Raciocínio Computacional

UNIDADE 08

Métodos e classes em Python

Esta unidade tem por objetivo dar uma visão introdutória aos conceitos de programação orientada a objetos em Python. Seguindo a linha de várias linguagens do mercado, o Python se apresenta em dois paradigmas de programação: procedural e orientado a objetos. Até este momento, ele foi trabalhado nesta disciplina em seu paradigma procedural, fazendo uso das estruturas de programação divididas em módulos lógicos que interagem entre si. Nesta unidade, será visto como distribuir as funcionalidades das aplicações em classes, que servirão de base para objetos, os quais vão interagir entre si em programas de computação.

Métodos e classes em Python

O paradigma de programação orientado a objetos demanda a organização da lógica de implementação em estruturas organizadas e de escopo próprio, chamadas classes. Cada classe tem um escopo interno definido, fazendo uso de suas próprias variáveis, as quais são acessadas e modificadas a partir de chamadas feitas por uma interface pública, formada por métodos de classe. Esses métodos nada mais são do que funções Python, que permitem que a classe interaja com o código fora de seu escopo para entrada e saída de dados.

Essas classes, porém, atuam como estruturas-modelo para criação de objetos, os quais são armazenados em variáveis de tipos de dado ditos customizáveis, que levam o nome e comportamento da classe que lhes dá origem. Cada objeto criado a partir de uma classe customizada recebe uma referência a seus métodos e uma cópia de cada uma de suas variáveis. Assim sendo, os vários objetos de uma classe apresentam estados individuais persistentes, com mesmos comportamentos lógicos definidos na estrutura que os originou.

Pela definição dada, é possível dizer que uma classe é como um tipo de variável, porém mais complexo. O tipo de dado primitivo diz como o dado pode ser manipulado, mas, para usá-lo, deve-se criar uma variável daquele tipo e sobre ela realizar as ações desejadas. Uma lista, por exemplo, é um tipo de dado mais complexo, sobre o qual se deve criar uma variável para ter acesso à forma de armazenamento de dados e aos métodos que permitem manipulá-los. Uma classe é um tipo ainda mais customizável, devendo uma variável ser criada com base em seu tipo e sobre ela se realizar as ações desejadas.

Classes em Python

Pensar em classes como um tipo customizável de dados é uma forma interessante de aprender programação orientada a objetos. É fácil confundir os conceitos de módulos em Python com o conceito de classes, porque normalmente a implementação da classe é feita em arquivo separado. No entanto, módulos são blocos de códigos que tratam de um mesmo assunto organizados em um mesmo arquivo, enquanto classes fazem uso de tipos de dado primitivos ou não, manipulando-os em uma estrutura de dados mais complexa.

Por exemplo, na unidade anterior, foi criado um módulo de gerenciamento de clientes, no qual foram utilizadas as diversas ações comuns para manipulação, como cadastro, edição, exclusão, entre outras. Os dados dos clientes foram armazenados em estruturas heterogêneas de dados, como listas e dicionários, uma vez que o dado **Cliente** não é uma informação simples, mas, sim, a composição de diversos dados primitivos, como nome, CPF etc.

A maneira mais simples de pensar esses dados seria criar um tipo de variável customizável, que internamente gerenciaria seus dados internos. Além disso, esse tipo personalizado poderia manipular suas informações de diversas formas, fazendo uso de funções próprias, as quais seriam seus métodos. Segue a implementação inicial de uma classe **Cliente**:

```
1. # Classe Cliente
2.
3. class Cliente:
4.     nome = "João"
5.     cpf = "123.456.789-00"
6.
7.     def imprimir(self):
8.         print(f"Cliente:", self.nome, "- CPF:", self.cpf)
```

Na linha 03, é possível perceber que a criação de uma classe demanda o uso da palavra reservada do Python ***class***. A classe deve possuir um nome, o qual deve ser relacionado com sua função no programa. Como toda estrutura em bloco em Python, o conteúdo da classe é limitado entre os dois-pontos após seu nome e o último nível de indentação apresentado no código.

