LINGUAGEM DE PROGRAMAÇÃO

AULA 3

Prof. Wellington Rodrigo Monteiro

CONVERSA INICIAL

Olá! Nas nossas conversas anteriores, comentamos sobre o uso das bibliotecas e, ainda, sobre o trabalho com algoritmos de *machine learning* (ML). Por outro lado, o Python oferece também outras possibilidades de desenvolvimento: a sua versatilidade é uma das razões pelas quais estamos trabalhando com esta linguagem dentro destes estudos.

Dito isso, vamos conversar aqui sobre programação orientada a objetos (POO). Até este momento, é provável que você não tenha um contato profundo com este tema em Python, por outro lado, o uso de POO pode ser bem útil em sistemas corporativos de larga escala, servidores, sistemas de simulação, entre outros. Vamos lá?

TEMA 1 – UM PROBLEMA PARA SER RESOLVIDO PELA PROGRAMAÇÃO ORIENTADA A OBJETOS

Quando ingressamos em um curso de Tecnologia da Informação (TI) e aprendemos a desenvolver algoritmos utilizando linguagens de programação, quase sempre aprendemos da mesma forma. Independentemente da universidade, do país ou do professor, (quase) sempre aprendemos na seguinte linha de pensamento.

- **Algoritmos são como receitas de bolo**: possuem começo, meio e fim bem definidos.
- Também como receitas de bolo, foram feitos para **sempre funcionarem de uma mesma forma***:* não importa se você está fazendo a receita de noite ou de dia; se quem faz a receita é um adolescente ou um idoso; se a receita está sendo feita em uma fazenda ou em uma cozinha em um arranha-céu. **Sempre esperamos o mesmo resultado como consequência das mesmas entradas.**
    - Da mesma forma, um algoritmo pode ser executado em um *notebook* ou um *desktop*; um computador antigo ou um que acabou de ser comprado: sempre esperamos que ele produza o mesmo comportamento. Peguemos um exemplo de um aplicativo de *internet banking* de um dado banco. É esperado que uma transferência sempre

funcione da mesma forma, independentemente do modelo de celular, da cor do celular ou da versão do sistema em uso.

- E, como receitas de bolo, é relativamente fácil mudarmos alguma coisa no **final** do processo. Por outro lado, se precisarmos mudar algo no **meio do caminho**, teremos problemas. Ou você nunca se deparou com alguma atividade em certa disciplina em que você arrumou algo no meio do seu código só para se deparar com um conjunto de novos erros em outras partes do código?

Este paradigma chama-se ***programação estruturada***. Essa forma de criar algoritmos possui suas vantagens: ao usar esse paradigma, por exemplo, é relativamente fácil começarmos a criar algoritmos simples e de entendermos o que ele faz, na prática. Isso inclui casos como:

- leitura e escrita de dados com base no que um usuário pode digitar;
- estruturas de seleção (como o *se/se-então/senão*);
- estruturas de repetição (como o *para/enquanto*);
- estruturas de dados (como vetores, listas e matrizes);
- estruturas de divisão de código como as funções; e
- também permite que você encontre os erros de forma mais rápida porque ela possui menos *overhead* – isto é, necessita que você produza menos código que serviriam somente para **servir como fundação** para a sua lógica.

Durante esta aula, voltaremos a usar várias analogias (igual a receita de bolo). Logo, se estamos falando em receitas culinárias, a programação estruturada serve como uma maneira mais simples de ensinar a cozinhar para quem nunca cozinhou na vida. Observe o quadro a seguir.

Quadro 1 – Programação estruturada

Cozinha	Programação
Fazer salada	Mostrar um texto na tela
Ferver água	Guardar em uma variável o que um usuário digitou
Cozinhar um ovo	Usar estruturas de seleção (se/se-então/senão)
Fazer omelete	Usar estruturas de seleção com variáveis que o usuário digitou
Cozinhar miojo	Usar estruturas de repetição (para/enquanto)

Cozinhar arroz	Usar listas, vetores e/ou matrizes
Cozinhar carne moída	Usar funções
Preparar um almoço com arroz, carne moída, ovo e salada	Criar um algoritmo com várias funções, estruturas de seleção, repetição, variáveis e listas

Entenda o quadro anterior como apenas formas de **comparar** o esforço entre ambos e não compreenda isto de forma literal. Dito isso, provavelmente, preparar um almoço com arroz, carne moída, ovo e salada não faria de você um *chef* de cozinha, muito menos o prepararia para todos os tipos de pratos possíveis. Por outro lado, poder-se-ia dizer que é a base necessária para preparar pratos mais complexos: cozinhar arroz envolve coisas como gerenciar a temperatura do fogão, saber lidar com os utensílios da cozinha, saber controlar o tempo e os ingredientes necessários, entre outros. Logo, quando fôssemos aprender sobre algo mais complexo, todos saberiam que você já dominaria o básico (e que alguns poderiam até mesmo dizer que seria o “óbvio”).

É exatamente a mesma coisa conosco: primeiro, você aprendeu os fundamentos da programação para então chegar ao momento de trabalhar com paradigmas mais complexos.

