List Mutation, Identity, and Nonlocal

Lists

List creation

Creating a list from scratch:

```
a = []
b = [1, 2, 3, 4, 5]
```

Creating a list from existing lists:

```
c = b + [20, 30]
d = c[:]
e = list(c)
```

Non-destructive or destructive?

Try in PythonTutor.

List creation

Creating a list from scratch:

```
a = []
b = [1, 2, 3, 4, 5]
```

Creating a list from existing lists:

```
c = b + [20, 30]
d = c[:]
e = list(c)
```

Non-destructive or destructive?
The operations above are **non-destructive.**

Try in PythonTutor.

List mutation

```
L[2] = 6

L[1:3] = [9, 8]

L[2:4] = []            # Deleting elements

L[1:1] = [2, 3, 4, 5]  # Inserting elements

L[len(L):] = [10, 11]  # Appending

L[0:0] = range(-3, 0)  # Prepending
```

Try in PythonTutor.

Non-destructive or destructive?

List mutation

```
L[2] = 6

L[1:3] = [9, 8]

L[2:4] = []             # Deleting elements

L[1:1] = [2, 3, 4, 5]  # Inserting elements

L[len(L):] = [10, 11]  # Appending

L[0:0] = range(-3, 0)  # Prepending
```

Try in PythonTutor.

Non-destructive or destructive?
All of the operations above are **destructive.**

List methods

`append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

Try in PythonTutor.

`extend()` adds all the elements in one list to a list:

```
s = [2, 3]
t = [5, 6]
s.extend(4)
s.extend(t)
t = 0
```

Try in PythonTutor.

Non-destructive or destructive?

List methods

`append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

Try in PythonTutor.

`extend()` adds all the elements in one list to a list:

```
s = [2, 3]
t = [5, 6]
s.extend(4) # 🚫 Error: 4 is not an iterable!
s.extend(t)
t = 0
```

Try in PythonTutor. (After deleting the bad line)

Non-destructive or destructive?

List methods

`append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

Try in PythonTutor.

`extend()` adds all the elements in one list to a list:

```
s = [2, 3]
t = [5, 6]
s.extend(4) # 🚫 Error: 4 is not an iterable!
s.extend(t)
t = 0
```

Try in PythonTutor. (After deleting the bad line)

Non-destructive or destructive?

`append()` and `extend()` are **destructive.**

List methods

`pop()` removes and returns the last element:

```
s = [2, 3]
t = [5, 6]
t = s.pop()
```

Try in PythonTutor.

`remove()` removes the first element equal to the argument:

```
s = [6, 2, 4, 8, 4]
s.remove(4)
s.remove(9)
```

Try in PythonTutor.

Non-destructive or destructive?

List methods

`pop()` removes and returns the last element:

```
s = [2, 3]
t = [5, 6]
t = s.pop()
```

Try in PythonTutor.

`remove()` removes the first element equal to the argument:

```
s = [6, 2, 4, 8, 4]
s.remove(4)
s.remove(9)
```

Try in PythonTutor.

Non-destructive or destructive?

`pop()` and `remove()` are **destructive.**

Equality and Identity

Equality of contents vs. Identity of objects

Identity: `exp0 is exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to the same object

Equality: `exp0 == exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to objects containing equal values

```
list1 = [1,2,3]
list2 = [1,2,3]
are_equal = list1 == list2
identical = list1 is list2
```

Try in PythonTutor.

Identical objects always have equal values.

Equality of contents vs. Identity of objects

```python
a = ["apples", "bananas"]
b = ["apples", "bananas"]
c = a

if a == b == c:
    print("All equal!")

a[1] = "oranges"

if a is c and a == c:
    print("A and C are equal AND identical!")

if a == b:
    print("A and B are equal!")

if b == c:
    print("B and C are equal!")
```

Try in PythonTutor.

Equality of contents vs. Identity of objects

```python
a = ["apples", "bananas"]
b = ["apples", "bananas"]
c = a

if a == b == c:
    print("All equal!")

a[1] = "oranges"

if a is c and a == c:
    print("A and C are equal AND identical!")

if a == b:
    print("A and B are equal!") # Nope!

if b == c:
    print("B and C are equal!") # Nope!
```

Try in PythonTutor.

Identity and immutables

Try this in your local friendly Python interpreter:

```
a = "orange"
b = "orange"
c = "o" + "range"
print(a is b)
print(a is c)

a = 100
b = 100
print(a is b)
print(a is 10 * 10)
print(a == 10 * 10)

a = 100000000000000000
b = 100000000000000000
print(a is b)
print(100000000000000000 is 100000000000000000)
```

Beware: `is` may not act like you expect for strings/numbers!

Scopes

Names inside local scopes

Does this work?

```
attendees = []

def mark_attendance(name):
    attendees.append(name)
    print("In attendance:", attendees)

mark_attendance("Emily")
mark_attendance("Cristiano")
mark_attendance("Samantha")
```

Does this work?

```
current = 0

def count():
    current = current + 1
    print("Count:", current)

count()
count()
```

Names inside local scopes

Does this work? 😊 Yes!

```
attendees = []

def mark_attendance(name):
    attendees.append(name)
    print("In attendance:", attendees)

mark_attendance("Emily")
mark_attendance("Cristiano")
mark_attendance("Samantha")
```

Does this work?

```
current = 0

def count():
    current = current + 1
    print("Count:", current)

count()
count()
```

Names inside local scopes

Does this work? 😊 Yes!

```
attendees = []

def mark_attendance(name):
    attendees.append(name)
    print("In attendance:", attendees)

mark_attendance("Emily")
mark_attendance("Cristiano")
mark_attendance("Samantha")
```

