CSCI 141

Lecture 17
String Manipulation
Happenings

CS Mentors Present: GITHUB Pages Workshop
  Tuesday, November 5\textsuperscript{th} 5:00 PM CF 165

Tech Talk: Pacific Northwest National Labs (PNNL)
  Tabling Wednesday, Nov 6\textsuperscript{th} 10:00-3:30 PM CF 4\textsuperscript{th} Floor Foyer
  Tech Talk Wednesday, Nov 6\textsuperscript{th} 5:00-6:00 PM CF 115

ACM Hosts: Fast Enterprises
  Career Prep Presentation Wednesday, Nov 6\textsuperscript{th} 6:00-7:00 PM CF 316

Group Advising Session for CS Premajors
  Thursday, November 7\textsuperscript{th} 3:00-4:30 PM
Announcements
Announcements

• Midterm grades are out - see Canvas announcement for full details.
Announcements

• Midterm grades are out - see Canvas announcement for full details.
  
  • Review your exam on Gradescope by Wednesday night for 2 bonus points on your exam score
Announcements

• Midterm grades are out - see Canvas announcement for full details.
  
  • Review your exam on Gradescope by Wednesday night for 2 bonus points on your exam score

  • If you do better on the final, it will replace your midterm grade.
Feedback Survey Results
## Feedback Survey Results

### Lecture Pace

On average, how would you describe the pace of lectures?

<table>
<thead>
<tr>
<th>Way too slow</th>
<th>3 respondents</th>
<th>2%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Somewhat too slow</td>
<td>31 respondents</td>
<td>18%</td>
</tr>
<tr>
<td>Just right</td>
<td>85 respondents</td>
<td>49%</td>
</tr>
<tr>
<td>Somewhat too fast</td>
<td>49 respondents</td>
<td>28%</td>
</tr>
<tr>
<td>Way too fast</td>
<td>4 respondents</td>
<td>2%</td>
</tr>
</tbody>
</table>
# Feedback Survey Results

## QOTD Review

<table>
<thead>
<tr>
<th>Rank</th>
<th>Feedback</th>
<th>Responses</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not helpful at all</td>
<td>1</td>
<td>1%</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>17</td>
<td>10%</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>42</td>
<td>24%</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>64</td>
<td>37%</td>
</tr>
<tr>
<td>5</td>
<td>Extremely helpful</td>
<td>48</td>
<td>28%</td>
</tr>
</tbody>
</table>
What shouldn't change?

Common themes, approximately in order of frequency:

• QOTDs
• In-class demos
• Socrative
What should change?

Common themes, approximately in order of frequency:

• Reviewing the QOTD is helpful but takes too much time, so new material is rushed.

• A3 was too hard; I gave too little guidance.

• Demos are helpful, often more so than slides.

• You need more in-class practice, especially coding on paper.
What am I doing about it?

Reviewing the QOTD is helpful but takes too much time, so new material is rushed.

• Short term: Spend less time on the QOTD
• Short term experiment: Written explanations of QOTD
• Long term: Video explanations of QOTD
What am I doing about it?

A3 was too hard; I gave too little guidance.

• Short term: Friday's lecture was all about how to approach A4.

• Long term: Adjust A3 difficulty and give more tips for how to approach each problem.
What am I doing about it?

Demos are helpful, often more so than slides.
You need more in-class practice, especially coding on paper.

• Strive to make slides more concise, talk a little less, and allow time for:
  • More frequent demos with more examples per demo.
  • More in-class exercises, including on-paper coding.
Other

• Many people don't like spending class time answering out-of-scope questions.

• Many people do not like taking CS exams on paper.

• Many people would like me to go over assignment solutions.
Goals

• Review what we know already about strings:
  • the `str` type, `+` and `*` operators, `len` function

• Know how to iterate over tuples and strings using `for` loops

• Know how to `index` into a string

• Know how Python interprets `negative indices` into strings.

