CS253: Software Development

Welcome to Lecture 14!

Daniel George

October 17, 2023

Announcements

- Please vote (this is required!) for your top choices for class project!
- Today: object-oriented programming

Object-Oriented Programming

- Last week, we talked about software engineering process: agile, iterative, and flexible
- To that end, how can we write code, concretely, which supports this new paradigm?
- Older programming languages (C, C++) asked the question: “what should this program do next”?
    - Split into tasks and sub-tasks
    - Make functions for the tasks
    - Instruct the computer to perform them sequentially

Object-Oriented Programming

- But…with large numbers of tasks, and large amounts of data, this can get unwieldy
- To see this, suppose we were trying to model the operation of a car, in code.
- Such a program would have lots of separate variables storing information on various car parts, and there’d be no way to group together all the code that relates to, say, the wheels.
- It’s hard to keep all these variables and the connections between all the functions in mind.

Object-Oriented Programming

- To manage this complexity, it’s nicer to package up self-sufficient, modular pieces of code.
- People think of the world in terms of interacting objects: we’d talk about interactions between the steering wheel, the pedals, the wheels, etc.
- OOP allows programmers to pack away details into neat, self-contained boxes **(objects)** so that they can think of the objects more abstractly and focus on the interactions between them.

Object-Oriented Programming

- Three primary features of OOP:
  - **Encapsulation:** grouping related data and functions together as objects and defining an interface to those objects
  - **Inheritance:** allowing code to be reused between related types
  - **Polymorphism:** allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type

Encapsulation

- **Encapsulation** just refers to packaging related stuff together. We've already seen how to package up data and the operations it supports in Python: with **classes.**
- If someone hands us a class, we do not need to know how it actually works to use it; all we need to know about is its public methods/data – its **interface.**
- This is often compared to operating a car: when you drive, you don't care how the steering wheel makes the wheels
- turn; you just care that the interface the car presents (the steering wheel) allows you to accomplish your goal.

Encapsulation

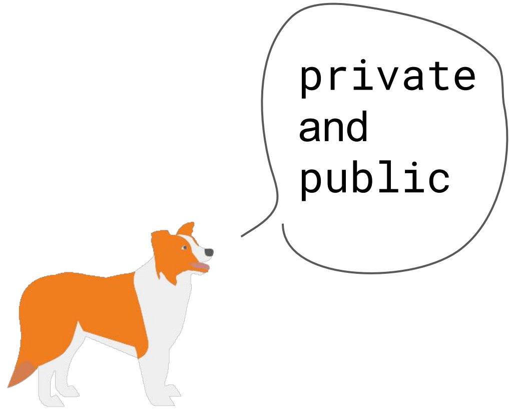
- This is why, in Python, you can specify **public** and **private** attributes: by default, it assumes that the things you define in a class are internal details which someone using your code should not have to worry about. The practice of hiding away these details from client code is called “**data hiding,**” or making your class a “**black box.**”
- One way to think about what happens in an object-oriented program is that we define what objects exist and what each one knows, and then the objects send messages to each other (by calling each other’s methods) to exchange information and tell each other what to do.

Inheritance

- **Inheritance** allows us to define hierarchies of related classes.
- Imagine we're writing an inventory program for vehicles, including cars and trucks. We could write one class for representing cars and an unrelated one for representing trucks, but we'd have to duplicate the functionality that all vehicles have in common.
- Instead, Python allows us to specify the common code in a Vehicle class, and then specify that the Car and Truck classes share this code.

Inheritance

```
class Vehicle:
    def __init__(self, myLicense, myYear):
        self._license = myLicense
        self._year = myYear

    def getDesc(self):
        return f"{self.license} from
            {str(self.year)}"

    def getLicense(self):
        return self._license

    def getYear(self):
        return self._year
```



Inheritance

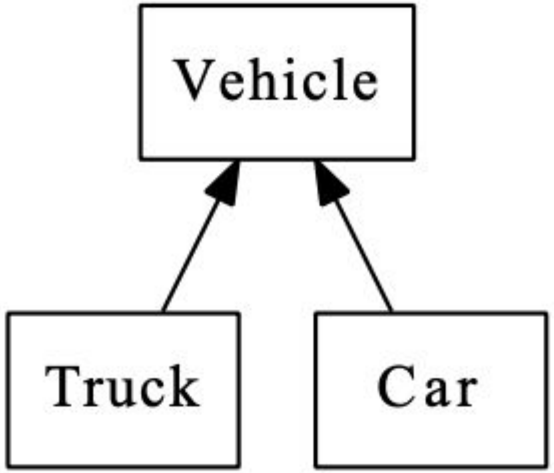
```
class Car(Vehicle):
    def __init__(self, myLicense, myYear, myStyle):
        super().__init__(myLicense, myYear)
        self.style = myStyle

    def getStyle(self):
        return self.style
```

Inheritance

- Now class Car has all the data members and methods of Vehicle, as well as a style data member and a getStyle method.
- Class Car *inherits* from class Vehicle. This is equivalent to saying that Car is a **derived class,** while Vehicle is its **base class.**
- You may also hear the terms **subclass** and **superclass** instead.

Inheritance

- Similarly, we could make a Truck class that inherits from Vehicle and shares its code. This would give a *class hierarchy* like the following:



Inheritance

- Now class Car has all the data members and methods of Vehicle, as well as a style data member and a getStyle method.
- Class Car *inherits* from class Vehicle. This is equivalent to saying that Car is a **derived class,** while Vehicle is its **base class.**
- You may also hear the terms **subclass** and **superclass** instead.

Is-a vs. Has-a

- There are two ways we could describe some class A as depending on some other class B:
  - Every A object *has a* B object. For instance, every Vehicle has a string object (called “license”).
  - Every instance of A *is a* B instance. For instance, every Car is a Vehicle, as well.
- Inheritance allows us to define “is-a” relationships, but it should not be used to implement “has-a” relationships. It would be a design error to make Vehicle inherit from “string” because every Vehicle has a license; a Vehicle is not a string. “Has-a” relationships should be implemented by declaring data members, not by inheritance.

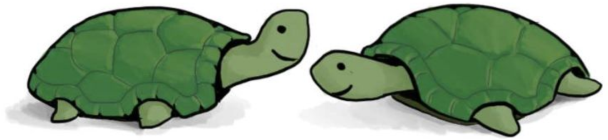
Polymorphism

- **Polymorphism** means “many shapes.” It refers to the ability of one object to have many types.
- If we have a function that expects a Vehicle object, we can safely pass it a Car object, because every Car is also a Vehicle.

Polymorphism

```
def process_vehicle(vehicle):
    print(f"License: {vehicle.getLicense()}")
    print(f"Year: {vehicle.getYear()}")

car = Car("Daniel’s Kia", 2023, "Sedan")
process_vehicle(car)
```

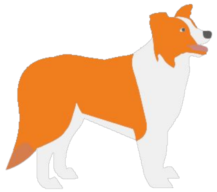
Think-Pair-Share Terrapins!



Given what you just learned about object-oriented programming, what do you think are its advantages and disadvantages? When would it be useful to use OOP and when would it not?

Practice

What are your questions?



Thank you!