Does this work? 😿 No!

```
current = 0

def count():
    current = current + 1
    print("Count:", current)

count()
count()
```

UnboundLocalError: local variable 'current' referenced before assignment

Scope rules

Action	Global code	Local code
Access names that are bound in the global scope?	✅ Yes	✅ Yes
Re-assign names that are bound in the global scope?	🚫 No (unless declared `global`)	🚫 No (unless declared `global`)

```
current = 0

def count():
    current = current + 1      # 🚫  Error!
    print("Count:", current)

count()
count()
```

Try in PythonTutor

Re-assigning globals

```python
current = 0

def count():

    current = current + 1
    print("Count:", current)

count()
count()
```

Try in PythonTutor

Re-assigning globals

```python
current = 0

def count():
    global current
    current = current + 1
    print("Count:", current)

count()
count()
```

Try in PythonTutor

Avoiding `global`

"Just because you can do something in a language, it doesn't mean you should." - Prof Fox

Re-assigning global variables inside functions can lead to more brittle and unpredictable code.

How about...

```python
current = 0

def count(current):
    current = current + 1
    print("Count:", current)
    return current

current = count(current)
current = count(current)
```

✨❤️🥰🌻💖✨

Names inside nested scopes

Does this work?

```python
def make_tracker(class_name):
    attendees = []

    def track_attendance(name):
        attendees.append(name)
        print(class_name, ": ", attendees)

    return track_attendance

tracker = make_tracker("CS61A")
tracker("Emily")
tracker("Cristiano")
tracker("Julian")
```

Names inside nested scopes

Does this work? 😊 Yes!

```python
def make_tracker(class_name):
    attendees = []

    def track_attendance(name):
        attendees.append(name)
        print(class_name, ": ", attendees)

    return track_attendance

tracker = make_tracker("CS61A")
tracker("Emily")
tracker("Cristiano")
tracker("Julian")
```

Names inside nested scopes

Does this work?

```python
def make_counter(start):
    current = start

    def count():
        current = current + 1
        print("Count:", current)

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

Names inside nested scopes

Does this work? 😿 No!

```
def make_counter(start):
    current = start

    def count():
        current = current + 1
        print("Count:", current)

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

UnboundLocalError: local variable 'current' referenced before assignment

Scope rules

Can code inside functions...	
Access names that are bound in the enclosing function?	✅ Yes
Re-assign names that are bound in the enclosing function?	🚫 No (unless declared `nonlocal`)

```python
def make_counter(start):
    current = start

    def count():
        current = current + 1      # 🚫  Error!
        print("Count:", current)

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

Try in PythonTutor

Re-assigning names in parent scope

```python
def make_counter(start):
    current = start

    def count():

        current = current + 1
        print("Count:", current)

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

Try in PythonTutor

Re-assigning names in parent scope

The `nonlocal` declaration tells Python to look in the parent frame for the name lookup.

```python
def make_counter(start):
    current = start

    def count():
        nonlocal current
        current = current + 1
        print("Count:", current)

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

Try in PythonTutor

Avoiding `nonlocal`

The `nonlocal` keyword was only added to Python 3, so most code that might use it can be done in more Pythonic ways.

For the example, the counter can be done with a generator:

```python
def make_counter(start):
    current = start
    while True:
        current = current + 1
        print("Count:", current)
        yield

counter = make_counter(30)
next(counter)
next(counter)
```

⚠️But we haven't learned about generators yet! Stay tuned! ⚠️

Avoiding `nonlocal`

We could also use a mutable value like a list or dict:

```
def make_counter(start):
    current = [0]

    def count():
        current[0] = 1
        print("Count:", current[0])

    return count

counter = make_counter(30)
counter()
counter()
counter()
```

Try in PythonTutor

Another use of `nonlocal`

We saw it earlier when making a pair data abstraction:

```python
def pair(a, b):
    def pair_func(which, v=None):
        nonlocal a, b
        if which == 0:
            return a
        elif which == 1:
            return b
        elif which == 2:
            a = v
        else:
            b = v
    return pair_func

def left(p):
    return p(0)

def right(p):
    return p(1)
```

Avoiding `nonlocal`

But then we learned about tuples, lists, and dicts...

```python
def pair(a, b):
    return [a, b]

def left(p):
    return p[0]

def right(p):
    return p[1)

def set_left(p, v):
    p[0] = v

def set_right(p, v):
    p[1] = v

aPair = pair(3, 2)
set_left(aPair, 5)
print(left(aPair))
```

Avoiding `nonlocal`

And we'll soon be learning how to use classes!

```python
class Pair:

    def __init__(left, right):
        self.left = left
        self.right = right

    def set_left(left):
        self.left = left

    def set_right(right):
        self.right = right

aPair = Pair(3, 2)
aPair.set_left(5)
print(aPair.left)
```

⚠️ You don't need to understand that code yet! Stay tuned! ⚠️

When to use `nonlocal` or `global`

Rarely! Once you finish this class, you will have many tools in your toolbox, and you will often find a way to write your code that doesn't need to re-assign names in parent scopes.

Scope rules

Action	**Global code**	**Local code**	**Nested function code**
Access names that are bound in the global scope?	✅Yes	✅Yes	✅Yes
Re-assign names that are bound in the global scope?	✅Yes	🚫No (unless declared `global`)	🚫No (unless declared `global`)
Access names in enclosing function?	N/A	N/A	✅Yes
Re-assign names in enclosing function?	N/A	N/A	🚫No (unless declared `nonlocal`)