CSCI 145 · Introduction to Computing · Fall 2026

Encapsulation, Inheritance, Polymorphism

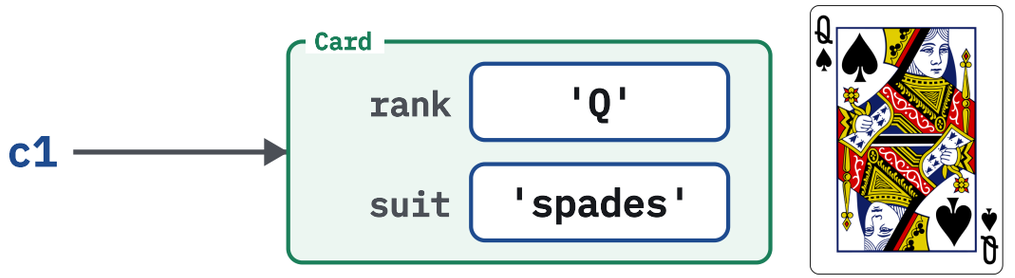
Lecture 19 — special methods, child classes, and overriding methods

Outline

- Encapsulation: special methods and a Rational class
- Inheritance: child classes
- Polymorphism: one loop, many kinds of objects

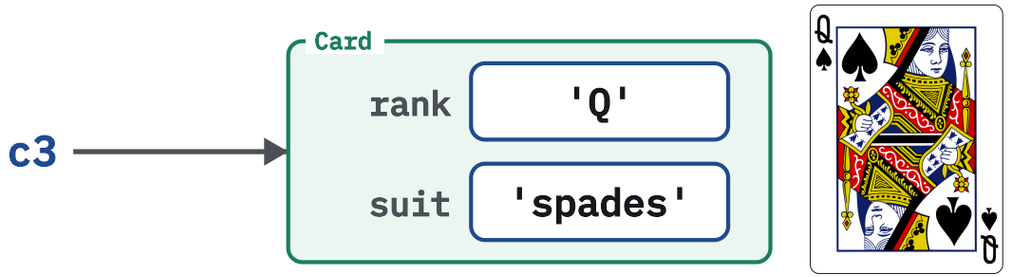
Two queens of spades: is `c1 == c3`?

```
c1 = Card("Q", "spades")
c3 = Card("Q", "spades")
print(c1 == c3)
```



OUTPUT

```
False
```



- Python doesn't know
  when two Cards are equal
  It checks: are they the same object?
- We'd rather get `True`

SOLUTION

Give the class a method that tells Python what `==` means for Cards

Teaching `Card` what `==` means

```
class Card:
    # __init__, __str__, value, and beats as before

    def __eq__(self, other):
        """Returns True if the two cards match"""
        return self.rank == other.rank and self.suit == other.suit
```

```
>>> c1 == c3
True
>>> c1 is c3
False
```

- `c1 == c3` calls `c1.__eq__(c3)`
- `self` is the left side
  `other` is the right side
- `is` still asks: same object?

Part 1 of 3

Encapsulation

> An object bundles its data and the methods that work on it: **encapsulation**
>
> Code outside the class uses the methods, not the attributes
>
> **Special methods** let `print`, `==`, and `+` use them too

Special methods: Python calls them for us

YOU WRITE		PYTHON CALLS
`print(c1)`	*calls*	`c1.__str__()`
`c1 == c3`	*calls*	`c1.__eq__(c3)`
`a + b`	*calls*	`a.__add__(b)`
`a * b`	*calls*	`a.__mul__(b)`
`a < b`	*calls*	`a.__lt__(b)`

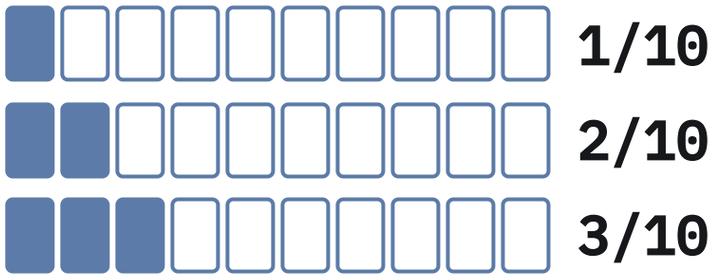
- Two underscores on each side
- Write them, and your objects work with Python's operators
- Next: a class that does arithmetic

