CS2204

Program Design and Data Structures for Scientific Computing

Announcements

- Project #1 is posted to Blackboard.
  - We will be creating a class that allows us to represent & manipulate DNA strands
  - We will use a string as our underlying storage container

- Continuing our discussion of classes in Python…

Acknowledgement: many of these slides were taken from PPT files found at UMBC.edu

Two Kinds of Attributes

- The non-method data stored by objects are called attributes
- *Data* attributes
  - Variable owned by a *particular instance* of a class
  - Each instance has its own value for it
  - These are the most common kind of attribute
- *Class* attributes
  - Owned by the *class as a whole*
  - *All class instances share the same value for it*
  - Called “static” variables in some languages
  - Good for (1) class-wide constants and (2) building counter of how many instances of the class have been made

Data Attributes

- Data attributes are created and initialized by an `__init__()` method.
  - Simply assigning to a name creates the attribute
  - Inside the class, refer to data attributes using **`self`**
    - for example, `self._full_name`

```
class teacher:
  """A class representing teachers."""
  def __init__(self,n):
      self._full_name = n
  def print_name(self):
      print(self._full_name)
```

Class Attributes

- Because all instances of a class share one copy of a class attribute, when *any* instance changes it, the value is changed for *all* instances
- Class attributes are defined *within* a class definition and *outside* of any method
- Since there is one of these attributes *per class* and not one *per instance*, they're accessed via a different notation:
  - Access class attributes using **`self.__class__.name`** notation -- This is just one way to do this & the safest in general.

```
class sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

```
>>> a = sample()
>>> a.increment()
>>> a.__class__.x
24
```

Data vs. Class Attributes

```
class counter:
   overall_total = 0
        # class attribute
   def __init__(self):
       self.my_total = 0
         # data attribute
   def increment(self):
      counter.overall_total = \
      counter.overall_total + 1
      self.my_total = \
      self.my_total + 1
```

```
>>> a = counter()
>>> b = counter()
>>> a.increment()
>>> b.increment()
>>> b.increment()
>>> a.my_total
1
>>> a.__class__.overall_total
3
>>> b.my_total
2
>>> b.__class__.overall_total
3
```

Built-In Members of Classes

- Classes contain many methods and attributes that are included by Python even if you don’t define them explicitly.
  - Most of these methods define automatic functionality triggered by special operators or usage of that class.
  - The built-in attributes define information that must be stored for all classes.

- All built-in members have double underscores around their names: `__init__` `__doc__`

Special Methods

- For example, the method `__str__` exists for all classes, and you can always redefine it
- The definition of this method specifies how to turn an instance of the class into a string
  - **`print f`** sometimes calls **`f.__str__()`** to produce a string for object f
  - If you type **`f`** at the prompt and hit ENTER, then you are also calling **`__str__`** to determine what to display to the user as output

Special Methods – Example

```
class student:
    ...
     def __str__(self):
        return "I'm named " + self._full_name
    ...

>>> f = student("Bob Smith", 23)
>>> print(f)
I'm named Bob Smith
>>> f
"I'm named Bob Smith"
```

Special Methods

- You can redefine these as well:

  `__init__` : The constructor for the class

  `__eq__` : Define how **==** works for class

  `__len__` : Define how **`len(`** obj **`)`** works

  `__copy__` : Define how to copy a class

- Other built-in methods allow you to give a class the ability to use [ ] notation like an array or ( ) notation like a function call

Private Data and Methods

- Any attribute/method with 2 leading under-scores in its name (but none at the end) is **private** and can't be accessed outside of class
- Note: Names with two underscores at the beginning ***and the end*** are for built-in methods or attributes for the class
- Note: There is no 'protected' status in Python; so, subclasses would be unable to access these private data either.

- Now that we seen enough about strings and classes to complete project 1, let’s continue our exploration of Python by looking at additional built-in data types.

- This will prepare us for project 2.

Sequence Types

1. Tuple
   - A simple *immutable* ordered sequence of items
   - Items can be of mixed types, including collection types

2. Strings
   - A simple *immutable* ordered sequence of characters
   - Conceptually very much like a tuple

3. List
   - *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
  - We've already been introduced to many via the string class

- Key difference:
  - Tuples and strings are *immutable*
  - Lists are *mutable*

- The operations shown in the following slides can be applied to *all* sequence types
  - most examples will just show the operation performed on one

Sequence Types 1

- Define tuples using parentheses and commas

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Define lists are using square brackets and commas

```
>>> li = ["abc", 34, 4.34, 23]
```

- Define strings using quotes (", ', or """).

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket “array” notation
- *Note that all are 0 based…*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'
>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34
>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Positive and negative indices

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

Positive index: count from the left, starting with 0

```
>>> t[1]
'abc'
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset

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

•Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

• You can also use negative indices

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset

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

- Omit first index to make a copy starting from the beginning of the container

```
>>> t[:2]
(23, 'abc')
```

- Omit second index to make a copy starting at the first index and going to the end of the container

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

- [:] makes a *copy* of an entire sequence

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

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to 1 object,
            # changing the object affects both
>>> l2 = l1[:] # Independent copies, two refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*

The + Operator

- The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

The * Operator

- The * operator produces a *new* tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

Lists are 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're done.

