclaimer: I am not a psychologist
People from underrepresented groups face extra obstacles.  

Disclaimer: I am not a psychologist

This claim is (heavily) backed by scientific research.

**Stereotype threat:**
Stereotypes become self-fulfilling when the subjects of the stereotype are conscious of them.

**Implicit bias:**
Well-intentioned people exhibit biases that they're not even aware they have.

**Impostor syndrome:**
Successes are attributed to luck
Failures are attributed to ability
Dr. Moushumi Sharmin is new Community Ambassador for CS.

Community office hours for Fall 2019 are **Wednesdays from 10:30-11:30 am (CF 465).** Students, faculty and staff are welcome to discuss issues related to **equity, inclusion, and diversity.**
MIX IT UP: Inclusion in STEM Mixer
Last time: Indexing Strings, Negative Indices

Negative indices count backwards from len(s):

<table>
<thead>
<tr>
<th>Index:</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>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S u m m e r</td>
<td>i s n e a r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index:</td>
<td>-14</td>
<td>-13</td>
<td>-12</td>
<td>-11</td>
<td>-10</td>
<td>-9</td>
<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>
Last time: Indexing Strings, Negative Indices

Negative indices count backwards from len(s):

<table>
<thead>
<tr>
<th>Index:</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>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum</td>
<td>mer</td>
<td>i</td>
<td>s</td>
<td>n</td>
<td>e</td>
<td>a</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index:</td>
<td>-14</td>
<td>-13</td>
<td>-12</td>
<td>-11</td>
<td>-10</td>
<td>-9</td>
<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>

Two possible ways to remember how this works:
Last time: Indexing Strings, Negative Indices

Negative indices count backwards from len(s):

<table>
<thead>
<tr>
<th>Index:</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>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Summer is near</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index:</td>
<td>-14</td>
<td>-13</td>
<td>-12</td>
<td>-11</td>
<td>-10</td>
<td>-9</td>
<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>

Two possible ways to remember how this works:

```
a_string[-5]
``` is equivalent to
```
a_string[len(a_string)-5]
```
Last time: Indexing Strings, Negative Indices

Negative indices count backwards from \( \text{len(s)} \):

<table>
<thead>
<tr>
<th>Index</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>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>u</td>
<td>m</td>
<td>m</td>
<td>e</td>
<td>r</td>
<td>i</td>
<td>s</td>
<td>n</td>
<td>e</td>
<td>a</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>-14</td>
<td>-13</td>
<td>-12</td>
<td>-11</td>
<td>-10</td>
<td>-9</td>
<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>

Two possible ways to remember how this works:

-1 is always the last character, and indices count backwards from there.

\[
a_{\text{string}}[-5] \quad \text{is equivalent to} \quad a_{\text{string}}[\text{len}(a_{\text{string}})-5]
\]
QOTD

```python
def flop(value, number):
    output = ""

    for i in range(number, 0, -1):
        output = output + value[i-1]

    for i in range(number, len(value)):
        output = output + value[i]

    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if `a` is an integer?
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]

    for i in range(number, len(value)):
        output = output + value[i]
    return output

Which of the following is not a possible return value of flop("no time", a) if a is an integer?
Which of the following is **not** a possible return value of `flop("no time", a)` if `a` is an integer?
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output

Which of the following is not a possible return value of flop("no time", a) if a is an integer?

mit one  t onime  emit on
nomite  on time  timeno time
```python
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is not a possible return value of `flop("no time", a)` if `a` is an integer?

- `mit one`
- t onime
- emit on
- nomite
- on time
- timenono time
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output

Which of the following is not a possible return value of flop("no time", a) if a is an integer?

✓ mit one   t onime   emit on

✗ nomite   on time   timeno time
```python
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if `a` is an integer?

- mit one
- t onime
- emit on
- nomite
- on time
- timeno time
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output

Which of the following is not a possible return value of flop("no time", a) if a is an integer?

✓mit one ✓t onime emit on
✗nomite ✓on time timeno no time
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output

Which of the following is not a possible return value of flop("no time", a) if a is an integer?

✓ mit one    ✓ t onime    ✓ emit on
✗ nomite    ✓ on time
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output

Which of the following is not a possible return value of
flop("no time", a) if a is an integer?

✔ mit one ✔ t onime ✔ emit on
✖ nomite ✔ on time ✔ time no time
def remove_comments(string):
    ''' Return a copy of string, but with all characters starting with the first # symbol removed. If there is no # in the string, return input unchanged. '''

    Hint: try a while loop!

# Example:

remove_comments("a = b # assign b to a")
# => "a = b "
Slicing: indexing substrings

```
alph = "abcdefgghij"
alph[0]  # => "a"
alph[4]  # => "e"
```

<table>
<thead>
<tr>
<th>str</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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
</tr>
</tbody>
</table>

**Ind**

<table>
<thead>
<tr>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>j</td>
</tr>
</tbody>
</table>
Slicing: indexing substrings

alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"

What if I want to "index" more than one character at a time?
Slicing: indexing substrings

```ruby
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

What if I want to "index" more than one character at a time?

```ruby
alph[???] # => "cdef"
```
Slicing: indexing substrings

\[
\text{alph} = "abcdefgij"
\]
\[
\text{alph}[0] \quad \Rightarrow \quad "a"
\]
\[
\text{alph}[4] \quad \Rightarrow \quad "e"
\]

\text{Slicing syntax: } \text{string}[\text{start:end}]
Slicing: indexing substrings

alph = "abcdefgij"
alph[0] # => "a"
alph[4] # => "e"

index of first character

Slicing syntax: string[start:end]
Slicing: indexing substrings

```
alph = "abcdefgij"
alph[0] # => "a"
alph[4] # => "e"
```

Index of first character = 1 + index of last character

**Slicing syntax:** `string[start:end]`
Slicing: indexing substrings

```
alph = "abcdefgghij"
alph[0] # => "a"
alph[4] # => "e"
```

<table>
<thead>
<tr>
<th>( \text{Ind} )</th>
<th>( \text{Val} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
</tr>
</tbody>
</table>

Slicing syntax: \( \text{string[start:end]} \)

- Index of first character: \( 1 + \text{index of last character} \)
- Just like the `range` function: the end index is **not** included.
Slicing: indexing substrings

\[ \text{alph} = \text{"abcdefghij"} \]
\[ \text{alph}[0] \# => \text{"a"} \]
\[ \text{alph}[4] \# => \text{"e"} \]

### Slicing syntax:
```
string[start:end]
```

index of first character \( 1 + \) index of last character

\[ \text{alph}[2:6] \# => \text{"cdef"} \]

just like the \text{range} function:
the end index is \textbf{not} included
Slicing: indexing substrings

alph = "abcdefghij"

alph[0] # => "a"
alph[4] # => "e"

Slicing syntax: string[start:end]

index of first character
1 + index of last character

alph[2:6] # => "cdef"

alph[0:10] # => "abcdefghij"
Slicing: indexing substrings

alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"

index of first character 1 + index of last character

Slicing syntax: string[start:end]

alph[2:6] # => "cdef"
alph[0:10] # => "abcdefghij"
alph[5:-2] just like the range function: the end index is not included
Slicing: indexing substrings

\[
\text{alph} = "abcdefg\text{hij}"
\]
\[
\text{alph}[0] \quad \# \Rightarrow "a"
\]
\[
\text{alph}[4] \quad \# \Rightarrow "e"
\]

\[
\begin{array}{c|cccccccccc}
\text{str} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
\text{Ind} & & & & & & & & & & \\
\text{Val} & a & b & c & d & e & f & g & h & i & j \\
\end{array}
\]

\[
\text{index of first character} \quad 1 + \text{index of last character}
\]

**Slicing syntax:** \(\text{string}[\text{start}:\text{end}]\)

\[
\text{alph}[2:6] \quad \# \Rightarrow "cdef"
\]
\[
\text{alph}[0:10] \quad \# \Rightarrow "abcdefg\text{hij}"
\]
\[
\text{alph}[5:-2] \quad \# \Rightarrow "fgh"
\]

just like the \texttt{range} function: the end index is **not** included
Slicing: indexing substrings

alph = "abcdefgij"

<table>
<thead>
<tr>
<th>Ind</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
</tr>
</tbody>
</table>

index of first character = 1 + index of last character

Slicing syntax: `string[start:end]`
Slicing: indexing substrings

```python
alph = "abcdefgij"
```

<table>
<thead>
<tr>
<th>Ind</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
</tr>
</tbody>
</table>

**Slicing syntax:**  `string[start:end]`

- index of first character: `start`
- 1 + index of last character: `end`
- If omitted, `start` defaults to 0
Slicing: indexing substrings

alpha = "abcdefg hij"

<table>
<thead>
<tr>
<th>Ind</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
</tr>
</tbody>
</table>

index of first character 1 + index of last character

Slicing syntax: `string[start:end]`

- If omitted, `start` defaults to 0
- If omitted, `end` defaults to `len(string)`
**Slicing: indexing substrings**

```
alph = "abcdefgij"
```

<table>
<thead>
<tr>
<th>Ind</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
</tr>
</tbody>
</table>

**index of first character**  \[ \text{1 + index of last character} \]

**Slicing syntax:** `string[start:end]`

- If omitted, `start` defaults to `0`
- If omitted, `end` defaults to `len(string)`

```
alph[::4] # => "abcd"
```
**Slicing: indexing substrings**

```python
alph = "abcdefgij"
```

### Indexing Table

<table>
<thead>
<tr>
<th>Ind</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
<tr>
<td>7</td>
<td>h</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>j</td>
</tr>
</tbody>
</table>

### Slicing Syntax

`slicing syntax: string[start:end]`

- If omitted, `start` defaults to 0
- If omitted, `end` defaults to `len(string)`

- `alph[:4]` # => “abcd”
- `alph[5:]` # => “fghij”
String Slicing: Demo
String Slicing: Demo

- \texttt{s = "fibonacci"}
- Positive indices: \texttt{s[1:3]}
- Negative indices!? \texttt{s[-4:9]}
- Leaving out start/endpoint: \texttt{s[:6]}, \texttt{s[4:]}
- Indices past the end in a slice: \texttt{s[1:21]}
- Single indices past the end: \texttt{s[9]}, \texttt{s[21]}
- Loop over a slice of a string
  
  \begin{verbatim}
  for c in s[2:6]:
      print(c, "!", sep=" ", end=" ")
  \end{verbatim}
String Slicing: Exercise

Which of these evaluates to "in"?

A. last_name[7:8]
B. last_name[6:-1]
C. last_name[-3:]  
D. last_name[-2:8]
String Slicing: Exercise

last_name = "Wehrwein"

Which of these evaluates to "in"?

A. last_name[7:8]
B. last_name[6:-1]
C. last_name[-3:]
D. last_name[-2:8]
Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:
Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

```python
t = turtle.Turtle()
```

![Diagram showing t as an object with data and methods](turtle_diagram.png)
Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

```python
t = turtle.Turtle()
t.forward(100)
```
Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

```
t = turtle.Turtle()
t.forward(100)
```
Strings are **objects**.

We’ve seen other objects before: turtles!

Turtles had methods:

```
t = turtle.Turtle()
t.forward(100)
```
Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

```
t = turtle.Turtle()
t.forward(100)
```

variable that refers to a turtle object

turtle module
module function (turtle constructor)
Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

```python
import turtle

t = turtle.Turtle()
t.forward(100)
```

- `t = turtle.Turtle()` is a module function (turtle constructor).
- `t.forward(100)` is a method of a turtle object.

The variable `t` is a variable that refers to a turtle object.

# data and methods
Strings are objects.

Strings are objects too - they also have methods.

```
last_name = "Wehrwein"
```
Strings are **objects**.

Strings are objects too - they also have methods.

```
last_name  
str
Ind  0 1 2 3 4 5 6 7
Val  W e h r w e i n
```

variable that refers to a string object

```
last_name = "Wehrwein"
```
Strings are objects. 

Strings are objects too - they also have methods.

```
last_name = "Wehrwein"
```

Ind  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  
---   |----|----|----|----|----|----|----|----
Val   | W  | e  | h  | r  | w  | e  | i  | n  

variable that refers to a string object

a string literal

last_name = "Wehrwein"
Strings are objects.

Strings are objects too - they also have methods.

```
last_name = "Wehrwein"
last_name.upper()
```
Strings are objects.

Strings are objects too - they also have methods.

variable that refers to a string object

a string literal

last_name = "Wehrwein"

last_name.upper()
Strings are **objects**.

last_name

Strings are objects too - 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

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**.
Worksheet - Exercise 2

```python
phrase = "WWU is in Bellingham"
phrase = phrase[:19] + phrase[19].upper()
```

Write a function that capitalizes the last letter of any string:

```python
def capitalize_last(in_str):
    """ Return a copy of in_str with its last letter capitalized. """
```

# Example:
capitalize_last("Mix") # => "MiX"
Problem: write an expression to determine if a string user_input contains the word "yes", with any capitalization and with any amount of spaces.
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.

```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.

```
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.

```
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!*
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"
=> True

dot (method call) operators are evaluated left-to-right!
**def remove_comments(string):**

""" Return a copy of string, but with all characters starting with the first # symbol removed. If there is no # in the string, return input unchanged. """

Do this without a loop!

For reference:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.upper()</td>
<td>Returns s in all uppercase</td>
</tr>
<tr>
<td>s.lower()</td>
<td>Returns s in all lowercase</td>
</tr>
<tr>
<td>s.strip()</td>
<td>Returns s with the leading and trailing whitespace removed</td>
</tr>
<tr>
<td>s.count(t)</td>
<td>Returns the number of occurrences of t in s</td>
</tr>
<tr>
<td>s.replace(u, v)</td>
<td>Replaces all occurrences of substring u with v in s</td>
</tr>
<tr>
<td>s.find(t)</td>
<td>Returns the leftmost index where the substring item is found, or -1 if not found</td>
</tr>
</tbody>
</table>
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:

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

Unfamiliar, but intuitive:

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

Unfamiliar, but intuitive:

```python
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

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 (possibly) 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"
Familiar, but (possibly) 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 (possibly) 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 (possibly) 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"

"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 \texttt{ord} function takes a character and returns its numerical (ASCII) code, which determines its ordering.

The \texttt{chr} function takes a numerical (ASCII) code and returns the corresponding character.

\begin{verbatim}
ord("?")  # => 63
ord("!")  # => 33
\end{verbatim}
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
ord(? )  # => False
```
Lexicographic Ordering

**ABCD:** Which of the these evaluates to True?

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