PYTHON CLASSES

(download slides and .py files to follow along)

6.100L Lecture 17

Ana Bell

OBJECTS

- Python supports many different kinds of data

`1234`   `3.14159`   `"Hello"`   `[1, 5, 7, 11, 13]`

`{"CA": "California", "MA": "Massachusetts"}`

- Each is an **object**, and every object has:
  - An internal **data representation** (primitive or composite)
  - A set of procedures for **interaction** with the object
- An object is an **instance** of a **type**
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a `str`

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- Can **create new objects** of some type
- Can **manipulate objects**
- Can **destroy objects**
  - Explicitly using `del` or just "forget" about them
  - Python system will reclaim destroyed or inaccessible objects – called "garbage collection"

WHAT ARE OBJECTS?

- Objects are **a data abstraction** that captures…

(1) An **internal representation**

  - Through data attributes

(2) An **interface** for interacting with object

  - Through methods (aka procedures/functions)
  - Defines behaviors but hides implementation

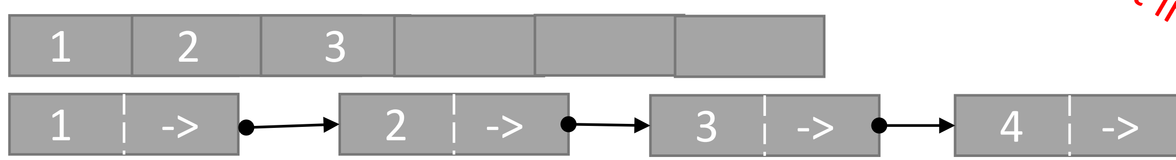
EXAMPLE:
[1,2,3,4] has type list

- (1) How are lists **represented internally**?
  Does not matter for so much for us as users (private representation)

follow pointer to the next index

L =

or **L =**



- (2) How to **interface with, and manipulate,** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- Internal representation should be private
- Correct behavior may be compromised if you manipulate internal representation directly

REAL-LIFE EXAMPLES

- **Elevator**: a box that can change floors
  - Represent using length, width, height, max_capacity, current_floor
  - Move its location to a different floor, add people, remove people
- **Employee**: a person who works for a company
  - Represent using name, birth_date, salary
  - Can change name or salary
- **Queue at a store**: first customer to arrive is the first one helped
  - Represent customers as a list of str names
  - Append names to the end and remove names from the beginning
- **Stack of pancakes**: first pancake made is the last one eaten
  - Represent stack as a list of str
  - Append pancake to the end and remove from the end

ADVANTAGES OF OOP

- **Bundle data into packages** together with procedures that work on them through well-defined interfaces
- **Divide-and-conquer** development
  - Implement and test behavior of each class separately
  - Increased modularity reduces complexity
- **Classes** make it easy to **reuse** code
  - Many Python modules define new classes
  - Each class has a separate environment (no collision on function names)
  - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

BIG IDEA

You write the class so you make the design decisions.

You decide what data represents the class.

You decide what operations a user can do with the class.

