# Three Python Nuances

## I wish I'd known earlier

@thejessleigh | #chipy

```
l = [[0] * 5] * 5

print(l)

[[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```



Over advent of code a friend of mine was taking the opprotunity to learn Python and there was a problem which required creating a large matrix and then
changing and manipulating specific coordinates in that matrix. This is how they originally set it up, and to test, they wanted to assign the element at index 3 in the first list to 1. Simple enough, right?

```
l[0][3] = 1

print(l)

[[0, 0, 0, 1, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 1, 0],
[0, 0, 0, 1, 0]]
```

😟

@thejessleigh | #chipy

They posted in our discord group and wondered where they went wrong. My typica answer for things like this is to shrug and blindly assert that a list comprehension will solve things.

^ Which is an accurate but unhelpful answer. Another friend of ours who used to do more Python but hasn't much in years suggested that there's probably something wonky happening with pointers, but wasn't able to help much either.

# Nuance Number One

Lists don't contain objects

They contain references to objects

And it's hella weird

@thejessleigh | #chipy

Turns out my second friend was right. Lists don't actually contain the objects that you think they contain.

They contain references to a specific location in memory.

That specific location contains information, and if you change the object stored at that location in memory, you change every instance where it appears.

So because each of the five arrays inside that matrix are actually just five separate references to the *same* array, and when you change one, you change them all.

# Let's try again

```
a = [0, 0, 0, 0, 0]

b = []

for _ in range(5):
    b.append(copy(a))
```



In order to accomplish the goal of having a matrix where each element can be manipulated independently without inadvertently affecting the whole,
we need to make sure that we're creating a new object at a new location in memory for each nested list. In this case, we can use the copy library
to create a new container object that we can change independently.

# Clean it up

## with list comprehensions

```
l = [[0 for i in range(5)] for j in range(5)]
print(l)
```

```
[[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```



It's a little ugly to use copy in a range like that. So like I said to my friend when they were doing the advent of code problem, mooooost of your Python problems can be solved wit a well placed list comprehension.

# Nuance Number Two

Default arguments are evaluated when a function is defined

*Not* each time a function is called

# Arbitrary example:

## Let's write a function to append an item to a list

```
def list_append(element, input_list=[]):
    input_list.extend([element])
    return input_list

print(list_append(3))
[3]
print(list_append([5, 7]))
[3, 5, 7]
```

@thejessleigh | #chipy

Now, this is a very dumb function and there is no reason to do this, but for the sake of the example, let's say we wanted to do a roundabout reimplementation of appending an item to a list.

When we call this function as written multiple times, it seems that it's always adding to and returning the same object, rather than starting with a fresh empty list each time it's called.

This is because the default arguments for a function are created and evaluated when the fuction is first declared.
So when you start up your app, or if you restart it, you'll be starting with a fresh list. But each time the function is called while your app is running
it's going to be working off of the same mutable default object, and it will just keep growing until you restart the app.

# Here's how to do it

```
def list_append(element, input_list=None):
    if not input_list:
        input_list = []
    return input_list.extend([element])
print(list_append(3))
[3]
print list_append([5, 7])
[5, 7]
```

@thejessleigh | #chipy

Elements that are declared *inside* a function, as opposed to function defaults, are evaluated when the function runs.

So the solution is to declare the function with a nullable default argument, and if it isn't present when the function is called, declare a fresh, shiny, new empty list within the scope of the function so it isn't bogged down by the history of previous calls.

# Nuance Number Three

## == vs is

@thejessleigh | #chipy

# Equality vs identity: truthy and falsey values

```
print(1 == True)
True
print(1 is True)
False
a = True
print(a is True)
True
print(0 == False)
True
print(0 is False)
False
b = False
print(b is False)
True
```



True checks for exact identity, while the == comparator checks to make sure that the values are equivalent

Some weird cases, like the numbers 1 and 0, evaluate as equal to True and False respectively when using the == comparator

But they don't register as the same value when using the `is` keyword because `is` checks for exact identity, so only a
variable actually assigned to `True` is `True`.

# Equality vs identity: equivalent variables

`a` and `b` variables pointing to the same id

```
a = [1, 2, 3, 4, 5]
b = a
print(b == a)
True
print(b is a)
True
print(id(a), id(b))
4437740104, 4437740104
```

# `a` and `b` are different objects, but the values of their contents are identical

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
print(b == a)
True
print(b is a)
False
print(id(a), id(b))
4437740104, 4442640968
```

@thejessleigh | #chipy

# Thank You!

@thejessleigh | #chipy