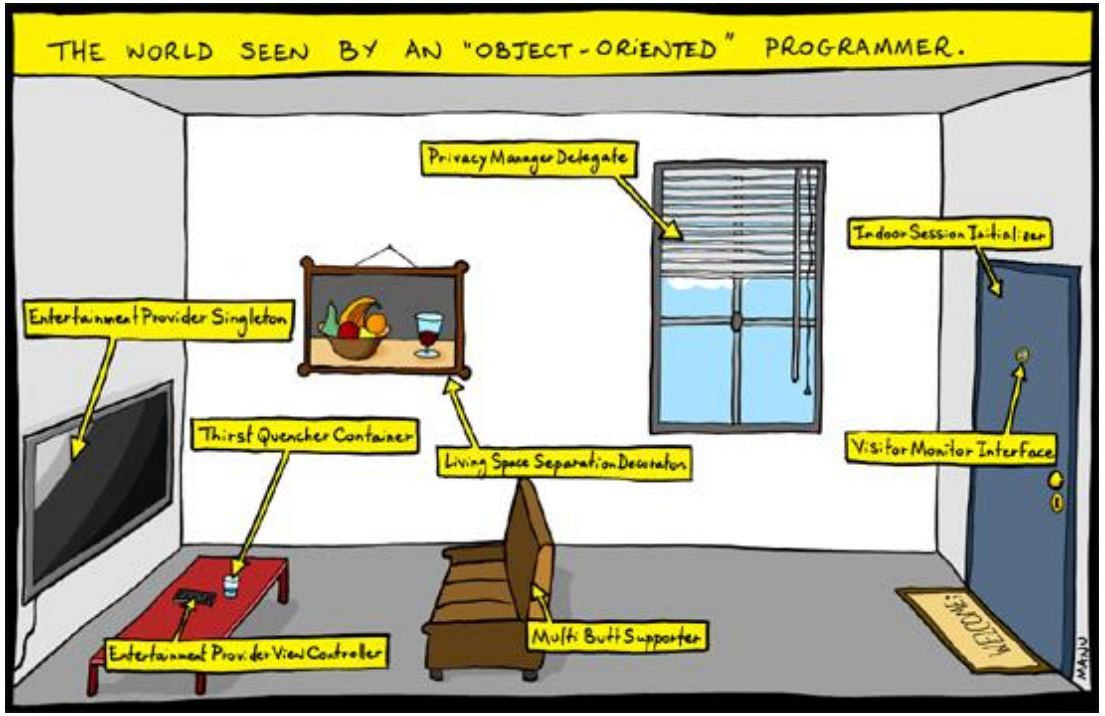
CSc 110, Spring 2017

Lecture 32: Objects

Adapted from slides by Marty Stepp and Stuart Reges



Pseudocode for finding the distance – Version1

initialize a current set of friends to name1
initialize distance to zero
while name2 not found in current set of friends
    increment the distance
    make a new set of friends from the current set using the dictionary
        to reference the sets of friends
    set the current set of friends to the union of the current set and new set of friends

print the distance

Sarah to Joshua

- This works but what if we looked for someone out of the friend network?
- What is the problem with `current_friends`?

```
new_friends
{'Christopher', 'Andrew', 'Emily'}
current_friends
{'Christopher', 'Sarah', 'Andrew', 'Emily'}
new_friends
{'Sarah', 'Ashley', 'Andrew', 'Emily', 'Jacob', 'Joshua',
 'Christopher'}
current_friends
{'Ashley', 'Jacob', 'Joshua', 'Sarah', 'Andrew', 'Emily',
 'Christopher'}
distance is:  2
```

We are never removing names that we have already seen.

