Variables are References

Mutability's Implications
Goals

• 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
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...

Sometimes I got lazy and wrote:

```
number 2
```

but what's truly happening is:

```
number
```

All variables store references to objects.

```
int
2
```

Objects can have any type.
All variables store references to objects

In code:

```
number = 2
```

In memory:

```
number
```

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

In code:

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

In memory:

```
number
```

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:

```
number = 2
```

In memory:

```
number
```

```
2
```

```
4
```

```
int
```

```
int
```

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, the fact that variables hold references doesn't have many interesting consequences.
Example

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

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

![Memory diagram](image)
All variables store references to objects

What about mutable objects?

In code:

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

In memory:

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

In memory:

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

```python
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 the object to a new variable.
List elements store references to objects

List elements are just like variables!

In code:

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

In memory:

```
a

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

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

List elements are just like variables!

In code:

\[
\text{a} = [4, 5]
\]

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

```python
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

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

Draw and update the memory diagram as the following code is executed.

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

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.

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

```python
abc = ["B", "C"]
abc.index("C")
abc.index("F")
abc.insert(0, "A")
abc.remove("C")
abc.remove("F")
del abc[0]
```
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. """
```

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 """