Dito isso, vamos pensar no seguinte desafio: você foi encarregado de desenvolver o sistema de software de uma empresa de aluguel de bicicletas. Utilizando sensores e tecnologia recente, a missão é construir um software em Python que registre dados das bicicletas, por exemplo, a troca de marchas, a velocidade e a cadência (a quantidade de pedaladas por minuto). Utilizando todas as técnicas que listamos e com o que já vimos anteriormente, poderíamos fazer algo assim:

```
def alterar_marcha(bicicleta, marcha):
    bicicleta[0] = marcha if marcha >= 0 else 0
    return bicicleta

def alterar_velocidade(bicicleta, velocidade):
    bicicleta[1] = velocidade
    return bicicleta

def alterar_cadencia(bicicleta, cadencia):
    bicicleta[2] = cadencia if cadencia >= 0 else 0
    return bicicleta

bicicletas = {
    'Bicicleta 1': [0, 0, 0],
    'Bicicleta 2': [0, 0, 0],
    'Bicicleta 3': [0, 0, 0],
    'Bicicleta 4': [0, 0, 0],
    'Bicicleta 5': [0, 0, 0]
}
```

Veja que temos um dicionário chamado *bicicletas* e que possui cinco bicicletas. As chaves são referentes ao nome das bicicletas (como *Bicicleta 1* e *Bicicleta 4*), e os valores são uma lista contendo os dados referentes às bibliotecas. Por exemplo:

```
'Bicicleta 1': [0, 0, 0]
```

Armazena três valores para a *Bicicleta 1*. Sem olhar o código das funções, fica bem difícil de sabermos a que se refere o primeiro valor (ou o segundo, ou o terceiro). Ao analisarmos novamente as funções apresentadas, sabemos que o primeiro valor equivale à marcha, o segundo valor equivale à velocidade e o terceiro valor equivale à cadência.

Aqui vai a primeira dor de cabeça: e se, em vez de três características, estivéssemos trabalhando com 300 características? Em grandes sistemas corporativos, isto pode ser bem comum: imagine que não estamos mais trabalhando com bicicletas, mas sim com notas fiscais ou com dados cadastrais de empresas – já imaginou a quantidade de características que existem? Agora, pense se em algum dia um determinado algoritmo está dando um erro, e você precisa analisar o código. Pense só: se estivéssemos trabalhando com uma lista com 300 itens, como poderíamos encontrar rapidamente **qual é o elemento** que

está com problemas, principalmente se nunca vimos aquele código antes em nossa vida?

Para falar da segunda dor de cabeça, vamos pensar em um código no qual alteraríamos marcha, velocidade e cadência da Bicicleta 1:

```
# escolhendo uma bicicleta e mostrando seus dados iniciais
bicicleta = 'Bicicleta 1'
print(f'Marcha, velocidade e cadência da {bicicleta}: {bicicletas[bicicleta]}')

# alterando os dados da bicicleta
bicicletas[bicicleta] = alterar_marcha(bicicletas['Bicicleta 1'], 3)
bicicletas[bicicleta] = alterar_velocidade(bicicletas['Bicicleta 1'], 15)
bicicletas[bicicleta] = alterar_cadencia(bicicletas['Bicicleta 1'], -100)
print(f'Marcha, velocidade e cadência da {bicicleta}: {bicicletas[bicicleta]}')
```

Ao rodar esse código, temos os seguintes resultados:

```
Marcha, velocidade e cadência da Bicicleta 1: [0, 0, 0]
Modificadas a marcha, velocidade e cadência da Bicicleta 1: [3, 15, 0]
```

Veja que a marcha original (primeiro valor da lista) era 0 e foi alterada para 3; a velocidade (segundo valor da lista) era 0 e foi alterada para 15; a cadência (terceiro valor da lista) era 0 **e permaneceu em 0**. Por outro lado, observe novamente o código para a alteração da cadência: tentamos mudar a cadência de 0 para -100, mas sem sucesso:

```
bicicletas[bicicleta] = alterar_cadencia(bicicletas['Bicicleta 1'], -100)
```

Ao analisarmos novamente a função *"alterar_cadencia"*, vemos que isto, na realidade, é esperado: dentro da função existe um **if** que altera quaisquer cadências negativas para zero. Poderíamos dizer que é uma forma de evitarmos que tenhamos cadências abaixo de zero na nossa lista:

```
def alterar_cadencia(bicicleta, cadencia):
    bicicleta[2] = cadencia if cadencia >= 0 else 0
    return bicicleta
```

Mas é justamente aqui que a dor de cabeça se apresenta: se realmente quisermos, há a possibilidade de burlarmos essa regra – basta alterar diretamente o valor da lista:

```
# mudando novamente a cadência para um valor negativo
bicicletas[bicicleta][2] = -100
print(f'Marcha, velocidade e cadência da {bicicleta}: {bicicletas[bicicleta]}')
```

E, assim, a cadência estará negativa:

```
Marcha, velocidade e cadência da Bicicleta 1: [3, 15, -100]
```

É claro que o exemplo aqui é meramente ilustrativo, mas pensemos nos problemas de **segurança** (ou **confiança**) que isso poderia causar. E se, em vez da cadência de uma bicicleta, estivéssemos falando das notas do sistema da universidade? Ou do valor da nota fiscal em um sistema contábil? Ou do saldo na conta em um *internet banking*?

*Como garantir que as **mesmas regras de negócio** sempre serão aplicadas para todos os casos sendo analisados, sem espaço para possíveis alterações indesejadas e tendo uma boa confiabilidade?*

É aí que entra o paradigma da programação orientada a objetos.