Nas linhas 04 e 05, os dados primitivos que compõem a classe são criados e inicializados, sendo usados internamente na classe em suas funções. Nas linhas 07 e 08, é feita a definição do método de classe **imprimir**. Diz-se que é um método de classe, pois não pode ser chamado diretamente pelo programa, a não ser a partir da criação de um objeto da classe **Cliente**. Por motivos de sintaxe do Python, todo método de classe deve passar como primeiro parâmetro, obrigatoriamente, o objeto ***self***, que é uma referência à própria classe à qual o método pertence.

Como mencionado anteriormente, para fazer uso da classe, deve ser criada uma variável para armazenar o seu tipo, podendo chamar os seus métodos internos para interagir com ela. Segue exemplo de como seria criado um objeto da classe **Cliente**:

```
11.  cliente = Cliente()
12.  cliente.imprimir()
13.  print(cliente.nome)
```

O resultado dessa execução será:

```
C:\
```

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Cliente.py
Cliente: João - CPF: 123.456.789-00
João


Process finished with exit code 0
```

Na linha 11, é criado um objeto da classe **Cliente**, atribuído à variável **cliente**. A partir dela, o método de classe **imprimir** pode ser acessado, bem como as demais variáveis da classe (que passam a receber o nome de atributos de classe).

Para usar a mesma classe para representar outro cliente, é necessário trocar os dados iniciais dela, como segue:

```
11.  cliente = Cliente()
12.  cliente.nome = "Maria"
13.  cliente.cpf = "987.654.321-00"
14. cliente.imprimir()
```

```
C:\
```

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Cliente.py
Cliente: Maria - CPF: 987.654.321-00


Process finished with exit code 0
```

Comportamento genérico de classes

As classes devem apresentar comportamento genérico, uma vez que podem ser criados diversos objetos diferentes a partir delas, cada um com suas próprias informações. Assim, a implementação da classe **Cliente** em sua primeira versão não ficou muito boa, devendo ser repensada:

```
1. class Cliente:
2.
3.     def definir_dados(self, nome, cpf):
4.         self.nome = nome
5.         self.cpf = cpf
6.
7.     def imprimir(self):
8.         print(f"Cliente:", self.nome, "- CPF:", self.cpf)
9.
10.
11. cliente = Cliente()
12. cliente.definir_dados("Maria", "987.654.321-00")
13. cliente.imprimir()
```

Como no método **definir_dados**, os valores iniciais dos dados internos à classe são definidos antes de seu uso. Segue resultado desta implementação:

```
C:\


/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Cliente.py
Cliente: Maria - CPF: 987.654.321-00


Process finished with exit code 0
```

É importante observar que se torna obrigatório chamar o método **definir_dados()** antes de utilizar a classe com todas as suas funcionalidades, pois esse método é responsável por definir os dados internos da classe. Uma chamada que tente usar um dado não inicializado vai resultar em um erro. Por exemplo:

```
11.  cliente = Cliente()
12. cliente.imprimir()
```

C:\

```
Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Cliente.py
Cliente: Maria - CPF: 987.654.321-00

Traceback (most recent call last):
    File  "/Users/user/projects/untitled1/ExAplic01.py  ",
line 12, in <module>
    cliente.imprimir()
    File  "/Users/user/projects/untitled1/ExAplic01.py  ",
line 07, in imprimir
    print(f"Cliente:", self.nome, "- CPF:", self.cpf)
AttributeError: 'Cliente' object has no attribute 'nome'


Process finished with exit code 1
```

Construtor de classe

Uma forma de evitar o esquecimento de chamar uma função específica para inicializar os dados da classe é utilizar um construtor, que é uma chamada obrigatória na criação do objeto que solicita as informações internas iniciais da classe. No caso da classe **Clientes**, seria possível pedir que um cliente sempre fosse criado passando, no mínimo, seu nome e CPF. Segue implementação:

```
1. class Cliente:
2.
3.     def __init__(self, nome, cpf):
4.         self.nome = nome
5.         self.cpf = cpf
6.
7.     def alterar_dados(self, nome, cpf):
8.         self.nome = nome
9.         self.cpf = cpf
10.
11.     def imprimir(self):
12.         print(f"Cliente:", self.nome, "- CPF:", self.cpf)
13.
14.
15. cliente = Cliente("João", "123.456.789-00")
```

```
16. cliente.imprimir()
```

Na linha 03, é possível observar o método construtor, que possui o nome ***__init__*** e obriga que os dados sejam informados na criação do objeto, por isso o nome de construtor. Na linha 07, foi alterado o nome do método **definir_dados()** para **alterar_dados()**, visto ser esta sua nova função. Na linha 15, por fim, a chamada à criação de um objeto da classe **Cliente** demanda a passagem dos dois valores: **nome** e **cpf**.

Ao contrário de outras linguagens de programação, o Python não permite a criação de diversos construtores, porém, conforme as regras de funções, é possível dar valores-padrão para os parâmetros do construtor, como, por exemplo:

```
def __init__(self, nome, cpf="000.000.000-00"):
```

Neste caso, se o valor de **cpf** não for informado na criação do objeto, o atributo **cpf** receberá o valor-padrão “000.000.000-00”.

Escopo de acesso em classes

Nos exemplos trabalhados até o momento, o escopo de todas as informações internas à classe é público, uma vez que é possível acessar os dados internos a partir da sintaxe do ponto. Por exemplo:

```
Print(cliente.nome)
```

Esta é uma ação perfeitamente possível.

A fim de garantir uma proteção aos dados internos da classe, pode-se definir um escopo privado para eles, ou seja, os dados só podem ser alterados pelo código interno da classe, e não por uma chamada externa sem controle.

Essa implementação é feita colocando dois ***underscores*** (_) antes do nome do parâmetro. Segue um exemplo:

```
1. class Cliente:
2.
3.     def __init__(self, nome, cpf):
4.         self.__nome = nome
5.         self.__cpf = cpf
6.
7.     def alterar_dados(self, nome, cpf):
8.         self.__nome = nome
```

```
 9.         self.__cpf = cpf
10.
11.     def imprimir(self):
12.         print(f"Cliente:", self.__nome, "- CPF:",
    self.__cpf)
```

Dessa forma, uma chamada ao parâmetro **cpf** ou mesmo a nomenclatura de escopo privado **__cpf** vão gerar um erro de ***AttributError*** na execução, como segue:

```
17.  cliente = Cliente("João", "123.456.789-00")
18.  print(cliente.__cpf)
```

C:\

```
Cliente: Maria - CPF: 987.654.321-00

Traceback (most recent call last):
    File  "/Users/user/projects/untitled1/ExAplic01.py  ",
line 18, in <module>
    print(cliente.__cpf)
AttributeError: 'Cliente' object has no attribute '__cpf'


Process finished with exit code 1
```

A partir deste momento, o acesso aos dados deve ser feito a partir de métodos públicos da classe, tanto para alteração quanto para leitura de valores. Segue um exemplo:

```
 1. class Cliente:
 2.
 3.     def __init__(self, nome, cpf):
 4.         self.__nome = nome
 5.         if self.__cpf_valido(cpf):
 6.             self.__cpf = cpf
 7.         else:
 8.             self.__cpf = "CPF inválido"
 9.
10.     def imprimir(self):
11.         print(f"Cliente:", self.__nome, "- CPF:",
    self.__cpf)
12.
13.     def mudar_cpf(self, cpf):
14.         if self.__cpf_valido(cpf):
```

```
            self.__cpf = cpf
        else:
            self.__cpf = "CPF inválido"

    def ler_cpf(self):
        return self.__cpf

    def __cpf_valido(self, cpf):
        if len(cpf) == 11:
            return True
        else:
            return False


cliente = Cliente("João", "12345678900")
print(cliente.ler_cpf())
cliente.mudar_cpf("12345")
print(cliente.ler_cpf())
```

A saída desta implementação será:

C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Cliente.py
12345678900
CPF inválido