Pseudocode for finding the distance – Version2

initialize a current set of friends to name1
Initialize a set of already seen friends to name1
initialize distance to zero
while name2 not found in current set of friends and length of current friends not zero
    increment the distance
    make a new set of friends from the current set using the dictionary
        to reference the sets of friends
    already seen friends is assigned to the union of itself and current friends
    set the current set of friends to the new set of friends minus the already seen friends

if the length of the current set of friends is not zero
    print the distance
else
    print not connected

```
# Reads in a dot file with friendship data - Version2
def main():
    file = open("friends.dot")
    lines = file.readlines()
    friends = create_dict(lines)
      name1 = input("Enter a name: ")
    name2 = input("Enter a name: ")

    #Are name1 and name2 friends?
    current_friends = {name1}
    already_seen = {name1}
    distance = 0
    # stops when the friend is found or there is no possibility of a connection
    while(name2 not in current_friends and len(current_friends) != 0):
        distance += 1
        new_friends = set()
        # builds up a set of the friends of the current friends
        for friend in current_friends:
            new_friends = new_friends | friends[friend]
        already_seen = already_seen | current_friends
        # replaces current friends and gets rid of friends looked at before
        current_friends = new_friends - already_seen

    if(len(current_friends) != 0):
        print("found at distance " + str(distance))
    else:
        print("sorry they are not connected")
```

Objects

- To human beings, an object is:

  "A tangible and/or visible thing; or,

  (a computer, a chair, a noise)

  Something that may be apprehended intellectually; or,

  (the intersection of two sets, a disagreement)

  Something towards which thought or action is directed"

  (the procedure of planting a tree)

  — Grady Booch

Objects

- Objects have state and behavior
  - the state of an object can influence its behavior
  - the behavior of an object can change its state
- State:

  All properties of an object and the values of those properties.
- Behavior:

  How an object acts and reacts, in terms of changes in state and interaction with other objects.

- **object**: An entity that combines state and behavior.

The Class concept

- It is often useful to think of objects as being members of a class:

  a set of objects having the same behavior and underlying structure

- A class is a template for defining a new type of object

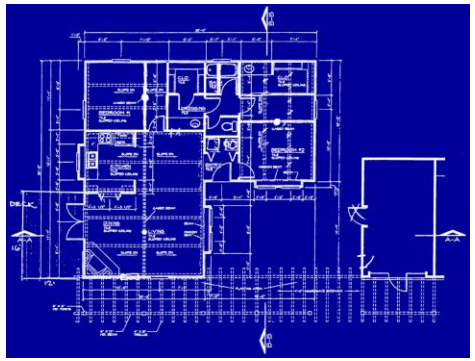
An object is an instance of a class.

Blueprint analogy

iPod blueprint

state:
- current song
- volume
- battery life

behavior:
- power on/off
- change station/song
- change volume
- choose random song



used to create instances of an iPod

iPod #1

state:
- song = "1,000,000 Miles"
- volume = 17
- battery life = 2.5 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod #2

state:
- song = "Letting You"
- volume = 9
- battery life = 3.41 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

iPod #3

state:
- song = "Discipline"
- volume = 24
- battery life = 1.8 hrs

behavior:
- power on/off
- change station/song
- change volume
- choose random song

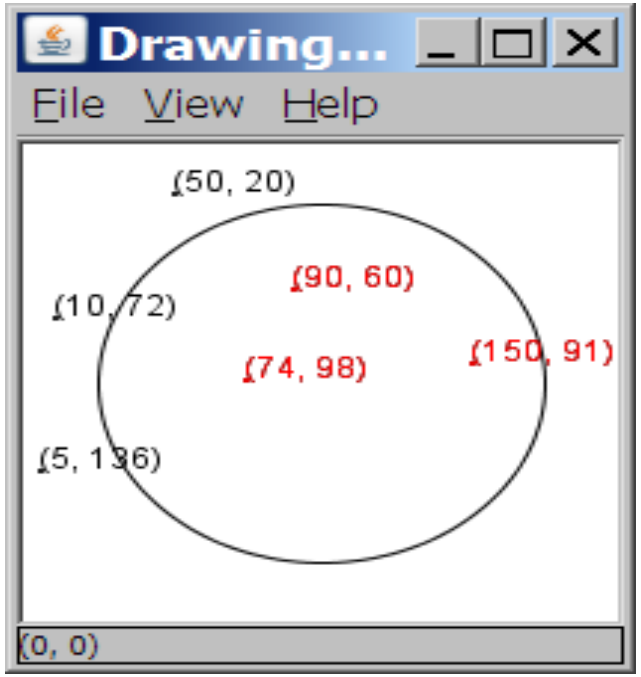
Classes

- In Python, that blueprint is expressed by a class definition
- A *class* describes the state and behavior of similar objects
- The *attributes* of a class represent the state of an instance
- The *methods* of a class describe the behavior