TEMA 2 – NOÇÕES DE PROGRAMAÇÃO ORIENTADA A OBJETOS

A programação orientada a objetos (POO) e os bancos de dados relacionais que usam o SQL possuem uma origem muito similar: ambos basicamente procuram representar o mundo real em um código. Da mesma forma que dois objetos não ocupam jamais o mesmo lugar, duas bicicletas também são coisas completamente diferentes. Esta lógica também se aplica a duas notas fiscais ou duas pessoas. Em SQL, temos tabelas e, dentro das tabelas, temos registros e colunas. Em POO, algo similar também é possível.

Veja como seria o mesmo código apresentado, mas implementado usando técnicas de POO:

```
class Bicicleta():
    def __init__(self):
        self.__marcha = 0
        self.__velocidade = 0
        self.__cadencia = 0

    def alterar_marcha(self, marcha):
        self.__marcha = marcha if marcha >= 0 else 0

    def alterar_velocidade(self, velocidade):
        self.__velocidade = velocidade

    def alterar_cadencia(self, cadencia):
        self.__cadencia = cadencia if cadencia >= 0 else 0

    def mostrar_dados(self):
        return f'Marcha: {self.__marcha}, velocidade: {self.__velocidade}, cadênci
a: {self.__cadencia}.'

bicicletas = {
    'Bicicleta 1': Bicicleta(),
    'Bicicleta 2': Bicicleta(),
    'Bicicleta 3': Bicicleta(),
    'Bicicleta 4': Bicicleta(),
    'Bicicleta 5': Bicicleta()
}

# escolhendo uma bicicleta e mostrando seus dados iniciais
bicicleta = 'Bicicleta 1'
print(bicicletas[bicicleta].mostrar_dados())

# alterando os dados da bicicleta
bicicletas[bicicleta].alterar_marcha(3)
bicicletas[bicicleta].alterar_velocidade(15)
bicicletas[bicicleta].alterar_cadencia(-100)
print(bicicletas[bicicleta].mostrar_dados())

# mudando novamente a cadência para um valor negativo
bicicletas[bicicleta].__cadencia = -100
print(bicicletas[bicicleta].mostrar_dados())
```

Vamos explicar cada um dos conceitos. Antes disso, observe como ficaram os resultados – veja que, apesar de tentarmos “forçar” a cadência para um número negativo (-100), o algoritmo não permite:

```
Marcha: 0, velocidade: 0, cadência: 0.
Marcha: 3, velocidade: 15, cadência: 0.
Marcha: 3, velocidade: 15, cadência: 0.
```

2.1 Objetos

O principal conceito da POO são, como o próprio nome já diz, os objetos. Para falarmos dos objetos e de POO, deixemos um pouco de lado todos os conceitos aplicáveis ao mundo do desenvolvimento de algoritmos e pensemos no mundo real. Olhe ao seu redor. O que você vê agora? Provavelmente deve estar vendo uma tela, um teclado, um celular, uma janela, uma porta, quem sabe até mesmo um copo de água (a propósito, você já bebeu água recentemente?).

Agora, perceba que “uma tela” ou “uma porta” ou “um celular” são únicos em si: alguém não teria “um celular do modelo Apple iPhone 12” em mãos, mas sim “um celular do modelo Apple iPhone 12 com o número de série A1234567890”. A porta é única em sua estrutura: ainda que várias outras portas sejam bem parecidas, a porta mais próxima de você é única, e só ela existe exatamente do jeito que ela é (talvez com alguns riscos ou com a pintura um pouco descascada).

Observe que todas as coisas do mundo real são únicas em si: até mesmo se nós estivermos segurando uma folha de papel em branco cada, teremos em mãos *folhas diferentes*. Elas podem até ser bem parecidas, mas perceba que a sua folha de papel **não é** a minha folha de papel.

Ainda que tenhamos uma bicicleta azul cada um, fabricadas pela mesma fábrica e pertencentes ao mesmo modelo, teremos também bicicletas totalmente diferentes: a nossa bicicleta pertence a nós e, a sua, a você. Pode ser que você cuide melhor da sua bicicleta do que nós ou, independentemente do caso, ainda que estejam juntas uma da outra, elas **nunca ocuparão o mesmo lugar ao mesmo tempo** – lembra quando mencionam na física que dois objetos jamais ocupam o mesmo lugar? Pois bem.

Os objetos da vida real possuem **estados** e **comportamento**. Estados são aquelas coisas que **definem** um objeto. Para um carro, pode ser placa, número de chassi, cor, ano e modelo. Para um cachorro, pode ser cor, raça, idade e nome. Logo, entendemos estados como sendo as suas **características**. Já os comportamentos são **ações** que um objeto pode fazer. Um carro pode virar

o volante, pode estar em movimento ou parado, pode estar ligado ou desligado. Um cachorro pode correr, abanar o seu rabo ou latir, por exemplo.

Em POO, mudamos um pouco os nomes:

- **estados** (características) – são armazenados em variáveis e constantes (que agora serão chamadas de **atributos**); e
- **comportamentos** (ações) – são geridos por funções (que agora serão chamadas de **métodos**).