Process finished with exit code 0
```

Nesta implementação, além de os atributos da classe serem privados, o método **__cpf_valido()** também é, não havendo motivos para ele ser acessado de fora da classe, uma vez que executa apenas uma função interna sobre os dados. Para fazer o método privado, basta seguir os mesmos princípios dos atributos e colocar, antes de seu nome, dois ***underscores***.

Quando da execução do construtor, é verificado, por meio da função **__cpf_valido()**, se o CPF possui 11 dígitos. Caso não possua, a função retorna ***False***. Assim, o construtor guarda o valor “CPF inválido” para o **cpf** em caso de não conseguir validar e o valor do

cpf passado no caso de validação correta. Esse processo é anterior à atribuição do valor interno, o que é mais seguro e garante os valores.

O usuário ainda pode mudar o valor do **cpf** pela função **mudar_cpf()**, que é um método público e está sujeito às mesmas regras de validação do construtor. Da mesma forma, o método público **ler_cpf()** permite ler o valor do **cpf** armazenado.

Esta é uma forma correta e elegante de interagir com os valores de escopo de classe, pois podem ser feitas diversas validações, permitindo que o estado dos atributos internos sempre seja consistente.

O uso de métodos e atributos privados em classes é dito uma boa prática de programação.

Trabalhando com classes em módulos diferentes

As classes normalmente são criadas em arquivos de módulo separados, para fins de organização de código e facilidade de manutenção do projeto. Assim, para fazer uso da classe em outro módulo do sistema, deve-se utilizar a mesma sintaxe do trato de módulos. Por exemplo, para usar a classe **Clientes** em outro módulo, deve-se fazer:

```
1. from Cliente import Cliente
2.
3. cliente = Cliente("João", "12345678900")
4. print(cliente.ler_cpf())
5. cliente.mudar_cpf("12345")
6. print(cliente.ler_cpf())
```

Da mesma forma, no arquivo **Cliente.py**, é possível continuar a verificação da execução do arquivo como módulo principal para manter testes da classe, caso deseje:

```
1. class Cliente:
2.
3.     def __init__(self, nome, cpf):
4.         self.__nome = nome
5.         if self.__cpf_valido(cpf):
6.             self.__cpf = cpf
7.         else:
8.             self.__cpf = "CPF inválido"
9.
10.     def imprimir(self):
11.         print(f"Cliente:", self.__nome, "- CPF:",
    self.__cpf)
```

```
12.
13.     def mudar_cpf(self, cpf):
14.         if self.__cpf_valido(cpf):
15.             self.__cpf = cpf
16.         else:
17.             self.__cpf = "CPF inválido"
18.
19.     def ler_cpf(self):
20.         return self.__cpf
21.
22.     def __cpf_valido(self, cpf):
23.         if len(cpf) == 11:
24.             return True
25.         else:
26.             return False
27.
28.
29. if __name__ == '__main__':
30.     cliente = Cliente("Maria", "11111111111")
31.     cliente.imprimir()
```

As linhas 30 e 31 só serão executadas caso o arquivo **Cliente.py** seja executado diretamente. Caso seja feita sua importação, esses códigos não serão executados, ou seja, mesmo comportamento visto para o trabalho com módulos e importação de módulos.

Exemplos de aplicação

Exemplo de aplicação 1: Elabore uma classe Quadrado, que receba em seu construtor o lado e tenha como função para seus métodos calcular perímetro, área e diagonal, bem como um método de impressão informando: “Quadrado de lado {lado}”. Ao final, deve testar a classe e todos os seus métodos.

