CSCI 141

Lecture 21
Mutability; Variables are References: Implications
Intro to Dictionaries
Announcements
Announcements

- A5 Code out this afternoon. Get started early!
Announcements

- A5 Code out this afternoon. Get started early!
  - It's worth 100 points.
Announcements

• A5 Code out this afternoon. Get started early!
  • It's worth 100 points.

• A5 Written out Soon™
Announcements

• A5 Code out this afternoon. Get started early!
  • It's worth 100 points.

• A5 Written out Soon™
  • It'll be worth ~20-30 points, and is primarily intended to help you identify things you need to study for the final exam.
Announcements

• A5 Code out this afternoon. Get started early!
  • It's worth 100 points.

• A5 Written out Soon™
  • It'll be worth ~20-30 points, and is primarily intended to help you identify things you need to study for the final exam.

• No class Monday
Announcements

• A5 Code out this afternoon. Get started early!
  • It's worth 100 points.

• A5 Written out Soon™
  • It'll be worth ~20-30 points, and is primarily intended to help you identify things you need to study for the final exam.

• No class Monday
  • No lab next week
Announcements

• A5 Code out this afternoon. Get started early!
  • It's worth 100 points.

• A5 Written out Soon™
  • It'll be worth ~20-30 points, and is primarily intended to help you identify things you need to study for the final exam.

• No class Monday
  • No lab next week
  • I will move my Monday office hours to Tuesday - time TBA
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!
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!

Reminder
Just in: there will be ice cream and cookies
Goals

• Understand the implications of variables holding references to mutable objects:
  • multiple variables can refer to the same object
  • function parameters can refer to objects that are also referred to by global variables

• Know how to modify lists using the following:
  • indexed assignment, slice assignment, insert, remove, del

• Know the basics of how to use dictionaries (dicts):
  • Creation, assignment, indexing
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.
- Know how to use the list methods `append`, and `extend`.

```python
a = ["Tony", "Steve", "Natasha", "T'Challa", "Carol"]
```

What is the value of: `a[2:3]`?

A. ["Steve", "Natasha"]
B. ["Natasha", "T'Challa"]
C. ["Steve"]
D. ["Natasha"]]
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.
- Know how to use the list methods append, and extend

```python
a = ['Tony', 'Steve', 'Natasha', 'T'Challa', 'Carol']
a.append(['Bruce', 'Peter'])
```

What is the value of: `len(a)`?

- A. 5
- B. 6
- C. 7
- D. 8
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.
- Know how to use the list methods `append`, and `extend`

```python
a = ["Tony", "Steve", "Natasha", "T'Challa", "Carol"]
a.extend(["Bruce", "Peter"])
```

What is the value of: `len(a)`?

A. 5
B. 6
C. 7
D. 8
List assignment + slicing
List assignment + slicing

We can assign to indices:
List assignment + slicing

We can **assign** to indices:

```python
a = [5, 6, 7, 8]
a[0] = 10
```
List assignment + slicing

We can **assign** to indices:

\[
a = [5, 6, 7, 8]
a[0] = 10
\]

We can **slice** out sublists:
List assignment + slicing

We can **assign** to indices:

\[
\begin{align*}
 a &= [5, 6, 7, 8] \\
 a[0] &= 10
\end{align*}
\]

We can **slice** out sublists:

\[
\begin{align*}
 a[0:3] \# &= [5, 6]
\end{align*}
\]
List assignment + slicing

We can **assign** to indices:

\[
a = [5, 6, 7, 8]
a[0] = 10
\]

We can **slice** out sublists:

\[
a[0:3] \quad \# \quad \Rightarrow \quad [5, 6]
\]

Can we **assign** to slices?
List assignment + slicing

We can **assign** to indices:

```python
a = [5, 6, 7, 8]
a[0] = 10
```

We can **slice** out sublists:

```python
a[0:3] # => [5, 6]
```

Can we **assign** to **slices**?

**You betcha!** (demo)
List assignment + slicing

slice_assign.py
Last time

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

**String and Tuples** are immutable  
**Lists** are mutable

```plaintext
a_string = "Scott"  
a_tuple = ("a", 14, 27.6)  
a_list = ["a", 14, 27.6]

a_string[1]  # => "c"
```  
```plaintext
a_tuple[1]  # => 14  
a_list[1]  # => 14
```

```plaintext
a_string[1] = "C"  # causes an error  
a_tuple[1] = 0  # causes an error  
a_list[1] = 0  # a_list is now ["a", 0, 27.6]
```
Today’s Quiz

• 5 minutes - collaborate at will!
All variables store references to objects
All variables store references to objects

number = 2
All variables store references to objects

number = 2
What's actually happening:
All variables store references to objects

number = 2

What's actually happening:
All variables store references to objects

number = 2
What's actually happening:

```
number
```

```
int
2
```
All variables store references to objects

```
number = 2
What's actually happening: number
```

![Diagram of variable and object relationship]
All variables store references to objects

number = 2
What's actually happening: number

number = 4
All variables store references to objects

number = 2
What's actually happening: number

number = 4

```
number
```

```
2
```

```
int
4
```
All variables store references to objects

number = 2

What's actually happening: number

number = 4

int 2

int 4
All variables store references to objects

number = 2
What's actually happening: number
number = 4

For immutable objects, we don't have to think about this much.
All variables store references to objects

number = 2
What's actually happening: number

number = 4

Aside: What happens to the 2 object?

For immutable objects, we don't have to think about this much.
All variables store references to objects

number = 2
What's actually happening: number

number = 4

Aside: What happens to the 2 object?
• If no variables refer to it, Python deletes it automatically.

For immutable objects, we don't have to think about this much.
All variables store references to objects

number = 2

What's actually happening:

number

number = 4

Aside: What happens to the 2 object?
- If no variables refer to it, Python deletes it automatically.
- This is called garbage collection.

For immutable objects, we don't have to think about this much.
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

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

After creating the initial list:

On the board: how does this picture change as the code is executed?
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.
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.

Changes to an object via one variable are reflected when accessing it via another variable!
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.

Changes to an object via one variable are reflected when accessing it via another variable!

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

\[
a = [1, 2, 3]
b = a
\]

you did not just create a copy of a
Don't make this mistake

```
a = [1, 2, 3]
b = a
```

you **did not** just create a copy of `a`

To create a true copy of a `mutable` object, you **can't** simply assign a new variable to the object.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a *reference* to the list:

```python
def z1(a_list):
a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```

*a_list* points to the *same* list as the global variable *a*.
When you pass a list into a function, you're actually passing a reference to the list:

```python
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

The local variable `a_list` is reassigned to point to a new (different) list.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

The local variable `a_list` is reassigned to point to a new (different) list.

The list referenced by `a` is unchanged.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z3(x):
    a_list = [x, x, x]
    return a_list
b = 2
a = z3(b)
print(a)
```
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z3(x):
    a_list = [x, x, x]
    return a_list
```

```python
b = 2
a = z3(b)
print(a)
```

The function creates a new list, with the local variable `a_list` referring to it.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z3(x):
    a_list = [x, x, x]
    return a_list

b = 2
a = z3(b)
print(a)
```

The function creates a new list, with the local variable `a_list` referring to it.

The reference to the list is returned and assigned to `a`. 
Exercise

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

```python
def copy_list(in_list):
    """ Return a new list object containing the same elements as in_list. Precondition: in_list's contents are all immutable. """
```
Exercise

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

```python
def copy_list(in_list):
    """ Return a new list object containing the same elements as in_list.
    Precondition: in_list's contents are all immutable. """
```

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

• Lists, tuples, strings are all **sequences** (their contents are ordered)

• Python also has some types that handle non-sequential collections, including dictionaries (type `dict`):

  • A **dictionary** is an unordered collection of **key-value mappings**
Another way to think about **lists**: 

A *list* is a **mapping** from *integer indices* to *arbitrary values*. 
Another way to think about lists:

A list is a mapping from integer indices to arbitrary values.
Dictionaries

Another way to think about lists: A list is a mapping from integer indices to arbitrary values.

Example:

\[
[ "B", "A", 7 ]
\]
Another way to think about **lists**: A list is a **mapping** from *integer indices* to *arbitrary values*.

**Example:**

```
[ "B", "A", 7 ]
```

represents the following **mapping**:
Dictionaries

Another way to think about lists:

A list is a mapping from integer indices to arbitrary values.

Example:

```
[ "B", "A", 7 ]
```

represents the following mapping:

```
0: "B"
1: "A"
2: 7
```
Another way to think about lists:

A list is a mapping from integer indices to arbitrary values.

Example:

```
[ "B", "A", 7 ]
```

represents the following mapping:

```
0: "B"
1: "A"
2: 7
```

the index 0 maps to the value "B"
Dictionaries

Another way to think about lists:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

**Example:**

\[
[ "B", "A", 7 ]
\]

represents the following **mapping**:

\[
0: "B"
1: "A"
2: 7
\]

the index 0 maps to the value "B".

A **dictionary** is a **mapping**

from *arbitrary immutable keys*

to *arbitrary values*. 
Another way to think about **lists**: A list is a **mapping** from **integer indices** to **arbitrary values**.

**Example:**

\[ \text{["B", "A", 7]} \]

represents the following **mapping**:

\[
\begin{align*}
0: & \text{"B"} \\
1: & \text{"A"} \\
2: & 7
\end{align*}
\]

the index 0 maps to the value "B"

\{
"B": 6,  
"A": 7
\}

A **dictionary** is a **mapping** from **arbitrary immutable keys** to **arbitrary values**.
Dictionaries

Another way to think about lists:

A list is a mapping from integer indices to arbitrary values.

Example:

\[
[ "B", "A", 7 ]
\]

represents the following mapping:

0: "B"
1: "A"
2: 7

the index 0 maps to the value "B"

A dictionary is a mapping from arbitrary immutable keys to arbitrary values.

Example:

\{
"B": 6,
"A": 7
\}

represents the following mapping:
Dictionaries

Another way to think about lists:

A list is a mapping from integer indices to arbitrary values.

Example:

\[[ "B", "A", 7 \] 

represents the following mapping:

0: "B"  
1: "A"  
2: 7

the index 0 maps to the value "B"

A dictionary is a mapping from arbitrary immutable keys to arbitrary values.

Example:

\{"B": 6, "A": 7 \}

represents the following mapping:

"B": 6  
"A": 7
Dictionaries

Another way to think about **lists**: A list is a **mapping** from **integer indices** to **arbitrary values**.

A **dictionary** is a **mapping** from **arbitrary immutable keys** to **arbitrary values**.

**Example:**

```
[ "B", "A", 7 ]
```

represents the following mapping:

```
0: "B"
1: "A"
2: 7
```

the index 0 maps to the value "B"

```
{"B": 6, "A": 7}
```

represents the following mapping:

```
"B": 6
"A": 7
```

the key B maps to the value 6
Dictionaries

Why do we want this?

Suppose I want to store...

```python
english = {}
english["aardvark"] = """"a nocturnal burrowing mammal with long ears, a tubular snout, and a long extensible tongue, feeding on ants and termites. Aardvarks are native to Africa and have no close relatives."""
```
Dictionaries

Why do we want this?

Suppose I want to store...

A list of W#s of all the students in each of the lab sections.

```python
sections = {}
sections[20891] = ["W0183782", "W0243810", # ...
sections[20892] = ["W0184582", "W0182368", # ...
# ...
```
Dictionaries

Why do we want this?

Suppose I want to store...

A bunch of different information about a WWU employee:

```
employee = {
    "First": "Scott",
    "Last": "Wehrwein",
    "Type": "Faculty",
    "W#": 98765438,
    # ...
```
Dictionaries

Why do we want this?

Suppose I want to store...

The number of students with each letter grade in my class:

```python
grade_counts = {"A": 6, "B": 12, "C": 8, "D": 2}
```
Dictionaries: Let's play

- Creation
- Indexing
- Assignment
Dictionaries: Let's play

- Creation
- Indexing
- Assignment

```python
grades = {"A": 10, "B": 18, "C": 6, "D": 2}
```