2.2 Classes

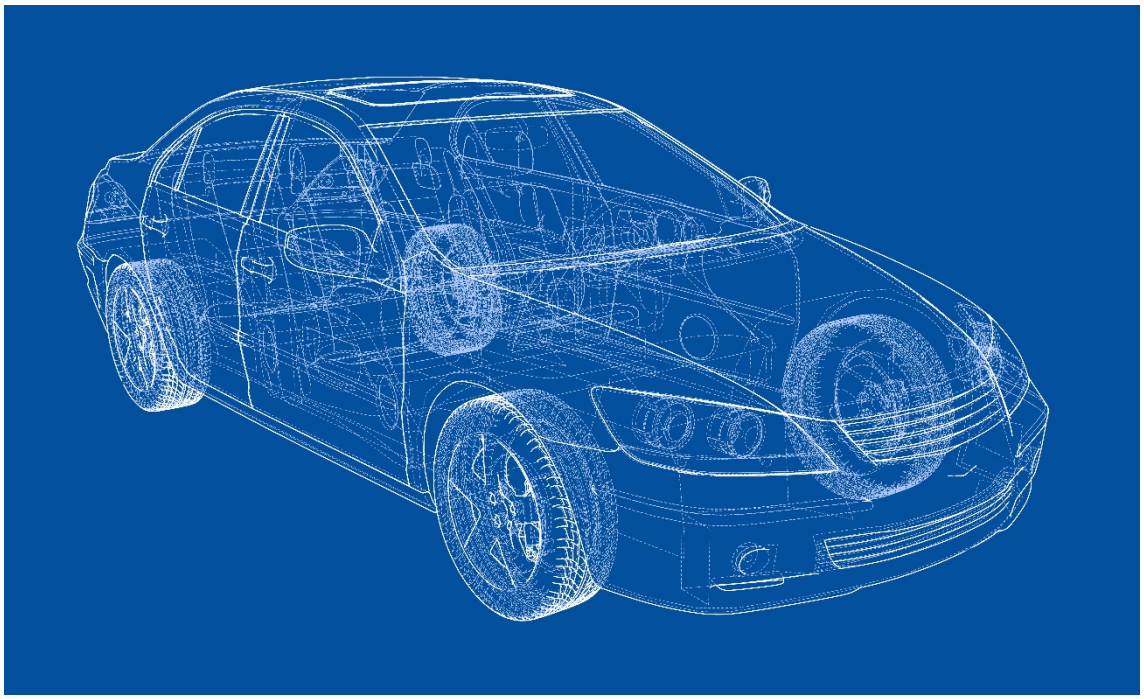
Como você deve imaginar, na realidade os objetos não possuem os mesmos estados e comportamentos ao mesmo tempo. Neste exato momento, todos os cachorros do mundo **não** estão latindo, nem todos os carros estão em movimento. Logo, cada carro e cada cachorro terá os seus estados e comportamentos de forma independente uns dos outros **ainda que possam compartilhar** os tipos de estados que são possíveis: todos os carros do mundo possuem alguma cor e foram fabricados em algum ano. Todos os cachorros do mundo também possuem alguma cor e alguma raça (ainda que a raça seja “sem raça definida”). É como se todos os objetos de um mesmo “tipo” seguissem sempre um molde mínimo contendo todos os estados e comportamentos esperados.

Figura 1 – Interface



Crédito: premium design/Shutterstock.

Figura 2 – Classe



Crédito: cherezoff/Shutterstock.

Figura 3 – Objetos da classe



Crédito: Paisan579/Shutterstock.

O nome que damos a este molde é ***classe***: uma classe chamada *Carro* terá um conjunto predefinido de atributos e métodos, e isso também vale para

uma classe chamada *Cachorro*. Imagine uma classe como um manual de como gerar um novo objeto ou um cardápio – a classe *Carro* não é um carro em si, mas é tão somente uma lista de definições (como um manual de instruções, mesmo) contendo tudo o que um novo objeto (um novo "Carro") deveria ter.

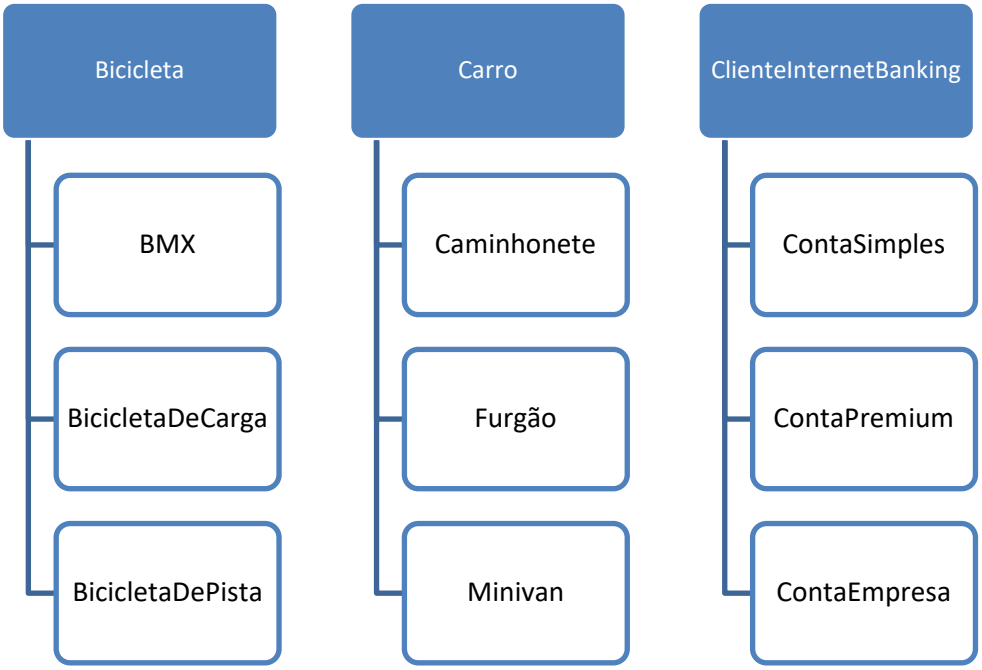
Ao criarmos um objeto de uma classe, estamos **instanciando-a**, ou seja, estamos criando um objeto seguindo o "manual" preestabelecido. Se a classe chamada *Carro* possui cinco atributos (placa, número de chassi, ano, cor e modelo) e três métodos (virar o volante, movimentar/parar, ligar/desligar), sabemos que todos os carros que instanciarmos deverão sempre seguir estes comportamentos: todos os carros virarão um volante seguindo o mesmo comportamento (para a direita e para a esquerda, e não como para cima e para baixo); vão se movimentar do mesmo jeito (ou seja, para frente ou para trás, mas não começarão a flutuar do nada); e terão um conjunto de atributos predeterminado. Isto nos dá muita segurança e previsibilidade ao trabalharmos com aplicações mais complexas: já pensou o que aconteceria se um método chamado *mostrar saldo* funcionasse de formas completamente diferentes a cada dia para um mesmo conjunto de pessoas? Ou já pensou o que aconteceria se cada "Carro" não tivesse nenhum **padrão** de direção?

