OBJECT ORIENTED PROGRAMMING

Objectives

- Define what Object Oriented Programming is
- Understand encapsulation and abstraction
- Create classes and instances and attach methods and properties to each

What is OOP?

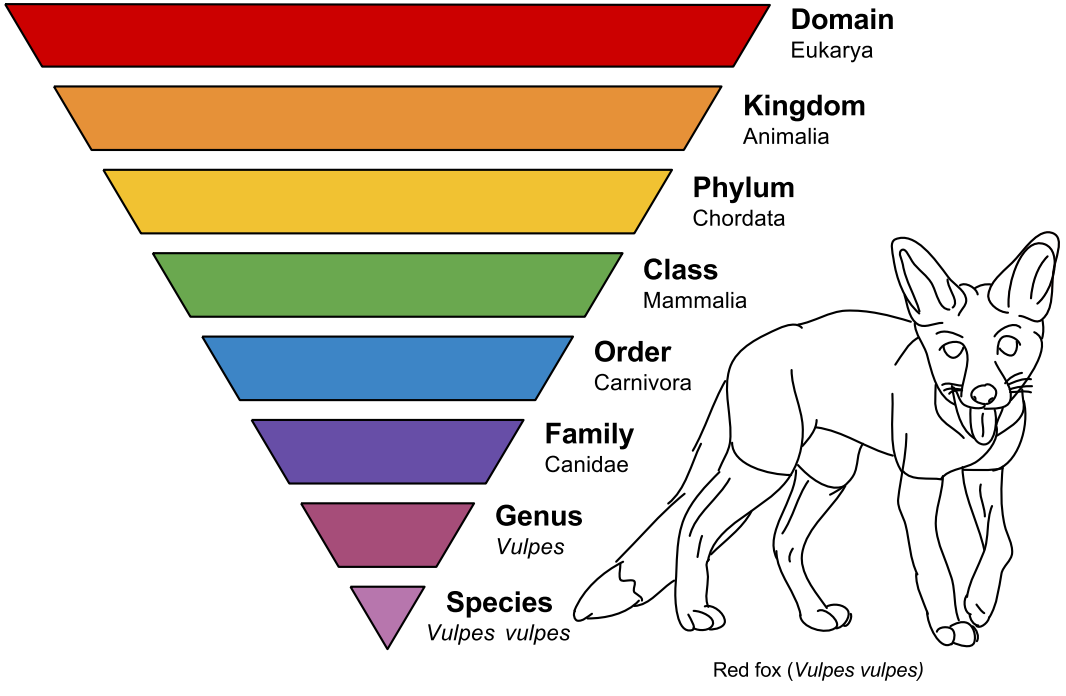
Object oriented programming is a method of programming that attempts to model some process or thing in the world as a **class** or **object**.

class - a blueprint for objects. Classes can contain methods (functions) and attributes (similar to keys in a dict).

instance - objects that are constructed from a class blueprint that contain their class's methods and properties.

Why OOP?

With object oriented programming, the goal is to *encapsulate* your code into **logical, hierarchical groupings using classes** so that you can reason about your code at a higher level.



Example

Say we want to model a game of **poker** in our program.

We could have the following entities:

- Game
- Player
- Card
- Deck
- Hand
- Chip
- Bet

Each entity could be its own class in our program!

Example (continued)

Card Deck Possible Implementation (Pseudocode)

`Deck {class}`

- `_cards` `{private list attribute}`
- `_max_cards` `{private int attribute}`
- `shuffle` `{public method}`
- `deal_card` `{public method}`
- `deal_hand` `{public method}`
- `count` `{public method}`

Encapsulation

Encapsulation - the grouping of public and private attributes and methods into a programmatic class, making **abstraction** possible

Example

- Designing the Deck class, I make **cards** a private attribute (a list)
- I decide that the length of the cards should be accessed via a public method called **count()** -- i.e. **Deck.count()**

Abstraction

Abstraction - exposing only "relevant" data in a class interface, hiding private attributes and methods (aka the "inner workings") from users

Example

- As a user of the **Deck** class, I never call **len(Deck.cards)**, only **Deck.count()** because **Deck.cards** is "abstracted away" for me.

