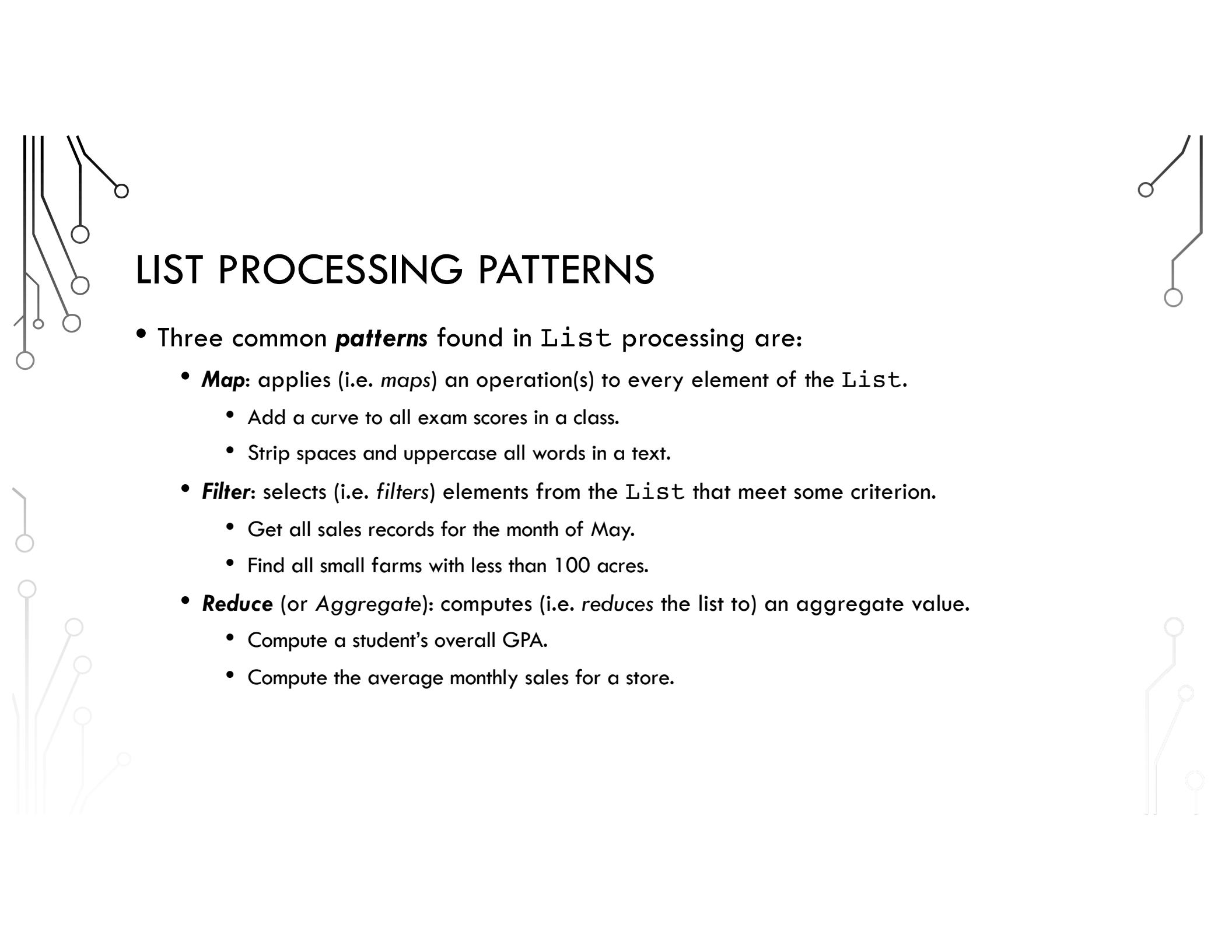




# LIST PROCESSING

COMP130 – INTRODUCTION TO COMPUTING

DICKINSON COLLEGE



# LIST PROCESSING PATTERNS

- Three common **patterns** found in List processing are:
  - **Map:** applies (i.e. *maps*) an operation(s) to every element of the List.
    - Add a curve to all exam scores in a class.
    - Strip spaces and uppercase all words in a text.
  - **Filter:** selects (i.e. *filters*) elements from the List that meet some criterion.
    - Get all sales records for the month of May.
    - Find all small farms with less than 100 acres.
  - **Reduce** (or Aggregate): computes (i.e. *reduces the list to*) an aggregate value.
    - Compute a student's overall GPA.
    - Compute the average monthly sales for a store.

# MAP PATTERN

- **Map:** applies (i.e. *maps*) an operation(s) to every element of the List.
- The common elements of the map pattern are:

Create a new List

Visit each element in the List

Apply the operation(s) to the current element and add it to the new List.

