Implementing relations in Python

In Python

- You create an aggregation / composition relationship by keeping a "reference" from one object to another.

Composition

- Example: a hand *has* 5 fingers
- We choose to use a composition relationship.

```
class Finger:
    def flick(self):
        print("Oh no.")

class Hand:
    def __init__(self):
        self._fingers = [Finger() for _ in range(5)]

    def thumbs_up(self):
        self._fingers[0].flick()
```

- The `Finger` objects are created by the `Hand` objects (when `Hand` disappears, so do the fingers)
- The `Hand` can keep track and interact with the `Finger` objects through the private variable `self._fingers`: `self._fingers[2].flick()`

Aggregation

- Example: the car and the driver

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

class Car:
    def __init__(self, driver):
        self._driver = driver
```

- You can associate the objects when creating the car:

```
tim = Driver("Tim")
my_car = Car(tim)
my_car = Car(driver=tim) # Also works
```

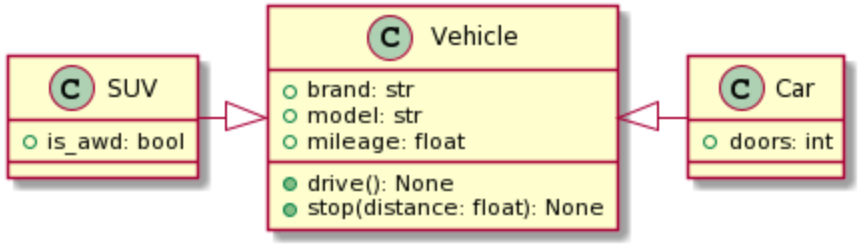
Composition and aggregation

- In a composition relationship, composite objects are usually created when the main object is created (in the `__init__` method).
- But it can also happen elsewhere (`def add_finger(self)`).
- In aggregation relationships, there are usually methods in the public interface that allow to associate / disassociate objects. They can sometimes be related to setters.

```
class EvoCar:
    def __init__(self):
        self._driver = None

    def start_rental(self, new_driver):
        self._driver = new_driver

    def end_rental(self):
        self._driver.charge_rental()
        self._driver = None

tim = Driver("Tim")
prius = EvoCar()
prius.start_rental(tim)
prius.end_rental()
```

Inheritance

- Inheritance is used when several objects are part of the same family:
  - they share similar attributes
  - they share similar behaviours
- In UML diagrams, inheritance is represented with an arrow.
- The arrow points to the "parent" class.



Inheritance in Python

```
class Vehicle:
    def drive(self):
        pass

    def stop(self, distance):
        self._mileage += distance

class Car(Vehicle):
    """ Car will automatically INHERIT from all attributes and methods of Vehicle """
    pass

class SUV(Vehicle):
    """ SUV will automatically INHERIT from all attributes and methods of Vehicle
        But you can add attributes / methods too """
    is_awd = True
```

Inheritance in Python: continued

- All the attributes and methods of the parent class are **inherited** by the child class.
- If an attribute is defined in the parent class and also in the child class, the child class has precedence.
- You can travel up the inheritance tree by using `super()`
- `super()` will give access to the parent class (and its methods / attributes)

```
class Car:
    def __init__(self, model, brand, mileage=0):
        self.brand = brand
        self.model = model
        self.mileage = mileage

class Chevrolet:
    def __init__(self, model, mileage=0):
        super().__init__(model, "Chevrolet", mileage)
```

Inheritance: limitations

- Inheritance is great - but should be used sparingly.
- It can be complicated to understand which method or variable comes from which class.
- There are lots of occurrences where inheritance can be replaced with composition.
- Inheritance can be really useful when used to create "better versions" of standard Python types.

```
class Grades(list):     # Grades derives from builtin `list`
    @property
    def average(self):
        return sum(self)/len(self)

grades = Grades([83, 90, 74, 23])
print(grades.average)       # 67.5
```

Combine them with other objects

```
class Student:
    def __init__(self, grades):
        self._grades = Grades(grades)

    @property
    def does_pass(self):
        return self._grades.average > 50

tim = Student(grades=[83, 90, 74, 23])
print(tim.does_pass)    # True
```

Using inheritance to create abstract classes

- You can use inheritance to define "generic classes" that only define a specific public interface, **without implementing it**.
- The child classes will inherit all attributes from the parent class, and will override the public interface methods.

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

    def sound(self):
        raise NotImplementedError("Abstract method for animal sound must be implemented by the child class")

    def show_name(self):
        print(self._name)

class Cow(Animal):
    def sound(self):
        return "MOO"

class Dog(Animal):
    def sound(self):
        return "WOOF"
```

From inheritance to polymorphism 🐄 🐕

```
daisy = Cow("Daisy")
daisy.sound()               # MOO!
daisy.show_name()           # prints "Daisy"
isinstance(daisy, Animal)   # TRUE!
isinstance(daisy, Cow)      # TRUE!

snoop = Dog("Snoopy")
snoop.sound()               # WOOF!
snoop.show_name()

for animal in (daisy, snoop):
    animal.sound()          # POLYMORPHISM!
```

- `animal` can be a cow or a dog, but it has an interface that defines `sound()`
- from the Python interpreter perspective, `animal` is just an object that happens to have a matching method available
- Note that `isinstance(daisy, Animal)` returns `True` even though daisy is a cow: that's because `Cow` is a child class of `Animal`. A cow is also an animal.

Defining abstract methods in Python with `@abstractmethod` and `ABC`

```
from abc import abstractmethod, ABC

class Animal(ABC):
    def __init__(self, name):
        self._name = name

    @abstractmethod
    def sound(self):
        pass

    def show_name(self):
        print(self._name)
```