```
1. import math
2.
3.
4. class Quadrado:
5.
6.     def __init__(self, lado):
7.         self.__lado = lado if lado >= 0 else 0
8.
9.     def calc_perimetro(self):
10.         return self.__lado * 4
```

```
11.
12.     def calc_area(self):
13.         return self.__lado ** 2
14.
15.     def calc_diagonal(self):
16.         return self.__lado * math.sqrt(2)  # math.sqrt
    calcula raiz quadrada
17.
18.     def imprimir(self):
19.         print("Quadrado de lado", self.__lado)
20.
21.
22. if __name__ == '__main__':
23.     quadrado = Quadrado(5)
24.     quadrado.imprimir()
25.     print("Perímetro:", quadrado.calc_perimetro())
26.     print("Área:", quadrado.calc_area())
27.     print(f"Diagonal: {quadrado.calc_diagonal():.2f}")
```

C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Quadrado.py
Quadrado de lado 5
Perímetro: 20
Área: 25
Diagonal: 7.07


Process finished with exit code 0
```

Exemplo de aplicação 2: Elabore uma classe que simule um dado de jogo. No construtor, deve-se informar quantas faces possui o dado. Caso o valor informado seja inválido, o dado será padrão, de seis faces. Deve ter um método privado de calcular a jogada e um método público de retorno do valor do dado para o usuário. Ao final, deve testar a classe e todos os seus métodos.

```
1. import random
2.
3.
4. class Dado:
5.
```

```
6.      def __init__(self, faces=6):
7.          self.__faces = faces if faces > 2 else 6
8.
9.      def valor_jogada(self):
10.         return self.__calcular_jogada()
11.
12.     def __calcular_jogada(self):
13.         return random.randint(1, self.__faces)
14.
15.
16. if __name__ == '__main__':
17.     dado4 = Dado(4)
18.     print(dado4.valor_jogada())
19.     dado = Dado(1)
20.     print(dado.valor_jogada())
```

C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Dado.py
2
2


Process finished with exit code 0
```

EXERCÍCIO

Classes

Exercício de fixação 1: Crie uma classe Televisor, em que o usuário é capaz de trocar de canais e aumentar/diminuir o volume. O limite de canais válidos vai de 1 a 15 e o limite de volume válido vai de 0 a 10. No caso de um canal inválido, deve ser definido como padrão o canal 1.

 Resolução do exercício 

Resolução exercício 1

```
 1. class Televisor:
 2.
 3.     def __init__(self):
 4.         self.__canal = 1
 5.         self.__volume = 0
 6.
 7.     def aumentar_volume(self):
 8.         volume = self.__volume + 1
 9.         self.__volume = volume if volume <=
    10 else 10
10.         self.__imprimir()
11.
12.     def diminuir_volume(self):
13.         volume = self.__volume - 1
14.         self.__volume = volume if volume >= 0
    else 0
15.         self.__imprimir()
16.
17.     def trocar_canal(self, canal):
18.         if 1 <= canal <= 15:
19.             self.__canal = canal
20.         else:
21.             self.__canal = 1
22.         self.__imprimir()
23.
24.     def __imprimir(self):
25.         print("Televisor")
26.         print("Volume:", self.__volume, "-
    Canal:", self.__canal)
27.
28.
29. if __name__ == '__main__':
30.     televisor = Televisor()
31.     televisor.trocar_canal(5)
32.     televisor.aumentar_volume()
33.     televisor.aumentar_volume()
34.     televisor.trocar_canal(20)
35.     televisor.diminuir_volume()
```

C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Televisor.p
y
Televisor
Volume: 0 - Canal: 5
Televisor
Volume: 1 - Canal: 5
Televisor
Volume: 2 - Canal: 5
Televisor
Volume: 2 - Canal: 1
Televisor
Volume: 1 - Canal: 1


Process finished with exit code 0
```

Exercício de fixação 2: Crie uma classe de cartas de baralho com o nome de Carta. Ao imprimir a carta, deve ser mostrada uma carta de 1 a 10 ou figuras (dama, valete ou rei) e um naipe (ouros, espadas, copas ou paus).

Resolução do exercício

Resolução exercício 2

```
1. import random
2.
3.
4. class Carta:
5.     __valor = 0
6.     __valores = ["1", "2", "3", "4", "5",
   "6", "7", "7", "9", "10", "dama", "valete",
   "rei"]
7.     __naipe = 0
8.     __naipes = ["ouros", "espadas", "copas",
   "paus"]
9.
10.    def __init__(self):
11.        self.__sortear()
```

```
12.
13.     def imprimir(self):
14.         print(self.__valores[self.__valor],
    "de", self.__naipes[self.__naipe])
15.         self.__sortear()
16.
17.     def __sortear(self):
18.         self.__valor = random.randint(0,
    len(self.__valores) - 1)
19.         self.__naipe = random.randint(0,
    len(self.__naipes) - 1)
20.
21.
22. if __name__ == '__main__':
23.     carta = Carta()
24.     for i in range(5):
25.         carta.imprimir()
```

C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Carta.py
9 de copas
2 de ouros
rei de paus
4 de copas
1 de espadas


Process finished with exit code 0
```

Exercício de fixação 3: Crie uma classe Baralho que possua um vetor com cada uma das cartas do baralho. A classe deve possuir um método sortear, que retorne uma das cartas. Essa carta não pode mais ser retornada. O baralho deve possuir, ainda, dois coringas. Ao retornar todas as 54 cartas, deve informar que as cartas acabaram.

Resolução do exercício

Resolução exercício 3

```
 1. import random
 2.
 3.
 4. class Baralho:
 5.
 6.     __valores = ["1", "2", "3", "4", "5",
    "6", "7", "7", "9", "10", "dama", "valete",
    "rei"]
 7.     __naipes = ["ouros", "espadas", "copas",
    "paus"]
 8.     __cartas = []
 9.
10.     def __init__(self):
11.         self.__criar_baralho()
12.         self.__embaralhar()
13.
14.     def __criar_baralho(self):
15.         for valor in
    range(len(self.__valores)):
16.             for naipe in
    range(len(self.__naipes)):
17.
    self.__cartas.append(self.__valores[valor] +
    " de " + self.__naipes[naipe])
18.         self.__cartas.append("** Coringa **")
19.         self.__cartas.append("** Coringa **")
20.
21.     def __embaralhar(self):
22.         for i in range(1000):
23.             pos1 = random.randint(0, 53)
24.             pos2 = random.randint(0, 53)
25.             aux = self.__cartas[pos1]
26.             self.__cartas[pos1] =
    self.__cartas[pos2]
27.             self.__cartas[pos2] = aux
28.
29.
30.     def sortear(self):
31.         if len(self.__cartas) == 0:
32.             return "Acabaram as cartas"
33.         else:
34.             return self.__cartas.pop()
35.
36.
37. if __name__ == '__main__':
38.     baralho = Baralho()
39.     for i in range(57):
40.         print(baralho.sortear())
```

C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Baralho.py
9 de espadas
9 de ouros
1 de copas
5 de ouros

...

2 de paus
6 de paus
Acabaram as cartas
Acabaram as cartas
Acabaram as cartas


Process finished with exit code 0
```

Exercício de fixação 4:

Crie uma classe Carro com as seguintes funcionalidades:

a. Um veículo tem certo consumo de combustível (medido em km/l) e certa quantidade de combustível no tanque.
b. O consumo é especificado no construtor e o nível de combustível inicial é 0.
c. Forneça um método abastecer(), que abasteça o carro com o valor passado de litros de combustível.
d. Forneça um método andar(), que simule o ato de dirigir o veículo por certa distância, reduzindo o nível de combustível no tanque de gasolina.
e. Forneça um método nivel_combustivel(), que retorne o nível atual de combustível.

 Resolução do exercício 

Resolução exercício 4

```
 1. class Carro:
 2.
 3.     __combustivel = 0
 4.
 5.     def __init__(self, autonomia):
 6.         self.__autonomia = autonomia
 7.
 8.     def abastecer(self, litros):
 9.         if litros >= 0:
10.             self.__combustivel += litros
11.
12.     def andar(self, distancia):
13.         litros = distancia / self.__autonomia
14.         if litros >= self.__combustivel:
15.             print("O carro parou sem
    combustível!")
16.             self.__combustivel = 0
17.         else:
18.             self.__combustivel -= litros
19.
20.     def nivel_combustivel(self):
21.         print(f"O carro ainda tem
    {self.__combustivel:.2f} litros de
    combustível!")
22.
23.
24. if __name__ == '__main__':
25.     carro = Carro(12)
26.     carro.abastecer(100)
27.     carro.andar(1000)
28.     carro.nivel_combustivel()
29.     carro.andar(300)
```

C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Carro.py
O carro ainda tem 16.67 litros de
combustível!
O carro parou sem combustível!


Process finished with exit code 0
```

Desafio

Exercício de fixação 5:

Crie uma classe Bomba que simule uma bomba de combustíveis de um posto. A bomba pode injetar dois tipos de combustível: gasolina ou álcool, podendo definir no construtor da classe a quantidade e o valor do litro de cada um. Deve possuir os seguintes métodos:

a. abastecer_valor(valor, tipo_combustivel), que mostra quantos litros foram colocados no veículo.
b. abastecer_litros(litros, tipo_combustivel), que mostra o valor no abastecimento.
c. fechar(), que mostra ao final do dia quanto foi ganho no valor total da bomba e quanto sobrou de cada tipo de combustível.

Resolução do exercício

Resolução exercício 5

```
1. class Bomba:
2.
3.     __total = 0
4.
```

```
    def __init__(self, gasolina, valor_gasolina, alcool, valor_alcool):
        self.__gasolina = gasolina
        self.__valor_gasolina = valor_gasolina
        self.__alcool = alcool
        self.__valor_alcool = valor_alcool

    def abastecer_valor(self, valor, combustivel):
        if combustivel == "gasolina":
            litros = valor / self.__valor_gasolina
            if litros > self.__gasolina:
                print("Não há gasolina suficiente na bomba!")
            else:
                self.__total += valor
                self.__gasolina -= litros
        elif combustivel == "alcool":
            litros = valor / self.__valor_alcool
            if litros > self.__alcool:
                print("Não há álcool suficiente na bomba!")
            else:
                self.__total += valor
                self.__alcool -= litros
        else:
            print("Tipo de combustível inválido!")

    def abastecer_litros(self, litros, combustivel):
        if combustivel == "gasolina":
            valor = litros * self.__valor_gasolina
            if litros > self.__gasolina:
                print("Não há gasolina suficiente na bomba!")
            else:
                self.__total += valor
                self.__gasolina -= litros
        elif combustivel == "alcool":
            valor = litros * self.__valor_alcool
            if litros > self.__alcool:
                print("Não há álcool suficiente na bomba!")
            else:
                self.__total += valor
                self.__alcool -= litros
        else:
            print("Tipo de combustível inválido!")

```

```
47.     def fechar(self):
48.         print(f"Valor do dia: R$
    {self.__total:.2f}!")
49.         print(f"Sobraram
    {self.__gasolina:.2f} litros de gasolina!")
50.         print(f"Sobraram {self.__alcool:.2f}
    litros de álcool!")
51.
52.
53. if __name__ == '__main__':
54.     bomba = Bomba(100, 4.3, 100, 3.1)
55.     bomba.abastecer_valor(100, "gasolina")
56.     bomba.abastecer_valor(50, "alcool")
57.     bomba.abastecer_litros(30, "gasolina")
58.     bomba.abastecer_litros(70, "alcool")
59.     bomba.fechar()
```

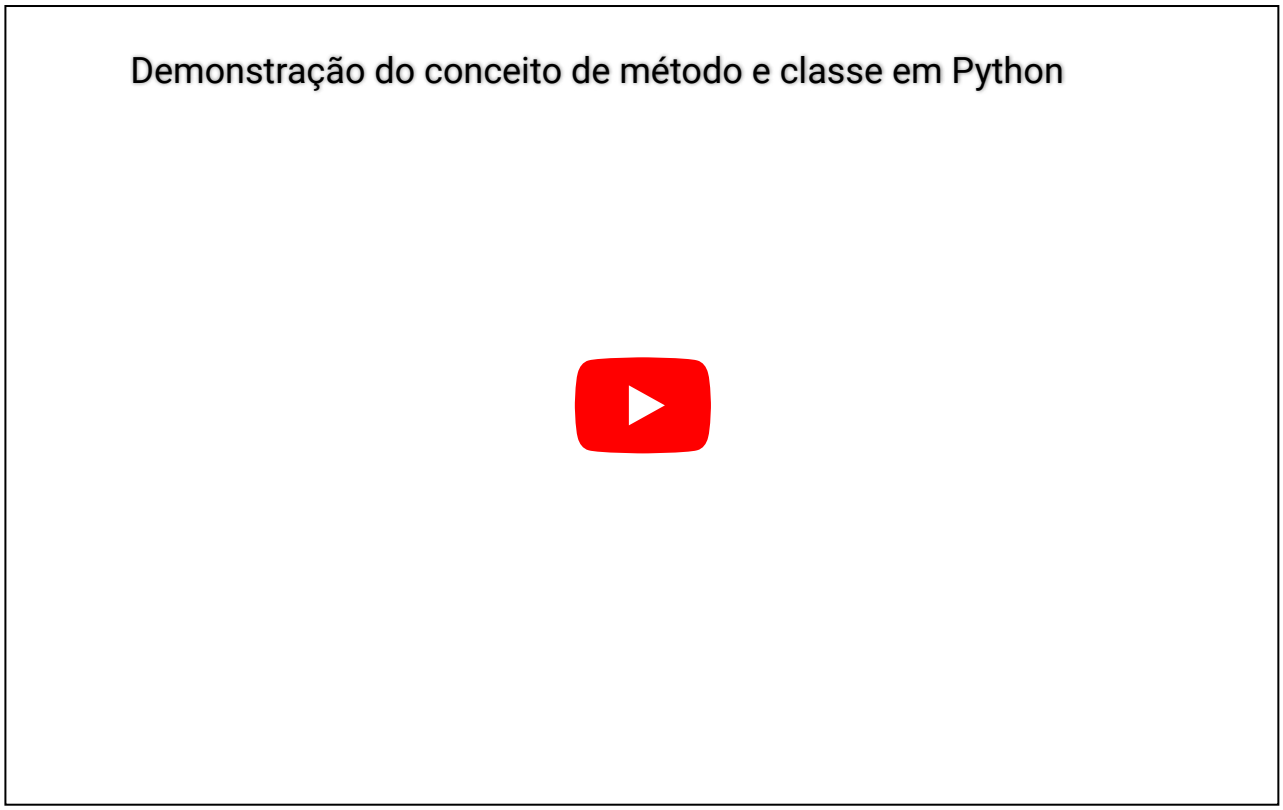
C:\

```
/Users/user/untitled1/bin/python
/Users/user/projects/untitled1/Bomba.py
Valor do dia: R$ 496.00!
Sobraram 46.74 litros de gasolina!
Sobraram 13.87 litros de álcool!


Process finished with exit code 0
```

Demonstração do conceito de método e classe em Python

Neste vídeo, veremos os conceitos de utilização de métodos e classes em Python, fazendo uma pequena introdução ao paradigma de orientação a objetos.



Conclusão

A respeito da utilização de métodos e classes em Python, podemos concluir que:

- Como outras linguagens comerciais do mercado, o Python permite trabalhar com orientação a objetos.
- Uma classe é um tipo de dado mais especializado, composto por atributos e métodos.
- Atributos em uma classe são variáveis de tipos de dado primitivos ou não.
- Métodos em uma classe são funções definidas no escopo interno dela.
- Para interagir com classes, é preciso criar objetos a partir delas, que são variáveis do seu tipo.
- Objetos de uma classe possuem referência aos métodos dela e uma cópia das variáveis de cada um de seus atributos.
- Métodos e atributos de classe podem ter escopo privado, colocando na frente de seus nomes dois símbolos de ***underscore*** (_).
- Uma classe pode ser importada da mesma forma que módulos quando se apresenta em um arquivo separado.

Referências

BANIN, S. L. **Python 3**: conceitos e aplicações uma abordagem didática. São Paulo: Érica, 2018.

PYTHON SOFTWARE FOUNDATION. **Documentação Python 3.9.0**. Disponível em: https://docs.python.org/pt-br/3/. Acesso em: 27 out. 2020.