Recall earthquake program

- Given a file of cities' names and (x, y) coordinates:

```
Winslow 50 20
Tucson 90 60
Phoenix 10 72
Bisbee 74 98
Yuma 5 136
Page 150 91
```



- Write a program to draw the cities on a `DrawingPanel`, then simulates an earthquake that turns all cities red that are within a given radius:

```
Epicenter x? 100
Epicenter y? 100
Affected radius? 75
```

Observations

- The data in this problem is a set of points.
- Used tuples before. Now use objects with state and behavior.

- A `Point` object:

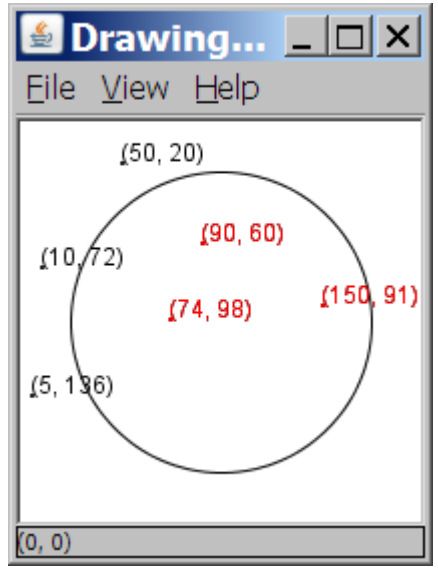
  attributes (state):

  a city's x/y data

  methods (behavior):

  Draw its x/y location on a `DrawingPanel` object

  Compare the distances between `Points`
  to see whether the earthquake hit a given city



`Point` objects (desired)

```
p1 = Point()
p2 = Point()
```

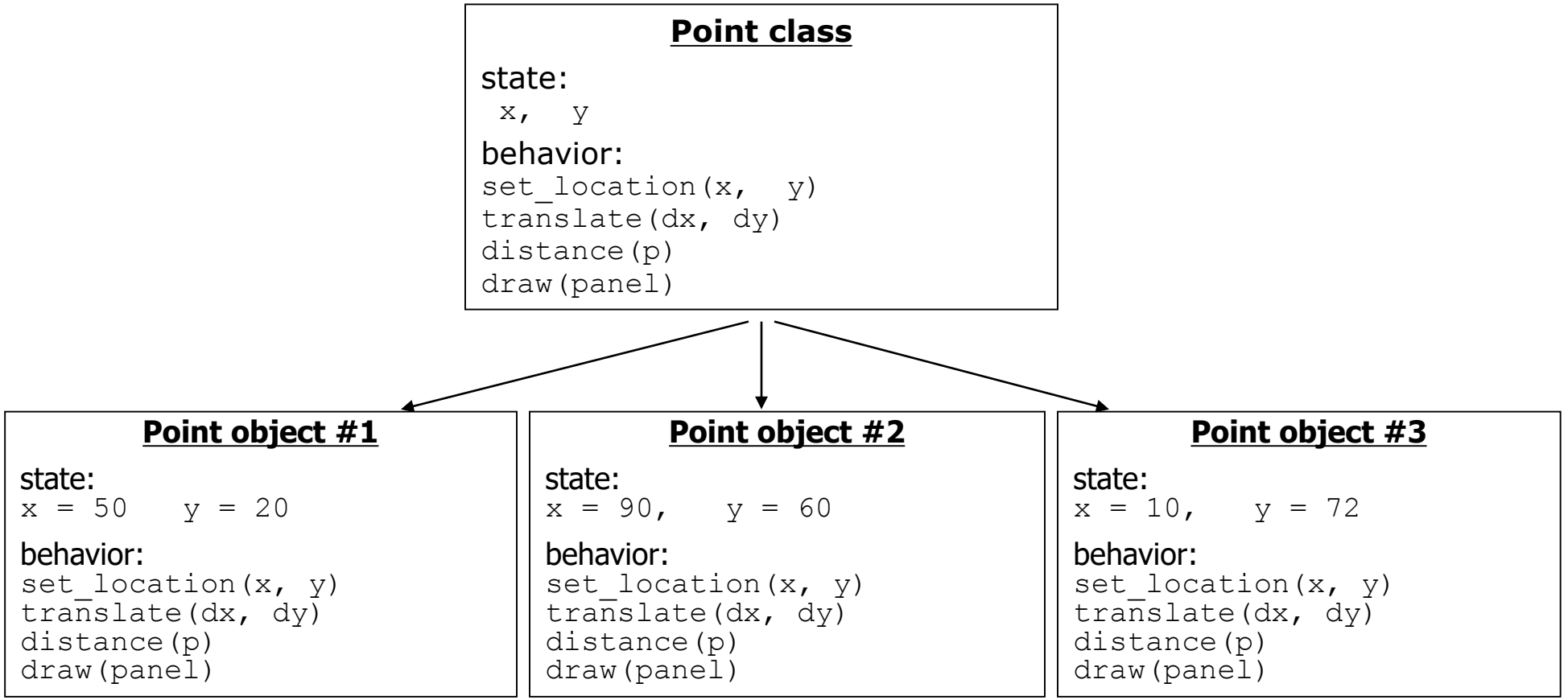
- Attributes of each `Point` object:

attribute	Description
`x`	the point's x-coordinate
`y`	the point's y-coordinate

- Methods in each `Point` object:

Method name	Description
`set_location(`**x, y**`)`	sets the point's x and y to the given values
`translate(`**dx, dy**`)`	adjusts the point's x and y by the given amounts
`distance(`**p**`)`	how far away the point is from point *p*
`draw(`**panel**`)`	displays the point on a drawing panel

`Point` class as blueprint



- The class (blueprint) will describe how to create objects.
- Each object will contain its own data and methods.

Attribute Syntax

- **attribute**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each attribute
  - Also called *an instance variable*

- Declaration syntax:

```
self.name = value
```

Method Syntax

- **method** : Defines the behavior of objects.

```
def name(self, parameters, ...):
    statements
```

  - Same syntax as functions, but with an extra `self` parameter

  - There is a special method that is called when an object is created
  - Used to initialize the object's instance variables

```
def __init__(self, parameters, ...):
    statements
```

Point class, version 1

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0
```

- The above code defines a new type named `Point`.
  - Each `Point` object contains two pieces of data:
    - an `int` named `x`, and
    - an `int` named `y`.
  - `__init__` method initializes `x` and `y`

Point class, version 1

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0
```

Given this version of the `Point` class, every `Point` object will have an x and y set to 0.