Creating a Class

```
class Vehicle:

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
```

Classes in Python can have a special __init__ method, which gets called every time you create an instance of the class (instantiate).

Instantiating a Class

Creating an object that is an instance of a class is called **instantiating** a class.

```
v = Vehicle("Honda", "Civic", 2017)
```

In this case, **v** becomes a Honda Civic, a new instance of Vehicle

```
v
<__main__.Vehicle at 0x10472f5c0>
v.make
'Honda'
v.model
'Civic'
v.year
2017
```

self

```
class Vehicle:

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
```

The **self** keyword refers to the current class instance.

self must always be the first parameter to **__init__** and any methods and properties on class instances.

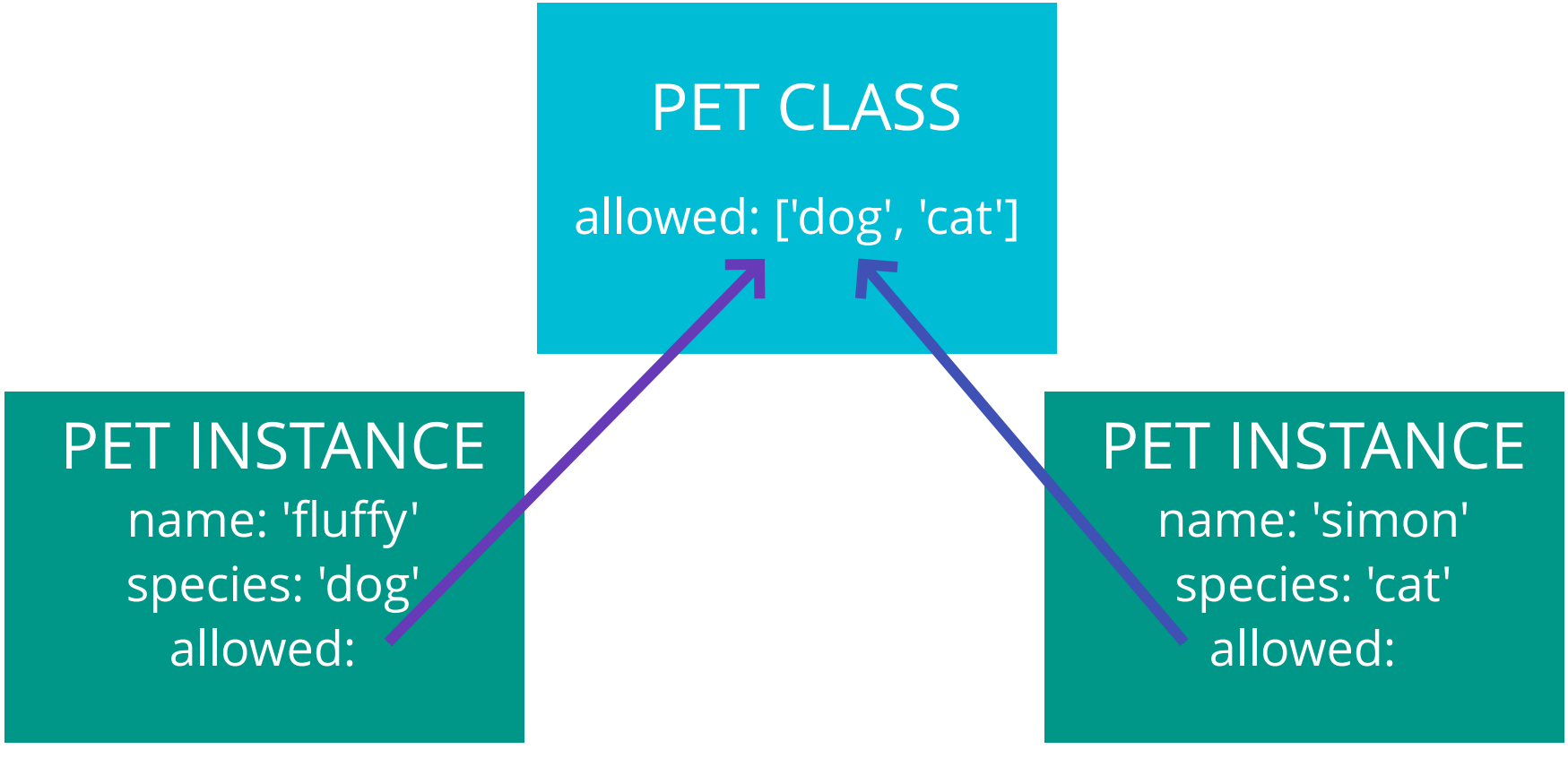
You never have to pass it directly when calling instance methods, including __init__.

Instance Attributes and Methods

```
class Person():

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def full_name(self):
        return f"My name is {self.first_name} {self.last_name}"

    def likes(self, thing):
        return f"{self.first_name} likes {thing}!"

p = Person("Colt", "Steele")

p.full_name()

Colt Steele
p.likes("Python")
Colt likes Python!
```

Class Attributes

We can also define attributes directly on a class that are shared by all instances of a class and the class itself.



Class Attributes

We can also define attributes directly on a class that are shared by all instances of a class and the class itself.

```
class Pet():

    allowed = ("cat", "dog", "bird", "lizard", "rodent")

    def __init__(self, kind, name):

        if kind not in self.allowed:
            raise ValueError(f"You can't have a {kind} as a pet here!")

        self.kind = kind
        self.name = name

fluffy = Pet("cat", "Fluffy")

fluffy.allowed
("cat", "dog", "bird", "lizard", "rodent")
Bro = Pet("bear", "Bro")
ValueError: You can't have a bear as a pet here!
```

Class Methods

Class methods are methods (with the @classmethod decorator) that are not concerned with instances, but the class itself.

```
class Person():
    # ...

    @classmethod
    def from_csv(cls, filename):
        return cls(*params) # this is the same as calling Person(*params)

Person.from_csv(my_csv)
```

The first argument is **cls** (for class) instead of self. Like self, it does not need to be passed in explicitly.

Class methods are available on the class itself and any instances of the class, and are mostly used for building new instances of classes.

YOUR TURN

Objectives

- Implement inheritance, including multiple
- Understand Method Resolution Order
- Understand polymorphism
- Add special methods to classes

Inheritance

A key feature of OOP is the ability to define a class which inherits from another class (a "base" or "parent" class).

In Python, inheritance works by passing the parent class as an argument to the definition of a child class:

```
class Animal:
    def make_sound(self, sound):
        print(sound)

    cool = True

class Cat(Animal):
    pass

gandalf = Cat()
gandalf.make_sound("meow")  # meow
gandalf.cool  # True
```

super

The **super()** keyword allows us to call the __init__ function of a parent class
In the example below, we initialize the child with both its own __init__ method *and* its parent's __init__ method:

```
class Animal:
    def __init__(self, species):
        self.species = species

class Dog(Animal):
    def __init__(self, name):
        super().__init__("canine")
        self.name = name

bro = Dog("Bro")

bro.name  # Bro

bro.species  # canine
```

Multiple Inheritance

Python also allows classes to inherit from more than one parent class.

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

  def swim(self):
    return f"{self.name} is swimming"

  def greet(self):
    return f"I am {self.name} of the sea!"

class Ambulatory:
  def __init__(self,name):
    self.name = name

  def walk(self):
    return f"{self.name} is walking"

  def greet(self):
    return f"I am {self.name} of the land!"

class Penguin(Aquatic, Ambulatory):
  def __init__(self,name):
```

```
jaws = Aquatic("Jaws")
lassie = Ambulatory("Lassie")
captain_cook = Penguin("Captain Cook")
```

Multiple Inheritance (continued)

```
jaws = Aquatic("Jaws")
lassie = Ambulatory("Lassie")
captain_cook = Penguin("Captain Cook")
```

```
jaws.swim() # 'Jaws is swimming'
jaws.walk() # AttributeError: 'Aquatic' object has no attribute 'walk'
jaws.greet() # 'I am Jaws of the sea!'

lassie.swim() # AttributeError: 'Ambulatory' object has no attribute 'swim'
lassie.walk() # 'Lassie is walking'
lassie.greet() # 'I am Lassie of the land!'

captain_cook.swim() # 'Captain Cook is swimming'
captain_cook.walk() # 'Captain Cook is walking'
captain_cook.greet() # 'I am Captain Cook of the sea!'
```

Penguin inherits from both **Aquatic** and **Ambulatory**, therefore instances of Penguin can call both the **walk** and **swim** methods.

What about the greet method for our instance of Penguin?

It is calling the Aquatic.greet() instead of Ambulatory.greet().

Method Resolution Order (MRO)

Whenever you create a class, Python sets a **M**ethod **R**esolution **O**rder, or **MRO**, for that class, which is the order in which Python will look for methods on instances of that class.

You can programmatically reference the MRO three ways:

- **__mro__** attribute on the class
- use the **mro()** method on the class
- use the builtin **help(**cls**)** method

```
Penguin.__mro__

# (<class 'multiple.Penguin'>, <class 'multiple.Aquatic'>,
#  <class 'multiple.Ambulatory'>, <class 'object'>)

Penguin.mro()

# [__main__.Penguin, __main__.Aquatic, __main__.Ambulatory, object]

help(Penguin) # best for HUMAN readability -> gives us a detailed cha
```

Polymorphism

A key principle in OOP is the idea of polymorphism - an object can take on many (poly) forms (morph).

While a formal definition of polymorphism is more difficult, here are two important practical applications:

1. The same class method works in a similar way for different classes

```
Cat.speak()  # meow
Dog.speak()  # woof
Human.speak()  # yo
```

2. The same operation works for different kinds of objects

```
sample_list = [1,2,3]
sample_tuple = (1,2,3)
sample_string = "awesome"

len(sample_list)
len(sample_tuple)
len(sample_string)
```

Polymorphism & Inheritance

1. The same class method works in a similar way for different classes

A common implementation of this is to have a method in a base (or parent) class that is overridden by a subclass. This is called **method overriding**.

- Each subclass will have a different implementation of the method.
- If the method is not implemented in the subclass, the version in the parent class is called instead.

```
class Animal():
    def speak(self):
        raise NotImplementedError("Subclass needs to implement this method"

class Dog(Animal):
    def speak(self):
        return "woof"

class Cat(Animal):
    def speak(self):
        return "meow"
```

Special Methods

2. (Polymorphism) The same operation works for different kinds of objects

How does the following work in Python?

```
8 + 2  # 10
"8" + "2"  # 82
```

The answer is "special methods"!

Python classes have special (also known as "magic") methods, that are dunders (i.e. double underscore-named).

These are methods with special names that give instructions to Python for how to deal with objects.

Special Methods Example

What is happening in our example?

```
8 + 2  # 10
"8" + "2"  # 82
```

The **+** operator is shorthand for a special method called **__add__()** that gets called on the first operand.

If the first (left) operand is an instance of **int**, __add__() does mathematical **addition**. If it's a **string**, it does string **concatenation**.

Special Methods Applied

Therefore, you can declare special methods on your own classes to mimic the behavior of builtin objects, like so using **__len__**:

```
class Human:
    def __init__(self, height):
        self.height = height  # in inches

    def __len__(self):
        return self.height

Colt = Human(60)
len(Colt)  # 60
```

String Representation

The most common use-case for special methods is to make classes "look pretty" in strings.

By default, our classes look ugly:

```
class Human:
    pass

colt = Human()
print(colt)  # <__main__.Human at 0x1062b8400>
```

We can use special methods to make it look way better!

String Representation Example

The __repr__ method is one of several ways to provide a nicer string representation:

```
class Human:

    def __init__(self, name="somebody"):
        self.name = name

    def __repr__(self):
        return self.name

dude = Human()
print(dude)  # "somebody"

colt = Human(name="Colt Steele")
print(f"{colt} is totally rad (probably)")
# "Colt Steele is totally rad (probably)"
```

There are also several other dunders to return classes in string formats (notably __str__ and __format__), and choosing one is a bit complicated!

Recap

- OOP is a way of programming by grouping things into hierarchical classes (objects)
- OOP uses encapsulation and abstraction to provide a class interface to call methods and read/modify attributes
- Classes can inherit from other classes and use the same methods to do different things (polymorphism)
- We can enhance our custom classes with special/magic/dunder methods

YOUR TURN