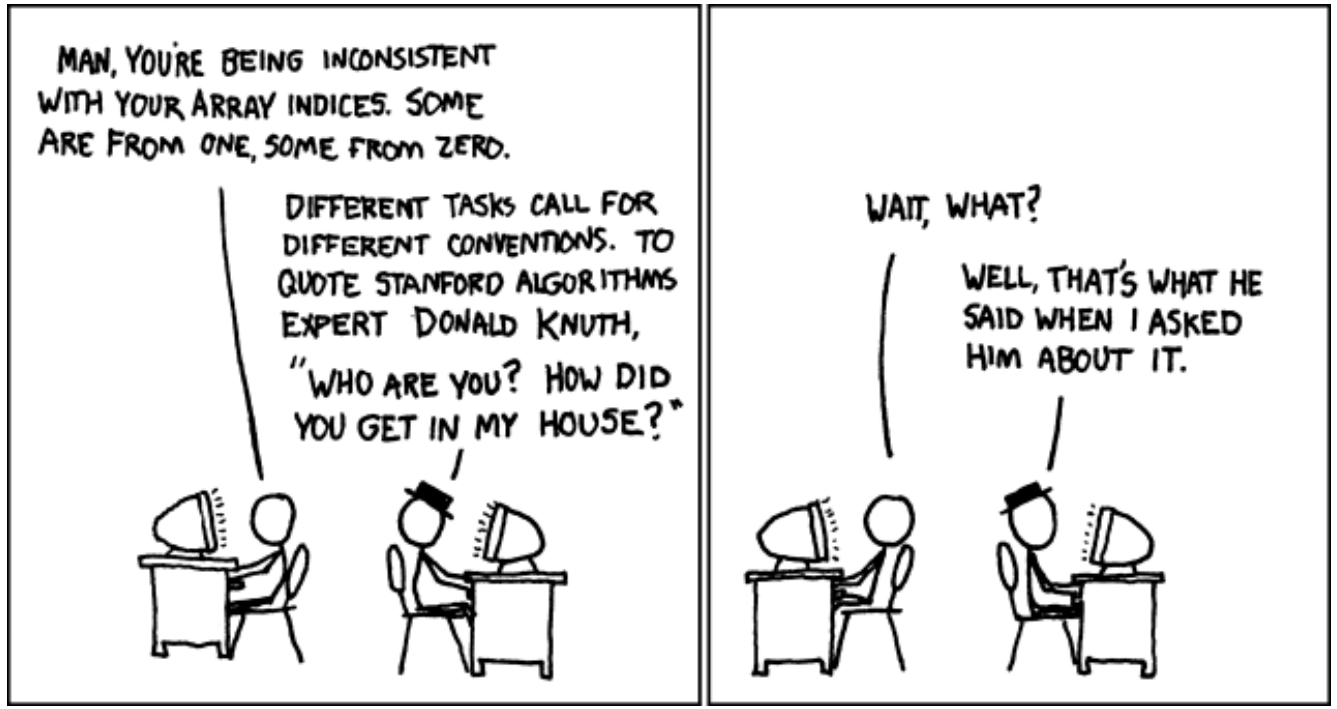
CSc 110, Autumn 2016

Lecture 30: Methods

Adapted from slides by Marty Stepp and Stuart Reges



Why objects?

- Primitive types don't model complex concepts well
  - Cost is a float. What's a person?
  - Classes are a way to define new types
  - Many objects can be made from those types

- Values of the same type often are used in similar ways
  - Promote code reuse through methods

Initializing objects

- Currently it takes 3 lines to create a `Point` and initialize it:

```
p = Point()
p.x = 3
p.y = 8                          # tedious
```

- We'd rather specify the fields' initial values at the start:

```
p = Point(3, 8)    # desired; doesn't work (yet)
```

  - We are able to this with most types of objects in Python.

Client code redundancy

- Suppose our client program wants to draw `Point` objects:

```
# draw each city
p1 = Point()
p1.x = 15
p1.y = 37
panel.canvas.create_oval(p1.x, p1.y, p1.x + 3, p1.y + 3)
panel.canvas.create_string(p1.x, p1.y, "(" + p1.x + ", " + p1.y + ")")
```

- To draw other points, the same code must be repeated.
  - We can remove this redundancy using a method.

Eliminating redundancy, v1

- We can eliminate the redundancy with a method:

```
# Draws the given point on the DrawingPanel.
def draw(self, p, panel):
    panel.canvas.create_oval(p.x, p.y, p.x + 3, p.y + 3)
    panel.canvas.create_string("(" + str(p.x) + ", " + str(p.y) + ")", p.x, p.y)
```

- `main` would call the method as follows:

```
draw(p1, g)
```

Problems with function solution

- We are missing a major benefit of objects: code reuse.
  - Every program that draws `Point`s would need a `draw` method.

- The syntax doesn't match how we're used to using objects.

```
draw(p1, panel)     # function (bad)
```

- The point of classes is to combine state and behavior.
  - The `draw` behavior is closely related to a `Point`'s data.
  - The method belongs *inside* each `Point` object.

```
p1.draw(panel)      # inside the object (better)
```

Instance methods

- **method** (or **object function**): Exists inside each object of a class and gives behavior to each object.

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

  - same syntax as functions, but with a `self` parameter

  Example:

```
def shout():
    print("HELLO THERE!")
```

`Point` class, version 2

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

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

- Each `Point` object contains a `draw` method that draws that point at its current `x`/`y` position.

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` and client code to use these methods.

Class method answers

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

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

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

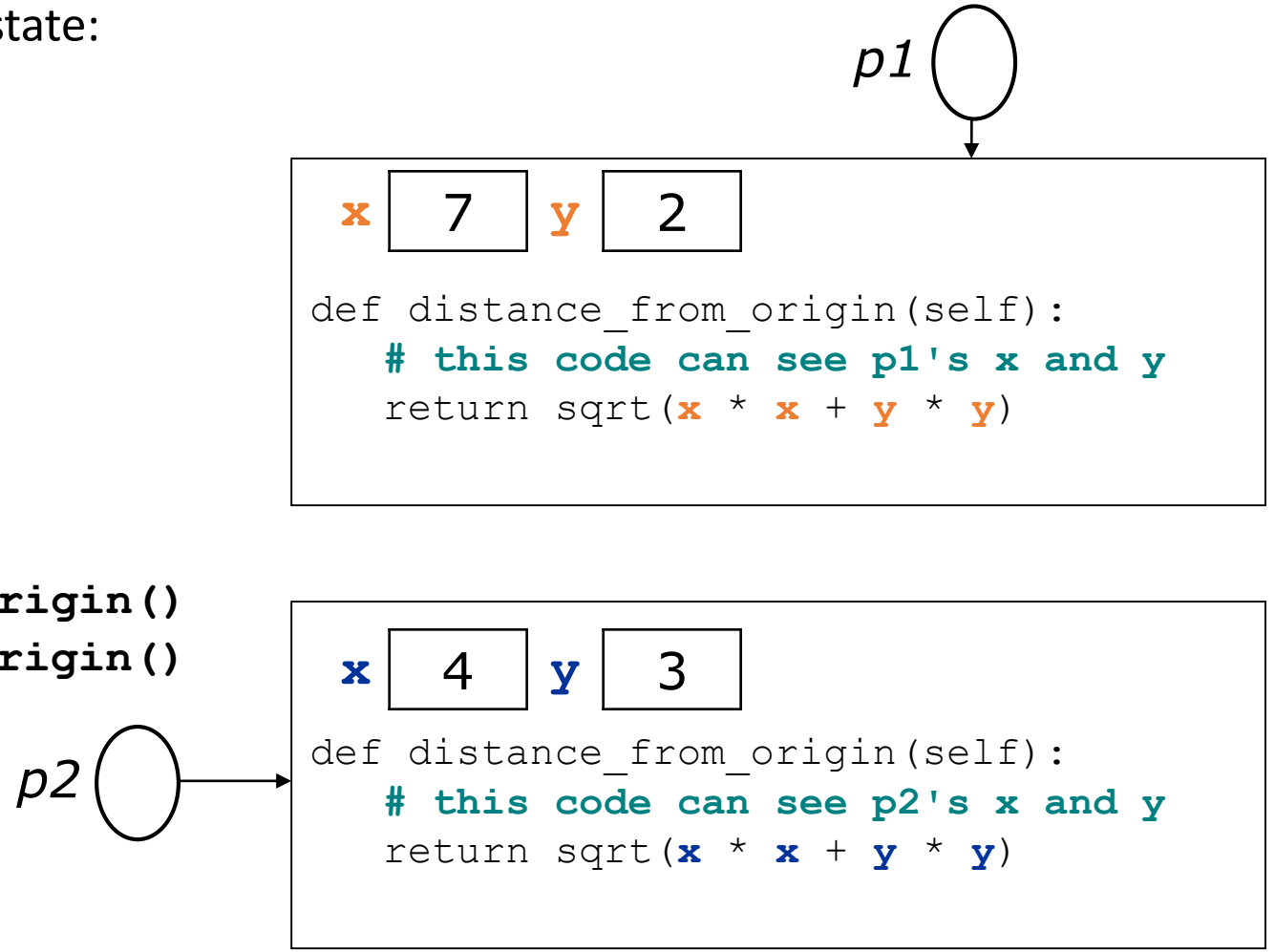
`Point` objects w/ method

- Each `Point` object has its own copy of the `distance_from_origin` method, which operates on that object's state:

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

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

p1.distance_from_origin()
p2.distance_from_origin()
```



Kinds of methods

- **accessor**: A method that lets clients examine object state.
  - Examples: `distance`, `distance_from_origin`
  - often returns something

- **mutator**: A method that modifies an object's state.
  - Examples: `set_location`, `translate`

Printing objects

- By default, Python doesn't know how to print objects:

```
p = Point()
p.x = 10
p.y = 7
print("p is " + str(p))  # p is
                         # <p.Point object at 0x000001BA6AE0BF28>

# better, but cumbersome;              p is (10, 7)
print("p is (" + str(p.x) + ", " + str(p.y) + ")")


# desired behavior
print("p is " + str(p))  # p is (10, 7)
```

The `__str__` method

tells Python how to convert an object into a `string`

```
p1 = Point(7, 2)
print("p1: " + str(p1))
```

- Every class has a `__str__`, even if it isn't in your code.

```
<point.Point object at 0x000001BA6AE0BF28>
```

`__str__` syntax

```
def __str__(self):
    code that returns a String representing this object
```

- Method name, return, and parameters must match exactly.
- Example:

```
# Returns a String representing this Point.
def __str__(self):
    return "(" + str(x) + ", " + str(y) + ")"
```