Tuples are 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 can't change a tuple.
- You can make a fresh tuple and assign its reference to a previously used name.

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

- *The immutability of tuples means they're faster than lists.*
  - *But making copies or new tuples is not free*

Operations on Lists Only

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

>>> li.append('a')          # Note the 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 +

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

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

- *Potentially 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

- Lists have many methods, including index, count, remove, reverse, sort

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')   # index of 1st occurrence
1
>>> li.count('b')   # number of occurrences
2
>>> li.remove('b') # remove 1st 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.
    # remember that function are first-class objects.
```

Summary: Tuples vs. Lists

- Lists are slower but more powerful than tuples
  - Lists can be modified, and they have lots of handy operations and methods
    - Being mutable means that you do not have to create a new structure each time you want to make a change
  - 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

- Assignment manipulates references
    - `x = y` does not make a copy of the object y references
    - `x = y` makes x refer to the object y references

- Very useful; but beware!, e.g.

```
>>> 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?

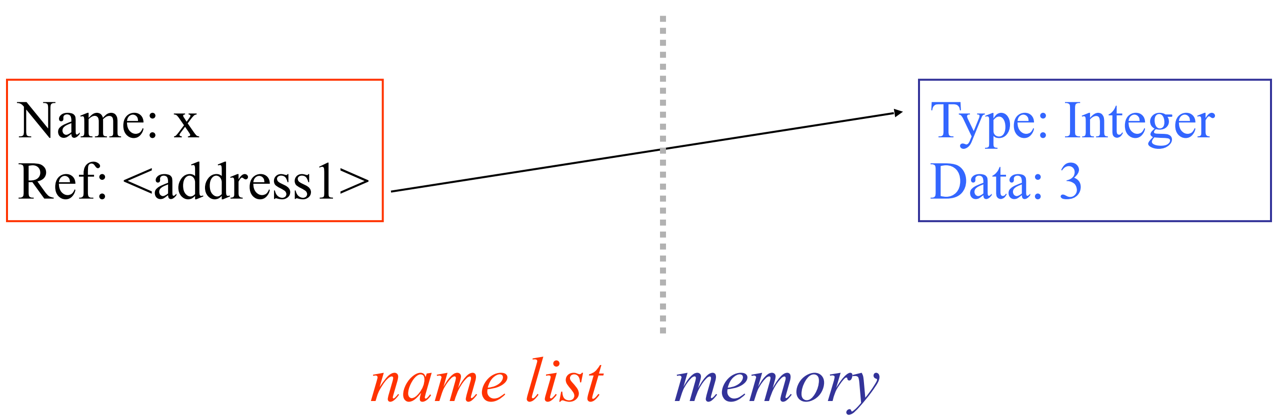
list is mutable, tuples are immutable

if a is a list, b=a points to a then to what a points to. if a changes, b changes.

if a is a tuple, b=a points to the value. if a changes, b doesn’t change.

Understanding Reference Semantics

- There's a lot going on with `x = 3`
- 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*



Understanding Reference Semantics

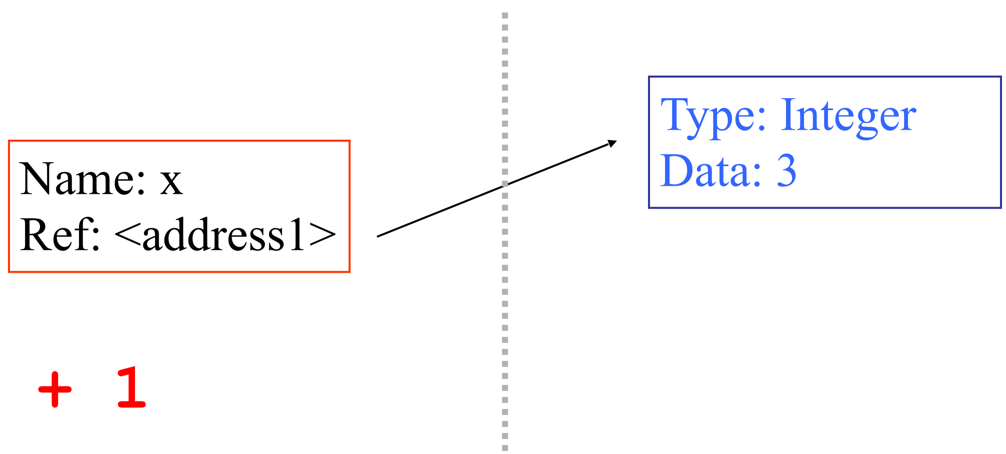
- The data 3 we created is of type integer – objects are typed, variables are not
- In Python, the datatypes integer, float, and string (and tuple) are “immutable”
- This doesn’t mean we can’t change the value of x, i.e. *change what x refers to* …
- For example, we could increment x:

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

Understanding Reference Semantics

When we increment x, then what happens is:

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

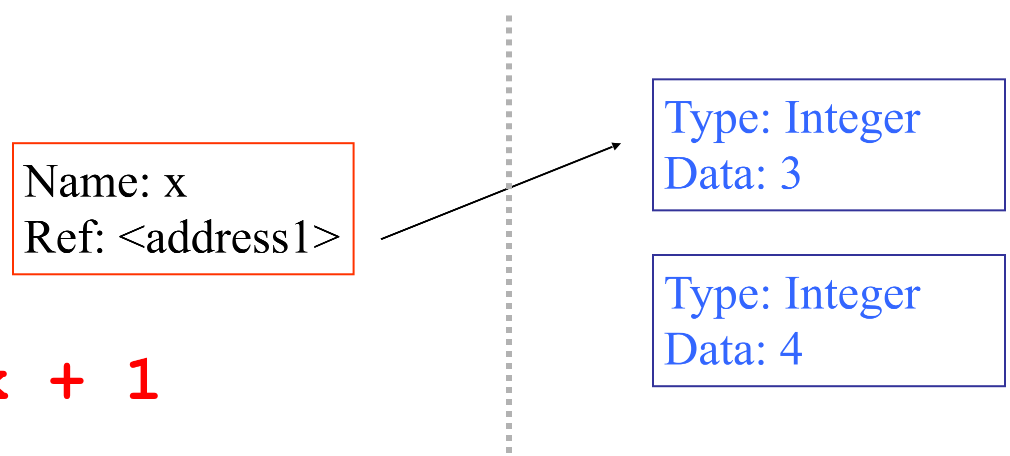



```
>>> x = x + 1
```

Understanding Reference Semantics

When we increment x, then what 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*

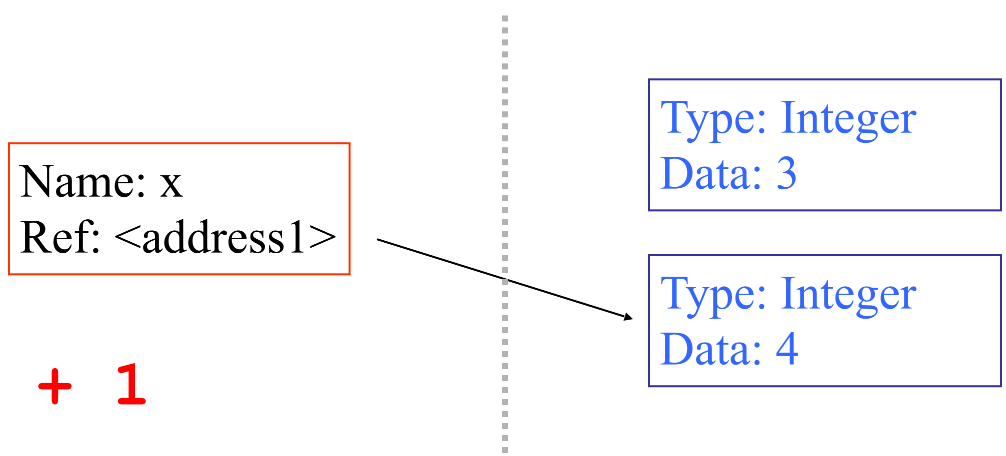



```
>>> x = x + 1
```

Understanding Reference Semantics

When we increment x, then what 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 new ref*




