Assignment

- ***Binding a variable* in Python means setting a *name* to hold a *reference* to some *object*.**
  - *Assignment creates references, not copies*
- **Names in Python do not have an intrinsic type. Objects have types.**
  - Python determines the type of the reference automatically based on the data object assigned to it.
- **You create a name the first time it appears on the left side of an assignment expression:**

```
        x = 3
```

- **A reference is deleted via garbage collection after any names bound to it have passed out of scope.**

Understanding Reference Semantics in Python

Understanding Reference Semantics

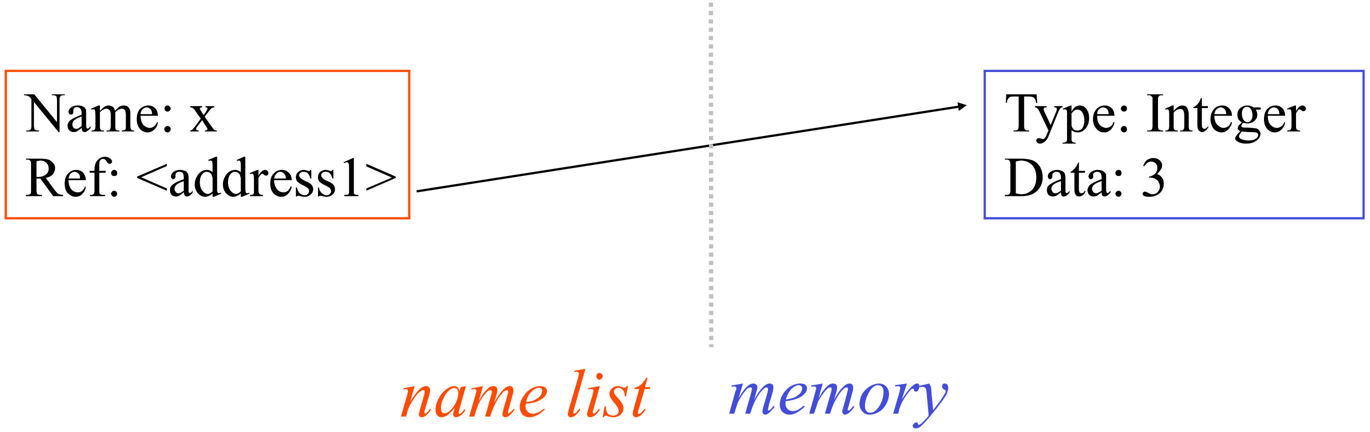
- **Assignment manipulates references**
  - —x = y **does not make a copy** of the object y references
  - —x = y makes x **reference** the object y references
- **Very useful; but beware!**
- **Example:**

```
>>> a = [1, 2, 3]    # a now references the list [1, 2, 3]
>>> b = a            # b now references what a references
>>> a.append(4)      # this changes the list a references
>>> print b          # if we print what b references,
[1, 2, 3, 4]         # SURPRISE!  It has changed…
```

Why??

Understanding Reference Semantics II

- **There is a lot going on when we type:**
  `x = 3`
- **First, an integer *3* is created and stored in memory**
- **A name *x* is created**
- **An *reference* to the memory location storing the *3* is then assigned to the name *x***
- **So: When we say that the value of *x* is *3***
- **we mean that *x* now refers to the integer *3***



Understanding Reference Semantics III

- The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are "immutable."
- This doesn't mean we can't change the value of x, i.e. *change what x refers to* …
- For example, we could increment x:

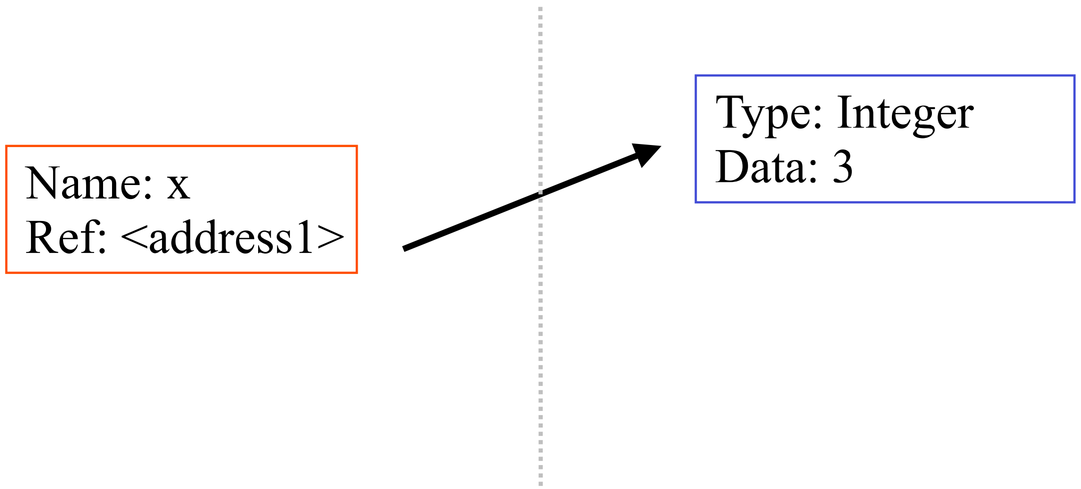
```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

  1. *The reference of name **x** is looked up.*
  2. *The value at that reference is retrieved.*

```
>>> x = x + 1
```



Name: x
Ref: <address1>

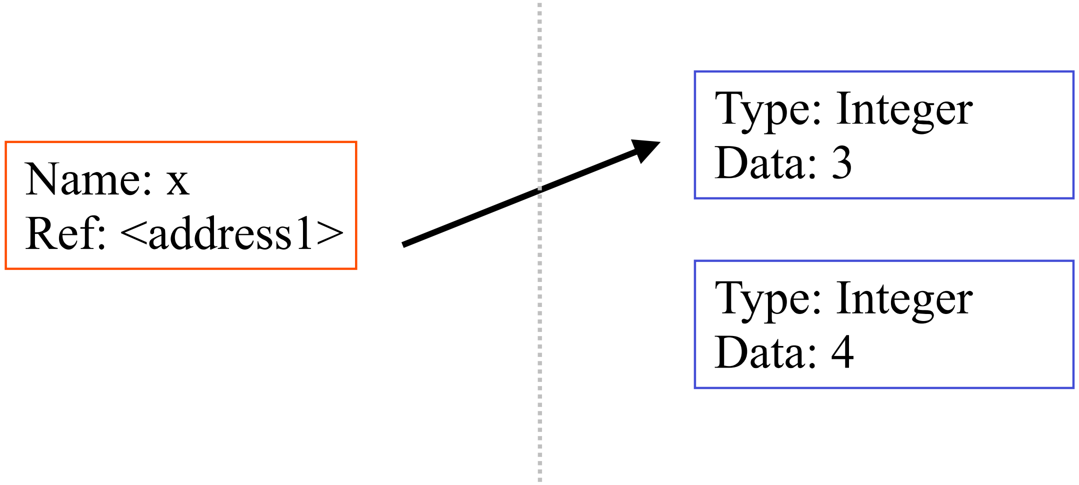
Type: Integer
Data: 3

Understanding Reference Semantics IV

- **If we increment x, then what’s really happening is:**

  1. The reference of name ***x*** is looked up.
  2. The value at that reference is retrieved.
  3. *The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.*

```
>>> x = x + 1
```

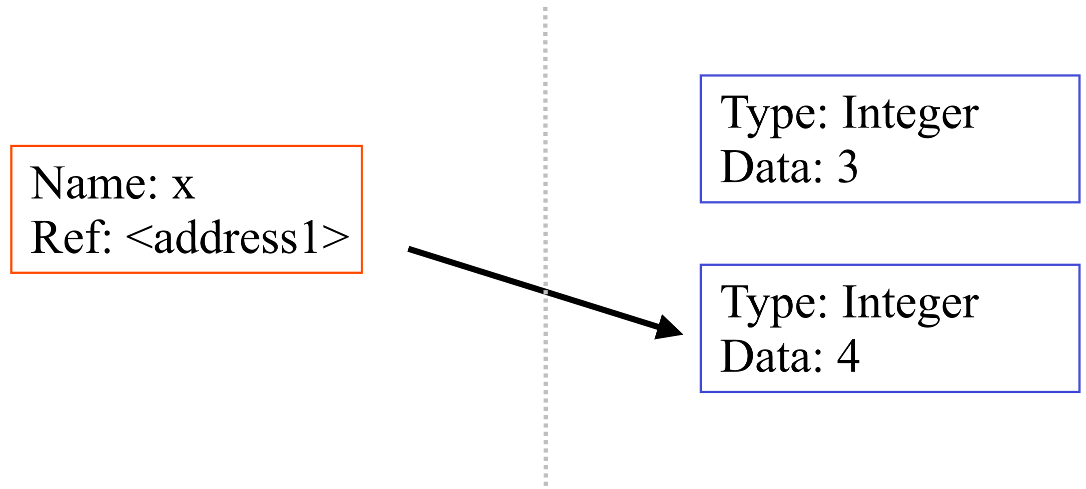


Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

  1. The reference of name ***x*** is looked up.
  2. The value at that reference is retrieved.
  3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
  4. *The name **x** is changed to point to this new reference.*

