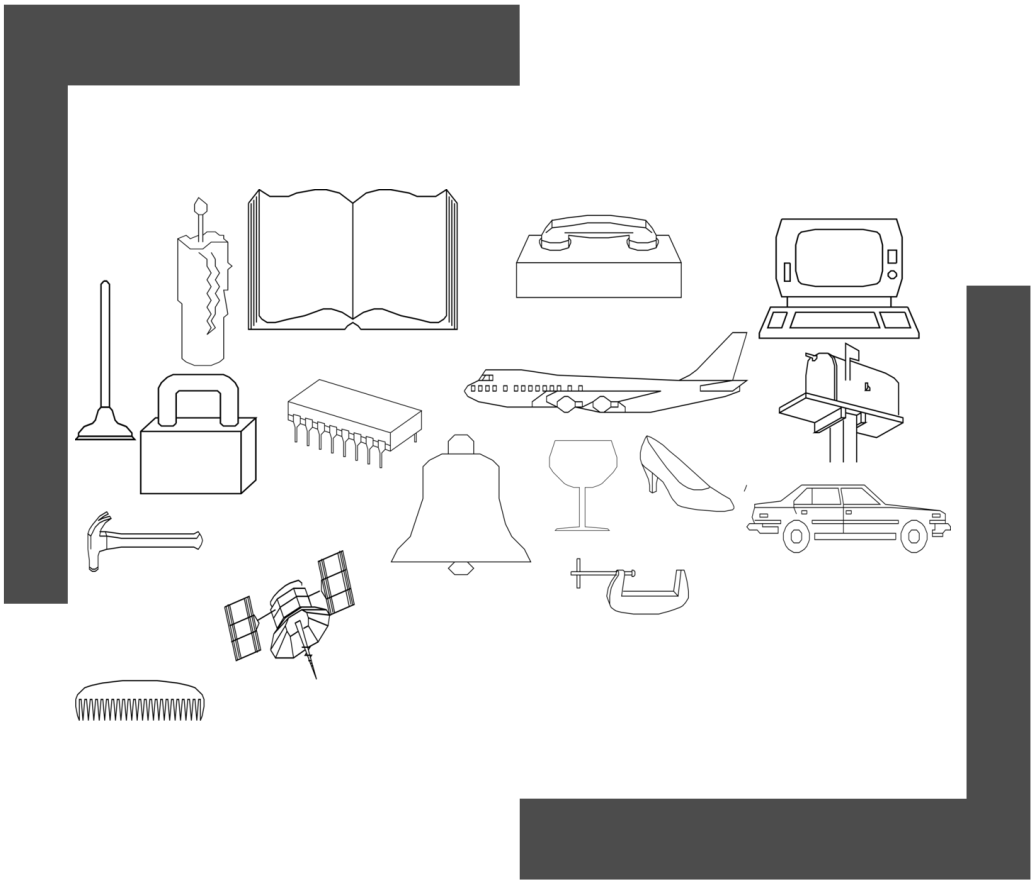
Objects. Introduction

Lecture 07.01

By Marina Barsky

What is an object?

Real objects vs. software objects



- Real objects in the real world have
  - things that they can do (*actions*, *methods*)
  - things that describe them (*attributes*, *properties*)
- **In programming, we have the same kind of thing**

What sounds more natural?

```
cook (microwave, chicken)
```

```
microwave.cook (chicken)
```

- The functionality of real-world objects tends to be tightly bound up **inside the objects themselves**

Change of perspective

- So far, we've been looking at different ways of organizing data and actions
  - *Lists* or *dictionaries* are a way to group variables (data) together
  - *Functions* are a way to group commands (actions) into a single unit of code and use it over and over again

- Now we will learn a way of **bundling both data and actions together in a single unit – an object**

Modeling a *Ball*

- Important ***attributes*** of a ball

color
radius
weight
shape

- Important ***actions*** of a ball

bounce
inflate
move

What are **attributes** (fields)?

- Attributes are all things you know (or can find out) about the ball
- The ball's attributes are chunks of data—numbers, strings, and so on
- They're just ***variables*** that are ***included inside the object***

```
print (ball.size)
ball.color = 'green'
my_color = ball.color
my_ball.color = your_ball.color
```

Dot – to show that a variable is a part of an object

What are **methods**?

- Methods are things you can make the object *do* or that you can *do* with an object
- They're chunks of code that you can *call* to do something
- ***Methods*** are just **functions** that are **included inside the object**
- You can do all the things with methods that you can do with any other function, including *passing arguments* and *returning values*

```
ball.bounce()
ball.move()
ball.inflate()
```

Object = attributes + methods

Data stored in **attribute variables**

```
ball.color = 'green'
ball.shape = 'round'
ball.radius = 20
```

Actions as **methods**

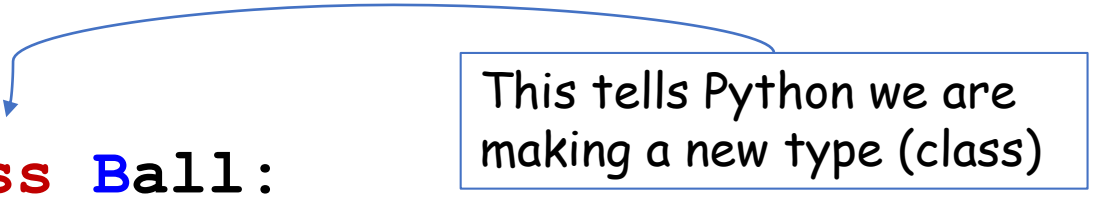
```
ball.bounce()
ball.move()
ball.inflate()
```

Creating objects of the same type (class)

There are **two steps** in creating any object

- Make object **template** – define a *class*
- Bake an **instance** of an object using the definition of a class as a blueprint

Step 1: define new type - *class* (template, blueprint)




```
class Ball:
```

This is a method

```
def bounce(self):
    if self.direction == "down":
        self.direction = "up"
```

Step 2: Create an instance (*object*) according to the template (*class*)

```
my_ball = Ball()
```

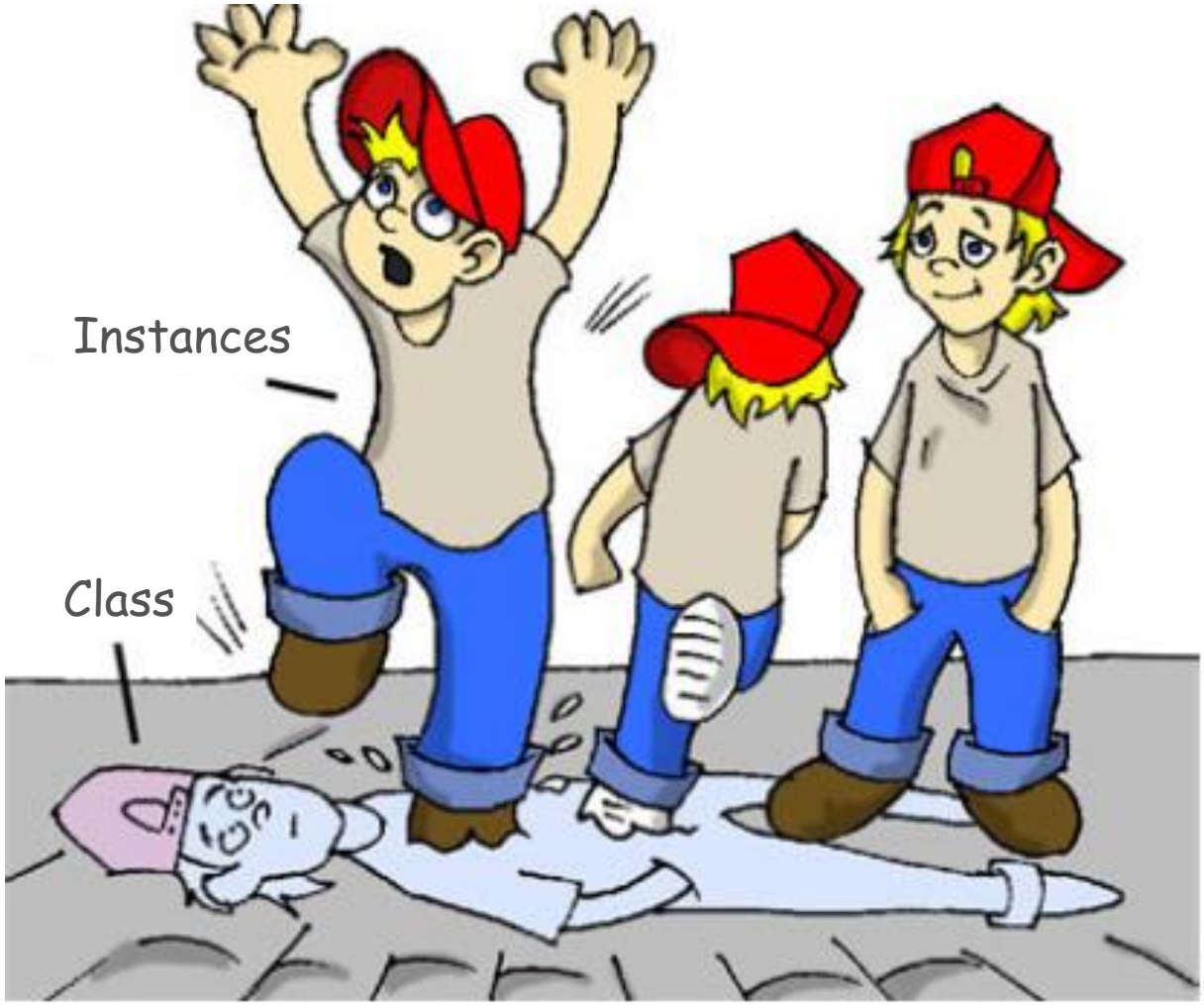
We can add new attributes directly to the object

```
my_ball.direction = "down"
```

