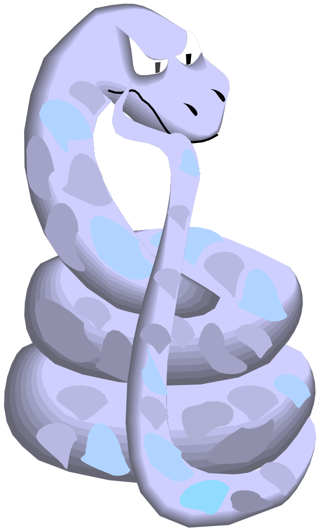
Mutability:
Tuples vs. Lists



Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You cannot change a tuple.

However, you can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
  ['abc', 45, 4.34, 23]
```

- We can change lists *in place.*
- Name *li* still points to the same memory reference when we are done.
- The mutability of lists means that operations on lists are not as fast as operations on tuples.

Operations on Lists Only

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')  # Our first exposure to method syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

The *extend* method vs the **+** operator.

- **+ creates a fresh list (with a new memory reference)**
- ***extend* operates on list `li` in place.**

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

***Confusing*:**

- **Extend takes a list as an argument.**
- **Append takes a singleton as an argument.**

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only

```
>>> li = [′a′, ′b′, ′c′, ′b′]

>>> li.index(′b′)     # index of first occurrence
1

>>> li.count(′b′)     # number of occurrences
2

>>> li.remove(′b′)    # remove first occurrence
>>> li
  [′a′, ′c′, ′b′]
```

Operations on Lists Only

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()    # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

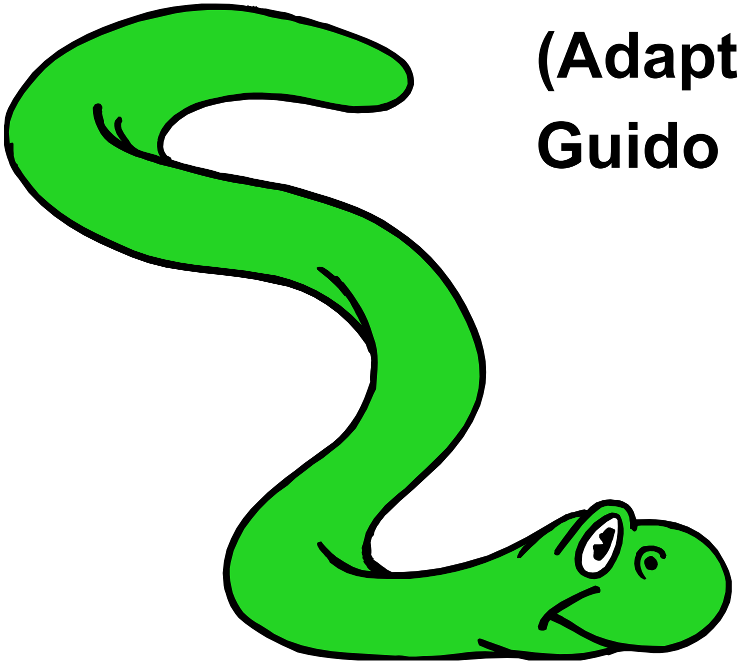
Tuples vs. Lists

- **Lists are slower at runtime, but more flexible than tuples.**
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- **To convert between tuples and lists use the list() and tuple() functions:**

```
li = list(tu)
tu = tuple(li)
```

Understanding Reference Semantics in Python



(Adapted from several slides by Guido van Rossum)

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…
```

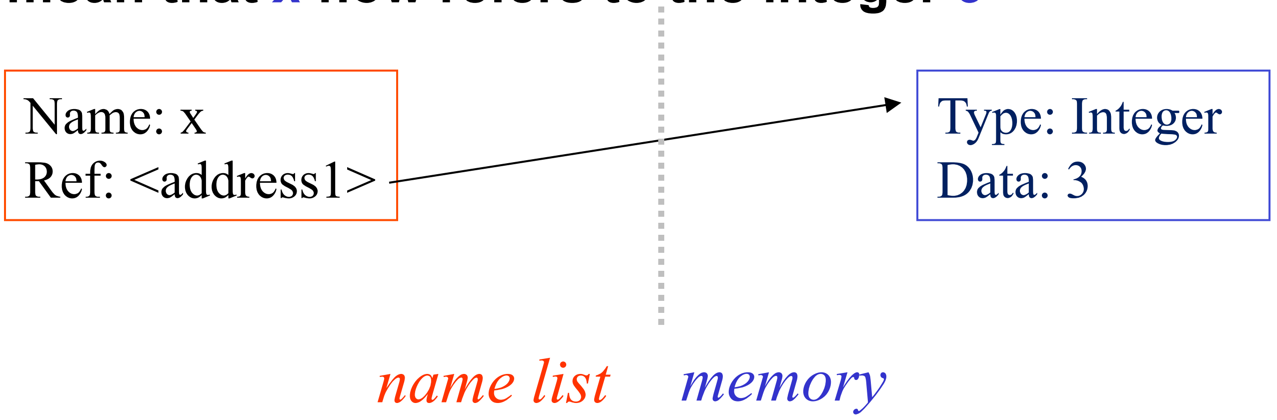
Why??

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
- A *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*



Name: x
Ref: <address1>

Type: Integer
Data: 3

name list

memory

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 does not mean we cannot change the value of x, i.e. *change what x refers to* …**
- **For example, we could increment x:**

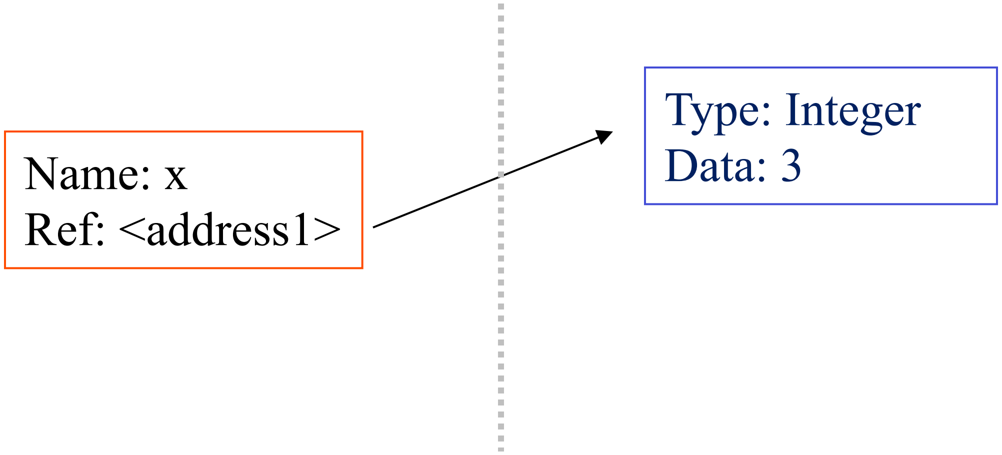
```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

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
```

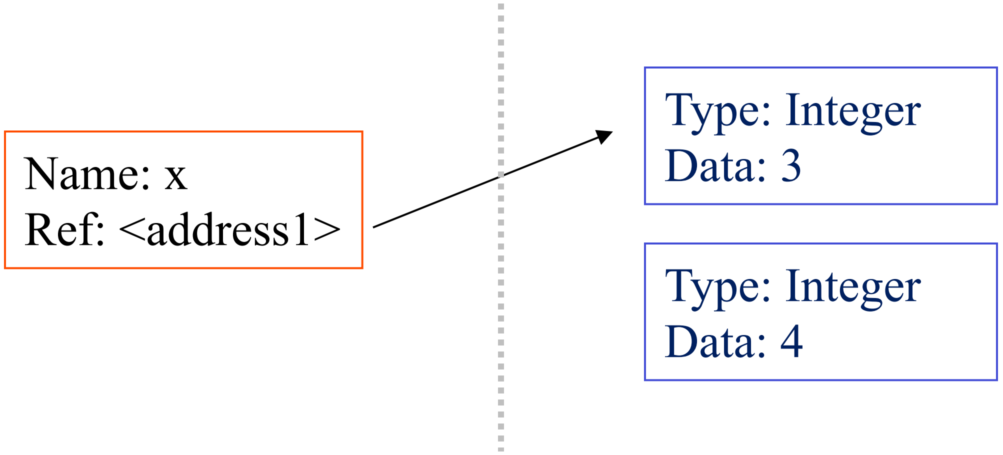


Understanding Reference Semantics IV

- **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
```

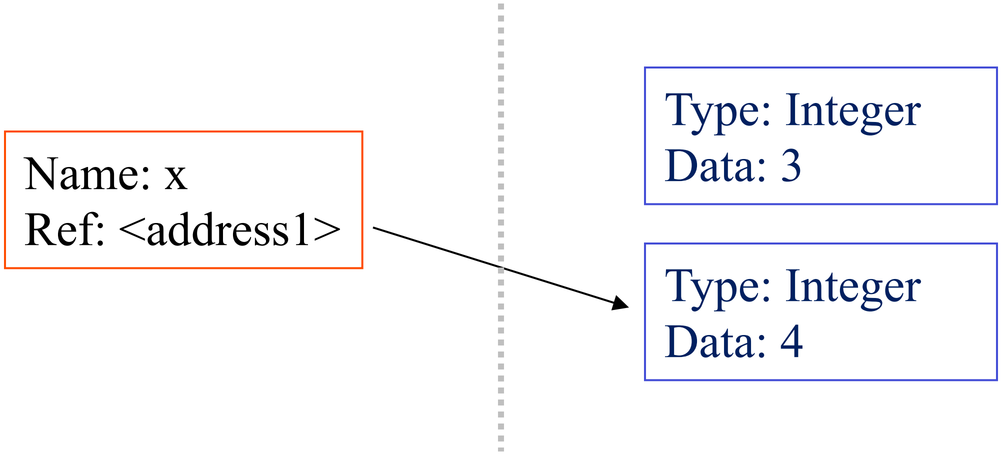


Understanding Reference Semantics IV

- **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

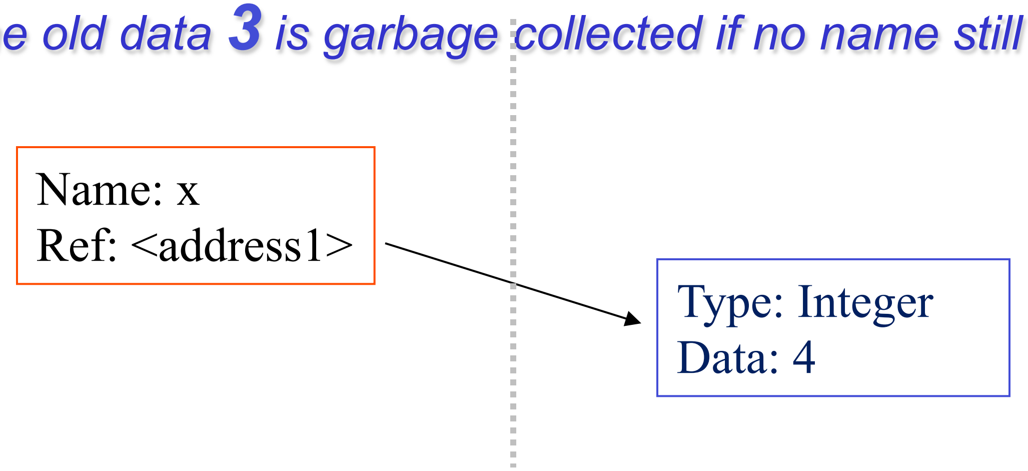
Type: Integer
Data: 4

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.*



Assignment (part 1)

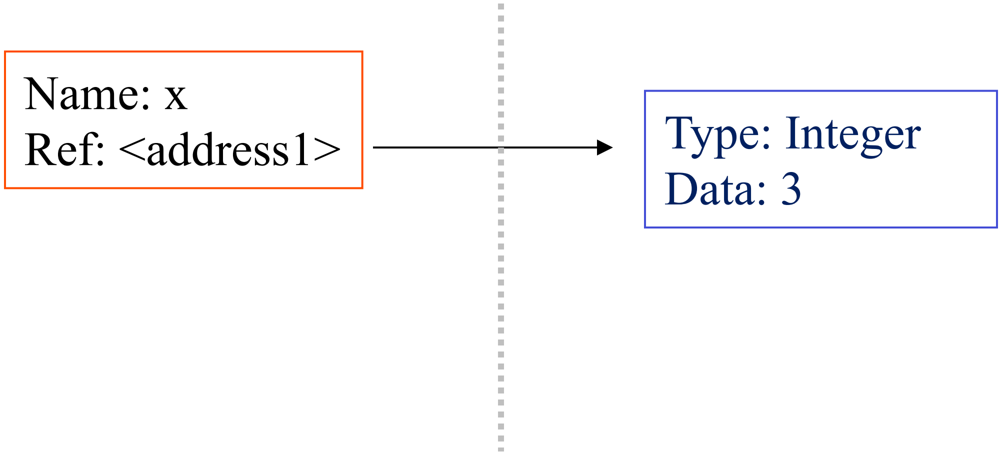
- **So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:**

