WHAT YOU SHOULD KNOW ABOUT

IMMUTABILITY VERSUS Mutability

Python has **mutable** and **immutable** objects.

- **Mutable objects** have methods that can modify the object itself.
  - Ex: `list`, `set`, `dictionary`.
- You can't change 'em after you make 'em.

**But what exactly does this mean??**

**WHY??**

I thought it was gonna add 1 to `a` and print 6.

WHAA?? BUT, BUT

**EX.** Here's a confusing example:

```python
>>> def add_one(num):
    num = num + 1
>>> a = 5
>>> add_one(a)
>>> print(a)
5 # WHAT THE -- ??!
```

**HERE'S WHAT'S HAPPENING:**

When you declare variables in Python, like:

```
a = 5
b = [1, 1, 0]
c = b
```

Three things happen:

1. Go into Python's memory and create a list with 1, 1, and 0.
2. Create a variable/label, "b"
3. Have "b" point to the object we just created.

```
b = [1, 1, 0]
```

Variables are like sticky node labels!
Going back to our confusing example:

```
Label          object
a   \rightarrow\ object\ Int: 5
num   \rightarrow\ object\ Int: 5
```

when we write `add_one(a),` Python creates a new variable “num” and has it point to what a points at.

**BUT WAIT!!!** In Python, the equals operator *can’t change objects* — it **reassigns variables** (changes what they point to) instead!!!

So after `num = num + 1`, this is what our depictions of the program look like:

```
Label          object
a   \rightarrow\ object\ Int: 5
num   \rightarrow\ object\ Int: 6
```

So when we print “a”, we end up with 5 instead of 6.  // END EXAMPLE

---

So how do you get the difference between immutability and mutability?

**Mutable objects** can only change their contents using certain methods (which you can look up in the Python documentation).

Ex. list objects have the `extend`, `append`, `insert`, `remove`, and `pop` methods.

**Immutable objects** don’t have any such nice methods that’ll reach in and do some surgery for us. They’re immutable — UNCHANGEABLE. Set for life.

So you can give up on changing the “value” of a variable that points to an immutable object (like an `int`). No matter what you do, the object’s value won’t change. You’re only solution is to reassign the variable to a new `int`.

```
blah = 4   # this'll never ever change its value
```

```
blah = 100 # you have to reassign it
```

<table>
<thead>
<tr>
<th>var</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>blah</td>
<td>4</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>var</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>blah</td>
<td>4</td>
</tr>
</tbody>
</table>
So let's do one more example before looking at that reverse list problem from HW5.

```
>>> def add_one(container):
    container.append(1)

>>> a = [5]
>>> add_one(a)
>>> print(a)
[5, 1]
```

// END EXAMPLE

OK, NOW LET'S LOOK AT HW5’S reverse_list problem.

COMMON PITFALL

```
>>> def rev(s):
    s = s[::-1]
    return none

>>> corazon = [1, 2, 3]
>>> rev(corozen)
>>> corazon
[1, 2, 3] # WHAT THE -- ??
```

You also just-assigned s to a completely new list. We online Python tutor to help you

while it's true that a list is a
mutable data type, thus won't work!!!

Remember what we started earlier?

YOU CAN ONLY CHANGE MUTABLE OBJECTS THROUGH CERTAIN METHODS!!!

In fact, typing

```
>>> id(corozen)
# Some location in memory
```

```
>>> id(corrozen[2])
# diff location
```

will show you

SO, GO CHECK YOUR SURGEON'S BAG FOR STUFF YOU CAN USE!!!

that

slicing a list actually make a copy of a list.

DID YOU SEE IT ??

WELL, DID YOU ??

NOW WRITE YOUR NEW SOLUTION
AND CHECK YOUR ANSWER
WITH THE NEXT PAGE !!!!
YOUR NEW, BEAUTIFUL SOLUTION DEMONSTRATING YOUR KNOWLEDGE OF MUTABLE & IMMU TABLE OBJECTS
SHOULD LOOK LIKE THIS:

```python
def reverse(s):
    s.reverse()  # you used the built-in method list.gape you
    return None
```

Ahah. So concise.

So hopefully, you now have a better idea of mutable vs. immutable objects
and how you can use that knowledge in your assignment.

Good luck!!