Floats aren't exact

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

Lecture 16: most real numbers can't be stored exactly in binary

Fractions are exact



Store a fraction as two ints: the top and the bottom

A `Rational` class

`Rational` An exact fraction, in lowest terms

Attribute	Example value
`num`	3
`den`	5

Method call	What it does or returns
`print(r)`	prints **`3/5`**
`r1 == r2`	**`True`** if they are equal
`r1 + r2`	returns a new **`Rational`**
`r1 * r2`	returns a new **`Rational`**



3 / 5

`num` is the numerator

`den` is the denominator

`__init__` constructs a reduced fraction

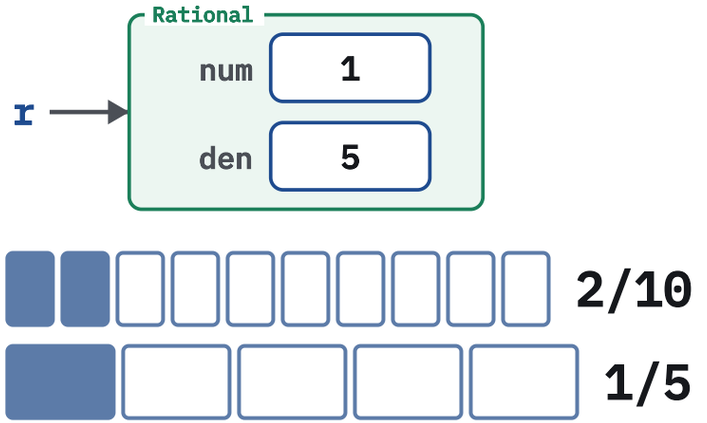
```
import math

class Rational:
    """An exact fraction, in lowest terms"""

    def __init__(self, num, den):
        g = math.gcd(num, den)
        self.num = num // g
        self.den = den // g
```

```
>>> math.gcd(2, 10)
2
>>> r = Rational(2, 10)
>>> print(r.num, r.den)
1 5
```

- `gcd` : greatest common divisor
  2 divides both 2 and 10
- `//` keeps them ints



Printing and comparing

```
    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def __eq__(self, other):
        return self.num == other.num and self.den == other.den
```

```
>>> print(r)
1/5
>>> Rational(2, 10) == Rational(1, 5)
True
```

- `str` turns each int into a string
- Both are stored as 1/5 so `__eq__` just compares the attributes

Adding returns a new `Rational`

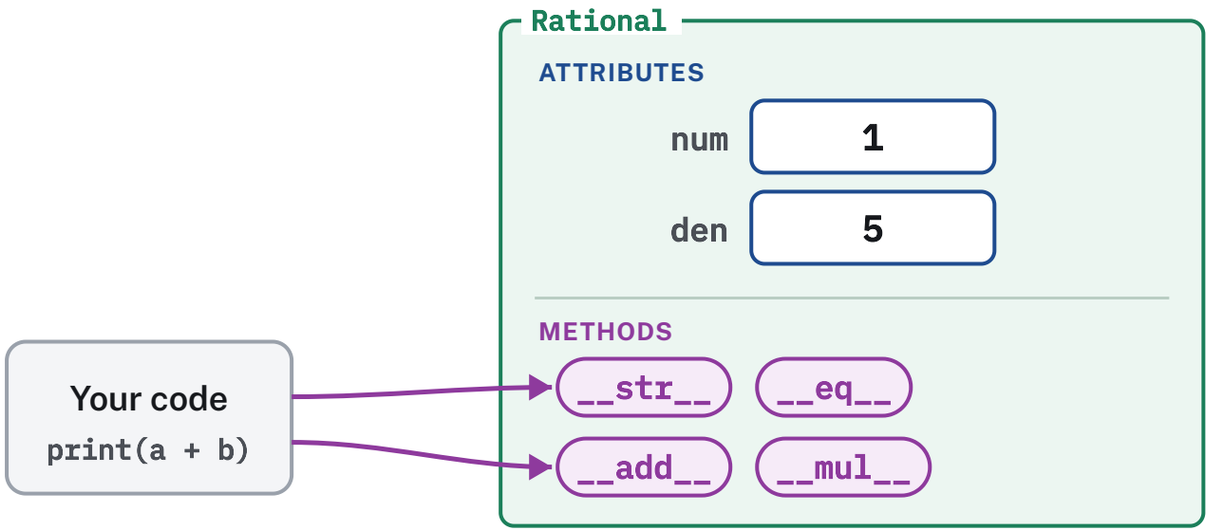
a/b + c/d = (a·d + c·b) / (b·d)
1/10 + 2/10 = (1·10 + 2·10) / (10·10) = 30/100

```
    def __add__(self, other):
        return Rational(self.num * other.den + other.num * self.den,
                        self.den * other.den)
```

```
>>> a = Rational(1, 10)
>>> b = Rational(2, 10)
>>> print(a + b)
3/10
>>> a + b == Rational(3, 10)
True
```

- `a + b` calls `a.__add__(b)`
- `__init__` reduces 30/100 to 3/10
- Exact: no 0.3000000000000004

Encapsulation: use the methods



- The object keeps its data and its methods together
- Outside code calls the methods
- How the data is stored is the class's business
- The class promises: always in lowest terms

Don't change attributes directly

CHANGING ATTRIBUTES DIRECTLY

```
r = Rational(1, 2)
r.num = 2
r.den = 4
print(r == Rational(1, 2))
```

OUTPUT

```
False
```

Now `r` is 2/4: not in lowest terms

`__eq__` gives the wrong answer

Outside the class, call the methods

Only the class's own methods should change `num` and `den`

A leading underscore means "private"

- MiddImage keeps its pixels in `_data`
  We use `img[x][y]`, not `img._data`
- Python doesn't stop you: it's a signal to other programmers

Activity A

Worksheet Part A

1. What does each line print?

```
print(Rational(3, 6))
print(Rational(4, 4))
print(Rational(1, 4) + Rational(1, 4))
print(Rational(3, 6) == Rational(2, 4))
```

2. Write `__mul__`

```
def __mul__(self, other):
    """Returns self times other"""
```

(a/b) · (c/d) = (a·c) / (b·d)

Activity A — answers

```
print(Rational(3, 6))
print(Rational(4, 4))
print(Rational(1, 4) + Rational(1, 4))
print(Rational(3, 6) == Rational(2, 4))
```

OUTPUT

```
1/2
1/1
1/2
True
```

2/4 reduces to 1/2 · 4/4 reduces to 1/1

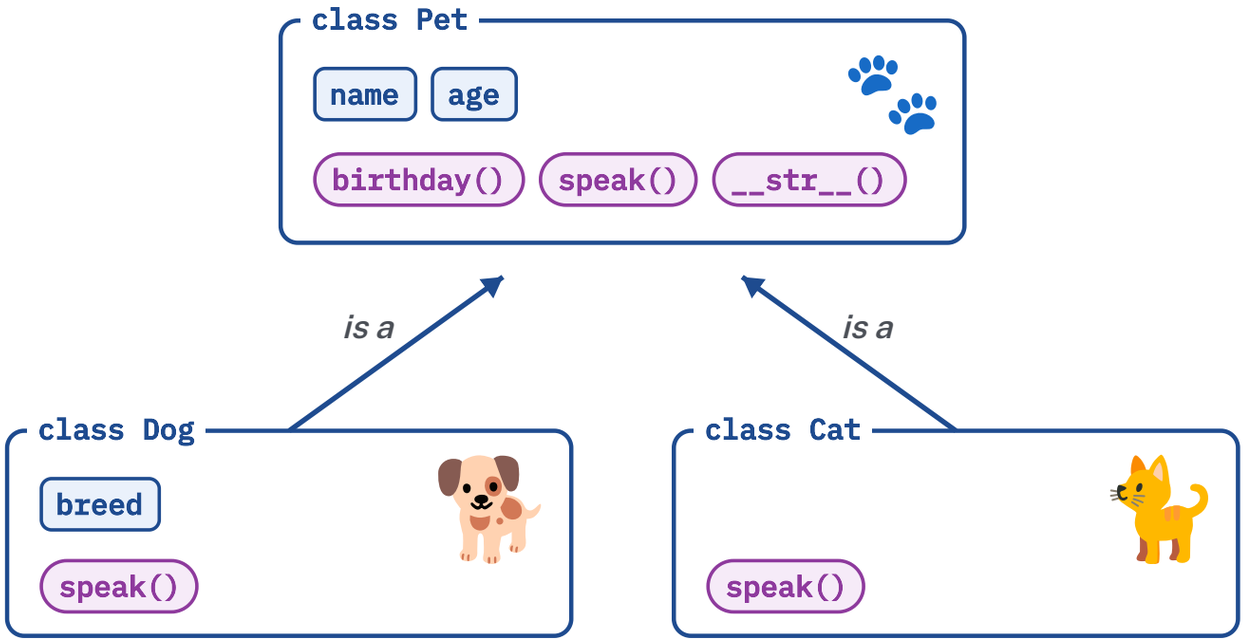
```
    def __mul__(self, other):
        """Returns self times other"""
        return Rational(self.num * other.num, self.den * other.den)
```

Part 2 of 3

Inheritance

A **child class** gets every method of its **parent class**
It can add new attributes and methods
and **override** methods to change what they do

A family of classes



- Every pet has a name and an age
- A Dog is a Pet
  A Cat is a Pet
- `Dog` and `Cat` are **child classes** (subclasses)
- `Pet` is their **parent class** (superclass)

The `Pet` class

```python
class Pet:
    """A pet with a name and an age"""

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def birthday(self):
        self.age = self.age + 1

    def speak(self):
        return "..."

    def __str__(self):
        return self.name + " says " + self.speak()
```

```
>>> goldie = Pet("Goldie", 1)
>>> print(goldie)
Goldie says ...
>>> goldie.birthday()
>>> goldie.age
2
```



- A plain Pet says `"..."`
- `__str__` calls `self.speak()`

Anatomy of a child class

```
class Dog(Pet):
    """A dog: a pet with a breed"""

    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def speak(self):
        return "Woof"
```

Dog(Pet) super().__init__ new attribute override

`class Dog(Pet):`

- The parent class goes in parentheses
- Dog inherits all of Pet's methods: `birthday`, `speak`, `__str__`

Anatomy of a child class

```
class Dog(Pet):
    """A dog: a pet with a breed"""

    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def speak(self):
        return "Woof"
```

Dog(Pet) | super().__init__ | new attribute | override

`super().__init__(…)`

- `super()` is the parent class, Pet
- Runs Pet's `__init__`
  It sets `self.name`
  and `self.age`
- Don't pass `self`:
  `super()` does that

Anatomy of a child class

```
class Dog(Pet):
    """A dog: a pet with a breed"""

    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def speak(self):
        return "Woof"
```

`self.breed = breed`

- A new **attribute**
- Only Dogs have a breed

Dog(Pet) · super().__init__ · **new attribute** · override

Anatomy of a child class

```
class Dog(Pet):
    """A dog: a pet with a breed"""

    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def speak(self):
        return "Woof"
```

Dog(Pet) super().__init__ new attribute **override**

```
def speak(self):
```

- **Overrides** Pet's `speak`
- Same name, new body
- A Dog says `"Woof"`

What a Dog has

Dog A pet with a breed

Attribute	Example value
`name`	'Biscuit' from Pet
`age`	3 from Pet
`breed`	'beagle' new

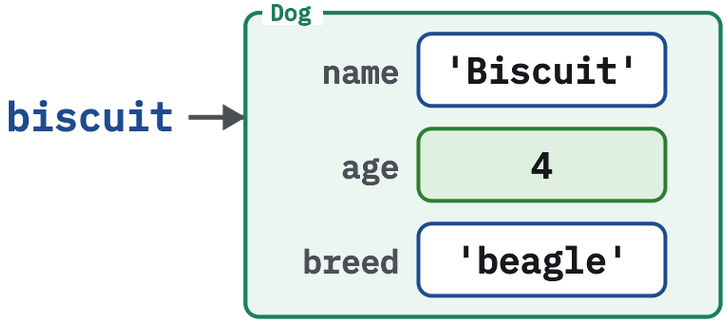
Method call	What it does or returns
`dog.birthday()`	adds 1 to `age` from Pet
`dog.speak()`	returns **`'Woof'`** overrides
`print(dog)`	prints `Biscuit says Woof` from Pet



We wrote 2 methods in `Dog`, but a Dog can do 4 things

Using a Dog

```
>>> biscuit = Dog("Biscuit", 3, "beagle")
>>> biscuit.birthday()
>>> biscuit.age
4
>>> biscuit.speak()
'Woof'
>>> print(biscuit)
Biscuit says Woof
```





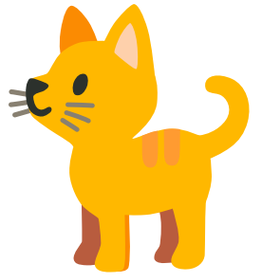
- `birthday` comes from Pet
- `speak` comes from Dog
- Pet's `__str__` calls `self.speak()`
  `self` is a Dog, so it says Woof

`Cat` is another child class of `Pet`

```
class Cat(Pet):
    """A cat. It uses Pet's __init__"""

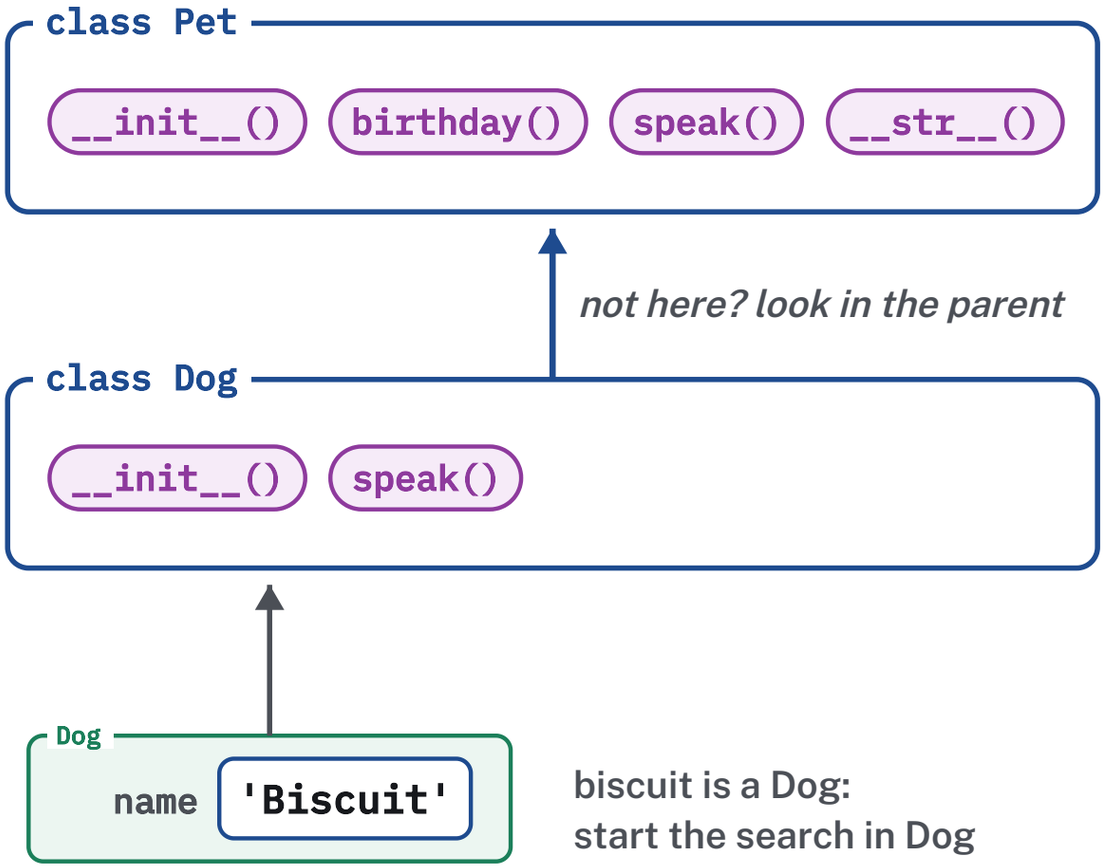
    def speak(self):
        return "Meow"
```

```
>>> mochi = Cat("Mochi", 2)
>>> print(mochi)
Mochi says Meow
>>> mochi.age
2
```



- No `__init__`: Cat uses Pet's (name and age)
- Pet's `__str__` calls Cat's `speak`

Which method runs?



biscuit is a Dog:
start the search in Dog

Python looks in the object's class first
then in the parent class

`biscuit.speak()`
Found in Dog: `'Woof'`

`biscuit.birthday()`
Not in Dog, found in Pet

`print(biscuit)`
`__str__` is found in Pet
It calls `self.speak()`
The search starts in Dog again

Python calls this order the
method resolution order

Activity B

Worksheet Part B

```
d = Dog("Rufus", 5, "poodle")
c = Cat("Luna", 4)
print(d.speak())
print(d.age, c.age)
print(c)
```

- For each call, whose method runs: Pet's, Dog's, or Cat's?
- Then write what lines 3, 4, and 5 print

Activity B — answers

Worksheet Part B

CALL		WHOSE METHOD
`Dog("Rufus", 5, "poodle")`	*runs*	Dog's **`__init__`**, which calls Pet's
`Cat("Luna", 4)`	*runs*	Pet's **`__init__`**
`d.speak()`	*runs*	Dog's **`speak`**
`print(c)`	*runs*	Pet's **`__str__`**, then Cat's **`speak`**

OUTPUT

```
Woof
5 4
Luna says Meow
```

Part 3 of 3

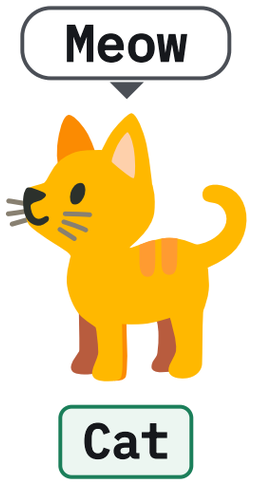
Polymorphism

> **Polymorphism**: the same method call works on many kinds of objects
> Each object runs its own class's version

One loop, three kinds of pets

```
pets = [Dog("Biscuit", 3, "beagle"), Cat("Mochi", 2), Pet("Goldie", 1)]
for pet in pets:
    print(pet.speak())
```







- The loop doesn't check the type
- Each pet runs its own `speak`

You've used polymorphism all along

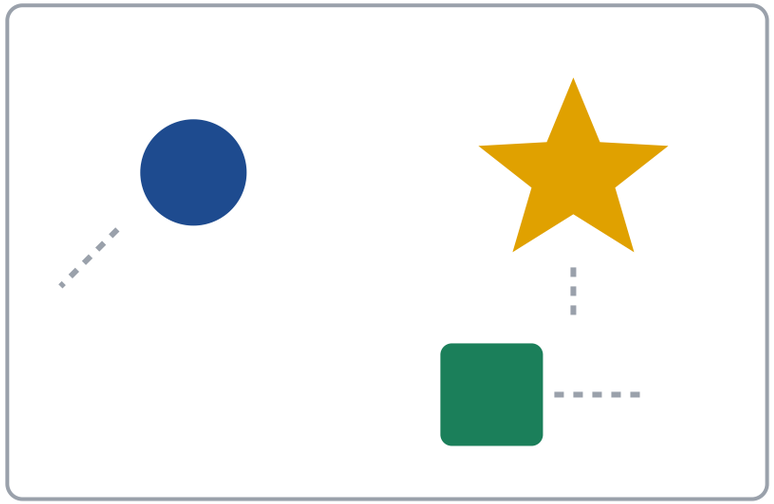
```
>>> len("hi")
2
>>> len([4, 5, 6])
3
>>> 2 + 3
5
>>> "ab" + "cd"
'abcd'
>>> [1] + [2]
[1, 2]
>>> print(Rational(1, 2) + Rational(1, 3))
5/6
```

- Same function, many types
- Same operator, the object decides what it means
- Our `Rational` joins in

Games put these ideas to work

```
for thing in things:
    thing.update()
    thing.draw(screen)
```

- Everything on the screen has `update` and `draw`
- Each kind moves and looks different: its own versions
- A new kind of thing?
  The loop doesn't change



Coming up in Lecture 20 and Homework 7

Common errors

```
self.breed = breed # no super()
```

`AttributeError: 'Dog' object has no attribute 'name'`

Forgot `super().__init__(name, age)`

Pet's attributes were never made

```
super().__init__(self, name, age)
```

`TypeError: Pet.__init__() takes 3 positional arguments but 4 were given`

`super()` already passes `self`

```
def speek(self):
```

`Biscuit says ...`

No error, wrong answer

A misspelled override doesn't override

```
Cat("Mochi", 2, "tabby")
```

`TypeError: Pet.__init__() takes 3 positional arguments but 4 were given`

Cat uses Pet's `__init__`: a name and an age

```
mochi.breed
```

`AttributeError: 'Cat' object has no attribute 'breed'`

Only Dogs have a breed

A child's new attributes don't go to its siblings

Activity C

Download day19.zip and extract it in your `class` folder
In `day19_pets.py`, write a `Parrot` class
A parrot says its own phrase: `Parrot("Polly", 30, "Hello!")`
Uncomment the Activity C tests at the bottom
Challenge: a `Puppy` is a Dog that says `"Yip"`



PATTERNS

Name the parent class	`class Parrot(Pet):`
Run the parent's `__init__`	`super().__init__(name, age)`
Store the new parameter	`self.phrase = phrase`
Override a method	`def speak(self):`

Activity C — answers

```
class Parrot(Pet):
    def __init__(self, name, age, phrase):
        super().__init__(name, age)
        self.phrase = phrase

    def speak(self):
        return self.phrase
```

Challenge

```
class Puppy(Dog):
    def speak(self):
        return "Yip"
```

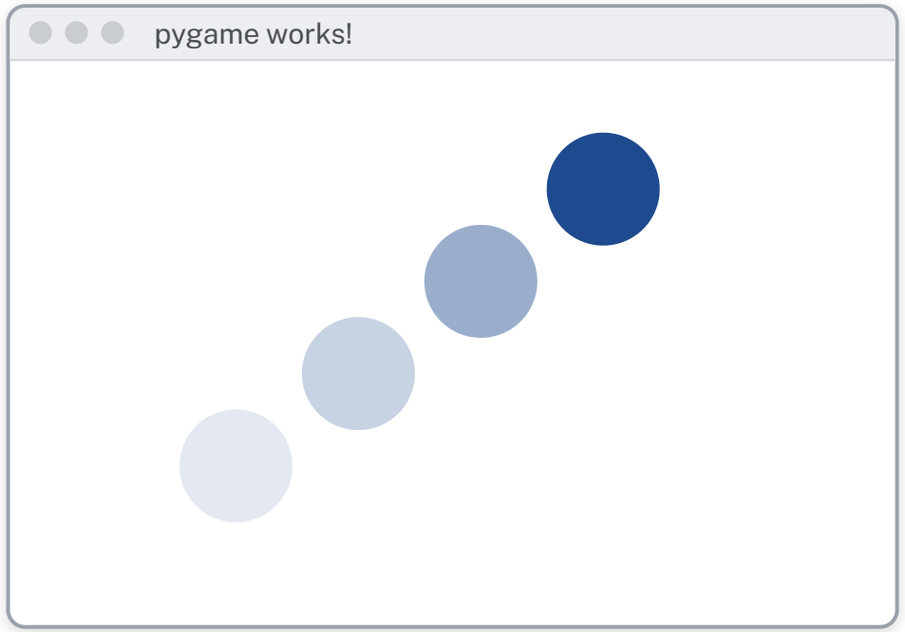
OUTPUT

```
Polly says Hello!
31
Biscuit says Woof
Mochi says Meow
Polly says Hello!
```

Puppy uses Dog's `__init__` which calls Pet's

Before you go: does pygame work?

1. Open `bounce.py` from `day19.zip`
2. Run it: a ball should bounce around a window
3. Close the window to stop
4. An error instead?
   Tools → Manage packages
   Search for `midd-media` and install it
5. Still stuck? Show us before you start Lecture 20



To do

- Review this slide deck and Notes 15
- Homework 6 is due Thursday 11/19
- Next class is Lecture 20 on video
  Watch the videos and do the activities before lab this Friday
- Lab 6 is Friday 11/20 and builds on Lecture 20

Next time

Lecture 20 — animation and games, on video

Make the ball in `bounce.py` *an object*