CREATING AND USING YOUR OWN TYPES WITH CLASSES

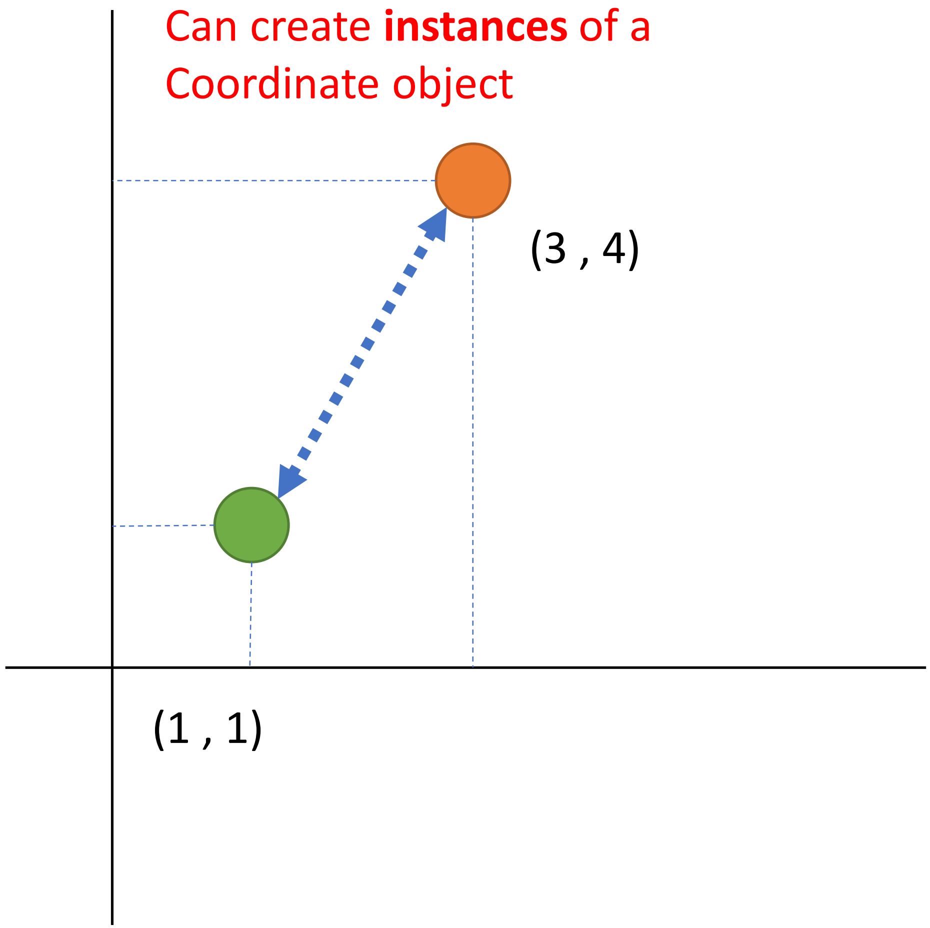
- Make a distinction between **creating a class** and **using an instance** of the class
- **Creating** the class involves
  - Defining the class name
  - Defining class attributes
  - *for example, someone wrote code to implement a list class*
- **Using** the class involves
  - Creating new **instances** of the class
  - Doing operations on the instances
  - *for example,* `L=[1,2]` *and* `len(L)`

A PARALLEL with FUNCTIONS

- **Defining a class** is like defining a function
  - With functions, we tell Python this procedure exists
  - With classes, we tell Python about a **blueprint for this new data type**
    - Its data attributes
    - Its procedural attributes

- **Creating instances of objects** is like calling the function
  - With functions we make calls with different actual parameters
  - With classes, we **create new object tinstances in memory of this type**
    - L1 = [1,2,3]
      L2 = [5,6,7]

COORDINATE TYPE DESIGN DECISIONS



- Decide what **data** elements constitute an object
- In a 2D plane
- A coordinate is defined by an **x and y value**

- Decide **what to do** with coordinates
- Tell us how far away the coordinate is on the x or y axes
- Measure the **distance** between two coordinates, Pythagoras

DEFINE YOUR OWN TYPES

- Use the `class` keyword to define a new type

class definition name/type class parent

```
class Coordinate(object):
    #define attributes here
```

- Similar to `def`, indent code to indicate which statements are part of the **class definition**
- The word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (will see in future lects)

WHAT ARE ATTRIBUTES?

- Data and procedures that "**belong**" to the class
- **Data attributes**
  - Think of data as other objects/variables that make up the class
  - *for example, a coordinate is made up of two numbers*
- **Methods** (procedural attributes)
  - Think of methods as functions that only work with this class
  - How to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- First have to define **how to create an instance** of class
- Use a **special method called `__init__`** to initialize some data attributes or perform initialization operations

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

what data initializes a Coordinate object

special method to create an instance is double __ underscore

parameter to refer to an instance of the class without having created one yet

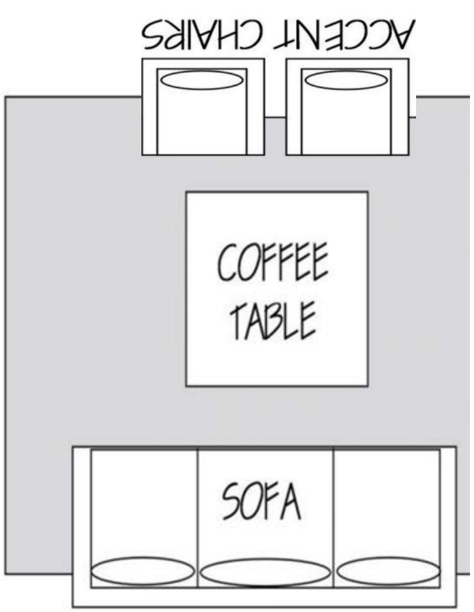
two data attributes make up your type

- `self` allows you to create **variables that belong to this object**
- Without `self`, you are just creating regular variables!