Using the Point class

- Create a new Point object:

```
p1 = Point()
```

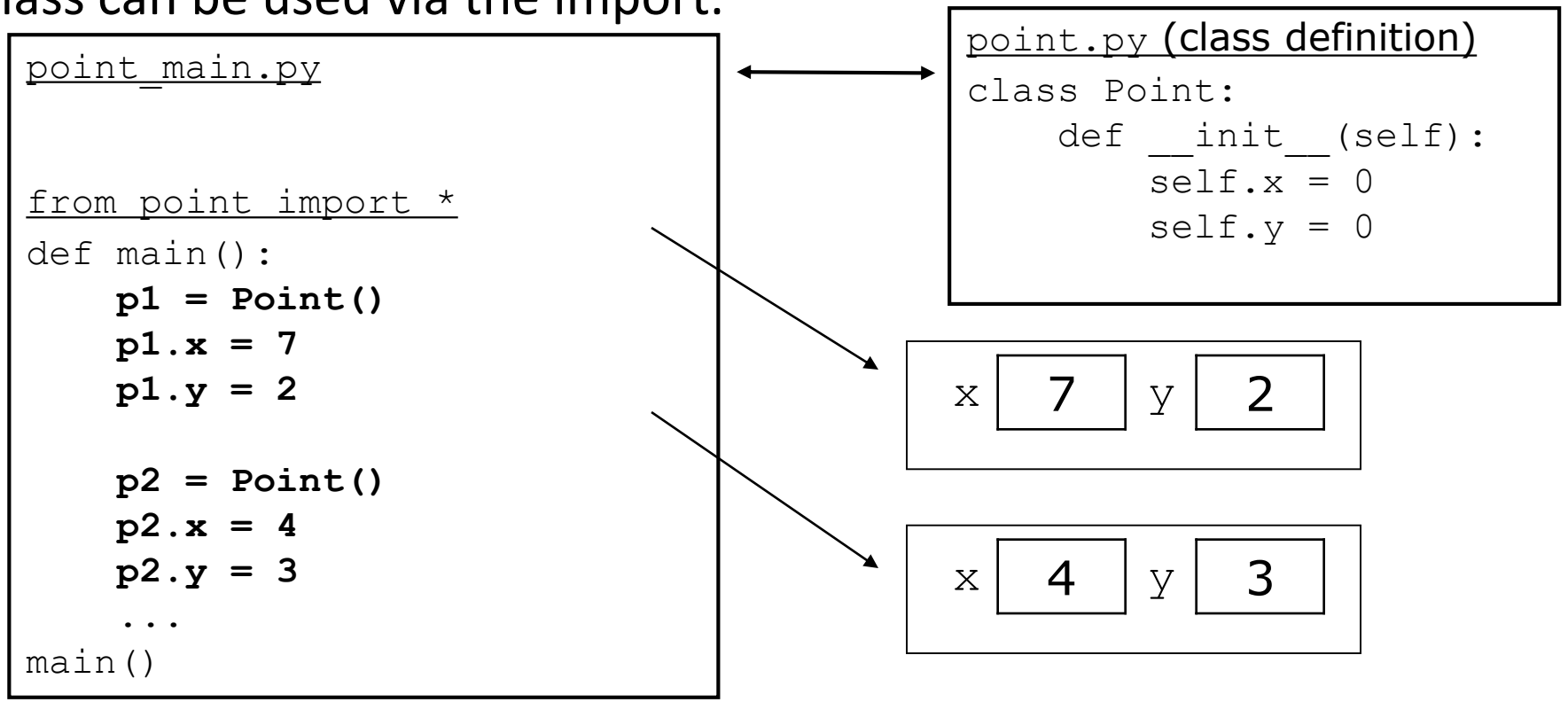
- access/modify an object's instance variables (attributes)
  - access: **variable . attribute**
  - modify: **variable . attribute = value**

- Example:

```
p1 = Point()
p2 = Point()
print("the x-coord is ", p1.x)     # access
p2.y = 13                          # modify
```

importing a Class definition

- Assume that class `Point` is in file `point.py`
  - A class can be used via the import.



Using `Point` objects

```
def main():
    # create two Point objects
    p1 = Point()
    p1.y = 2
    p2 = Point()
    p2.x = 4

    print(p1.x , p1.y)    # 0, 2

    # move p2 and then print it
    p2.x += 2
    p2.y += 1
    print(p2.x,p2.y)      # 6, 1
```