This will behave according to the template in class Ball

```
my_ball.bounce()
```

Big difference: class and instances



Try it out

```
class Ball:
    def bounce(self):
        if self.direction == "down":
            self.direction = "up"


my_ball = Ball()
my_ball.direction = "down"
my_ball.color = "red"
my_ball.radius = 20
print("I just created a ball.")
print("Ball's diameter is", my_ball.radius*2, "inches")
print("My ball is", my_ball.color)
print("My ball's direction is", my_ball.direction)
print("Now I'm going to bounce the ball")
print()
my_ball.bounce()
print("Now the ball's direction is", my_ball.direction)
```

Initializing an object

- When you create the class definition, you can define a ***special*** method called **__init__()**
- That code will run whenever a new instance of the class is created. You can pass arguments to the **__init__()** method to set up the future object however you want

```
class Ball:
    def __init__(self, color, radius, direction):
        self.color = color
        self.radius = radius
        self.direction = direction
```

Here are setup instructions

This will call __init__

```
my_ball = Ball("red", 20, "down")
```

What's "self"?

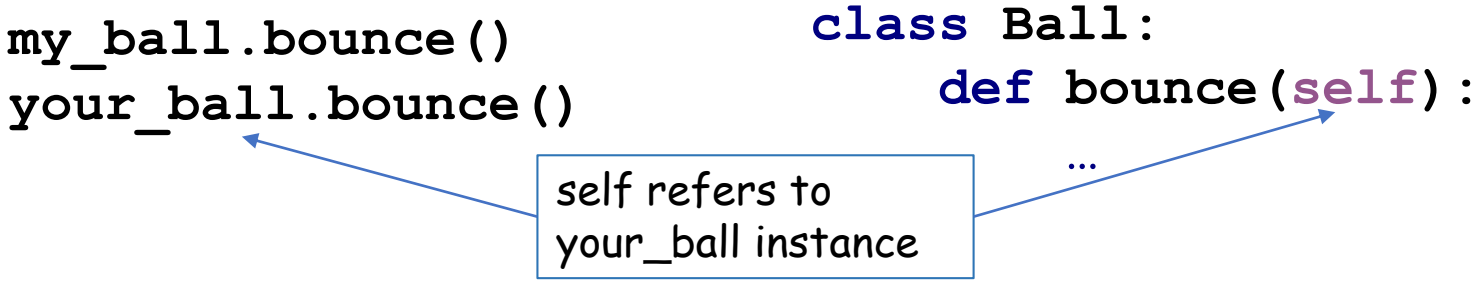
```
class Ball:
    def __init__(self, color, radius, direction):
        …

    def bounce(self):
        …
```

- We could use the class (blueprint) to create more than one ball:

```
my_ball = Ball("red", 20, "down")
your_ball = Ball("green", 10, "down")
```

- When we call a method for one of these instances, the method has to know which instance called it:
- Is it *my_ball* that needs to bounce, or *your_ball*?

Self is an *instance reference*



- The *self* argument is what tells the method which object called it
- Where did the reference to my_ball come from if we did not pass anything?
- When you call a class method, the information about which instance called—the *instance reference*—is automatically passed to the method using dot notation

```
my_ball.bounce()   same as   Ball.bounce(my_ball)
```

Printing the ball

```
print(my_ball)
```

- Default string representation for any new type of objects:

  <__main__.Ball object at 0x0173D410>

1. where the instance is defined (in __main__, which is the main part of the program)
2. the class name (Ball)
3. the memory location (the 0x0173D410 part)

We need to redefine (override) string representation of a Ball

- Special methods are surrounded with **double underscores**: **__init__**
- If we implement our own special **__str__** method this will override the default behavior of converting Ball to *str*

```
def __str__(self):
    msg = "Hi, I'm a {} ball with diameter " \
          "of {} inches".format(self.color,self.radius*2)
    return msg
```

Converts to str

```
print(my_ball)
```

Default arguments

```
class Ball:
    def __init__(self, color, radius=10,
                 direction="down"):
        self.color = color
        self.radius = radius
        self.direction = direction
```

- If radius is not provided, the default value is 10
- If direction is not provided, the default value is “down”

A list of balls

```
balls = [Ball("red", 20, "down"),
         Ball("green", 10),
         Ball("blue")]

print(balls)
```

What is direction of balls[1]?

What is radius of balls[2]?

```
[<__main__.Ball object at 0x017D4FB0>,
<__main__.Ball object at 0x017D4FD0>,
<__main__.Ball object at 0x017D4FF0>]
```

- We only defined how to convert a separate Ball to *str*, not how to make a string from a list of balls

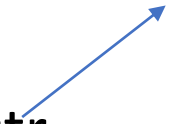
Another special method: __repr__

- To make sure that balls are always printed properly – whether they are elements of the list or of a dictionary or anything else – there is another special method __repr__

- **__str__** is intended for the user-readable string representation of an object
- **__repr__** is mostly used for debugging and complex nested objects

Printing list of balls

```
class Ball:
    def __init__(self, color, radius=10,
                 direction="down"):

    def __str__(self):
        msg = "Hi, I'm a {} ball with diameter " \
          "of {} inches".format(self.color,self.radius*2)
        return msg

    def __repr__(self):
        return self.__str__()
```



- Reused method __str__
- We could have implemented completely different __repr__

Virtual hotdogs

- Class ***HotDog***
- **Attributes**:
  - *cooked_level* – a number which shows how long it has been cooked
  - *condiments* – a list of what is on hotdog, like ketchup, mustard etc.

- **Methods**:
  - *__init__* – initializes hotdog to raw state
  - *__str__* – string representation of a hotdog
  - *cook()* – cooks hotdog for some period of time
  - *add_condiment()* – adds condiments to the hotdog

Setting it up: __init__

```
class HotDog():
    def __init__(self):
        self.cooked_level = 0
        self.condiments = []
        self.level_to_str = {(0,3):"raw",
                             (4,5):"medium",
                             (5,8):"well-done"}
```

Note the use of tuples as dictionary keys

Cooking time!

```
def cook(self, time):
    self.cooked_level += time

def add_condiment(self, condiment):
    self.condiments.append(condiment)
```

Print hot dog state

```
def __str__(self):
    state = "charcoal hot dog "
    for key,val in self.level_to_str.items():
        time_from, time_to = key
        if time_from <= self.cooked_level <= time_to:
            state = val + " hot dog "
            break
    s = state
    if len(self.condiments) > 0:
        s += "with:"
    for c in self.condiments:
        s += " " + c

    return s
```

Test your hotdog

```
hot_dog = HotDog()

for i in range(3):
    print("Cooking hot dog for 3 more minutes")
    hot_dog.cook(3)
    print(hot_dog)

print("Now I am going to add some condiments")
hot_dog.add_condiment('mustard')
hot_dog.add_condiment('pickles')
print(hot_dog)
```

Two ways of changing object state

- There are 2 possible ways of changing cooked level:
  - Assign directly:

```
hot_dog.cooked_level = 5
```

  - Use method to change an attribute:

```
hot_dog.cook(5)
```

- If we started with a raw hotdog, the result is the same
- Why did we bother to have a special method if we could do it directly?

Danger of accessing attributes directly

- If we were accessing the attributes directly, we could do some illegal assignments:

```
hot_dog.cooked_level -= 2
```

- But you cannot “uncook” the hotdog!

- Using the method, we make sure that our hotdogs behave logically
- The same for *add_condiment*: inside the method we can check that only proper condiments are being added to the hotdog – and we do not have this opportunity if we would add condiments directly to the list attribute:

```
hot_dog.condiments.append("milk")
```

Data hiding

- In programming terms, restricting the access to an object's data so you can only get it or change it by using methods is called ***data hiding***
- Python doesn't have any way to enforce data hiding, but you should write code that follows this rule
- This will protect your data fields (attributes) from illegal changes

Encapsulation

- Data hiding and protection of object's data from illegal changes is a part of a very important principle in OOP: *encapsulation*
- You expose methods to the user of your classes through method interface
- If you later decide to change the internal implementation of your class, the programs which use your classes would not need to change

Encapsulate your code under the stable interface

```
def add_condiment(self, condiment):
    self.condiments.append(condiment)
```

Condiments are stored in a list

changed to:

```
def add_condiment(self, condiment):
    self.condiments[condiment] += 1
```

Condiments are stored in a dictionary

- But the main program does not need to change – because the method interface remains unchanged

```
print("Now I am going to add some condiments")
hot_dog.add_condiment('mustard')
hot_dog.add_condiment('pickles')
print(hot_dog)
```

Summary: what to know for the next time

- Classes and objects – what is the difference
- Attributes and methods
- Blueprint for object initialization: *Class*
- String representation of an object: *dunder-str*
- Why use data hiding and encapsulation