2.3 Heranças

Heranças são outro conceito bem importante em POO. Ao ler as descrições anteriores, você deve ter pensado algo como: "beleza, mas nem todos os carros são iguais – existem carros que são à combustão que funcionam de forma diferente de carros elétricos. Já vi carros que nem possuem volante para virar! E aí?". Ou, ainda: "existem cachorros que fazem mais coisas do que abanar o rabo e latir. Existem cachorros que gostam de correr, e outros que gostam de caçar". Ou, voltando ao nosso exemplo das bicicletas: "Existem bicicletas que são especializadas para cada caso. Existem bicicletas para correr em velódromos, e outras, para transportar cargas. Também existem bicicletas para manobras, como a BMX. Nem todas são iguais!". Ou, no caso do *internet banking*: "nem todos os clientes são iguais! Alguns são pessoa física, e outros, jurídica. Existem também clientes com mais privilégios do que outros". E todos esses argumentos são justificáveis.

É para isso que serve a **herança**: podemos criar classes com base em outras classes, mas que respeitam as especificidades do seu caso de uso. No caso do carro.

Figura 4 – Exemplo de classes



Veja que uma classe herdada possui esse nome porque ela **herda** os métodos e atributos da classe principal: BMX herdou tudo de bicicleta, mas podemos modificar algumas coisas (as BMX possuem marchas diferentes) e podemos incluir novos métodos (como a execução de manobras).

2.4 Interfaces

Além disso, também podemos implementar **interfaces**: existem casos nos quais queremos somente definir quais métodos uma classe deve ter, mas não queremos implementar a **lógica** dentro dos métodos. No caso das bicicletas ou dos clientes de banco, pode ser que cada implementação de um método seja completamente diferente da de outro, mas, mesmo assim, precisamos garantir que todas as classes utilizem um conjunto predefinido de métodos: o nome disso é *interface*.

TEMA 3 – CLASSES E OBJETOS EM PYTHON

Da mesma forma que a biblioteca-padrão do Python oferece listas, dicionários de dados e funções, ela também possibilita a criação de algoritmos utilizando POO[1].

Vamos começar com o mesmo caso das bicicletas que vimos anteriormente, mas agora seguindo um passo a passo:

```
class Bicicleta:
    def alterar_velocidade(self, nova_velocidade):
        print(f'Velocidade recebida: {nova_velocidade}')
```

Veja que a classe foi **definida** ao usarmos a palavra-chave *class* seguida pelo nome (no caso, *Bicicleta*). Isto é bem parecido com a forma como criamos funções em Python, em que usamos o *def* seguido pelo nome da função. Após o nome da classe, é **opcional** usarmos parênteses, ao contrário das funções.

Dentro da classe, podemos criar uma função. Neste primeiro momento, só queremos testar para saber se estamos fazendo tudo certo. Para tal, só mostramos em tela a velocidade que recebemos no método *alterar_velocidade*. Este método também possui o parâmetro *self*. Isto significa que o método pode ser utilizado pelos objetos, e os seus resultados se aplicam somente ao escopo do objeto em si. Lembre-se: agora só temos a *classe*, mas nenhum *objeto* – ou seja, não temos nenhuma bicicleta instanciada no nosso código. Para instanciar, basta fazer isto:

```
bicicleta_professor = Bicicleta()
bicicleta_colega = Bicicleta()
```

Agora, sim, temos duas instâncias de bicicletas: uma chamada *bicicleta_professor*, e outra, *bicicleta_colega.* Espera-se que a lógica de ambos seja igual, mas um é independente do outro em sua essência. Veja como chamamos os métodos para cada um desses objetos e os seus resultados:

[1] Caso tenha interesse, sugerimos muito que leia a documentação oficial do Python sobre o assunto. Disponível em: <https://docs.python.org/3/tutorial/classes.html>. Acesso em: 30 nov. 2021.

```
print(f'bicicleta_professor (antes do 7): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(7)
print(f'bicicleta_professor (depois do 7): {bicicleta_professor.__dict__}')

print(f'bicicleta_colega (antes do 8): {bicicleta_colega.__dict__}')
bicicleta_colega.alterar_velocidade(8)
print(f'bicicleta_colega (depois do 8): {bicicleta_colega.__dict__}')

print(f'bicicleta_professor (antes do 10): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(10)
print(f'bicicleta_professor (depois do 10): {bicicleta_professor.__dict__}')
```

```
bicicleta_professor (antes do 7): {}
Velocidade recebida: 7
bicicleta_professor (depois do 7): {}

bicicleta_colega (antes do 8): {}
Velocidade recebida: 8
bicicleta_colega (depois do 8): {}

bicicleta_professor (antes do 10): {}
Velocidade recebida: 10
bicicleta_professor (depois do 10): {}
```

Os valores que informamos (7, 8 e 10) são as velocidades. Logo, aquele *self* não é considerado como o primeiro parâmetro, mas, sim, o parâmetro "*nova_velocidade*". Veja que usamos o ponto seguido pelo nome do método para chamar um método para um objeto em específico.