Implementing the `draw` method

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0

    # Draws this Point object on the given panel
    def draw(self, panel):
        panel.canvas.create_rectangle(x, y, x + 3, y + 3)
```

- How will the method know which point to draw?
  - How will the method access that point's x/y data?

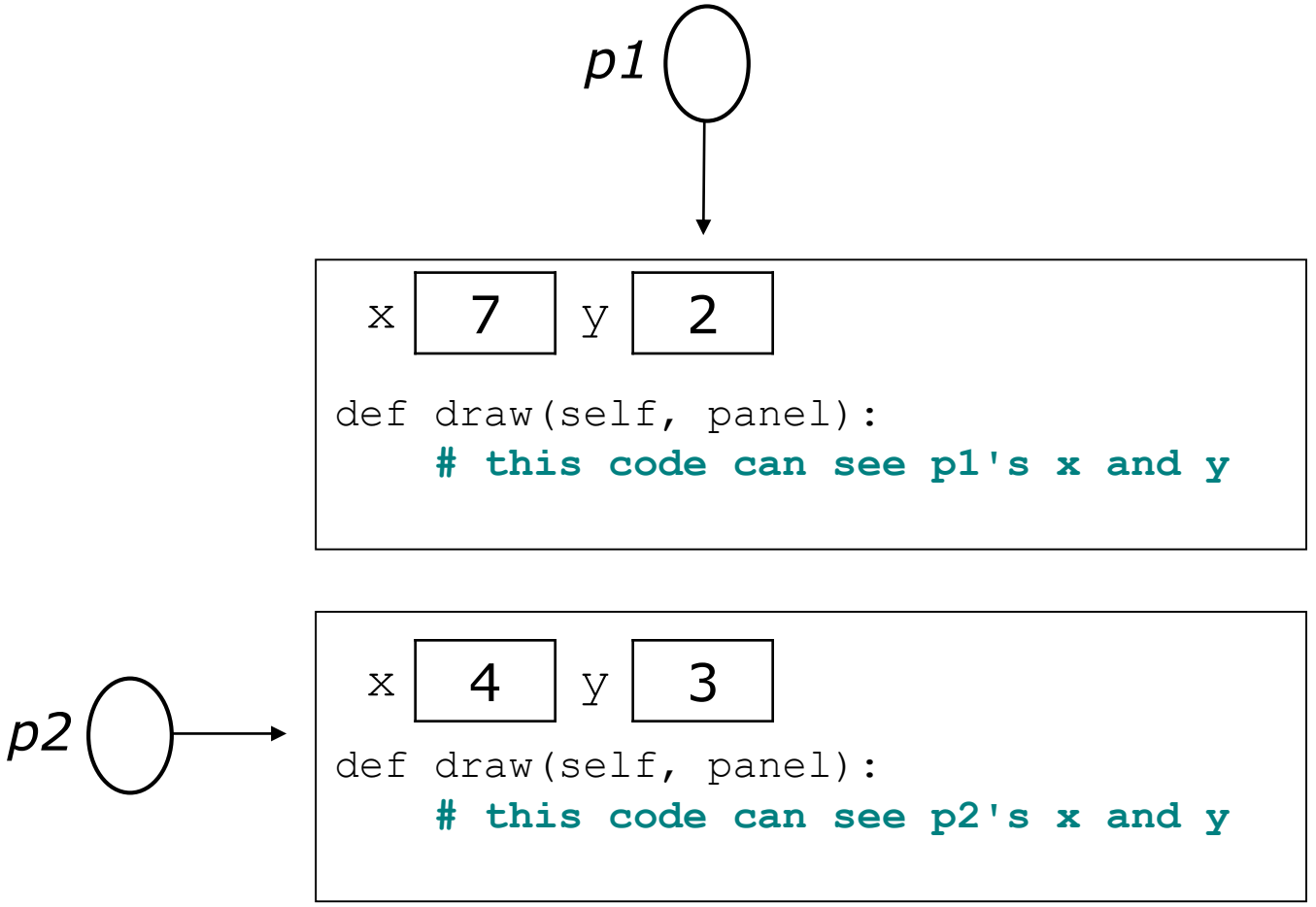
`Point` objects

- The object instance is passed as the first argument to the `draw` method, which operates on the object's state:

```
p1 = Point()
p1.x = 7
p1.y = 2

p2 = Point()
p2.x = 4
p2.y = 3

p1.draw(panel)
p2.draw(panel)
```



The implicit parameter

- **implicit parameter**:
  The object on which an instance method is called.
  - During the call `p1.draw(panel)`
    the object referred to by `p1` is the implicit parameter.
  - During the call `p2.draw(panel)`
    the object referred to by `p2` is the implicit parameter.
  - The instance method can refer to that object's fields.
    - We say that it executes in the *context* of a particular object.
    - `draw` can refer to the `x` and `y` of the object it was called on.

`Point` class, version 2

```
class Point:
    def __init__(self):
    self.x = 0
    self.y = 0

    # Draws this Point object on the given panel
    def draw(self, panel):
        panel.canvas.create_rectangle(x, y, x + 3, y + 3)
        panel.canvas.create_string("(" + str(x) + ", " +
                                   str(y) + ")", x, y)
```

The Object Concept

- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects

Class method questions

- Write a method `translate` that changes a `Point`'s location by a given *dx*, *dy* amount.

- Write a method `distance_from_origin` that returns the distance between a `Point` and the origin, (0, 0).

  Use the formula:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

  - Modify the `Point` class to use these methods.

Class method answers

```
class Point:
    def __init__(self):
        self.x
        self.y

    def translate(self, dx, dy):
        x = x + dx
        y = y + dy

    def distance_from_origin(self):
        return sqrt(x * x + y * y)
```