• Know how to use `slicing` to get substrings
import math

def square(x):
    return x ** 2

def quadratic(a, b, c):
    disc = discriminant(a, b, c)
    return (-b + disc) / (2 * a)

def discriminant(a, b, c):
    b2 = square(b)
    return math.sqrt(b2 - 4 * a * c)

print(quadratic(4, 6, 2))

Which of the following are local variables belonging to the discriminant function?
- x, a, b, c, disc, b2
Last time: **tuples**

- A tuple is a sequence of values, optionally enclosed in parens.
  
  \[(1, 4, "Mufasa")\]

- You can “pack” and “unpack” them using assignment statements:
  
  ```python
  v = (1, 4, "Mufasa")  # "packing"
  (a, b, c) = v  # "unpacking"
  ```
Run the following code. If a line causes an error, skip it and continue execution. Which lines, if any, cause errors?

```python
1 a, b, c = 6, 4, 2
2 (z, x) = c, b
3 print((x, z))
4 v = (x, z, c)
5 print(v)
6 a, b = v
```
What does line 5 print?

1   a, b, c = 6, 4, 2
2   (z, x) = c, b
3   print((x, z))
4   v = (x, z, c)
5   print(v)
6   a, b = v

A. (2, 4, 2)
B. (2, 2, 4)
C. (6, 4, 2)
D. (4, 2, 2)
E. (4, 4, 2)
fun fact:

Tuples are sequences,
so they can be used in for loops just like lists and ranges.

These two loops do the same thing:

```python
for number in [1, 3]:
    print(number, end=" ")
```

```python
for number in (1, 3):
    print(number, end=" ")
```
fun fact:

Tuples are sequences, so they can be used in for loops just like lists and ranges.

These two loops do the same thing:

```python
for number in [1, 3]:
    print(number, end=" ")

for number in (1, 3):
    print(number, end=" ")
```

Exercise: write a for loop that uses a range to print the same thing.
Today: Strings

Don’t we already know about strings?
Today: Strings

Don’t we already know about strings?

type("hello")
Today: Strings

Don’t we already know about strings?

def type("hello") # => <class 'str'>
Today: Strings

Don’t we already know about strings?

```python
print("Hello")
```

```python
type("hello")  # => <class 'str'>
```

```python
print("Hello")
```
Today: Strings

Don’t we already know about strings?

```python
type("hello")  # => <class 'str'>

print("Hello")  # prints Hello to the console
```
Today: Strings

Don’t we already know about strings?

```python
print("Hello")  # prints Hello to the console
"Hello" + "World"
```
Today: Strings

Don’t we already know about strings?

```python
type("hello")  # => <class 'str'>

print("Hello")  # prints Hello to the console

"Hello" + "World"  # => "HelloWorld"
```
Today: Strings

Don’t we already know about strings?

```python
type("hello")  # => <class 'str'>

print("Hello")  # prints Hello to the console

"Hello" + "World"  # => "HelloWorld"

len("abc")
```
Today: Strings

Don’t we already know about strings?

```python
type("hello")  # => <class 'str'>

print("Hello")  # prints Hello to the console

"Hello" + "World"  # => "HelloWorld"

len("abc")  # => 3
```
Today: Strings

Don’t we already know about strings?

type("hello")  # => <class ‘str’>

print("Hello")  # prints Hello to the console

"Hello" + "World"  # => “HelloWorld"

len("abc")  # => 3

"na" * 16 + " Batman!"
Today: Strings

Don’t we already know about strings?

```python
type("hello")  # => <class 'str'>
print("Hello")  # prints Hello to the console
"Hello" + "World"  # => "HelloWorld"
len("abc")  # => 3
"na" * 16 + " Batman!"
# => ...
```
Today: Strings

Don’t we already know about strings?

def type("hello")  # => <class ‘str’>
def print("Hello")  # prints Hello to the console

def "Hello" + "World"  # => “HelloWorld”
def len("abc")  # => 3

def "na" * 16 + " Batman!"
# => …"nananananananananananananananana Batman!"
Strings: What else is there?
Strings: What else is there?

def house_number(address_line):

Strings: What else is there?

def house_number(address_line):
    """ Return the house number portion of the given address line."""
Strings: What else is there?