Return the new List

```
def curve_grades(scores, curve):
    """ Apply the curve to each score
    in the list scores. """
```

```
new_scores = []
for score in scores:
    new_scores.append(score + curve)

return new_scores
```

```
def add_interest(balances, rate):
    """ Add interest at the given rate to
    each balance in the list balances. """
```

```
new_balances = []
for balance in balances:
    interest = balance * rate
    new_bal = balance + interest
    new_balances.append(new_bal)

return new_balances
```

# FILTER PATTERN

- **Filter:** selects (i.e. *filters*) elements from the List that meet some criterion.
- The common elements of the filter pattern are:

Create a new List

Visit each element in the List

Filter the element, adding it to the new List if it meets the criterion.

Return the new List

```
def comes_before(names, cutoff):
    """ Get all of the names in the list
        names that come before cutoff in
        the alphabet """

```

```
    names_before = []
    for name in names:
        if name.upper() < cutoff.upper():
            names_before.append(name)
```

```
return names_before
```

# REDUCE (AGGREGATE) PATTERN

- **Reduce** (or Aggregate): computes (i.e. *reduces the list to*) an aggregate value.
- The common elements of the reduce pattern are:

Initialize the aggregate value.

Visit each element in the List

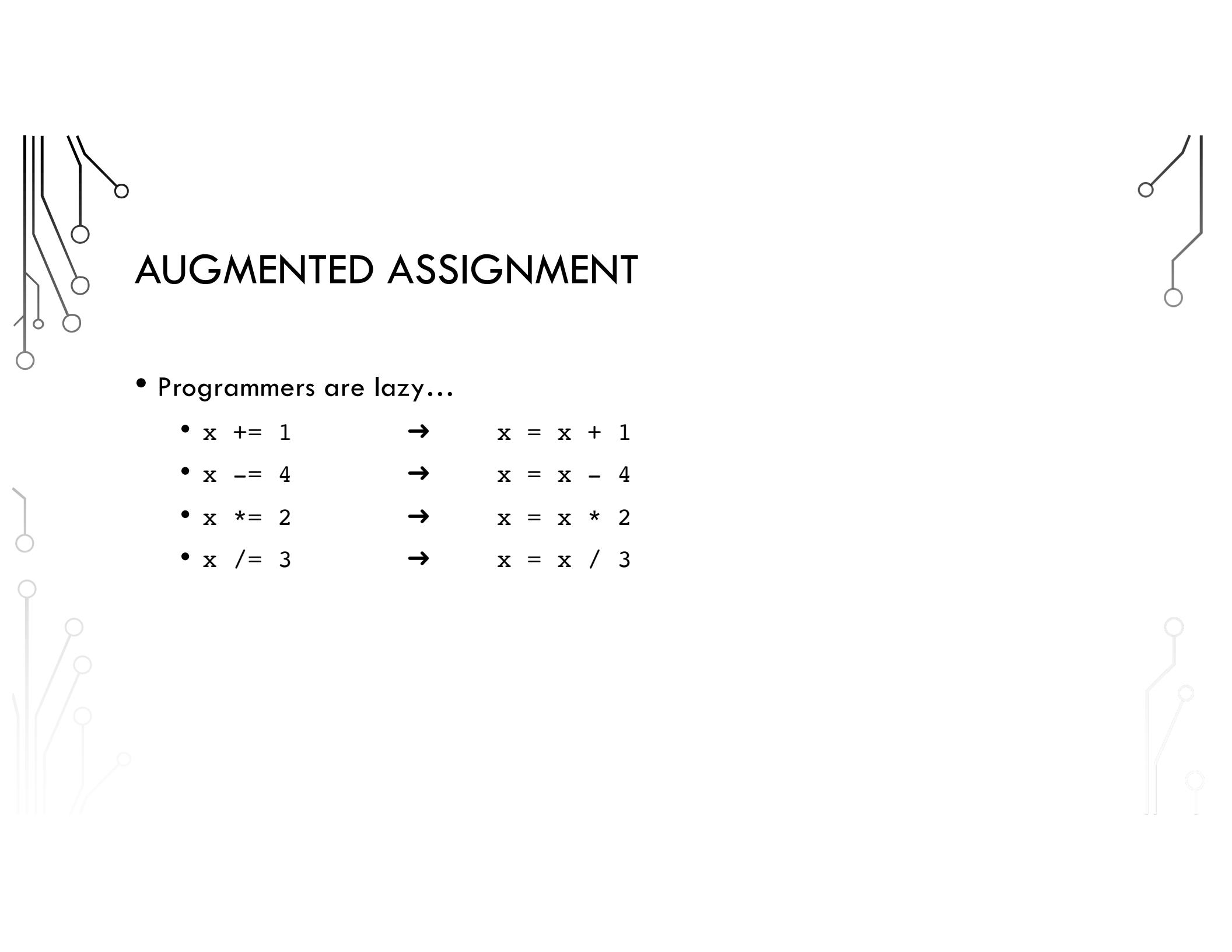
Update the aggregate value based on the current element of the List.