```
>>> x = x + 1
```



Name: x
Ref: <address1>

Type: Integer
Data: 3

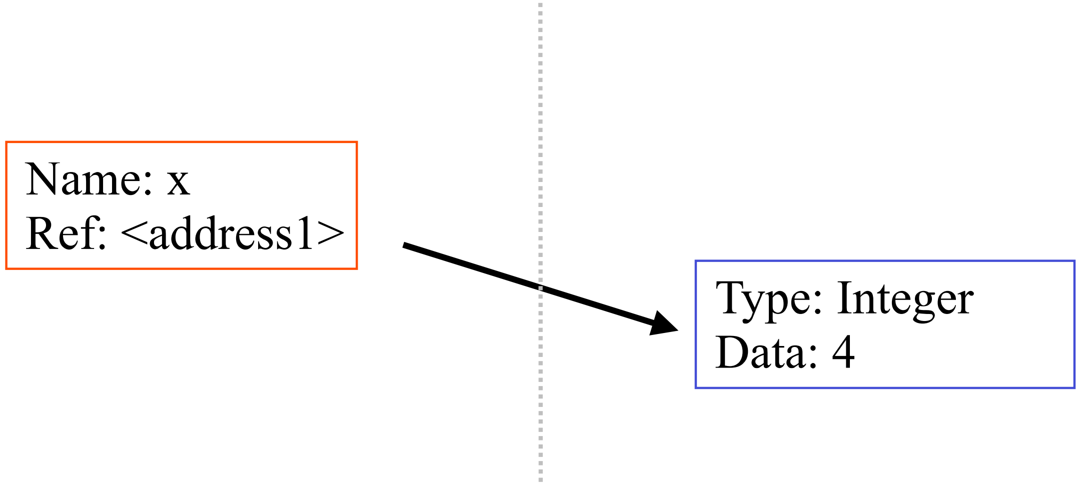
Type: Integer
Data: 4

Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

```
>>> x = x + 1
```

1. The reference of name ***x*** is looked up.
2. The value at that reference is retrieved.
3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
4. The name **x** is changed to point to this new reference.
5. *The old data* ***3*** *is garbage collected if no name still refers to it.*



Name: x
Ref: <address1>

Type: Integer
Data: 4

Assignment 1

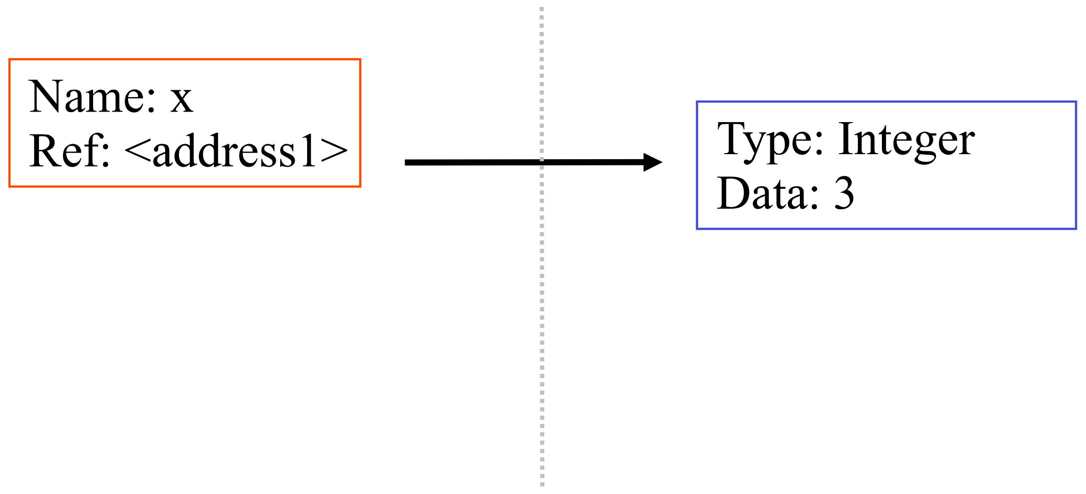
- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```

Assignment 1

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

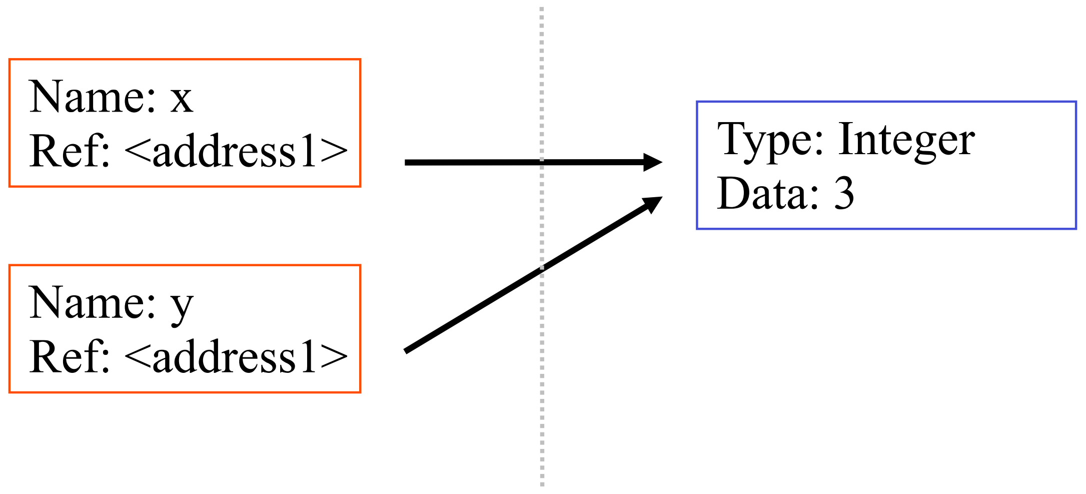
```
>>> x = 3           # Creates 3, name x refers to 3
>>> y = x           # Creates name y, refers to 3.
>>> y = 4           # Creates ref for 4. Changes y.
>>> print x         # No effect on x, still ref 3.
3
```



Assignment 1

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

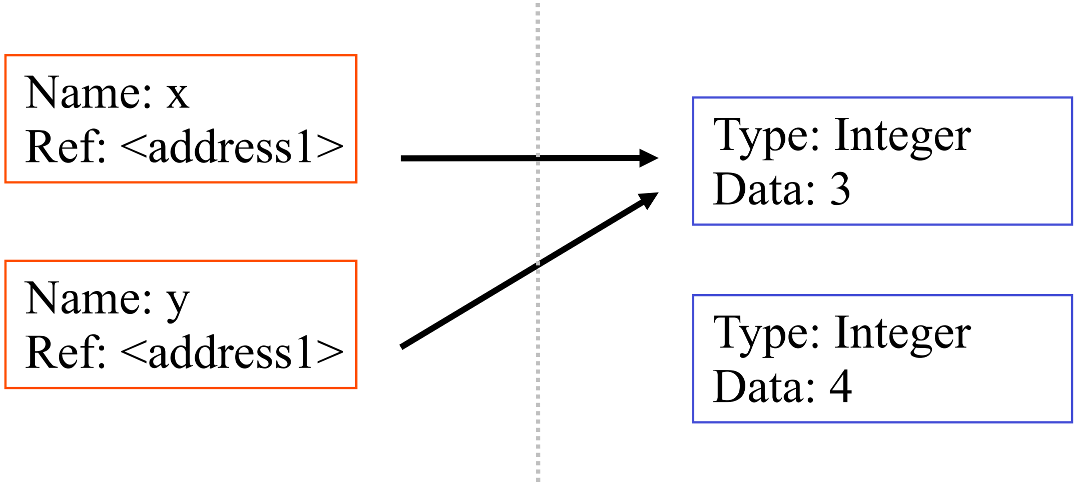
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```



Assignment 1

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

```
>>> x = 3         # Creates 3, name x refers to 3
>>> y = x         # Creates name y, refers to 3.
>>> y = 4         # Creates ref for 4. Changes y.
>>> print x       # No effect on x, still ref 3.
3
```



Name: x
Ref: <address1>

Name: y
Ref: <address1>

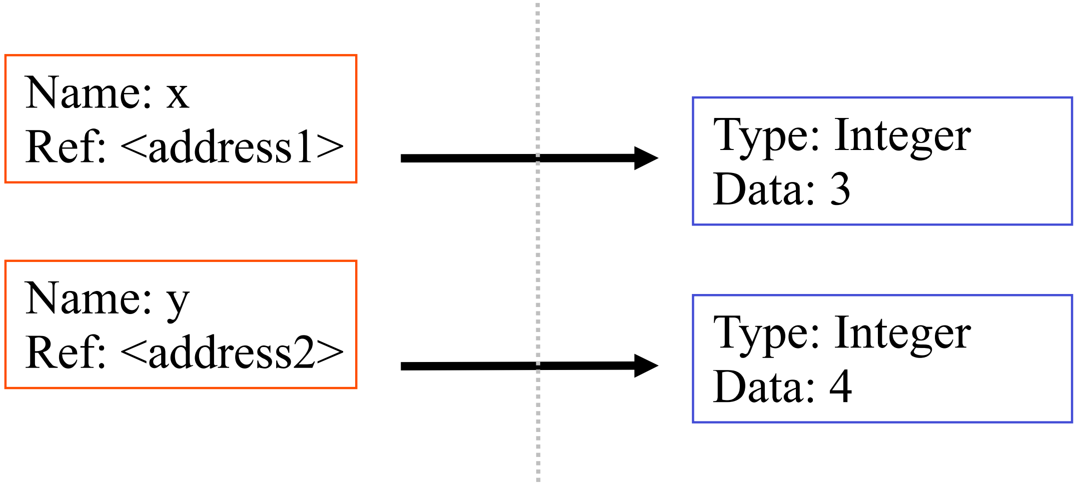
Type: Integer
Data: 3

Type: Integer
Data: 4

Assignment 1

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

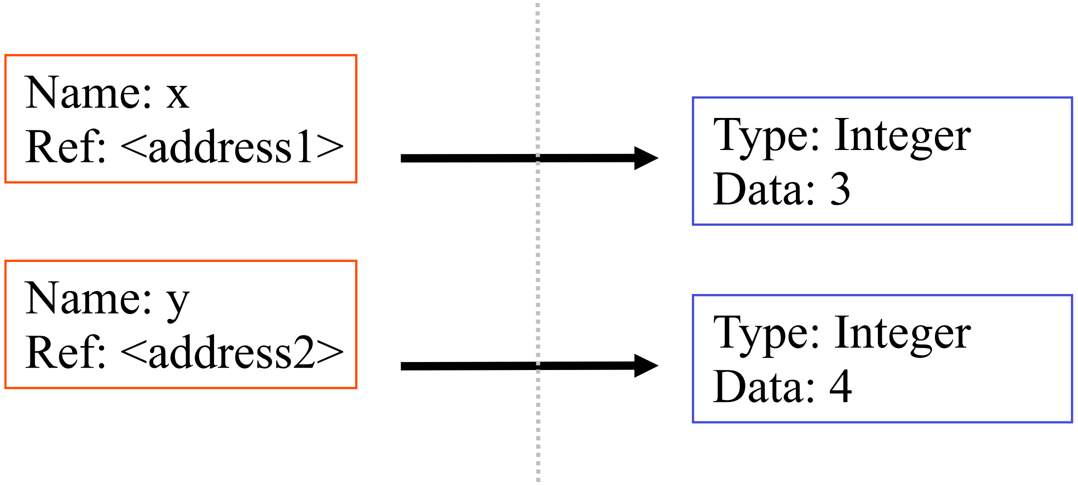
```
>>> x = 3          # Creates 3, name x refers to 3
>>> y = x          # Creates name y, refers to 3.
>>> y = 4          # Creates ref for 4. Changes y.
>>> print x        # No effect on x, still ref 3.
3
```



Assignment 1

- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

```
>>> x = 3         # Creates 3, name x refers to 3
>>> y = x         # Creates name y, refers to 3.
>>> y = 4         # Creates ref for 4. Changes y.
>>> print x       # No effect on x, still ref 3.
3
```



Name: x
Ref: <address1>

Type: Integer
Data: 3

Name: y
Ref: <address2>

Type: Integer
Data: 4

Assignment 2

- **For other data types (lists, dictionaries, user-defined types), assignment works differently.**
  - These datatypes are **“mutable.”**
  - When we change these data, we do it *in place.*
  - We don’t copy them into a new memory address each time.
  - If we type y=x and then modify y, both x and y are changed.

immutable

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

mutable

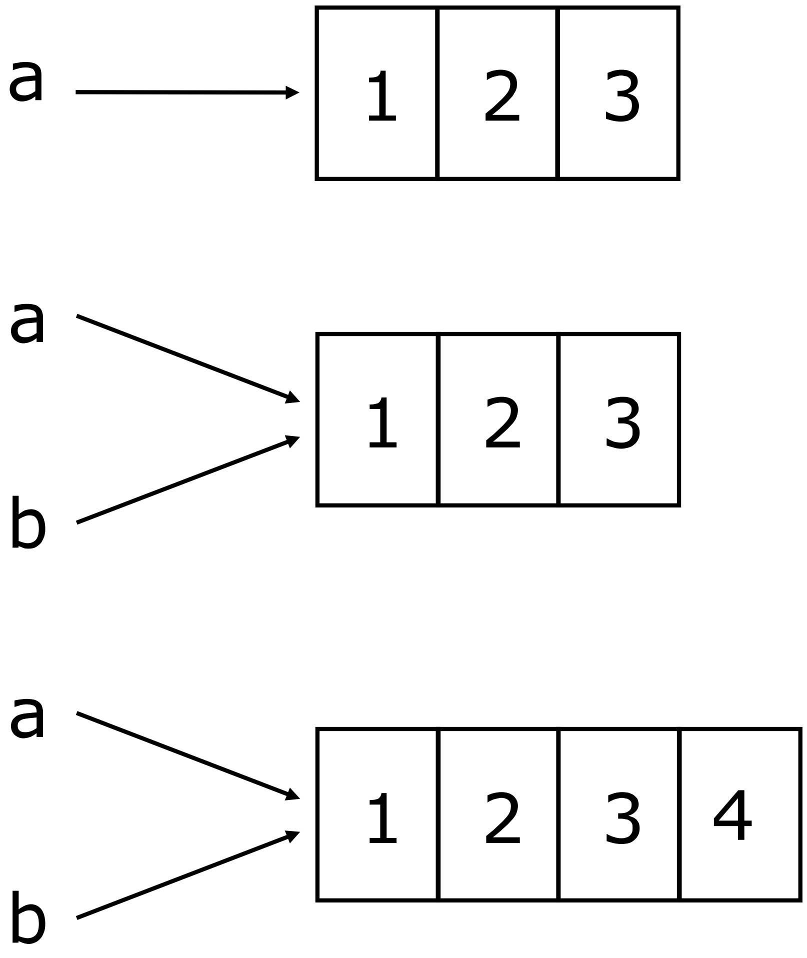
```
x = some mutable object
y = x
make a change to y
look at x
x will be changed as well
```

Why? Changing a Shared List

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

b = a

a.append(4)
```



Our surprising example surprising no more...

- **So now, here’s our code:**

```
>>> a = [1, 2, 3]    # a now references the list [1, 2, 3]
>>> b = a            # b now references what a references
>>> a.append(4)      # this changes the list a references
>>> print b          # if we print what b references,
[1, 2, 3, 4]         # SURPRISE!  It has changed…
```