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

Assignment

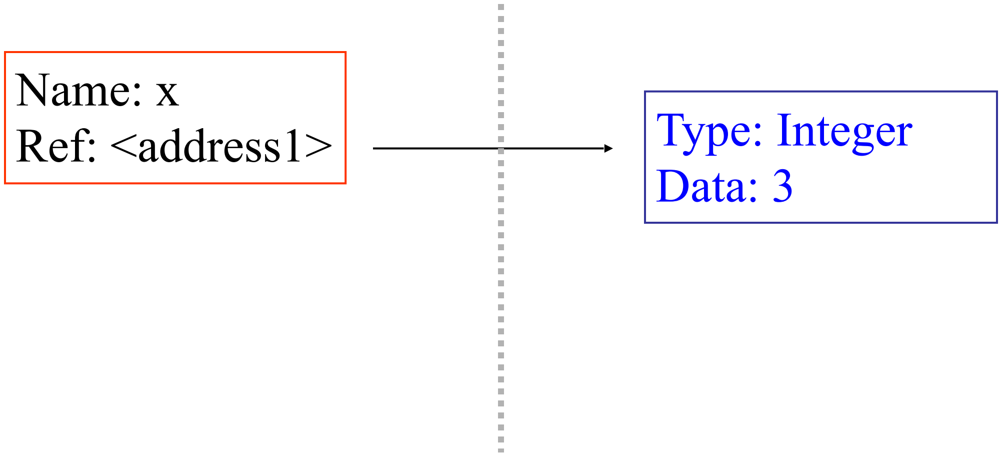
So, for simple built-in datatypes (integers, floats, strings) assignment behaves as expected

```
>>> 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

So, for simple built-in datatypes (integers, floats, strings) assignment behaves as expected

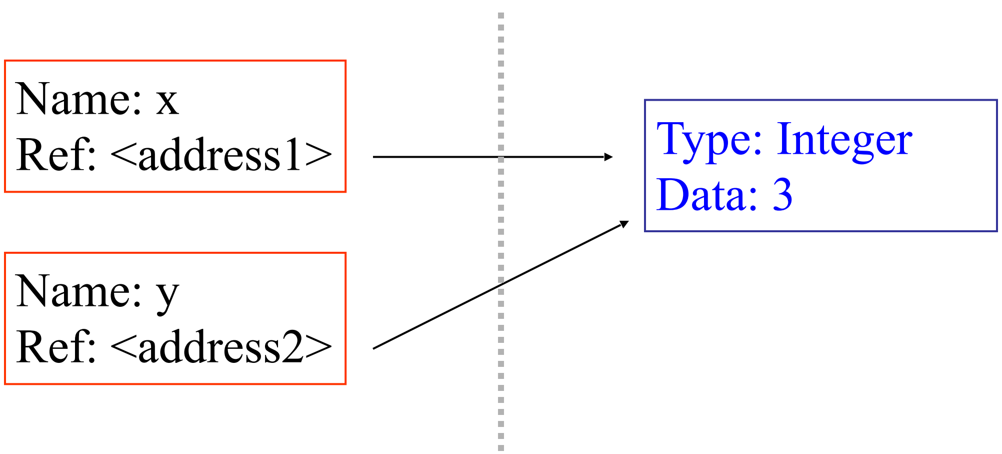
```
>>> x = 3     # Creates 3, name x refers to 3
>>> y = x     # Creates name y, refers to 3
>>> y = 4     # Creates ref for 4. Changes y
>>> print(x) # No effect on x, still ref 3
3
```



Assignment

So, for simple built-in datatypes (integers, floats, strings) assignment behaves as expected

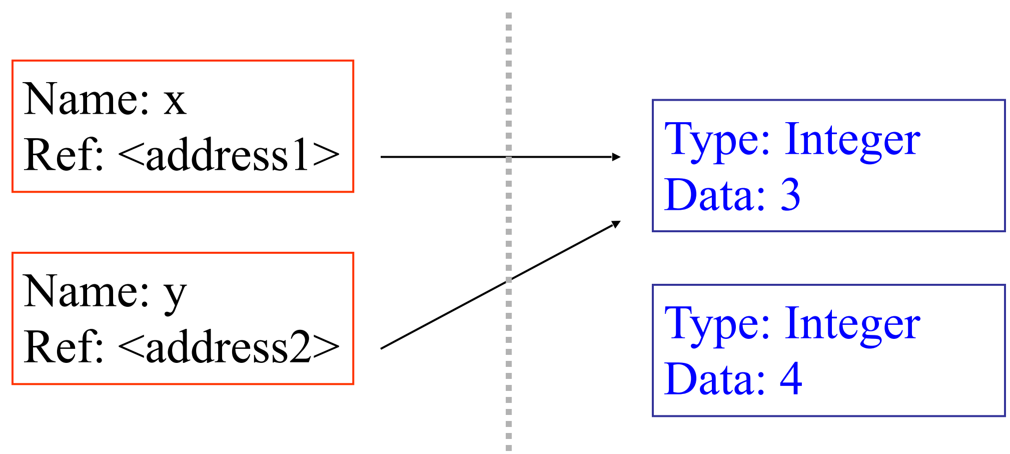
```
>>> x = 3     # Creates 3, name x refers to 3
>>> y = x     # Creates name y, refers to 3
>>> y = 4     # Creates ref for 4. Changes y
>>> print(x) # No effect on x, still ref 3
3
```



Assignment

So, for simple built-in datatypes (integers, floats, strings) assignment behaves as expected

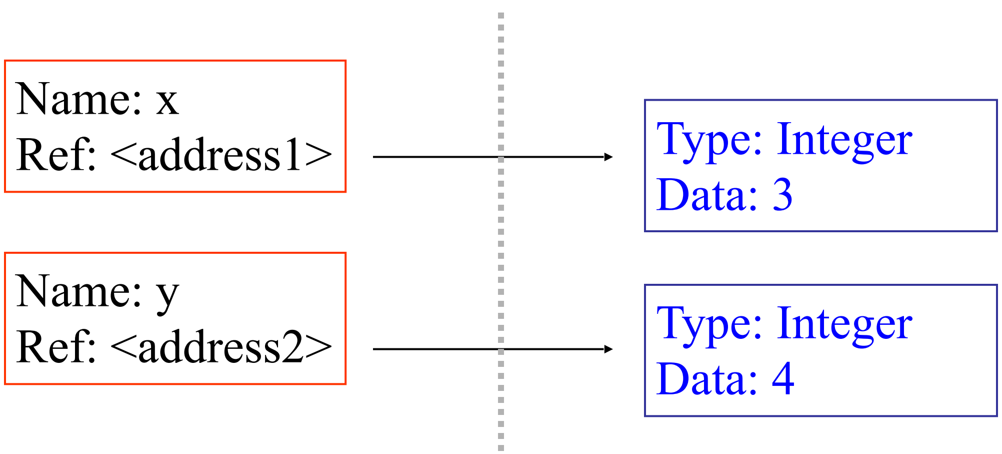
```
>>> x = 3     # Creates 3, name x refers to 3
>>> y = x     # Creates name y, refers to 3
>>> y = 4     # Creates ref for 4. Changes y
>>> print(x) # No effect on x, still ref 3
3
```



Assignment

So, for simple built-in datatypes (integers, floats, strings) assignment behaves as expected