Isto posto, a forma como está agora não é das melhores: como saber se a velocidade realmente mudou para um objeto? Ao mostrarmos o antes e o depois de chamarmos cada um dos métodos, o resultado é aparentemente a mesma coisa: um dicionário vazio de atributos (o *__dict__* é uma funcionalidade-padrão do Python que mostra o que existe dentro de uma classe). Logo, a pergunta então é outra: como **armazenar** esses valores dentro de um objeto? Vamos inserir um novo parâmetro chamado *velocidade* fora do método, mas dentro da classe.

```
class Bicicleta:
    velocidade = 0

    def alterar_velocidade(self, nova_velocidade):
        print(f'Velocidade recebida: {nova_velocidade}')
        self.velocidade = nova_velocidade

bicicleta_professor = Bicicleta()
bicicleta_colega = Bicicleta()

print(f'bicicleta_professor (antes do 7): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(7)
print(f'bicicleta_professor (depois do 7): {bicicleta_professor.__dict__}')

print(f'bicicleta_colega (antes do 8): {bicicleta_colega.__dict__}')
bicicleta_colega.alterar_velocidade(8)
print(f'bicicleta_colega (depois do 8): {bicicleta_colega.__dict__}')

print(f'bicicleta_professor (antes do 10): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(10)
print(f'bicicleta_professor (depois do 10): {bicicleta_professor.__dict__}')
```

```
bicicleta_professor (antes do 7): {}
Velocidade recebida: 7
bicicleta_professor (depois do 7): {'velocidade': 7}

bicicleta_colega (antes do 8): {}
Velocidade recebida: 8
bicicleta_colega (depois do 8): {'velocidade': 8}

bicicleta_professor (antes do 10): {'velocidade': 7}
Velocidade recebida: 10
bicicleta_professor (depois do 10): {'velocidade': 10}
```

Vejamos isso! Agora, a variável *velocidade* está sendo armazenada dentro de cada objeto, e de forma individual. A velocidade da *bicicleta_professor* não interfere na velocidade da *bicicleta_colega*. Agora, vamos aumentar um pequeno grau de complexidade: vamos modificar o método *alterar_velocidade* para não permitirmos velocidades negativas e tentar passar uma velocidade negativa de duas formas para a *bicicleta_professor*: a primeira, via método; a segunda, forçando diretamente a variável.

```
class Bicicleta:
    velocidade = 0

    def alterar_velocidade(self, nova_velocidade):
        print(f'Velocidade recebida: {nova_velocidade}')
        self.velocidade = nova_velocidade if nova_velocidade >= 0 else 0

bicicleta_professor = Bicicleta()

print(f'bicicleta_professor (antes do 7): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(7)
print(f'bicicleta_professor (depois do 7): {bicicleta_professor.__dict__}')

print(f'bicicleta_colega (antes do -5): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(-5)
print(f'bicicleta_colega (depois do -5): {bicicleta_professor.__dict__}')

print(f'bicicleta_colega (antes do -9): {bicicleta_professor.__dict__}')
bicicleta_professor.velocidade = -9
print(f'bicicleta_colega (depois do -9): {bicicleta_professor.__dict__}')
```

```
bicicleta_professor (antes do 7): {}
Velocidade recebida: 7
bicicleta_professor (depois do 7): {'velocidade': 7}

bicicleta_colega (antes do -5): {'velocidade': 7}
Velocidade recebida: -5
bicicleta_colega (depois do -5): {'velocidade': 0}

bicicleta_colega (antes do -9): {'velocidade': 0}
bicicleta_colega (depois do -9): {'velocidade': -9}
```

Veja que a classe permitiu que forçássemos o –9 sem maiores problemas. E agora, como endereçaríamos isso? É para isso que existem os **atributos privados**: atributos que somente deveriam ser acessados e modificados de dentro da classe e por seus respectivos métodos. Em Python, os atributos privados possuem dois *underscores* (“_”) em seu início. Observe:

```
class Bicicleta:
    __velocidade = 0

    def alterar_velocidade(self, nova_velocidade):
        print(f'Velocidade recebida: {nova_velocidade}')
        self.__velocidade = nova_velocidade if nova_velocidade >= 0 else 0

bicicleta_professor = Bicicleta()

print(f'bicicleta_professor (antes do 7): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(7)
print(f'bicicleta_professor (depois do 7): {bicicleta_professor.__dict__}')

print(f'bicicleta_colega (antes do -5): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(-5)
print(f'bicicleta_colega (depois do -5): {bicicleta_professor.__dict__}')

print(f'bicicleta_colega (antes do -9): {bicicleta_professor.__dict__}')
bicicleta_professor.velocidade = -9
print(f'bicicleta_colega (depois do -9): {bicicleta_professor.__dict__}')
```

```
bicicleta_professor (antes do 7): {}
Velocidade recebida: 7
bicicleta_professor (depois do 7): {'_Bicicleta__velocidade': 7}
bicicleta_colega (antes do -5): {'_Bicicleta__velocidade': 7}
Velocidade recebida: -5
bicicleta_colega (depois do -5): {'_Bicicleta__velocidade': 0}
bicicleta_colega (antes do -9): {'_Bicicleta__velocidade': 0}
bicicleta_colega (depois do -9): {'_Bicicleta__velocidade': 0, 'velocidade': -9}
```

Note que ainda conseguimos visualizar a variável (agora chamada de *_Bicicleta__velocidade*), mas o risco de mudarmos manualmente é menor (quando tentamos mudar para –9, acabamos não afetando a *_Bicicleta__velocidade*, mas, sim, só criando uma variável chamada *velocidade* que o objeto acaba não utilizando). A própria documentação do Python[2] diz que é possível modificarmos os valores de variáveis privadas, mas que, ao seguirmos essa convenção, diminuímos o risco de modificá-las acidentalmente.

E se quiséssemos instanciar o *bicicleta_professor* com um valor inicial para a velocidade? Há uma função-padrão em Python chamada *__init__* na qual podemos informar valores que devem ser usados logo ao inicializar a classe. Lembra do *random_state* que usamos no *scikit-learn* anteriormente? Este é um exemplo de parâmetro que é usado na inicialização da classe daquela biblioteca.

[2] Disponível em: <https://docs.python.org/3/tutorial/classes.html#private-variables>. Acesso em: 30 nov. 2021.

No nosso caso, vamos utilizar o *__init__* para definir uma velocidade inicial da bicicleta. Se nenhuma velocidade for informada, usaremos o zero como padrão.

```
class Bicicleta:
    __velocidade = 0

    def __init__(self, velocidade_inicial=0):
        self.__velocidade = velocidade_inicial

    def alterar_velocidade(self, nova_velocidade):
        print(f'Velocidade recebida: {nova_velocidade}')
        self.__velocidade = nova_velocidade if nova_velocidade >= 0 else 0

bicicleta_professor = Bicicleta(5)

print(f'bicicleta_professor (antes do 7): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(7)
print(f'bicicleta_professor (depois do 7): {bicicleta_professor.__dict__}')
```

```
bicicleta_professor (antes do 7): {'_Bicicleta__velocidade': 5}
Velocidade recebida: 7
bicicleta_professor (depois do 7): {'_Bicicleta__velocidade': 7}
```

Seguindo na mesma linha, o que precisaríamos fazer se quiséssemos usar a herança para criar uma classe chamada BMX, a qual usaria a classe Biblioteca como base? Ao analisarmos o código a seguir, perceba algumas coisas.

- Na definição da classe BMX, estamos colocando entre parênteses a classe Bicicleta. Isso significa que BMX herda de Bicicleta.
- O __init__ é próprio da classe BMX, mas ela também logo em seguida chama o método __init__ próprio da classe Bicicleta por meio do *super()*, um termo-padrão do Python para chamar a classe herdada.
- Veja que em momento algum estamos implementando o método *alterar_velocidade*. Logo, estamos reaproveitando exatamente a mesma lógica da *alterar_velocidade*. Se quiséssemos, também poderíamos reimplementar este mesmo método com uma lógica específica à classe BMX.

- Também estamos criando um método específico para a classe BMX chamado *executar_manobra* – este método é próprio da BMX, mas não da Bicicleta.

```
class BMX(Bicicleta):
    __manobra = 0

    def __init__(self, manobras_iniciais=0, velocidade_inicial=0):
        self.__manobra = manobras_iniciais
        super().__init__(velocidade_inicial)

    def executar_manobra(self):
        self.__manobra += 1

bicicleta_propria = BMX(velocidade_inicial=3)
print(f'bicicleta_propria (antes de chamar os métodos): {bicicleta_propria.__dict__}')
bicicleta_propria.alterar_velocidade(5)
bicicleta_propria.executar_manobra()
bicicleta_propria.executar_manobra()
print(f'bicicleta_propria (depois de chamar os métodos): {bicicleta_propria.__dict__}'
)
```

```
bicicleta_propria (antes de chamar os métodos): {'_BMX__manobra': 0,
'_Bicicleta__velocidade': 3}
Velocidade recebida: 5
bicicleta_propria (depois de chamar os métodos): {'_BMX__manobra': 2,
'_Bicicleta__velocidade': 5}
```

TEMA 4 – POLIMORFISMO

Outro conceito-chave em POO é o **polimorfismo**: a possibilidade de diferentes classes implementarem de formas diferentes um mesmo método e/ou de uma mesma função operar de formas diferentes com tipos de dados distintos.