WHAT is self? ROOM EXAMPLE

self is the blueprint's way of accessing attributes (data and methods)

- Think of the class definition as a **blueprint** with placeholders for actual items
  - self has a chair
  - self has a coffee table
  - self has a sofa



- Now when you create **ONE** instance (name it living_room), self becomes this actual object
  - living_room has a blue chair
  - living_room has a black table
  - living_room has a white sofa
- Can make **many instances** using the same blueprint

BIG IDEA

When defining a class, we don’t have an actual tangible object here.

It’s only a definition.

Recall the __init__ method in the class def:

```
def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

ACTUALLY CREATING AN INSTANCE OF A CLASS

- Don't provide argument for `self`, Python does this automatically

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

- Data attributes of an instance are called **instance variables**
  - Data attributes were defined with `self.XXX` and they are accessible with dot notation for the lifetime of the object
  - All instances have these data attributes, but with different values!

```
print(c.x)
print(origin.x)
```

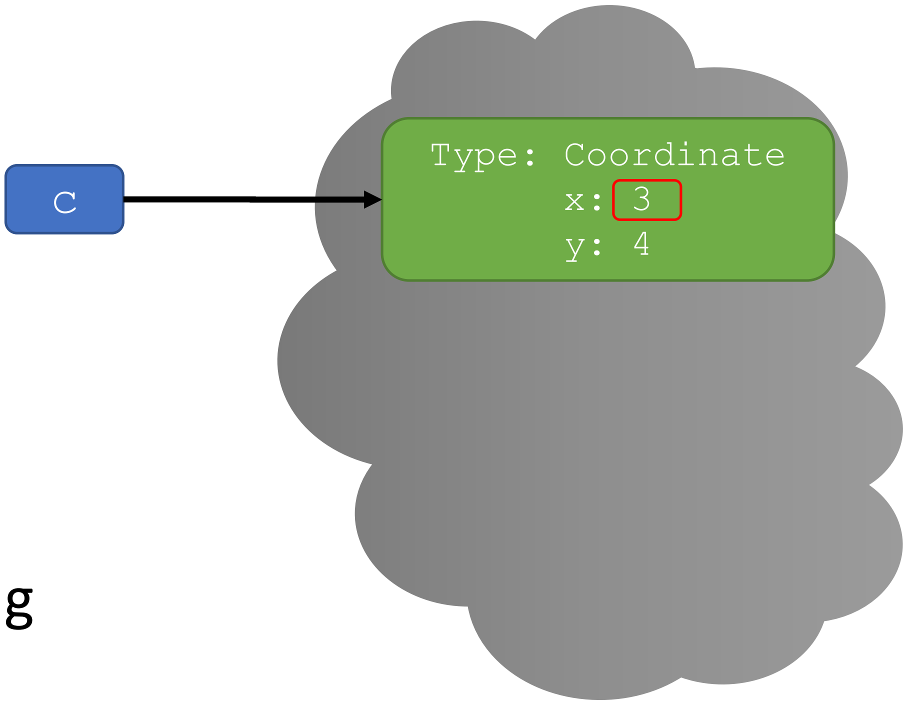
use the dot notation to access an attribute of instance `c`

VISUALIZING INSTANCES

- Suppose we create an instance of a coordinate

