Intro to Computer Science

Previous

- Objects

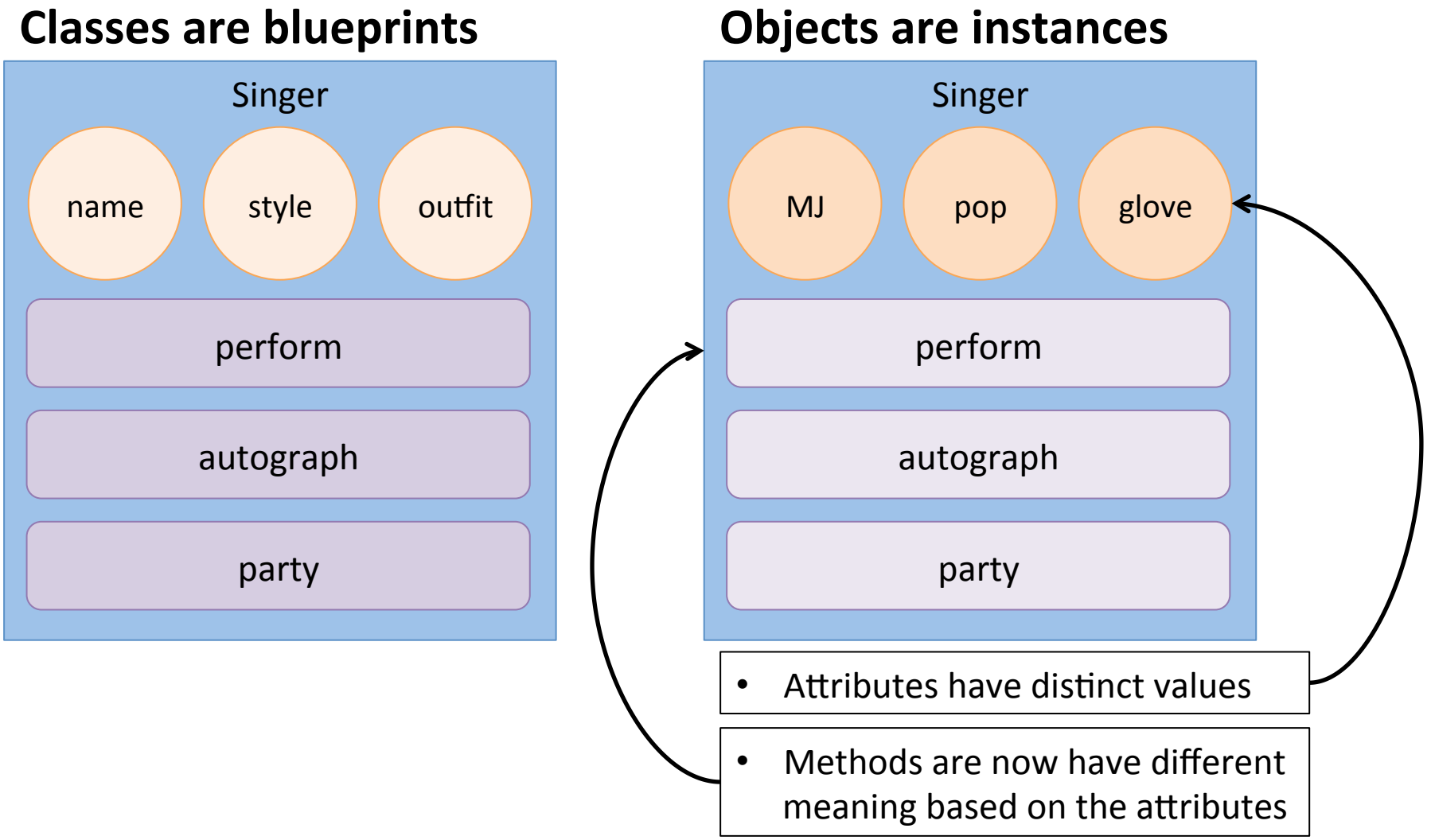
Next

- Objects (review)

Readings	
Gaddis	• Chapter 10

Readings	
Gaddis	• Chapter 10

Class versus object



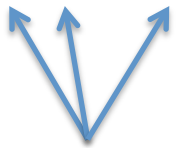
Instantiation (again)

You call

```
s = Singer()
```

Python does

```
class Singer:
    def __init__(self):
        return
```



What to notice:

1. Function name is special
   - Cannot change this!
2. Double underscore syntax
3. Obligatory `self` parameter

Instantiation (again)

You call

```
bieb = Singer('Bieber')
print(bieb.name)

mj = Singer('MJ')
print(mj.name)
```

Python does

```
class Singer:
    def __init__(self, name):
        self.name = name
        return
```

- `name` is now an additional variable in the `Singer` class
  - It can be used throughout the class, akin to a global variable within the class
- This assignment happens during instantiation
- Each particular instance will have the value that is specified

__ functions get called automatically

You call	Python does
`bieb = Singer('Bieber')`	`beib = Singer.__init__(name)`
`mj = Singer('MJ')`	
`print(bieb)`	`print(str(bieb))`
	`print(Singer.__str__(bieb))`
`bieb > mj`	`Singer.__gt__(bieb, mj)`
`madonna = bieb + mj`	`Singer.__add__(bieb, mj)`

Scope

- Methods and attributes *live* inside the class
- They stick around for as long as the variable sticks around
- Just like other values:
  - lists
  - strings
  - dictionaries
- You're accessing variables and functions that are *bound* to that class
  - Reaching in to a mini program that retains its state

self: reason 1

```
bieb = Singer('Bieber')
x = bieb.autograph()
y = bieb.name
```

- Methods stick around all the time
- Variables can stick around all the time as well

```
class Singer:
    def __init__(self, n):
        self.name = n

    def autograph(self):
        return self.name

    def party(self):
        self.name = 'prince'
```

- self helps Python know what name we are talking about
  - The current instances name

self: reason 2

You call	**Python does**
`bieb = Singer('Bieber')`	
`bieb.party()`	`Singer.party(bieb)`
`beib.turn_up(11)`	`Singer.turn_up(bieb, 11)`

- Python, internally, needs to know which instance of the class it's operating on:
  - `self` allows this to be distinguished

Journey through programming

- Started with sequential programming
  - Top-down collection of statements
  - Run one after the other
- Introduced control structures
  - Repeat certain actions (loops)
  - Decide when to execute certain actions (if-then)
  - Group certain actions (functions)
- Usage of functions known as *procedural* programming

Procedural programming

The good

- Modularity
  - Functions perform specific task
  - Operate over a well defined set of input
  - Have a distinct output
- Scoping
  - Protection of variables
- Reuse

The not so good

- Writing procedures around data types provided by the language

Example: the entourage

- To make data-types “do” what you want, you have to write code that takes all of the information

```
update(board, row, col)
```

- Essentially using separate variables to manage state
  - Gets hairy when a data-type has lots of extraneous information

```
draw_line(map, x1, y1, x2, x2)
```

The entourage is fragile

- Complex data is “faked” by carrying around all the built-in types they require
- In doing this, we write *fragile* programs
  - If something about the abstract data type changes, there’s lots of code to change

Fragile is as fragile does

- Move from 2D to tic-tac-toe to 3D tic-tac-toe
- Must change all functions that operate on a board to now take a third argument
  - Imagine thousands of files
  - Used by hundreds of people
  - #painful

A better way

- Have an abstract data type that everyone uses and only you manage
  - Everyone: board
  - You: lists of lists, contains strings, now 3D…
- Know as *encapsulation*

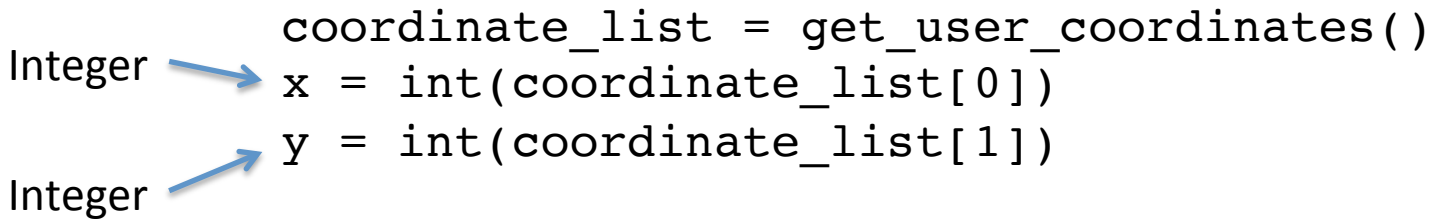
Example: this is not natural

- Additional *exposed* code to make what Python sees into what you need

```
user_coordinate = input('Coordinate 1 (x, y)')
coordinates = user_coordinate.split(',')
x = int(coordinates[0])
y = int(coordinates[1])
```

Add a function?

Sure, we could add a function…



… but in real life, these are *coordinates*!

- They have a distinct identity

➢ Remember: we want real-world abstractions to match programming formalisms

We need more than functions

Even if we used a function we're still missing the point:

- Still using integers to "talk" about something that is not a synonym for an integer
- In life we don't talk in terms of lists and strings
  - We abstract away what's underneath

Examples:

- GPS coordinates
- Game boards

➢ Underneath these may integers, or sets of strings, but we don't talk about them that way
  - We talk about their usage

Solving our procedural problems

```
board = make_board()
while True:
  c = input('Coords')
  l = c.split(',')
  x = int(l[0])
  y = int(l[1])
  update_board(board,x,y)
```

```
board = Board()
while True:
  c = input('Coords')
  coord = Coord(c)
  board.update(coord)
```