CSCI 141

Lecture 20
Lists
Mutability
Variables are References
Announcements
Announcements

• A4 is in! (tonight, if you're using all 3 slip days)
Announcements

• A4 is in! (tonight, if you're using all 3 slip days)
• I have office hours 2-3 today.
Announcements

• A4 is in! (tonight, if you're using all 3 slip days)
• I have office hours 2-3 today.
• A5 out tomorrow or Wednesday, due Friday 5/31
CS Stories: What’s it like to be a Female Professor?

Who: Dr. Sharmin, Dr. Liu, Dr. Islam, AWC professional guests from industry, alumni, friends, YOU!

What: Creating the space to open about experiences as students in education with various career goals in addition to equipping our friends to be allies for underrepresented friends.

When: Thursday May 23rd from 3-5pm. Doors open @2:45pm

Where: Wilson Library Reading Room #480 (yes the Harry Potter Reading Room)

Contact: awc.wwu@gmail.com for more info or questions!

See you there!
Goals

• Know how to create, index, slice, and check for membership in lists.

• Understand the behavior of the +, *, in, not in, operators on lists.

• Know how to use the assignment operator on list elements and slices.

• Know how to use the list methods append, and extend.

• Know the definition of mutability, and which sequence types are mutable (lists) and immutable (strings, tuples).

• Understand that Python variables actually hold references to objects.

  • Understand the implications of mutability when multiple variables reference the same mutable object.
Last time

Understand the behavior of the following operators on strings:

- `<`, `>`, `==`, `!=`, `in`, and `not in`
- Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"

"Bellingham"

"Bellevue"
Last time

Understand the behavior of the following operators on strings:

- `<`, `>`, `==`, `!=`, `in`, and `not in`
- Understand how Python orders strings using **lexicographic ordering**:

Example: "Bellingham" > "Bellevue"

"Bellingham"
"Bellevue"
Last time

Understand the behavior of the following operators on strings:

- `<`, `>`, `==`, `!=`, `in`, and `not in`
- Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"
Last time

Understand the behavior of the following operators on strings:

- `<`, `>`, `==`, `!=`, `in`, and `not in`
- Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"

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

Understand the behavior of the following operators on strings:

- `<`, `>`, `==`, `!=`, `in`, and `not in`
- Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"

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

Reminder: character ordering is based on `ord` function:
Last time

Understand the behavior of the following operators on strings:

- <, >, ==, !=, in, and not in
- Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"

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

Reminder: character ordering is based on ord function:
ord("a") => 97, ord("b") => 98, ...
Last time

Understand the behavior of the following operators on strings:

• `<`, `>`, `==`, `!=`, `in`, and `not in`

• Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"

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

Reminder: character ordering is based on `ord` function:

`ord("a")` => 97, `ord("b")` => 98, ...
`ord("A")` => 65, `ord("B")` => 66, ...
Last time

Understand the behavior of the following operators on strings:

- <, >, ==, !==, in, and not in
- Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"

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

Reminder: character ordering is based on ord function:

ord("a") => 97, ord("b") => 98, ...
ord("A") => 65, ord("B") => 66, ...
Last time

Understand the behavior of the following operators on strings:

- <, >, ==, !=, in, and not in
- Understand how Python orders strings using lexicographic ordering:

Example: "Bellingham" > "Bellevue"

Reminder: character ordering is based on ord function:

\[
\begin{align*}
\text{ord("a")} & \Rightarrow 97, \quad \text{ord("b")} \Rightarrow 98, \ldots \\
\text{ord("A")} & \Rightarrow 65, \quad \text{ord("B")} \Rightarrow 66, \ldots
\end{align*}
\]

All upper-case letters come before all lower-case letters.
Last time

• Know how to create, index, slice, and check for membership in lists.

• Understand the behavior of the +, *, in, not in, operators on lists.

more on this today
Today’s Quiz

• 3 minutes
Today’s Quiz

• 3 minutes

• Working with a neighbor: do your answers agree? (2 minutes)
**Lists: Yet Another Sequence Type**

A **list** is an object that contains a sequence of values. We've seen them before.

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values. We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values. We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Syntax:

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A **list** is an object that contains a sequence of values. We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

**Syntax:**

```python
[val0, val1, val2, val3]
```

Values can be of any type(s)!
Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values. We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```
[val0, val1, val2, val3]
```

comma-separated list of values

Values can be of any type(s)!

Lists: Yet Another Sequence Type

A list is an object that contains a sequence of values.
We've seen them before.

```python
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```
[val0, val1, val2, val3]
```

- comma-separated list of values
- surrounded by square brackets

Values can be of any type(s)!
What can we do with Lists?

A lot of this should look familiar.

These things work analogously to strings:

• Indexing
• Slicing
• The len function
• in and not in operators
• + and * operators
What can we do with Lists?

A lot of this should look familiar.

```python
a_list = ["Scott", 34, 27.7]
```

These things work analogously to strings:
- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators
Demo

A lot of this should look familiar.

These things work analogously to strings:
- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators
Demo

A lot of this should look familiar.

```python
a_list = ["Scott", 34, 27.7]
```

These things work analogously to strings:
- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators
Demo

A lot of this should look familiar.

make 'em

index 'em

index 'em

slice 'em
Demo

A lot of this should look familiar.

```python
a_list = ['Scott', 34, 27.7]  # make 'em
a_list[0]  # index 'em
a_list[-1]  # index 'em
a_list[1:]  # slice 'em
```
Demo

A lot of this should look familiar.
Demo

A lot of this should look familiar.

```python
a_list = ["Scott", 34, 27.7]
len(a_list)
len(["abc"])
len([])
34 in a_list
"34" not in a_list
a_list + ["Wehrwein", "WWU"]
["na"] * 16 + ["Batman"]
```
Demo

Lists can contain any type: lists, tuples, turtles, ...
Demo

Lists can contain any type: lists, tuples, turtles, ...

```python
da_list = ["Scott", [34, 27.7, (39, 70)]]
da_list[0]
da_list[1]
da_list[1][2]
da_list[1][2][0]
```
What can go in lists?

- Like tuples, *any* value can go in a list.
  - tuples, lists, Turtles, ... *anything*
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

"Ned" in starks
Lists: Yet Another Sequence Type

```
starks = ["Ned", "Arya", "Bran", "Sansa"]

"Ned" in starks
```
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks

"Sansa" in starks[1:3]
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

"Ned" in starks

"Sansa" in starks[1:3]
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

"Ned" in starks

"Sansa" in starks[1:3]

len(starks[1:4]) == 3
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

✓ "Ned" in starks
✗ "Sansa" in starks[1:3]
✓ len(starks[1:4]) == 3
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

- "Ned" in starks
- "Sansa" in starks[1:3] ✗
- len(starks[1:4]) == 3 ✓
- "Arya" in (starks + ["Jon"])[:2]
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

- "Ned" in starks
- "Sansa" in starks[1:3] ✗
- len(starks[1:4]) == 3 ✓
- "Arya" in (starks + ["Jon"]) [2:] ✗
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

- "Ned" in starks
- "Sansa" in starks[1:3]  ✗
- len(starks[1:4]) == 3 ✓
- "Arya" in (starks + ["Jon"])[2:]  ✗
- len(starks[1:2] * 4) == 8

A

B

False True
Lists: Yet Another Sequence Type

starks = ["Ned", "Arya", "Bran", "Sansa"]

- "Ned" in starks
- "Sansa" in starks[1:3]  // False
- len(starks[1:4]) == 3\n- "Arya" in (starks + ["Jon"])\n- len(starks[1:2] * 4) == 8 // False
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

...haven't we seen this before?
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

...haven't we seen this before?

**Tuples** are also objects that hold a sequence of values of any type(s).
Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

...haven't we seen this before?

**Tuples** are also objects that hold a sequence of values of any type(s).

("alpaca", 14, 27.6)
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).
Lists vs Tuples: What's the difference?

Tuples are also objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents cannot be changed.
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are *immutable*: their contents **cannot** be changed.

**Lists** are *mutable*: their contents **can** be changed.
Lists vs Tuples: What's the difference?

Tuples are also objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents cannot be changed.

Lists are mutable: their contents can be changed.

```python
a_tuple = ("a", 14, 27.6)
```
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.

**Lists** are **mutable**: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)

a_list = ["a", 14, 27.6]
```
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.

**Lists** are **mutable**: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are immutable: their contents cannot be changed.

**Lists** are mutable: their contents can be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents **cannot** be changed.

**Lists** are **mutable**: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
a_list[1]   # => 14

a_tuple[1] = 0  # causes an error
```
Lists vs Tuples: What's the difference?

Tuples are also objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents cannot be changed.

Lists are mutable: their contents can be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14

a_tuple[1] = 0 # causes an error
a_list[1] = 0 # a_list is now ["a", 0, 27.6]
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are **immutable**: their contents cannot be changed.

**Lists** are **mutable**: their contents can be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14

a_tuple[1] = 0 # causes an error
a_list[1] = 0 # a_list is now ["a", 0, 27.6]
```
Lists vs Tuples: What's the difference?

**Tuples** are *also* objects that hold a sequence of values of any type(s).

**Tuples** are immutable: their contents **cannot** be changed.

**Lists** are mutable: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
a_list[1]   # => 14

a_tuple[1] = 0  # causes an error
a_list[1] = 0   # a_list is now ["a", 0, 27.6]
```
Lists vs Tuples: What's the difference?

**Tuples** are also objects that hold a sequence of values of any type(s).

**Tuples** are immutable: their contents **cannot** be changed.

**Lists** are mutable: their contents **can** be changed.

```python
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1]  # => 14
a_list[1]   # => 14

a_tuple[1] = 0  # causes an error
a_list[1] = 0   # a_list is now ["a", 0, 27.6]
```
Lists are mutable

a_list = ["a", 14, 27.6]
Lists are mutable

```python
a_list = ["a", 14, 27.6]
a_list[0] = "b"
```

```
a_list  ➝ ["b", 14, 27.6]
```
Lists are mutable

```python
a_list = ["a", 14, 27.6]

a_list[0] = "b"

a_list.append(19)

a_list
```

a_list ➞ ["b", 14, 27.6, 19]
Lists are mutable

```python
a_list = ["a", 14, 27.6]
a_list[0] = "b"
a_list.append(19)
a_list.append(["12", 2])
```

```
["b", 14, 27.6, 19, ["12", 2]]
```
Lists are mutable

```python
a_list = ['a', 14, 27.6]

a_list[0] = 'b'

a_list.append(19)
a_list.append(['12', 2])
a_list.extend([22, 33])
```

`a_list` → ['b', 14, 27.6, 19, ['12', 2], 22, 23]
Lists are mutable

Notice the difference between string methods and list methods:

```
a_list.append(19)
a_list  
```

```
new_string = a_string.lower()
a_string  
```

```python
[
"b"
]
"JON"
```
Lists are mutable

Notice the difference between string methods and list methods:

```python
a_list.append(19)  
• **modifies** the list in-place  
• has **no** return value

new_string = a_string.lower()  
```

```
a_list ➞ ["b", 19]  
a_string ➞ "JON"
```

```python
new_string = a_string.lower()  
```
Lists are mutable

Notice the difference between string methods and list methods:

```python
a_list.append(19)  # a_list → ["b", 19]

• modifies the list in-place
• has no return value
```

```python
new_string = a_string.lower()