```
c = Coordinate(3,4)
```

- Think of this as creating a structure in memory
- Then evaluating
  `c.x`
  looks up the structure to which `c` points, then finds the binding for `x` in that structure

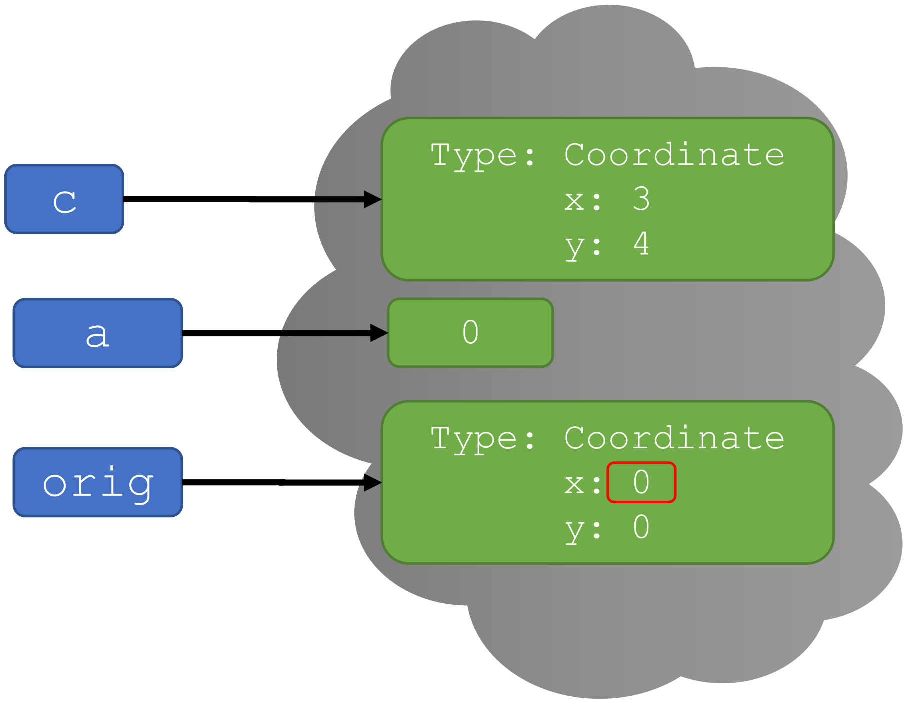


VISUALIZING INSTANCES: in memory

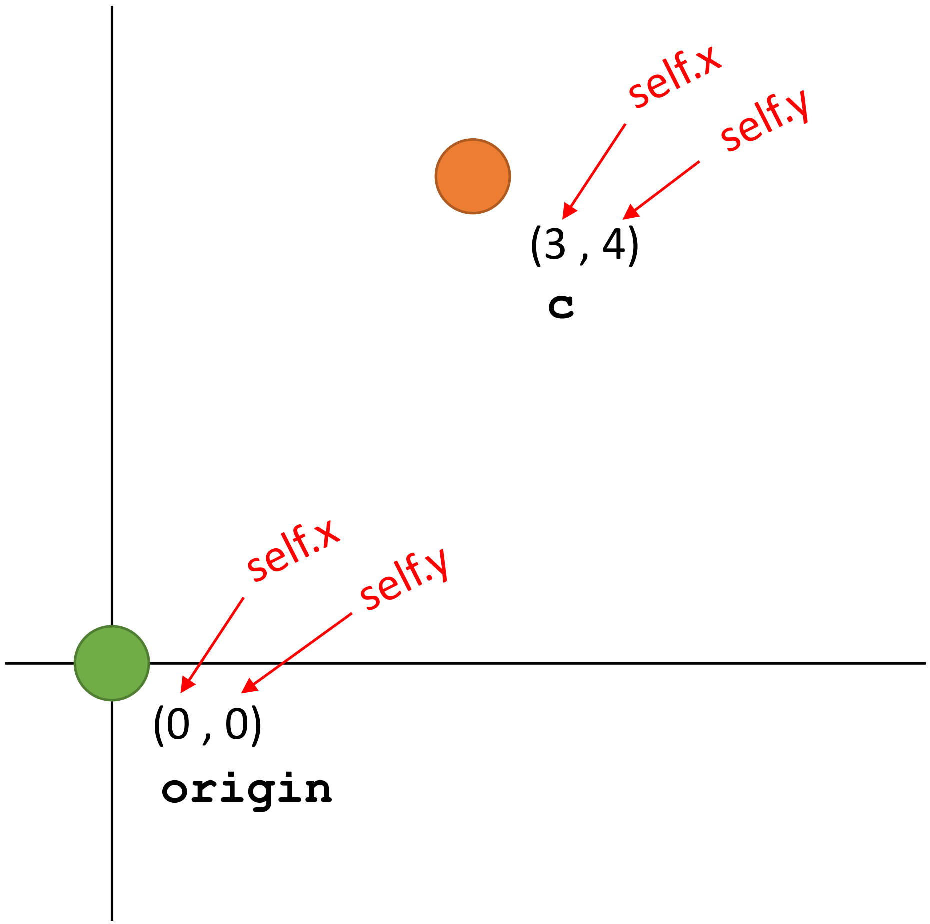
- Make another instance using a variable