```
>>> 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 & mutable objects

For other data types (lists, dictionaries, user-defined types), assignment works differently

- These datatypes are "mutable"
- Change occur *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

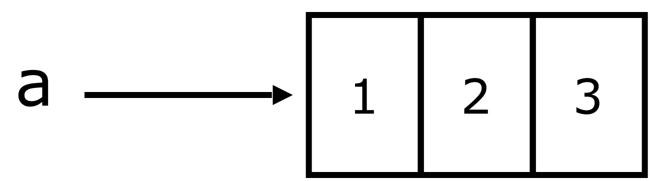
```
>>> x = 3
>>> y = x
>>> y = 4
>>> print(x)
3
```

mutable

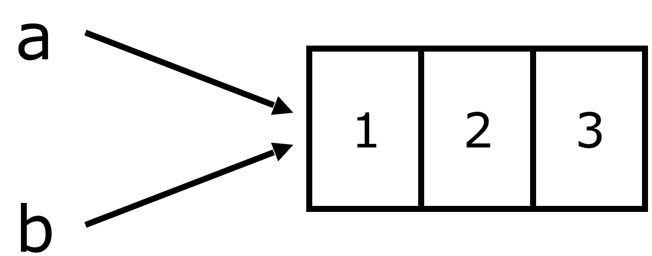
```
x = some mutable object
y = x
make a change to y
look at x
        x will be changed as well
```

Why? Changing a Shared List

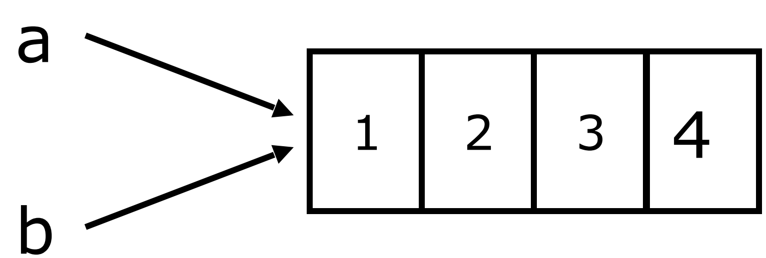
```
a = [1, 2, 3]
```




```
b = a
```




```
a.append(4)
```



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…
```