Return the aggregate value.

```
def count_e(words):
    """ Count the number of e's that
        appear in the strings in the
        list words. """

    total_e = 0
    for word in words:
        up = word.upper()
        total_e = total_e + up.count('E')

    return total_e
```



## AUGMENTED ASSIGNMENT

- Programmers are lazy...

- $x += 1$        $\rightarrow$        $x = x + 1$

- $x -= 4$        $\rightarrow$        $x = x - 4$

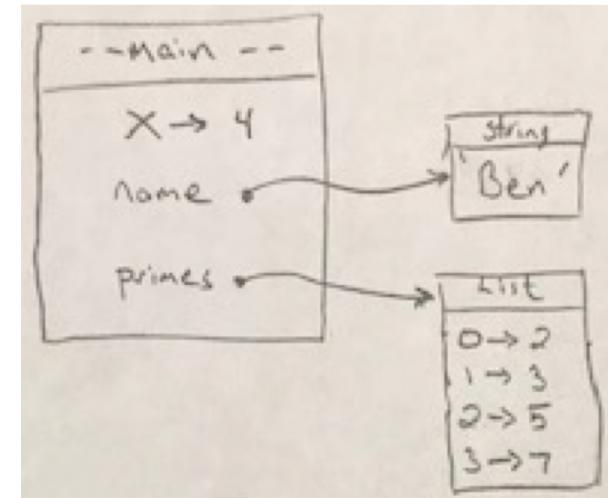
- $x *= 2$        $\rightarrow$        $x = x * 2$

- $x /= 3$        $\rightarrow$        $x = x / 3$

# OBJECTS AND OBJECT REFERENCES

- Variables that name an object do not hold the object, instead they hold a **reference** to the object.

```
x = 4
name = 'Ben'
primes = [2, 3, 5, 7]
```

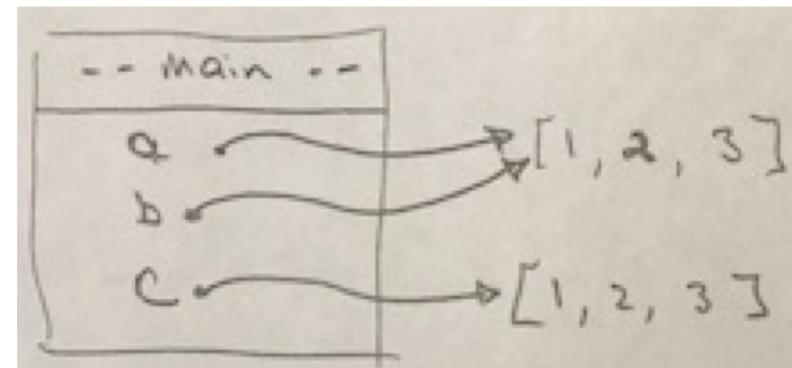


# OBJECT REFERENCES AND ALIASES

- When two variables refer to the same object they are *aliases* for the object.

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

c = [1, 2, 3]
```



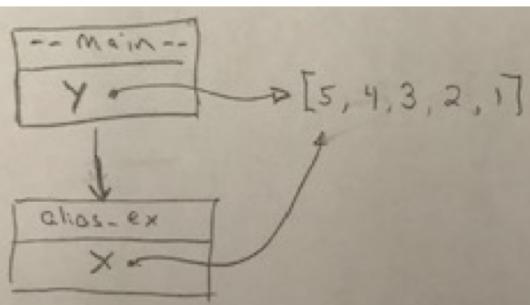
# FUNCTION PARAMETERS AND ALIASES

- If an argument passed to a function is an object reference the parameter becomes an alias to the object.

```
def alias_ex(x):
    print(x[2])

y = [5, 4, 3, 2, 1]
alias_ex(y)
```

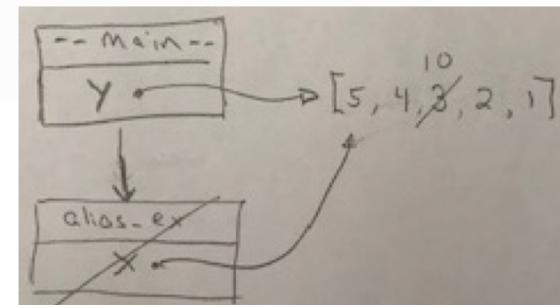
3



```
def alias_change(x):
    x[2] = 10

y = [5, 4, 3, 2, 1]
print(y[2])
alias_change(y)
print(y[2])
```

3  
10



# FUNCTION PARAMETERS AND VALUES

- By contrast, if the argument passed to a function is a value (not an object reference), the parameter is a copy of the value.

```
def not_today(x):  
    x = 7  
  
y = 3  
not_today(y)  
print(y)
```

3