```
a = 0
orig = Coordinate(a,a)
orig.x
```

- All these are just objects in memory!
- We just access attributes of these objects



VISUALIZING INSTANCES: draw it



The template for a Coordinate type

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
```

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

Code to make actual tangible Coordinate objects (aka instances)

WHAT IS A METHOD?

- Procedural attribute
  - Think of it like a **function that works only with this class**
- Python always passes the object as the first argument
  - Convention is to use **`self`** as the name of the first argument of all methods

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class Coordinate(object):
    def __init__(self, xval, yval):
        self.x = xval
        self.y = yval
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to the obj I call this method on

another parameter to method

dot notation to access x of self

dot notation to access x of other

- Other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO CALL A METHOD?

- The "." **operator** is used to access any attribute
  - A data attribute of an object (we saw `c.x`)
  - A method of an object
- Dot notation

```
<object_variable>.<method>(<parameters>)
```

Object to call method on, becomes self in the class def

Name of method

Not including self. `self` is the obj before the dot!

- Familiar?

```
my_list.append(4)
my_list.sort()
```

HOW TO USE A METHOD

Recall the definition of distance method:

```
def distance(self, other):
    x_diff_sq = (self.x-other.x)**2
    y_diff_sq = (self.y-other.y)**2
    return (x_diff_sq + y_diff_sq)**0.5
```

Using the class:

```
c = Coordinate(3,4)
orig = Coordinate(0,0)
print(c.distance(orig))
```

object to call method on

name of method

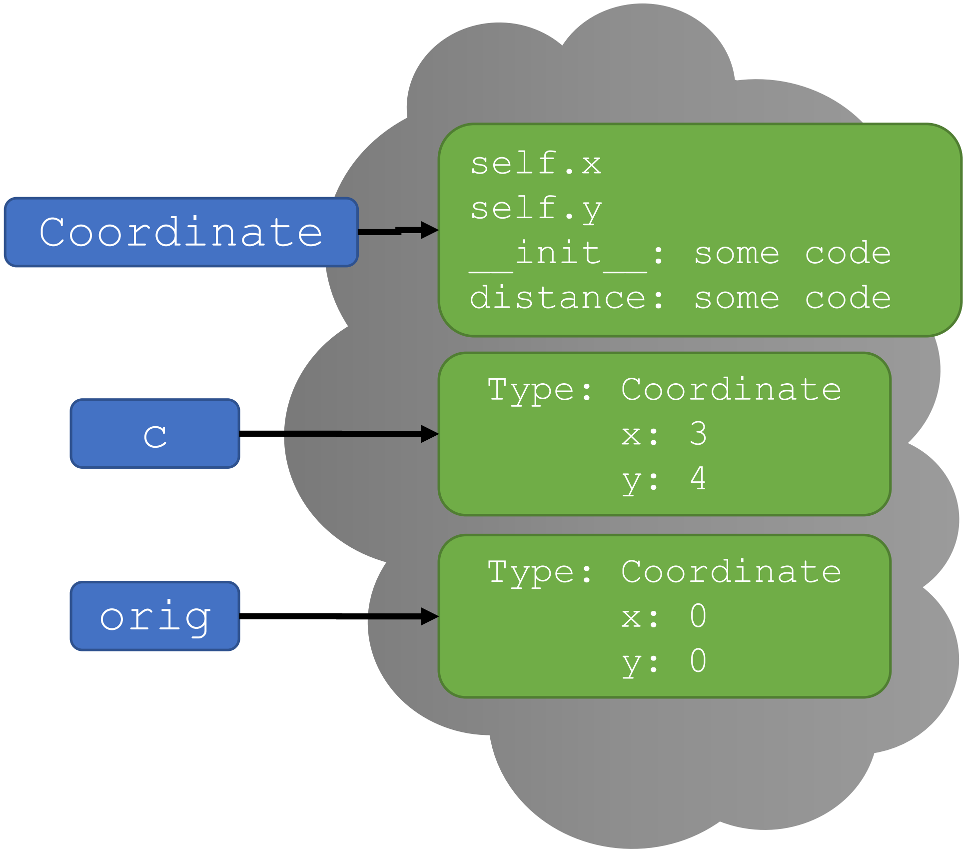
parameters not including self (self is implied to be c)

- Notice that `self` becomes the object you call the method on (the thing before the dot!)