• does not modify a_string  # a_string → "JON"
• returns a lower-case copy  # new_string → "jon"
```
Demo: a *bale* of turtles

- bale.py
Objects and Variables: Digging a little deeper

When we talked about variables...

All variables store references to objects. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

All variables store references to objects. Objects can be any type (that's why a variable can have any type):
Objects and Variables: 
Digging a little deeper

When we talked about variables...

I lied and told you:

number 2

All variables store references to objects. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

All variables store **references** to **objects**. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

what's actually happening:

All variables store references to objects. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

what's actually happening: number

All variables store references to objects. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

what's actually happening: number

All variables store references to objects. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

number 2

what's actually happening: number

All variables store **references** to **objects**. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

All variables store *references* to *objects*. Objects can be any type (that's why a variable can have any type):
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

```
number 2
```

what's actually happening:

```
number
```

All variables store references to objects. Objects can be any type (that's why a variable can have any type):

```
int 4
```
```
int 2
```

After `number = 4` is executed, `number` points at a different object.
Objects and Variables: Digging a little deeper

When we talked about variables...

I lied and told you:

```
number = 2
```

what's actually happening:

```
number
```

All variables store references to objects. Objects can be any type (that's why a variable can have any type):

```
int 4
```

After `number = 4` is executed,

```
number
```

points at a different object.

For immutable objects, we don't have to think about this much.
Objects and Variables: Digging a little deeper

On paper exercise (not collected)
Execute the following, drawing and updating the memory diagram for each variable and object involved.

```python
number = 2
number = 4
another_number = number
another_number += 1
```

```
int
2
number = 4
```
Objects and Variables: Digging a little deeper

On paper exercise (not collected)
**Execute the following**, drawing and updating the memory diagram for each variable and object involved.

```java
number = 2
number = 4
another_number = number
another_number += 1
number = 4
```

![Memory Diagram](image)
Objects and Variables: Digging a little deeper

Now let's talk about lists:
• each element is like its own variable

```python
weather = [63, "light rain", 8, "SSW", 29.75]
```
Now let's talk about lists:

- each element is like its own variable

```python
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```

```markdown
<table>
<thead>
<tr>
<th></th>
<th>list</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>1</td>
<td>int</td>
<td>str</td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td></td>
<td>8</td>
<td>&quot;SSW&quot;</td>
<td>29.75</td>
</tr>
<tr>
<td></td>
<td>&quot;light rain&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Objects and Variables: Digging a little deeper

Now let's talk about lists:
- each element is like its own variable

```python
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```
Objects and Variables: Digging a little deeper

Now let's talk about lists:
• each element is like its own variable

```python
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```
Objects and Variables: Digging a little deeper

Now let's talk about lists:
• each element is like its own variable

```python
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```
Implications of Mutability

```python
weather = [63, "light rain"]
tomorrow_weather = weather
tomorrow_weather[0] = 68
print(weather[0])
```

**ABCD:** What does the above code print?

A. "light rain"
B. Error
C. 63
D. 68
Implications of Mutability

```python
weather = [63, "light rain"]
tomorrow_weather = weather
tomorrow_weather[0] = 68
print(weather[0])
```
Implications of Mutability

```python
weather = [63, "light rain"]
tomorrow_weather = weather
tomorrow_weather[0] = 68
print(weather[0])
```
Implications of Mutability

```python
weather = [63, "light rain"]
tomorrow_weather = weather
tomorrow_weather[0] = 68
print(weather[0])
```

More than one variable can refer to the same object. Changing that object via one variable affects the other, because it's the same object!
Implications of Mutability

```python
weather = [63, "light rain"]
tomorrow_weather = weather
tomorrow_weather[0] = 68
print(weather[0])
```

More than one variable can refer to the same object. Changing that object via one variable affects the other, because it's the same object!

To create a true copy of a mutable object, you can't simply assign a new variable to the object.
Exercise

Write a function that returns a true copy (i.e., a different object that has the same values).

```python
def copy_list(in_list):
    """ Return a new list object containing the same elements as in_list. """
```
Exercise

Write a function that returns a true copy (i.e., a different object that has the same values).

```python
def copy_list(in_list):
    """ Return a new list object containing the same elements as in_list. """
```

Hint: one possible approach uses a loop and the `append` method.