07-POO-classes_abstratas

March 3, 2020

1 Programação Orientada aos Objetos (POO) - parte VII

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

1.1 Classes abstratas

Uma classe define as características e o comportamento de um conjunto de objetos. No entanto, nem todas as classes são projetadas para instanciar / permitir a criação de objetos, i.e., algumas classes são usadas apenas para agrupar características comuns a diversas classes e, então, ser herdada por outras classes.

- As classes que não podem ser instanciadas são conhecidas como **classes abstratas**
- **Classe abstrata** corresponde à declaração de uma classe para as qual nunca pretendemos criar objetos/instanciar.
- Como **classes abstratas** só são usadas como superclasses em hierarquias de herança, são chamadas **superclasses abstratas**.
- As **classe abstrata** não podem ser usadas para instanciar objetos, porque são incompletas.
- As classes que não são abstratas, as que podem ser instanciadas, são conhecidas como **classes concretas**.
- As subclasses devem implementar as *partes ausentes* para se tornarem classes concretas

As classes abstratas podem ser utilizadas para dar a definição de métodos que *têm* de ser implementados em todas as suas subclasses, sem apresentar uma implementação para esses métodos. - Tais métodos são chamados de **métodos abstratos**. - Toda classe que possui pelo menos um **método abstrato** é uma **classe abstrata**, mas uma classe pode ser abstrata sem possuir nenhum método abstrato. - Em algumas linguagens, um **método abstrato** "não tem corpo", ou seja, apresenta-se apenas uma "assinatura".

1.1.1 Solução 1

"Declaram-se" os métodos e depois levanta-se uma exceção de método não implementado

```
In [1]: class Vehicle:
            def __init__(self, owner, brand):
                self.owner = owner
                self.brand = brand
```

```
    def vehicle_info(self):
        raise NotImplementedError("vehicle_info: não implementado")

    @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
```

Mas a criação de um objeto é valida

```
In [2]: c = Vehicle('Fiat', 'Margarida')
```

Será levantada uma exceção que se chama o método `vehicle_info()`, mas não antes!

```
In [3]: c.vehicle_info()
```

```
---------------------------------------------------------------------------

NotImplementedError                       Traceback (most recent call last)

<ipython-input-3-bdd04d06f57c> in <module>()
----> 1 c.vehicle_info()


<ipython-input-1-79eec72b46a0> in vehicle_info(self)
      5
      6     def vehicle_info(self):
----> 7         raise NotImplementedError("vehicle_info: não implementado")
      8
      9     @property


NotImplementedError: vehicle_info: não implementado
```

1.1.2 Solução 2

Como 2ł solução podemos user o módulo `abc`, prevenindo que a classe seja instanciada

```
In [4]: from abc import ABC, abstractmethod

        class Vehicle(ABC):
            def __init__(self, owner, brand):
                self.owner = owner
                self.brand = brand

            @abstractmethod
            def vehicle_info(self):
                pass

            @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

In [5]: c = Vehicle('Fiat', 'Margarida')


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-5-47138bfa9fdd> in <module>()
    ----> 1 c = Vehicle('Fiat', 'Margarida')


        TypeError: Can't instantiate abstract class Vehicle with abstract methods vehicle_info
```

Temos pois de derivar a classe `Vehicle`

```
In [6]: class Car(Vehicle):
            def __init__(self, owner, brand, engine):
```

```
            super().__init__( owner, brand)
            self.engine = engine

        @property
        def engine(self):
            return self.__engine

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

mas não basta derivar da super classe

```
In [7]: c = Car('Margarida', 'Fiat', '1500 turbo')
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-7-9e5a5663fbd9> in <module>()
----> 1 c = Car('Margarida', 'Fiat', '1500 turbo')


TypeError: Can't instantiate abstract class Car with abstract methods vehicle_info
```

temos de implementar os métodos que foram identificados como abstratos

```
In [8]: class Car(Vehicle):
            def __init__(self, owner, brand, engine):
                super().__init__( owner, brand)
                self.engine = engine

            def vehicle_info(self): # implementação do método abstracto
                print(self.__dict__ )

            @property
            def engine(self):
                return self.__engine

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

```
In [9]: c = Car('Margarida', 'Fiat', '1500 turbo')
```

```
In [10]: c.vehicle_info()
```

```
{'_Vehicle__owner': 'Margarida', '_Vehicle__brand': 'Fiat', '_Car__engine': '1500 turbo'}
```

1.2 Exemplo

sugestão: corra o exemplo num terminal/pycharm/...

Comecemos por definir uma classe abstrata para jogos de tabuleiro com 2 jogadores que jogam alternadamente

```
In [11]: from abc import ABC, abstractmethod
         import random


         class Jogo(ABC):
             """ implementa uma classe para um jogo com 2 humanos"""

             def __init__(self):
                 print('bom jogo...')
                 self.inicializa_tabuleiro()

             @abstractmethod
             def joga_humano(self, jogador):
                 """ metodo que solicita ao humano :jogador: a proxima jogada e coloca-a no ta
                 :param jogador: numero do jogador (0 ou 1)
                 """
                 pass

             @abstractmethod
             def terminou(self):
                 """ devolve `True` se foi verificada a condicao de paragem, i.e., um jogador
                 devolve `False`caso contrário """
                 pass

             @abstractmethod
             def mostra_tabuleiro(self):
                 """desenha o tabuleiro"""
                 pass

             @abstractmethod
             def inicializa_tabuleiro(self):
                 """ inicializa o tabuleiro de jogo"""
                 pass

             @abstractmethod
             def ha_jogadas_possiveis(self):
                 """ verifica se ainda ha jogadas possiveis ou se o jogo esta empatado"""
                 pass

             def jogar(self):
                 """ corre o jogo..."""
                 jogador = random.randint(0, 1)
```

```
        while True:
            self.mostra_tabuleiro()
            self.joga_humano(jogador)
            if self.terminou():
                self.mostra_tabuleiro()
                print(f'o jogador {jogador} ganhou!!')
                return
            elif not self.ha_jogadas_possiveis():
                print(f'Empataram!!!')
                return
            jogador = (jogador+1) % 2
```

Agora podemos criar uma classe concreta, definindo somente os métodos abstratos

```
In [12]: class Galo(Jogo):

    def inicializa_tabuleiro(self):
        self.numero_de_jogadas_realizadas = 0  # conta as jogadas, serve para saber s
        self.tabuleiro = {(l, c): ' ' for l in range(3) for c in range(3)}  # o tabul

    def _le_linha_coluna_valida(self, s):
        """ metodo auxiliar para ler uma posicao que seja 0, 1 ou 2"""
        while True:
            x = input(s)
            if x in ['0', '1', '2']:
                return int(x)

    def joga_humano(self, jogador):
        print(f'jogador {jogador} insira a sua jogada')
        while True:
            linha = self._le_linha_coluna_valida('Linha?')
            coluna = self._le_linha_coluna_valida('Coluna?')
            if self.tabuleiro[(linha, coluna)] == ' ':  # verifica se a posicao nao e
                self.tabuleiro[(linha, coluna)] = ['X', 'O'][jogador]
                self.numero_de_jogadas_realizadas += 1
                return
            else:
                print('Jogada invalida. Tente de novo')

    def terminou(self):
        lista_posicaoes_ganhadoras = (
            ((0, 0), (0, 1), (0, 2)),  # linha 0
            ((1, 0), (1, 1), (1, 2)),  # linha 1
            ((2, 0), (2, 1), (2, 2)),  # linha 2
            ((0, 0), (1, 0), (2, 0)),  # coluna 0
            ((0, 1), (1, 1), (2, 1)),  # coluna 1
            ((0, 2), (1, 2), (2, 2)),  # coluna 2
            ((0, 0), (1, 1), (2, 2)),  # diagonal
```

```
                ((0, 2), (1, 1), (2, 0)),  # anti-diagonal
            )

            for teste in lista_posicaoes_ganhadoras:
                if self.tabuleiro[teste[0]] == self.tabuleiro[teste[1]] == self.tabuleiro
                        and self.tabuleiro[teste[0]] != ' ':
                    return True  # encontrou posicao ganhadora
            return False

        def mostra_tabuleiro(self):
            print(13 * '-')
            for l in range(3):
                for c in range(3):
                    print(f'| {self.tabuleiro[(l, c)]} ', end='')
                print('|\n' + 13*'-')

        def ha_jogadas_possiveis(self):
            return self.numero_de_jogadas_realizadas < 9
```

In [ ]: