Lecture 9.1 : Object-oriented programming: Inheritance

Introduction

- A company manages a fleet of vehicles. Vehicles include cars, motorcycles, trucks and vans. Vehicles have the following characteristics:
    - All vehicles have a make, model, year, cost price, mileage and driver
    - Trucks also have an associated mechanic
    - A vehicle's value depreciates at a rate dependent on vehicle-type:
        - Cars depreciate at a rate of 10% per annum
        - Motorcycles depreciate at a rate of 15% per annum
        - Goods vehicles (trucks and vans) depreciate at a rate of 20% per annum
    - Vehicles are serviced at intervals dependent on vehicle-type:
        - Cars are serviced every 10,000 miles
        - Motorcycles are serviced every 5,000 miles
        - Vans are serviced every 15,000 miles
        - Trucks are serviced every 20,000 miles
- We are approached by the above company and asked to implement a Python program that models their fleet of vehicles. The program must be able to output the current value of any particular vehicle and tell us in how many miles its next service is due.

First attempt

- We decide to adopt an object-oriented approach when modelling the company's fleet. For example, any particular car in the company's fleet will be a "car object" i.e. an instance of the class `Car` in our program. Obviously, Python does not come with a built-in `Car` class so we will define one of our own. We will have to do likewise for the `Motorcycle`, `Van` and `Truck` classes. It looks like we have considerable work to do so to get started we decide to model just the cars and motorcycles. Below is our first attempt at defining the `Car` and `Motorcycle` classes (followed by a demonstration of the code in action):

```
# vehicles_v01.py
from datetime import datetime
current_year = datetime.now().year

class Car(object):

    service_miles = 10000
    depreciation_rate = -0.1

    def __init__(self, make, model, year, cost, miles, driver):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver

    def value(self):
```

```
        age = current_year - self.year
        return round(self.cost * (1+self.depreciation_rate)**age)

    def service(self):
        return self.service_miles - self.miles % self.service_miles

class Motorcycle(object):

    service_miles = 5000
    depreciation_rate = -0.15

    def __init__(self, make, model, year, cost, miles, driver):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver

    def value(self):
        age = current_year - self.year
        return round(self.cost * (1+self.depreciation_rate)**age)

    def service(self):
        return self.service_miles - self.miles % self.service_miles
```

```
>>> from vehicles_v01 import Car, Motorcycle
>>> car1 = Car('Honda', 'Civic', 2013, cost=20000, miles=16000, driver='Joe')
>>> car1.driver
'Joe'
>>> car1.value()
16200
>>> car1.service()
4000
>>> bike1 = Motorcycle('Suzuki', '650', 2012, cost=10000, miles=23000, driver='
>>> bike1.driver
'Moe'
>>> bike1.value()
6141
>>> bike1.service()
2000
```

- Do you notice anything disconcerting about the code presented above? It contains serious *duplication*. Whenever you find yourself writing the same code over and over: **stop!** Avoiding duplication is one of the key motivations behind object-oriented programming. Specifically, it is the application of *inheritance* in object-oriented programming that helps us avoid duplication and write code that is more *compact*, *maintainable* and *extensible*.

Applying inheritance

- Whereas composition is modelled by *has-a* and *has-many* relationships, inheritance is modelled by *is-a* relationships. For example, a `Student` object may reference a number of `Module` objects. This is *composition* where a `Student` *has-many* `Modules`. A `Van` *is-a* `Vehicle`, a `Car` *is-a* `Vehicle`, a `Truck` *is-a* `Vehicle`. The latter is *inheritance*.

- A key observation, one that will allow us to apply inheritance in the above scenario is that, although cars and motorcycles are different, they share common characteristics and behaviour. Cars and motorcycles (and everything else in the fleet) are *vehicles*. In object-oriented programming we define a new class `Vehicle` which captures what cars and motorcycles have *in* *common*. We retain the `Car` and `Motorcycle` classes in order to capture only what is *specific* to each of those object-types. `Vehicle` *attributes* will capture characteristics shared by all vehicles. `Vehicle` *methods* will capture behaviour shared by all vehicles. `Car` *attributes* will capture characteristics specific to cars. `Car` *methods* will capture behaviour specific to cars. `Motorcycle` *attributes* will capture characteristics specific to motorbikes. `Motorcycle` *methods* will capture behaviour specific to motorbikes.

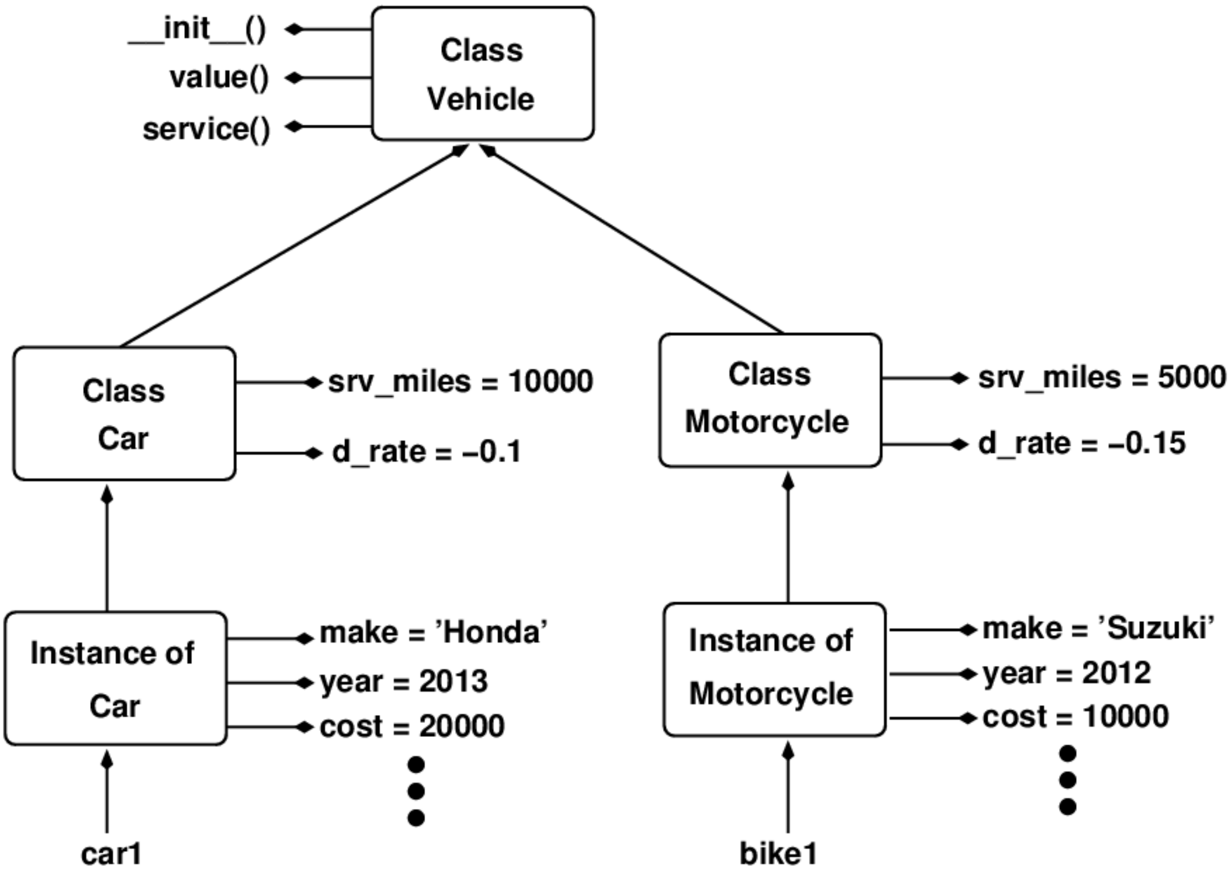
- Below is our new implementation where `Car` and `Motorcycle` classes *inherit from* the `Vehicle` class:

```
# vehicles_v02.py
from datetime import datetime
current_year = datetime.now().year

class Vehicle(object):

    def __init__(self, make, model, year, cost, miles, driver):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver

    def value(self):
        age = current_year - self.year
        return round(self.cost * (1+self.depreciation_rate)**age)

    def service(self):
        return self.service_miles - self.miles % self.service_miles

class Car(Vehicle):

    service_miles = 10000
    depreciation_rate = -0.1

class Motorcycle(Vehicle):

    service_miles = 5000
    depreciation_rate = -0.15
```

- Some key observations:
    - To have a *subclass* inherit from a *superclass* we place the name of the superclass in brackets when defining the subclass e.g. `class Car(Vehicle)` defines a class `Car` that inherits all of the class `Vehicle` attributes and methods. (All classes inherit from the `object` class so there will always be something between the brackets.)
    - Consider the `service()` method in the `Vehicle` class. It references `self.service_miles`. If `self` refers to an instance of `Car` then `self.service_miles` refers to the class variable in the `Car` class i.e. 10,000. If however `self` refers to an instance of `Motorcycle` then `self.service_miles` refers to the class variable in the `Motorcycle` class i.e. 5,000.
    - Some terminology: `Vehicle` is a *superclass* of `Car` and `Motorcycle`. (A superclass is also referred to as a *base class* or *parent class*.)
    - Some more terminology: `Car` is a *subclass* of `Vehicle` (as is `Motorcycle`). (A subclass is also referred to as a *derived class* or *child class*.)
    - In superclass attributes we capture shared characteristics. In superclass methods we capture shared behaviour.
- Let's verify that despite writing less code everything works as before:

```
>>> from vehicles_v02 import Car, Motorcycle
>>> car1 = Car('Honda', 'Civic', 2013, cost=20000, miles=16000, driver='Joe')
>>> car1.driver
'Joe'
>>> car1.value()
16200
>>> car1.service()
4000
>>> bike1 = Motorcycle('Suzuki', '650', 2012, cost=10000, miles=23000, driver='
>>> bike1.driver
'Moe'
>>> bike1.value()
6141
>>> bike1.service()
2000
```

- A diagram will help us understand how the inheritance approach works. The following is an *inheritance tree*:



- When a method or attribute is referenced through an object Python begins a search for a match at the bottom of the tree. It proceeds upwards until a match is found. (If no match is found then an `AttributeError` is returned.) Thus when we invoke the `car1.value()` method a search begins for the method:

    1. The `car1` object's attributes are checked for a match: **none found**.
    2. The `Car` class's attributes are checked for a match: **none found**.
    3. The `Vehicle` class's attributes are checked for a match: **match found** (and the method executes).

- When the `value()` method references the `self.year` attribute a search begins for that attribute:

    1. Since `self` is a reference to `car1` the latter's attributes are checked for a match: **match found** (and similarly for `self.cost`).

- When the `value()` method references the `self.depreciation_rate` attribute a search begins for that attribute:

    1. Since `self` is a reference to `car1` the latter's attributes are checked for a match: **none found**.
    2. We proceed up the tree and check the `Car` class attributes for a match: **match found**.

What about vans and trucks?

- Vans and trucks share a common `depreciation_rate` and are categorised as *goods vehicles* by the company. We define a new `Goods` class to reflect this relationship. `Goods` inherits from `Vehicle` while `Van` and `Truck` inherit from `Goods`. Making `depreciation_rate` a class attribute of `Goods` means all trucks and vans will share the same depreciation rate.

- Finally, trucks have an associated mechanic. This presents us with something of an issue. The `Vehicle` class's `__init__()` method knows nothing about mechanics so is not in a position to initialise this attribute for trucks. It looks like we will have to write a separate `__init__()` method for the `Truck` class:

```
# vehicles_v03.py
from datetime import datetime
current_year = datetime.now().year

class Vehicle(object):

    def __init__(self, make, model, year, cost, miles, driver):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver

    def value(self):
        age = current_year - self.year
        return round(self.cost * (1+self.depreciation_rate)**age)

    def service(self):
        return self.service_miles - self.miles % self.service_miles

class Goods(Vehicle):

    depreciation_rate = -0.20

class Van(Goods):

    service_miles = 15000

class Truck(Goods):

    service_miles = 20000

    def __init__(self, make, model, year, cost, miles, driver, mechanic):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver
        self.mechanic = mechanic
```

- The code excerpt above exhibits code duplication. The `Truck` class's `__init__()` method is nearly identical to that of the `Vehicle` class. The only difference is that the `__init__()` in `Truck` initialises one more attribute i.e. a mechanic. It would be nice if we could use the `Truck` class's `__init__()` method to initialise *just the mechanic attribute* and use the `Vehicle` class's `__init__()` method to initialise the rest.

- It turns out we can do just that! We can access a superclass from a subclass using the `super()` method. We use `super()` below to find and invoke the `__init__()` method of a superclass. (Note that `super()` will search the inheritance tree until it finds a match.) The `Truck` class's `__init__()` method invokes its superclass's `__init__()` method to initialise shared characteristics. Once that is done, it initialises the mechanic attribute to the supplied argument. Neat!

- Here is the updated code (complete with `Goods`, `Truck` and `Van` classes) and a demonstration follows:

```
# vehicles_v04.py
from datetime import datetime
current_year = datetime.now().year

class Vehicle(object):

    def __init__(self, make, model, year, cost, miles, driver):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver

    def value(self):
        age = current_year - self.year
        return round(self.cost * (1+self.depreciation_rate)**age)

    def service(self):
        return self.service_miles - self.miles % self.service_miles

class Car(Vehicle):

    service_miles = 10000
    depreciation_rate = -0.1

class Motorcycle(Vehicle):

    service_miles = 5000
    depreciation_rate = -0.15

class Goods(Vehicle):

    depreciation_rate = -0.20

class Van(Goods):

    service_miles = 15000

class Truck(Goods):

    service_miles = 20000

    def __init__(self, make, model, year, cost, miles, driver, mechanic):
        # 1. Invoke the __init__() method of superclass of current class
        # 2. Pass self as first argument to that __init__() method
        super().__init__(make, model, year, cost, miles, driver)
        self.mechanic = mechanic
```

```
>>> from vehicles_v04 import Van, Truck
>>> van1 = Van('Ford', 'Transit', 2010, cost=15000, miles=53000, driver='Lou')
>>> van1.driver
'Lou'
>>> van1.value()
4915
>>> van1.service()
7000
>>> truck1 = Truck('Scania', 'R420', 2014, cost=50000, miles=30000, driver='Max
>>> truck1.driver
'Max'
>>> truck1.mechanic
'Sue'
>>> truck1.value()
40000
>>> truck1.service()
10000
```

- Now when we create a `Van` instance called `van1` Python searches the inheritance tree for an `__init__()` method:

    1. It does not find one attached to the `van1` object.
    2. It does not find one attached to the `Van` class.
    3. It does not find one attached to the `Goods` class.
    4. It finds and executes the `__init__()` method attached to the `Vehicle` class.

- Now when we create a `Truck` instance called `truck1` Python searches the inheritance tree for an `__init__()` method:

    1. It does not find one attached to the `truck1` object.
    2. It finds and executes the `__init__()` method attached to the `Truck` class.
    3. The latter `__init__()` method invokes the `__init__()` method of its superclass and Python begins a new search for an `__init__()` method.
    4. It does not find one attached to the `Goods` class.
    5. It finds and executes the `__init__()` method attached to the `Vehicle` class to initialise what trucks have in common with all vehicles.

Finishing off

- As it currently stands our `Vehicle` class looks like this:

```
# vehicles_v04.py
from datetime import datetime
current_year = datetime.now().year

class Vehicle(object):

    def __init__(self, make, model, year, cost, miles, driver):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver

    def value(self):
        age = current_year - self.year
        return round(self.cost * (1+self.depreciation_rate)**age)
```

```
    def service(self):
        return self.service_miles - self.miles % self.service_miles
```

- Looking at the `value()` and `service()` methods we see that in addition to the attributes initialised in `__init__()` they also make reference to `depreciation_rate` and `service_miles` attributes **which are not initialised** by `__init__()`.

- This presents us with the following rather disconcerting situation:

```
>>> from vehicles_v04 import Vehicle
>>> v1 = Vehicle('Unknown', 'Generic', 2008, cost=10000, miles=16000(
>>> v1.value()
AttributeError: 'Vehicle' object has no attribute 'depreciation_rate
```

- In the example above we create an instance of the `Vehicle` class supplying everything required by the class's `__init__()` method. Having done so it turns our we cannot use some of the class's methods! Yikes!

- On a related note suppose we wanted to add a `Moped` class to our fleet. Also suppose that we are unsure both about how rapidly mopeds depreciate in value and when exactly they require to be serviced. As things currently stand, for each subclass of `Vehicle` we are required to define both a `service_miles` and `depreciation_rate`.

- We can solve the above issues by associating with the `Vehicle` class default values for `service_miles` and `depreciation_rate`. If a subclass requires to do so it can **override** its parent class's values for these attributes (as is done by cars, trucks, vans, etc.). However if a subclass is happy with the default attribute values it need not redefine them.

- Adding default values for the `service_miles` and `depreciation_rate` attributes to the parent class and defining suitable `__str__()` methods gives our finished class:

```
# vehicles_v05.py
from datetime import datetime
current_year = datetime.now().year

class Vehicle(object):

    service_miles = 100000
    depreciation_rate = -0.05

    def __init__(self, make, model, year, cost, miles, driver):
        self.make = make
        self.model = model
        self.year = year
        self.cost = cost
        self.miles = miles
        self.driver = driver

    def value(self):
        age = current_year - self.year
        return round(self.cost * (1+self.depreciation_rate)**age)

    def service(self):
        return self.service_miles - self.miles % self.service_miles

    def __str__(self):
        l = []
        l.append('Make: {}'.format(self.make))
```

```python
        l.append('Model: {}'.format(self.model))
        l.append('Year: {}'.format(self.year))
        l.append('Cost: {}'.format(self.cost))
        l.append('Miles: {}'.format(self.miles))
        l.append('Driver: {}'.format(self.driver))
        l.append('Service: {}'.format(self.service_miles))
        l.append('Depreciation rate: {}'.format(self.depreciation_rate))
        return '\n'.join(l)

class Car(Vehicle):

    service_miles = 10000
    depreciation_rate = -0.1

class Motorcycle(Vehicle):

    service_miles = 5000
    depreciation_rate = -0.15

class Goods(Vehicle):

    depreciation_rate = -0.20

class Van(Goods):

    service_miles = 15000

class Truck(Goods):

    service_miles = 20000

    def __init__(self, make, model, year, cost, miles, driver, mechanic):
        super().__init__(make, model, year, cost, miles, driver)
        self.mechanic = mechanic

    def __str__(self):
        l = []
        l.append('Mechanic: {}'.format(self.mechanic))
        l.append(super().__str__())
        return '\n'.join(l)
```

```python
>>> from vehicles_v05 import Vehicle, Car, Truck
>>> v1 = Vehicle('Unknown', 'Generic', 2008, cost=10000, miles=160000, driver='
>>> v1.value()
5987
>>> car1 = Car('Honda', 'Civic', 2013, cost=20000, miles=16000, driver='Joe')
>>> print(car1)
Make: Honda
Model: Civic
Year: 2013
Cost: 20000
Miles: 16000
Driver: Joe
Service: 10000
Depreciation rate: -0.1
>>> truck1 = Truck('Scania', 'R420', 2014, cost=50000, miles=30000, driver='Max
>>> print(truck1)
Mechanic: Sue
Make: Scania
Model: R420
Year: 2014
Cost: 50000
Miles: 30000
Driver: Max
Service: 20000
Depreciation rate: -0.2
```

Object-oriented design

- When seeking to apply an object-oriented approach to solving a particular problem you might try following these steps:
    1. Write or draw about the problem
    2. Extract key concepts
    3. Create a class hierarchy
    4. Code classes and tests
    5. Repeat and refine