06-POO-heranca_e_polimorfismos

March 3, 2020

1 Programação Orientada aos Objetos (POO) - parte VI

Pedro Cardoso
(ISE/UAlg - pcardoso@ualg.pt)

1.1 herança: subclasses

Suponhamos agora que pretendíamos tratar de vários tipos de veículos de transporte - Carros, Motos, Camiões, ... - Barcos sem/com motor, ... - Aviões ... - ...

Sendo (possivelmente) **más soluções**, poderíamos: - criar uma classe genérica, mantendo nela a marca, o modelo, o dono, nž de passageiros, tamanho dos pneus, etc. e se não existe valor para o atributo em questão (um barco não tem pneus...!) deixaríamos esse atributo vazio. - reescrevemos tudo para cada classe específica, apesar de ser exatamente o mesmo código. E se alterar num sítio tenho de alterar em todos!!! E se acrescentar alguma funcionalidade também tenho de adicionar a todas! - E em relação aos métodos? Não faria sentido ter para alguns dos veículos (p.e., barcos) métodos para definir o tamanho dos pneus...

Em POO podemos relacionar classes de tal maneira que uma delas **herda** o que a outra tem. - Isto é uma relação de **classe mãe** e **classe filha**. - A classe estendida diz-se **super classe** - A classe que estende diz-se **sub classe**

Em resumo, se **B estende A** então - B herda de A todas a variáveis e métodos de instância que não sejam declarados como private - B pode definir novas variáveis e novos métodos - B pode redefinir variáveis e métodos herdados

```
In [1]:  class Vehicle:
             def __init__(self, brand, model, number_of_passengers=0, owner=None):
                 self.owner = owner
                 self.brand = brand
                 self.model = model
                 self.number_of_passengers = number_of_passengers

             def vehicle_info(self):
                 return f'Veiculo da marca {self.brand}, modelo {self.model}, com capacidade pa

             @property
             def owner(self):
                 return self.__owner
```

```
    @owner.setter
    def owner(self, owner):
        self.__owner = owner

    @property
    def brand(self):
        return self.__brand

    @brand.setter
    def brand(self, brand):
        self.__brand = brand

    @property
    def model(self):
        return self.__model

    @model.setter
    def model(self, model):
        self.__model = model

    @property
    def number_of_passengers(self):
        return self.__number_of_passengers

    @number_of_passengers.setter
    def number_of_passengers(self, number_of_passengers):
        self.__number_of_passengers = number_of_passengers
```

```
In [2]: v = Vehicle(owner='Margarida', brand='Fiat', model='500', number_of_passengers=4)
        print(v.vehicle_info())
```

```
Veiculo da marca Fiat, modelo 500, com capacidade para 4. O dono é Margarida.
```

Agora podemos começar a particularizar, supondo que todos os veiculos terrestres tem rodas... podemos juntar atributos/propriedades como sejam `land_velocity`, `number_of_wheels` e `wheels`

Note-se ainda que o inicilizador chama o inicializador de `Vehicle` para inicializar os atributos/propriedades de `Vehicle`

```
In [3]: class  LandVehicle(Vehicle):

            def __init__(self, land_velocity, wheels, number_of_wheels, brand, model, numbe

                # chama o construtor de Vehicle para inicializar os atributos/propriedades
                super().__init__(owner=owner, brand=brand, model=model, number_of_passenger

                self.land_velocity = land_velocity;
```

```
        self.wheels = wheels;
        self.number_of_wheels = number_of_wheels;

    def vehicle_info(self): # redefinição do método
        return  f'''Veiculo da marca {self.brand}, modelo {self.model}, com capaci
            O dono é {self.owner}.
            Tem {self.number_of_wheels} rodas com as especificações {self.wheels}
            A velocidade em terra é {self.land_velocity}
            '''

    @property
    def land_velocity(self):
        return self.__land_velocity

    @land_velocity.setter
    def land_velocity(self, land_velocity):
        self.__land_velocity = land_velocity

    @property
    def number_of_wheels(self):
        return self.__number_of_wheels

    @number_of_wheels.setter
    def number_of_wheels(self, number_of_wheels):
        self.__number_of_wheels = number_of_wheels

    @property
    def wheels(self):
        return self.__wheels

    @wheels.setter
    def wheels(self, wheels):
        self.__wheels = wheels
```

```
In [4]: lv = LandVehicle(land_velocity=200, wheels='225/55 R 17 97 W', number_of_wheels=4, owne
        print(lv.vehicle_info())
```

```
Veiculo da marca Fiat, modelo 500, com capacidade para 4.
            O dono é Margarida.
            Tem 4 rodas com as especificações 225/55 R 17 97 W
            A velocidade em terra é 200
```

E podem pode ser ainda mais particularizado num **carro** juntando atributos/propriedades `engine` e `number_of_doors`

Note-se ainda que o inicilizador chama o inicializador de `LandVehicle` para inicializar os atributos/propriedades de `LandVehicle` (e implicitamente de `Vehicle`)

```
In [5]: class Car(LandVehicle):

            def __init__(self, engine, number_of_doors,  land_velocity, wheels, number_of_wheel

                # chama o construtor de LandVehicle para inicializar os atributos/propriedades
                super().__init__(land_velocity=land_velocity, wheels=wheels, number_of_wheels=n

                self.engine = engine
                self.number_of_doors = number_of_doors

            def vehicle_info(self): # redefinição do método
                return  f'''Veiculo da marca {self.brand}, modelo {self.model}, com capacidade
                    O dono é {self.owner}.
                    Tem {self.number_of_wheels} rodas com as especificações {self.wheels}
                    A velocidade em terra é {self.land_velocity}.
                    Tem um motor com {self.engine}cc e {self.number_of_doors} portas.
                    '''
            @property
            def engine(self):
                return self.__engine

            @engine.setter
            def engine(self, engine):
                self.__engine = engine

            @property
            def number_of_doors(self):
                return self.__number_of_doors

            @number_of_doors.setter
            def number_of_doors(self, number_of_doors):
                self.__number_of_doors = number_of_doors

In [6]: c = Car(engine='1500 cc', number_of_doors=5, land_velocity=200, wheels='225/55 R 17 97
        print(c.vehicle_info())

Veiculo da marca Fiat, modelo 500, com capacidade para 4.
            O dono é Margarida.
            Tem 4 rodas com as especificações 225/55 R 17 97 W
            A velocidade em terra é 200.
            Tem um motor com 1500 cccc e 5 portas.
```

Quais são os atributos de uma instância de Car (métodos e atributos começados por só '_')?

```
In [7]: list(filter(lambda x : (x[0] == '_' and x[1] != '_'), dir(c)))

Out[7]: ['_Car__engine',
         '_Car__number_of_doors',
```

```
         '_LandVehicle__land_velocity',
         '_LandVehicle__number_of_wheels',
         '_LandVehicle__wheels',
         '_Vehicle__brand',
         '_Vehicle__model',
         '_Vehicle__number_of_passengers',
         '_Vehicle__owner']
```

E que métodos e propriedades tem Car?

```
In [8]: list(filter(lambda x : (x[0] != '_'), dir(c)))
```

```
Out[8]: ['brand',
         'engine',
         'land_velocity',
         'model',
         'number_of_doors',
         'number_of_passengers',
         'number_of_wheels',
         'owner',
         'vehicle_info',
         'wheels']
```

E obviamente podemos usar os métodos/propriedades herdados pela classe `Car`

```
In [9]: c.owner = 'João Pedro'
        print(c.vehicle_info())
```

```
Veiculo da marca Fiat, modelo 500, com capacidade para 4.
            O dono é João Pedro.
            Tem 4 rodas com as especificações 225/55 R 17 97 W
            A velocidade em terra é 200.
            Tem um motor com 1500 cccc e 5 portas.
```

Notemos que `Car.__dict__` devolve um dicionário com espaço de nomes do módulo

```
In [10]: Car.__dict__
```

```
Out[10]: mappingproxy({'__module__': '__main__',
                       '__init__': <function __main__.Car.__init__>,
                       'vehicle_info': <function __main__.Car.vehicle_info>,
                       'engine': <property at 0x7f5b5c50d048>,
                       'number_of_doors': <property at 0x7f5b5c50d598>,
                       '__doc__': None})
```

que é diferente de `dir()` que mostra também o que herdou

```
In [11]: print(dir(Car))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__
```

1.2 Algumas notas

1.2.1 Sobreposição (Overriding) de métodos

- Por vezes, no mecanismo de herança, uma classe herda métodos que não lhe servem.
- Nesse caso podemos redefinir esses métodos (Polimorfismo)

Nos exemplos anteriores vimos que o método `vehicle_info(self):` foi (re)defenido em todas as classes

1.2.2 Sobrecarga

- Ao trabalhar com linguagens que podem discriminar tipos de dados em tempo de compilação, a seleção entre as alternativas pode ocorrer em tempo de compilação. O ato de criar tais *funções alternativas* para seleção em tempo de compilação é geralmente chamado de **sobrecarga de função**.
- O Python é uma linguagem dinamicamente tipada, portanto o conceito de sobrecarga simplesmente não se aplica. No entanto, podemos criar funções alternativas em tempo de execução usando por exemplo argumentos opcionais (como no exemplo atrás)

```
In [12]: c1 = Car(engine='1500 cc', number_of_doors=5, land_velocity=200, wheels='225/55 R 17 9
         c2 = Car(engine='1500 cc', number_of_doors=5, land_velocity=200, wheels='225/55 R 17 9
```

```
In [ ]:
```