def house_number(address_line):
    """ Return the house number portion of
    the given address line.
    Examples:
def house_number(address_line):
    """ Return the house number portion of the given address line."
    Examples:
    house_number("1600 Pennsylvania Ave")
    => 1600
def house_number(address_line):
    """ Return the house number portion of the given address line.
    Examples:
    house_number("1600 Pennsylvania Ave")
    => 1600
    house_number("221B Baker St")
    => 221
    """
def house_number(address_line):
    """ Return the house number portion of the given address line.
    Examples:
    house_number("1600 Pennsylvania Ave")
    => 1600
    house_number("221B Baker St")
    => 221
    """

    # ????
    return result
def remove_comments(string):
    """ Return a copy of string, but with all characters starting with and following the first instance of ‘#’ emoved. If there is no # in the string, return input unchanged. """
def remove_comments(string):
    """ Return a copy of string, but with all characters starting with and following the first instance of ‘#’ emoved. If there is no # in the string, return input unchanged. """
    # ????
def remove_comments(string):
    """Return a copy of string, but with all characters starting with and following the first instance of '#' removed. If there is no # in the string, return input unchanged."
    """
    # ????
    return result
fun fact:

Strings are sequences, so they can be used in for loops just like lists and ranges.

Check this out:

```python
for letter in "Bellingham":
    print(letter, "-", sep=" ", end=" ")
```
Strings are sequences, so they can be used in for loops just like lists and ranges.

Check this out:

```python
for letter in "Bellingham":
    print(letter, "-", sep=" ", end=" ")
```

Demo?
fun fact:

Strings are sequences, so they can be used in for loops just like lists and ranges.

Check this out:

```python
for letter in "Bellingham":
    print(letter, "-", sep=" ", end=" ")
```
Strings are sequences, so they can be used in for loops just like lists and ranges.

Check this out:

```python
for letter in "Bellingham":
    print(letter, "-", sep=" ", end="")
```

What does this print?

A. Bellingham  
B. B-e-l-l-i-n-g-h-a-m  
C. -B-e-l-l-i-n-g-h-a-m  
D. B-e-l-l-i-n-g-h-a-m-
Exercise (worksheet #1)

Write a function that prints a string with all vowels removed.

```python
def remove_vowels(string):
    """ Print string, but with no vowels. Don't count y as a vowel. """
```
Exercise (worksheet #1)

Write a function that **prints** a string with all vowels removed.

```python
def remove_vowels(string):
    """ Print string, but with no vowels. Don't count y as a vowel. ""
```

Possible modification: Return the result instead of printing it.
Strings are collections of individual characters. We can get access to an individual character by index.

outlook = “Winter is coming”

How is this stored in memory?
Indexing into Strings

Strings are collections of individual characters. We can get access to an individual character by index.

outlook = “Winter is coming”

How is this stored in memory?
Indexing into Strings

Strings are collections of individual characters. We can get access to an individual character by index.

`outlook = “Winter is coming”`

How is this stored in memory?

```
<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>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value:</td>
<td>W</td>
<td>i</td>
<td>n</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td>i</td>
<td>s</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>i</td>
<td>n</td>
<td>g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(just smaller strings!)
Strings are collections of individual characters. We can get access to an individual character by index.

```python
outlook = "Winter is coming"
```

How is this stored in memory?

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>t</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
</tr>
<tr>
<td>8</td>
<td>i</td>
</tr>
<tr>
<td>9</td>
<td>n</td>
</tr>
<tr>
<td>10</td>
<td>g</td>
</tr>
<tr>
<td>11</td>
<td>i</td>
</tr>
<tr>
<td>12</td>
<td>n</td>
</tr>
<tr>
<td>13</td>
<td>e</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td>ing</td>
</tr>
</tbody>
</table>

Indices in Python begin at 0.
Indexing into Strings

Strings are collections of individual characters. We can get access to an individual character by index.

outlook = "Winter is coming"

How is this stored in memory?

Syntax:

Indices in Python begin at 0.
Indexing into Strings

Strings are collections of individual characters. We can get access to an individual character by `index`.

```python
outlook = "Winter is coming"
```

How is this stored in memory?

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Index:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>outlook</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
<td></td>
</tr>
<tr>
<td>W i n t e r i s c o m i n g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:

```python
outlook[0] # => "W"
```

Indices in Python begin at 0.
Indexing into Strings

Strings are collections of individual characters. We can get access to an individual character by **index**.

```python
outlook = "Winter is coming"
```

How is this stored in memory?

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>t</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
</tr>
<tr>
<td>8</td>
<td>is</td>
</tr>
<tr>
<td>9</td>
<td>com</td>
</tr>
<tr>
<td>10</td>
<td>ing</td>
</tr>
</tbody>
</table>

Syntax:

```python
outlook[0]  # => "W"
outlook[4]  # => "e"
```

Indices in Python begin at 0.

(just smaller strings!)
Indexing into Strings

Strings are collections of individual characters. We can get access to an individual character by index.

`outlook = "Winter is coming"

How is this stored in memory?

Syntax:

```python
outlook[0] # => "W"
outlook[4] # => "e"
```

Indices in Python begin at 0.

Spaces are characters too!
Indexing into Strings

Strings are collections of individual characters. We can get access to an individual character by index.

```
outlook = "Winter is coming"
```

How is this stored in memory?

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>t</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
</tr>
<tr>
<td>8</td>
<td>is</td>
</tr>
<tr>
<td>9</td>
<td>coming</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Syntax:

```
outlook[0] # => "W"
outlook[4] # => "e"
```

Indices in Python begin at 0.
Spaces are characters too!

```
outlook[6] # => " "
```
Indexing into Strings

Assume \( s \) is a variable that refers to the above string object. How would I access the letter 'r'?

A. \( s[5] \)
B. \( s(5) \)
C. \( s[6] \)
D. \( s(6) \)
**Indexing into Strings**

What is the index of the last character of a string $s$? (not the specific string above - this should work for any string)

A. $\text{len}(s) - 1$
B. $\text{len}(s)$
C. $\text{len}(s) + 1$
D. 42
A consequence of indexing - Another way to loop through strings:

```python
for letter in a_string:
    print(letter, "-", sep=" ", end=" ")
```

is equivalent to

```python
for i in range(len(a_string)):
    print(a_string[i], "-", sep=" ", end=" ")
```
Nifty Python Feature: Negative Indices

Negative indices count backwards from len(s):

Index:

Also Index:
## Nifty Python Feature: 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>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also Index:</td>
<td>-16</td>
<td>-15</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>

Winter is coming
Nifty Python Feature: 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>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Winter is coming</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>
<td></td>
</tr>
<tr>
<td>Also Index:</td>
<td>-16</td>
<td>-15</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:
Nifty Python Feature: 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>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Winteriscoming</td>
<td>W</td>
<td>i</td>
<td>n</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td>i</td>
<td>s</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>i</td>
<td>n</td>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Also Index:</td>
<td>-16</td>
<td>-15</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]
```
# Nifty Python Feature: 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>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Winter</td>
<td>W</td>
<td>i</td>
<td>n</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td>i</td>
<td>s</td>
<td>c</td>
<td>o</td>
<td>m</td>
<td>i</td>
<td>n</td>
<td>g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Also Index: -16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

**Two possible ways to remember how this works:**

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

```python
a_string[-5] is equivalent to a_string[len(a_string)-5]
```
Negative Indices!

For which assignment of $a$ and $b$ does the above not print True?

A. $a = 1 \quad C. a = -8$
   $b = 5 \quad b = -4$

B. $a = 1 \quad D. a = -2$
   $b = 7 \quad b = 6$
Negative Indices!

last_name = "wehrwein"

For which assignment of a and b does the above not print True?

A. a = 1  b = 5
   C. a = -8  b = -4

B. a = 1  b = 7
   D. a = -2  b = 6
Negative Indices!

last_name = "wehrwein"

For which assignment of $a$ and $b$ does the above not print True?

A. $a = 1$  
   $b = 5$

B. $a = 1$  
   $b = 7$

C. $a = -8$  
   $b = -4$

D. $a = -2$  
   $b = 6$
Negative Indices!

last_name = "wehrwein"

print(last_name[a] == last_name[b])

For which assignment of $a$ and $b$ does the above not print True?

A. $a = 1$  \hspace{1cm} C. $a = -8$
    \hspace{1cm} $b = 5$  \hspace{1cm} \hspace{1cm} $b = -4$

B. $a = 1$  \hspace{1cm} D. $a = -2$
    \hspace{1cm} $b = 7$  \hspace{1cm} \hspace{1cm} $b = 6$
Negative Indices!

```
last_name = "wehrwein"

print(last_name[a] == last_name[b])
```

For which assignment of $a$ and $b$ does the above not print True?

A. $a = 1$  B. $a = 1$
   $b = 5$  $b = 7$

C. $a = -8$  D. $a = -2$
   $b = -4$  $b = 6$
Indexing gives us other ways to loop through strings:

```python
for letter in a_string:
    print(letter, end="")
```

is equivalent to

```python
for i in range(len(a_string)):
    print(a_string[i], end="")
```

and also

```python
i = 0
while i < len(a_string):
    print(a_string[i], end="")
    i += 1
```
def remove_comments(string):
    """ Return a copy of string, but with
    all characters starting with and following
    the first instance of '#' emoved. If there
    is no # in the string, return input unchanged.
    """

# Example:
remove_comments("a = b # assign b to a")
# => "a = b "
def remove_comments(string):
    """ Return a copy of string, but with all characters starting with and following the first instance of '#' removed. If there is no # in the string, return input unchanged.  """

Hint: use a while loop!

# Example:

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

str
0 1 2 3 4 5 6 7 8 9
a b c d e f g h i j

Slicing syntax: string[start:end]

just like the range function: the end index is not included

alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
Slicing: indexing substrings

Slicing syntax: `string[start:end]`

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

```
<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>
```

1 + index of last character

just like the `range` function:
the end index is not included
**Slicing: indexing substrings**

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

<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 + \text{index of last character})

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

just like the `range` function: the end index is **not** included.
Slicing: indexing substrings

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

<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]`

```
alph[0:5] # => "abcde"
```

just like the `range` function: the end index is **not** included
Slicing: indexing substrings

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

Slicing syntax: string[start:end]

just like the range function: the end index is not included

index of first character 1 + index of last character

alph[0:5] # => "abcde"
alph[0:10] # => "abcdefgij"
Slicing: indexing substrings

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

<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]

```
alph[0:5] # => "abcde"
alph[0:10] # => "abcdefghij"
alph[5:-2]  
```

just like the range function: the end index is **not** included
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]

alph[0:5] # => "abcde"
alph[0:10] # => "abcdefgij"
alph[5:-2] # => "fgh"
Slicing: indexing substrings

\[\text{alph} = "abcdefg hij"
\]

<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 \( \text{1 + index of last character} \)

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

\[ \text{alph} = "abcdefgij" \]

<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: \(1 + \text{index of last character}\)

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

If omitted, \(\text{start}\) defaults to 0
Slicing: indexing substrings

\[
\text{alph} = "abcdefg\text{hi}j"
\]

<table>
<thead>
<tr>
<th>Ind</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>Val</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>

index of first character $\rightarrow$ \( 1 + \text{index of last character} \)

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

- If omitted, \( \text{start} \) defaults to 0
- If omitted, \( \text{end} \) defaults to \( \text{len(string)} \)
Slicing: indexing substrings

```python
alph = "abcdefghij"

Slicing syntax:  `string[start:end]`

index of first character  1 + index of last character

If omitted, `start` defaults to 0

If omitted, `end` defaults to `len(string)`

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

alph = "abcdefglij"

index of first character 1 + index of last character

Slicing syntax: string[start:end]

<table>
<thead>
<tr>
<th>Ind</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>Val</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>

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

alph[:4] # => "abcd"
alph[5:] # => "fghij"
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]