```
>>> 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 (part 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
```



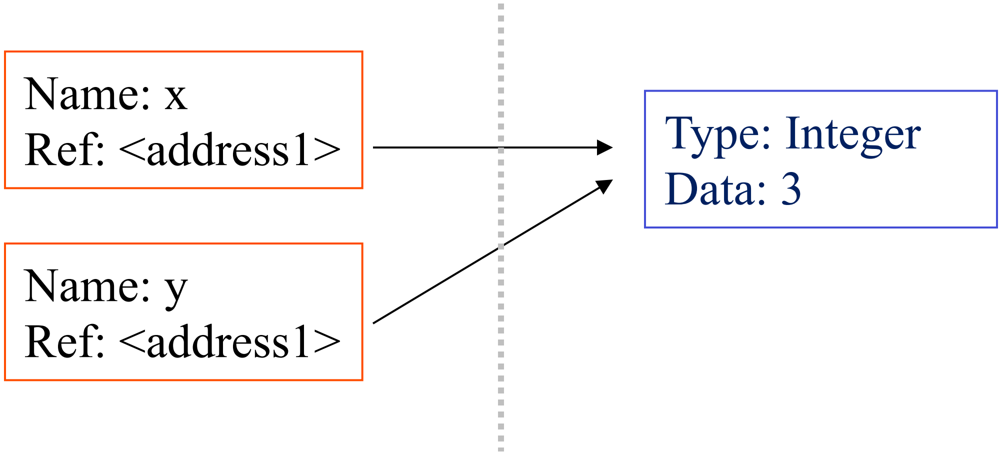
Name: x
Ref: <address1>

Type: Integer
Data: 3

Assignment (part 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>

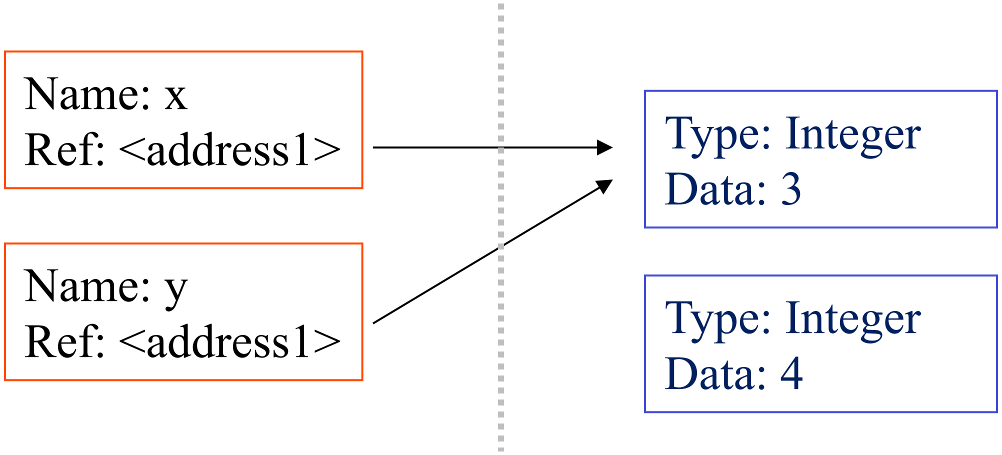
Name: y
Ref: <address1>

Type: Integer
Data: 3

Assignment (part 1)

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

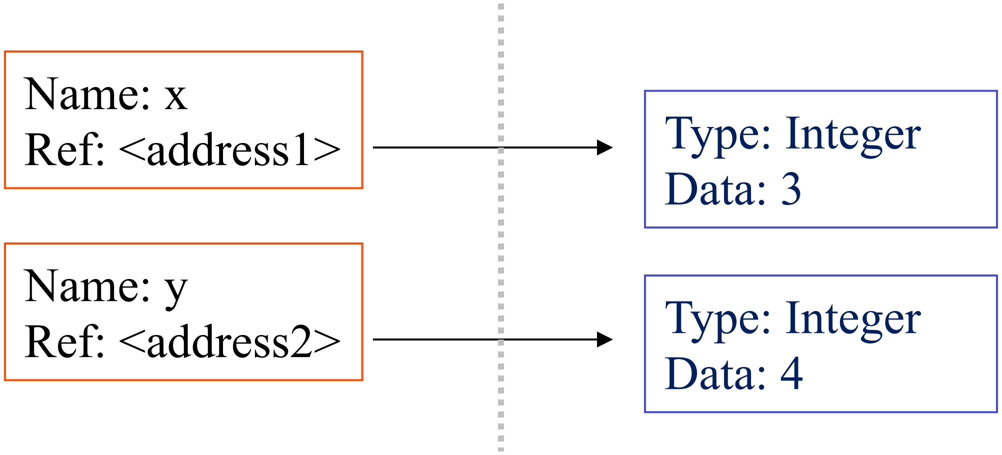
```
>>> x = 3        # Creates 3, name x refers to 3
>>> y = x        # Creates name y, refers to 3.
>>> y = 4        # Creates ref for 4. Changes y.
>>> print(x)     # No effect on x, still ref 3.
3
```



Assignment (part 1)

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

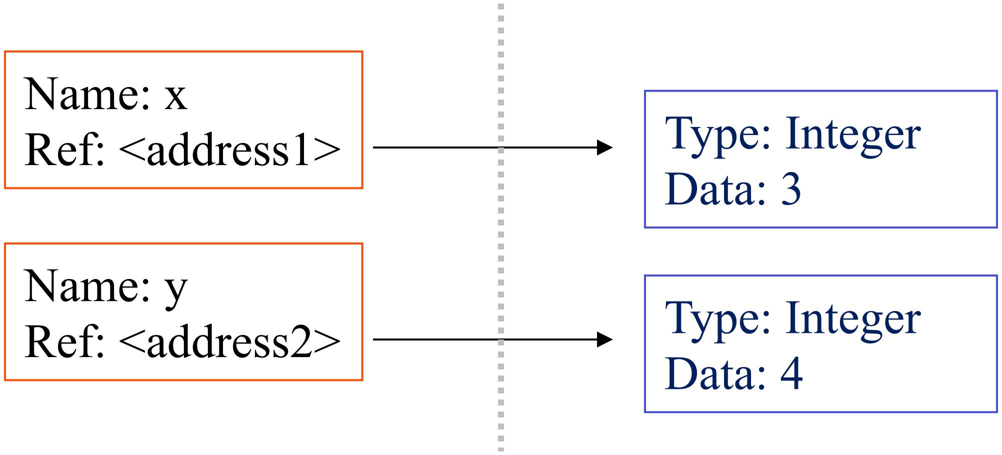
```
>>> x = 3        # Creates 3, name x refers to 3
>>> y = x        # Creates name y, refers to 3.
>>> y = 4        # Creates ref for 4. Changes y.
>>> print(x)     # No effect on x, still ref 3.
3
```



Assignment (part 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 (part 2)

- **For some 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

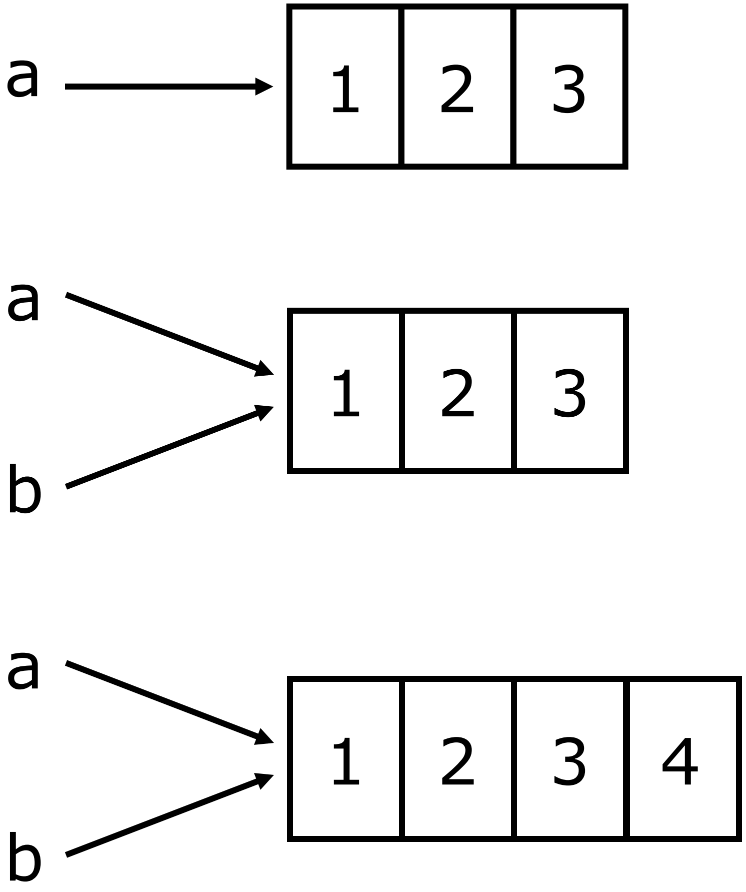
```
x = some mutable object
y = x
make a change to y
look at x
x will be changed as well
```

Why? Changing a Shared List

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

b = a

a.append(4)
```

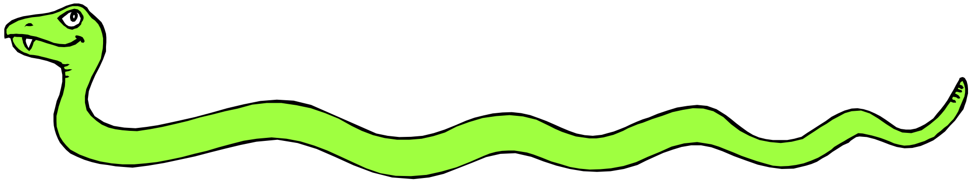


Our surprising example surprising no more...

- **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…
```

Dictionaries



Dictionaries: A *Mapping* type

- **Dictionaries store a *mapping* between a set of keys and a set of values.**
  - Keys can be any *immutable* type.
  - Values can be any type
  - A single dictionary can store values of different types
- **You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.**

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Updating Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries are unordered
  - New entry might appear anywhere in the output.
- (Dictionaries work by *hashing*)

Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']                 # Remove one.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                     # Remove all.
>>> d
{}
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()            # List of keys.
['user', 'p', 'i']

>>> d.values()          # List of values.
['bozo', 1234, 34]

>>> d.items()      # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```