VISUALIZING INVOCATION

- Coordinate class is an object in memory
  - From the class definition
- Create two Coordinate objects

```
c = Coordinate(3,4)
orig = Coordinate(0,0)
```

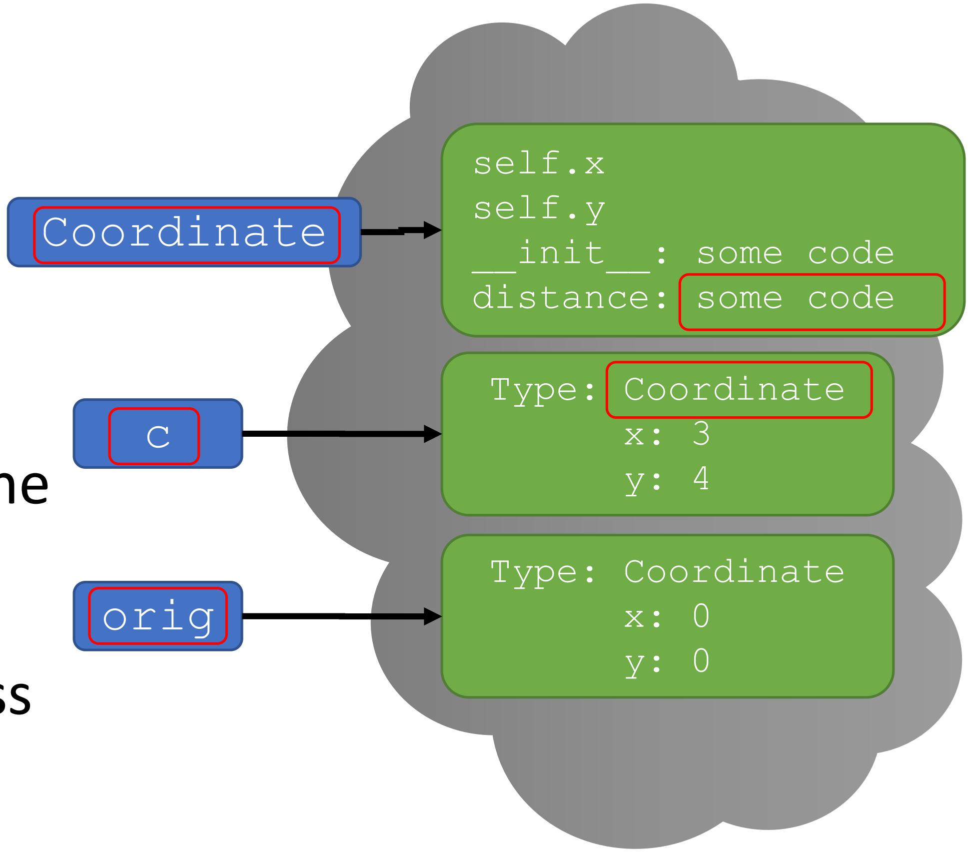


VISUALIZING INVOCATION

- Evaluate the method call

`c.distance(orig)`

- 1) The object is before the dot
- 2) Looks up the type of `c`
- 3) The method to call is after the dot.
- 4) Finds the binding for `distance` in that object class
- 5) Invokes that method with
  `c` as `self` and
  `orig` as `other`



HOW TO USE A METHOD

- Conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
c.distance(zero)
```

object to call method on, this is self in the class def

name of method

parameters not including self (self is implied to be c)

- Equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
Coordinate.distance(c, zero)
```

name of class (NOT an object of type Coordinate)

name of method

parameters, including an object to call the method on, representing self

BIG IDEA

The . operator accesses either data attributes or methods.

Data attributes are defined with `self`.something

Methods are functions defined inside the class with `self` as the first parameter.

THE POWER OF OOP

- **Bundle together objects** that share
  - Common attributes and
  - Procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object vs how to use the object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes