Lecture 22
Variables are References
Mutability's Implications
Announcements
Announcements

• A5 is out! Due in 2 weeks (after Thanksgiving break).
Announcements

- A5 is out! Due in 2 weeks (after Thanksgiving break).
  - Suggestion: get it mostly done this week.
Announcements

• A5 is out! Due in 2 weeks (after Thanksgiving break).
  • Suggestion: get it mostly done this week.

• Next week is Thanksgiving week
Announcements

• A5 is out! Due in 2 weeks (after Thanksgiving break).
  • Suggestion: get it mostly done this week.

• Next week is Thanksgiving week
  • No labs
Announcements

• A5 is out! Due in 2 weeks (after Thanksgiving break).
  • Suggestion: get it mostly done this week.

• Next week is Thanksgiving week
  • No labs
  • Class Monday
Announcements

• A5 is out! Due in 2 weeks (after Thanksgiving break).
  • Suggestion: get it mostly done this week.

• Next week is Thanksgiving week
  • No labs
  • Class Monday
  • No class Wednesday or Friday
Goals

• Understand what you're being asked to do in A5.

• Understand the implications of variables holding references to mutable objects:
  • Multiple variables can refer to the same object.

• Be able to draw memory diagrams for code snippets involving mutable objects.

• Know how to query or modify lists using the following: index, insert, remove, del
This is your machine learning system?

Yup! You pour the data into this big pile of linear algebra, then collect the answers on the other side.

What if the answers are wrong?

Just stir the pile until they start looking right.
A5: Machine Learning!
A5: Machine Learning!

\[
E[x] = \frac{1}{(2\pi)^{D/2}} \left| \Sigma \right|^{1/2} \int \exp \left\{ -\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right\} x \, dx
\]

\[
= \frac{1}{(2\pi)^{D/2}} \left| \Sigma \right|^{1/2} \int \exp \left\{ -\frac{1}{2} z^T \Sigma^{-1} z \right\} (z + \mu) \, dz \tag{2.58}
\]
okay but it's not actually that crazy

Let's talk about creatures.

Some creatures are monsters.
Some creatures are not monsters.

You can't always tell by looking at them.
okay but it's not actually that crazy

Let's talk about *creatures*.

Some creatures are monsters.
Some creatures are not monsters.

You can't always tell by looking at them.

Problem setup: we have a dataset of known monsters and non-monsters. and we want to look at their attributes to figure out how to decide whether a new, never-before-seen creature is a monster.
## Known Creatures:

<table>
<thead>
<tr>
<th>Size</th>
<th>Toothiness</th>
<th>Monster?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Y</td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>104</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>112</td>
<td>Y</td>
</tr>
</tbody>
</table>

## Unknown creature:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>22</td>
<td>?</td>
</tr>
</tbody>
</table>
A scheme for guessing whether an unseen creatures is a monster:

1. Find the average **size** of non-monsters
2. Find the average **size** of monsters
3. Cast a "vote" as follows:

4. Repeat the same procedure for the **toothiness** attribute.
5. Tally the votes and guess majority vote winner.

### Known Creatures:

<table>
<thead>
<tr>
<th>Size</th>
<th>Tooth</th>
<th>Mnstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Y</td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>104</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>112</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Unknown creature:**

| 4 | 22 | ? |
A scheme for guessing whether an unseen creature is a monster:

1. Find the average **size** of non-monsters

2. Find the average **size** of monsters

3. Cast a "vote" as follows:

4. Repeat the same procedure for the **toothiness** attribute.

5. Tally the votes and guess majority vote winner.
A scheme for guessing whether an unseen creature is a monster:

1. Find the average size of non-monsters:
   \[
   \text{size} = 4
   \]

2. Find the average size of monsters:

<table>
<thead>
<tr>
<th>Size</th>
<th>Tooth</th>
<th>Mnstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Y</td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>104</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>112</td>
<td>Y</td>
</tr>
</tbody>
</table>

3. Cast a "vote" as follows:

   Unknown creature:
   | Size | Tooth |?
   |------|-------|---
   | 4    | 22    | ?

4. Repeat the same procedure for the toothiness attribute.

5. Tally the votes and guess majority vote winner.
A scheme for guessing whether an unseen creatures is a monster:

1. Find the average size of non-monsters

   \[ \text{size} = 4 \]

2. Find the average size of monsters

3. Cast a "vote" as follows:

4. Repeat the same procedure for the toothiness attribute.

5. Tally the votes and guess majority vote winner.

Known Creatures:

<table>
<thead>
<tr>
<th>Size</th>
<th>Tooth</th>
<th>Mnstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Y</td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>104</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>112</td>
<td>Y</td>
</tr>
</tbody>
</table>

Unknown creature:

| 4    | 22    | ?     |
A scheme for guessing whether an unseen creature is a monster:

1. Find the average size of non-monsters = 4
2. Find the average size of monsters = 15
3. Cast a "vote" as follows:

<table>
<thead>
<tr>
<th>Size</th>
<th>Tooth</th>
<th>Mnstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Y</td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>104</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>112</td>
<td>Y</td>
</tr>
</tbody>
</table>

Unknown creature:

<table>
<thead>
<tr>
<th>Size</th>
<th>Tooth</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>22</td>
<td>?</td>
</tr>
</tbody>
</table>

4. Repeat the same procedure for the toothiness attribute.
5. Tally the votes and guess majority vote winner.
A scheme for guessing whether an unseen creatures is a monster:

1. Find the average size of non-monsters = 4
2. Find the average size of monsters = 15
3. Cast a "vote" as follows:
   • If the unknown creature's size is closer to the Monster average, vote Monster.

4. Repeat the same procedure for the toothiness attribute.
5. Tally the votes and guess majority vote winner.
A scheme for guessing whether an unseen creatures is a monster:

1. Find the average size of non-monsters
   \[ \text{size} = 4 \]

2. Find the average size of monsters
   \[ \text{size} = 15 \]

3. Cast a "vote" as follows:
   - If the unknown creature's size is closer to the Monster average, vote Monster.
   - If the creature's size is closer to the non-Monster average, vote non-Monster.

4. Repeat the same procedure for the toothiness attribute.
5. Tally the votes and guess majority vote winner.

Known Creatures:

<table>
<thead>
<tr>
<th>Size</th>
<th>Tooth</th>
<th>Mnstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>N</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>Y</td>
</tr>
<tr>
<td>21</td>
<td>84</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>104</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>112</td>
<td>Y</td>
</tr>
</tbody>
</table>

Unknown creature:

4 | 22 | ?
A5

- Creatures --> tumors
- Monster --> malignant
- Non-monster --> benign
- Size and toothiness --> radius, texture, area, ...
  (a total of 10 attributes)
I want to show you something weird.
I want to show you something weird.

- Demo:

```python
a = [4, 5]
b = a
b[0] = 1
print(a[0])
```
Objects and Variables: Digging a little deeper

When we talked about variables...
Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:
Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

```
number 2
```
Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:
Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:
Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:

\[ \text{number} \]
Objects and Variables: Digging a little deeper

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:

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

When we talked about variables...

Sometimes I got lazy and wrote:

but what's truly happening is:

All variables store references to objects.

Objects can have any type
All variables store references to objects

In code:                                       In memory:
All variables store references to objects

In code:    In memory:

number = 2
All variables store references to objects

In code:  

number = 2

In memory:

<table>
<thead>
<tr>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
All variables store references to objects

In code:

```
number = 2
```

In memory:

```
number

<table>
<thead>
<tr>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
```
All variables store references to objects

In code:

number = 2

In memory:

number

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

In code:

number = 2

In memory:

number

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

In code:

```
number = 2
number = 4
```

In memory:

```
number
```

![Diagram showing variable initialization and memory allocation]
All variables store references to objects

In code:

```python
number = 2
```

```python
number = 4
```

In memory:

```
    number
    └── int
        ├── 2
        └── 4
```
All variables store references to objects

In code:

```python
number = 2

number = 4
```

In memory:

```
In memory:
```

Like strings, `ints` are immutable:
You can't change its value.
You can only make a new one with a different value.
All variables store references to objects

In code:

```python
number = 2
number = 4
```

In memory:

```
number
```

```
int
2
```

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

In code:

```
number = 2
```

```
number = 4
```

Aside: What happens to the 2 object?
All variables store references to objects

In code:

```python
define variable:
number = 2
```

In memory:

```
number
2
```

Aside: What happens to the 2 object?

• If no variables refer to it, Python deletes it automatically.

```python
number = 4
```
All variables store references to objects

In code:

```python
number = 2
```

In memory:

```
number
2
```

```python
number = 4
```

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

In code:

```python
number = 2
```

In memory:

```
number
```

```
int
2
```

```python
number = 4
```

```
int
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, the fact that variables hold references doesn't have many interesting consequences.
Execute the following, drawing and updating the memory diagram for each variable and object involved.

```plaintext
number = 2
other_number = number
number += 1
```
Worksheet - Problem 1

**Execute the following**, drawing and updating the memory diagram for each variable and object involved.

```python
number = 2
other_number = number
number += 1
```

(whiteboard)
All variables store references to objects

What about mutable objects?

In code: 

In memory:
All variables store references to objects

What about mutable objects?

In code:  

\[
a = [4, 5]
\]  

In memory:
All variables store references to objects

What about mutable objects?

In code:

```python
a = [4, 5]
```

In memory:

```
<table>
<thead>
<tr>
<th>list</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```
All variables store references to objects

What about *mutable* objects?

In code:

```
a = [4, 5]
```

In memory:
All variables store references to objects

What about **mutable** objects?

In code:

```python
a = [4, 5]
b = a
```

In memory:
All variables store references to objects

What about mutable objects?

In code:

```python
a = [4, 5]
b = a
```

In memory:

```
<table>
<thead>
<tr>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>b</th>
</tr>
</thead>
</table>
```
All variables store references to objects

What about *mutable* objects?

In code:

```python
a = [4, 5]
b = a
```

In memory:

![Diagram showing that `a` and `b` point to the same list object.]

The value of `a` is a *reference* to that list object, so the new value of `b` is also a *reference* to that *same* list!
All variables store references to objects

What about mutable objects?

In code:

```python
a = [4, 5]
b = a
```

In memory:

```
<table>
<thead>
<tr>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
</tr>
<tr>
<td>4 5</td>
</tr>
</tbody>
</table>
```

```rust
[1, 5] # !!!
```
All variables store references to objects

What about **mutable** objects?

In code:

```python
a = [4, 5]
b = a
b[0] = 1
```

In memory:

```
<table>
<thead>
<tr>
<th>list</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
```

```bash
[1, 5] # !!!
```
All variables store references to objects

What about **mutable** objects?

In code:

```python
a = [4, 5]
b = a
b[0] = 1
```

In memory:

```
<table>
<thead>
<tr>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

```

[1, 5] # !!!
All variables store references to objects

What about mutable objects?

In code:

```python
a = [4, 5]
b = a
b[0] = 1
print(a)
[1, 5] #！！!
```
All variables store references to objects

What about **mutable** objects?

In code:

```python
a = [4, 5]
b = a
b[0] = 1
print(a)  # [1, 5] # !!!
```

In memory:

```
list
1 5
```

More than one variable can refer to the **same object**.
Don't make this mistake

\[
\begin{align*}
a & = [1, 2, 3] \\
b & = a
\end{align*}
\]

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

\[
\begin{align*}
a &= [1, 2, 3] \\
b &= a
\end{align*}
\]

you **did not** just create a copy of \(a\)

To create a true copy of a **mutable** object, you **can't** simply assign the object to a new variable.
List elements store references to objects

List elements are just like variables!

In code:

```python
a = [4, 5]
```

In memory:

```
a = list
0 1
1 5
```

I lied to you again!
List elements store references to objects

List elements are just like variables!

In code:

\[
a = [4, 5]
\]

In memory (the true picture):
List elements store references to objects

weather = [63, "light rain", 8, "SSW", 29.75]
List elements store references to objects

```python
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```
List elements store references to objects

```python
weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
```
List elements store references to objects

weather = [63, "light rain", 8, "SSW", 29.75]
weather[1] = "cloudy"
List elements store references to objects

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

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?
Creating lists vs Creating references

• A list literal creates a new list
  
  \[ a = [4, 5, 6] \]

• List assignment does not create a new list
  
  \[ b = a \]

• List concatenation creates a new list
  
  \[ c = a + b \]

• List slicing creates a new list
  
  \[ d = a[::1] \]
A few more list operations:
A few more list operations:

```python
my_list.index(value)
```

Return the index of the first occurrence of `value` in `my_list`

Throw an error if `value` is not in `my_list`. 
A few more list operations:

my_list.index(value)
Return the index of the first occurrence of value in my_list
Throw an error if value is not in my_list.

my_list.insert(index, value)
Inserts value into my_list at index, shifting all following elements one spot to the right.
A few more list operations:

my_list.index(value)
Return the index of the first occurrence of value in my_list
Throw an error if value is not in my_list.

my_list.insert(index, value)
Inserts value into my_list at index, shifting all following elements one spot to the right.

my_list.remove(value)
Removes the first item from the list whose value is equal to value.
Causes an error if value is not in my_list.
A few more list operations:

my_list.index(value)
Return the index of the first occurrence of value in my_list
Throw an error if value is not in my_list.

my_list.insert(index, value)
Inserts value into my_list at index, shifting all following elements one
spot to the right.

my_list.remove(value)
Removes the first item from the list whose value is equal to value.
Causes an error if value is not in my_list.

delete my_list[index]
Removes the element at index, shifting all following elements one spot
to the left.
index, insert, remove, del: Demo

```
abc = [ "B", "C" ]
abc.index("C")
abc.index("F")
abc.insert(0, "A")
abc.remove("C")
abc.remove("F")
```
Execute the following, drawing and updating the memory diagram for each variable and object involved.

```python
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
```
Worksheet - Problem 2

```python
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
print(a)
```

What does this print?
Worksheet - Problem 2

```python
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
print(a)
```

What does this print?

A. [1, 4]
B. [4, 4]
C. [[1], [4]]
D. [[4], [4]]
Problem 3

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

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

```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.
def snap(avengers):
    """ Remove a randomly chosen half of the elements from the given list of avengers """