Um exemplo básico de polimorfismo é a função a seguir: ela aceita dois ou três parâmetros e consegue gerar um resultado para ambas as combinações.

```python
def mostrar_nome(primeiro_nome, sobrenome, nome_meio=None):
    if nome_meio is None:
        print(f'{primeiro_nome} {sobrenome}')
    else:
        print(f'{primeiro_nome} {nome_meio} {sobrenome}')

mostrar_nome('Jéssica', 'Silva')
mostrar_nome('Rodrigo', 'dos Santos', 'André')
```

```
Jéssica Silva
Rodrigo André dos Santos
```

Para classes, algo similar também pode ocorrer – comentamos anteriormente que a classe BMX poderia reimplementar a função *alterar_velocidade*: este é um exemplo de polimorfismo também. Ambas as classes podem ter o mesmo nome da função, mas as lógicas podem ser completamente diferentes uma da outra.

```python
class Bicicleta:
    __velocidade = 0

    def __init__(self, velocidade_inicial=0):
        self.__velocidade = velocidade_inicial

    def alterar_velocidade(self, nova_velocidade):
        print(f'Velocidade recebida: {nova_velocidade}')
        self.__velocidade = nova_velocidade if nova_velocidade >= 0 else 0

class BMX(Bicicleta):
    __manobra = 0
    __velocidade = 0

    def __init__(self, manobras_iniciais=0, velocidade_inicial=0):
        self.__manobra = manobras_iniciais
        self.__velocidade = velocidade_inicial

    def alterar_velocidade(self, nova_velocidade):
        print(f'Velocidade recebida pela BMX: {nova_velocidade}')
        self.__velocidade = nova_velocidade + self.__velocidade

    def executar_manobra(self):
        self.__manobra += 1

bicicleta_propria = BMX(velocidade_inicial=3)
print(f'bicicleta_propria (antes de chamar os métodos): {bicicleta_propria.__dict__}')
bicicleta_propria.alterar_velocidade(-5)
bicicleta_propria.executar_manobra()
bicicleta_propria.executar_manobra()
print(f'bicicleta_propria (depois de chamar os métodos): {bicicleta_propria.__dict__}')

bicicleta_professor = Bicicleta(5)
print(f'bicicleta_professor (antes do 7): {bicicleta_professor.__dict__}')
bicicleta_professor.alterar_velocidade(7)
print(f'bicicleta_professor (depois do 7): {bicicleta_professor.__dict__}')
```

```
bicicleta_propria (antes de chamar os métodos): {'_BMX__manobra': 0,
'_BMX__velocidade': 3}
Velocidade recebida pela BMX: -5
bicicleta_propria (depois de chamar os métodos): {'_BMX__manobra': 2,
'_BMX__velocidade': -2}
bicicleta_professor (antes do 7): {'_Bicicleta__velocidade': 5}
Velocidade recebida: 7
bicicleta_professor (depois do 7): {'_Bicicleta__velocidade': 7}
```

Os resultados (e a lógica) de ambas as classes são diferentes. A BMX permite velocidades negativas – a Bicicleta, não.

Em Python também é possível termos **funções encapsuladas**, uma função que fica dentro de outra função. Essas funções que ficam dentro de outras funções e que usam variáveis atribuídas a elas são chamadas de ***closures***. O uso delas possui algumas vantagens, por exemplo, a redução de casos em que precisamos usar variáveis globais e, além disso, a oportunidade de encapsular a lógica (e protegê-la) do resto do algoritmo. O código a seguir implementa uma *closure* para transformar a escala de um valor – no caso, estamos convertendo números que estejam entre 0.0 e 1.0 para 0% a 100% e arredondando em 3 casas decimais quando necessário:

```
def converte_porcentagem(proporcao):
    # inicio closure
    def multiplicador(valor):
        return round(proporcao * valor, 3)
    # fim closure

    return multiplicador
```

```
45.0
```

O ***decorator*** é outra funcionalidade bem interessante em Python. Você deve ter percebido que em vários dos exemplos anteriores usamos muito a função *print* para mostrar como estavam as coisas antes e depois de aplicar determinada função. Felizmente, há uma forma de simplificarmos esse trabalho. Os *decorators* são *closures* que podem ser incluídas no código sem grandes alterações deste com o uso do "@".

Vamos criar um *decorator* chamado "*atualizarstatus*", o qual deve conter as mensagens que desejamos mostrar.

```
def atualizarstatus(func):
    def wrapper(self):
        print(f'Antes de chamar o método: {self.__dict__}')
        func(self)
        print(f'Depois de chamar o método: {self.__dict__}')

    return wrapper

class BMX(Bicicleta):
    __manobra = 0

    def __init__(self, manobras_iniciais=0, velocidade_inicial=0):
        self.__manobra = manobras_iniciais
        super().__init__(velocidade_inicial)

    @atualizarstatus
    def executar_manobra(self):
        self.__manobra += 1

bicicleta_propria = BMX(velocidade_inicial=3)
bicicleta_propria.executar_manobra()
```

```
Antes de chamar o método: {'_BMX__manobra': 0, '_Bicicleta__velocidade': 3}
Depois de chamar o método: {'_BMX__manobra': 1, '_Bicicleta__velocidade': 3}
```

Veja que o *decorator* é uma função a qual encapsula outra função qualquer dentro dela – logo, da mesma forma como é utilizada para a *executar_manobra*, ela também pode ser reaproveitada para a *atualizar_velocidade* e quaisquer outras funções. O Python também possui alguns *decorators* já prontos: um deles é o @staticmethod. O @staticmethod serve para métodos estáticos (isto é, quando a lógica do método **não depende** do objeto, mas sim é algo específico da classe). Veja, por exemplo, o caso a seguir, em que calculamos a velocidade com base no número de manobras. O resultado desse método não possui relação com os objetos, mas é algo que possui relação com a **classe**.

```
def atualizarstatus(func):
    def wrapper(self):
        print(f'Antes de chamar o método: {self.__dict__}')
        func(self)
        print(f'Depois de chamar o método: {self.__dict__}')

    return wrapper

class BMX(Bicicleta):
    __manobra = 0

    def __init__(self, manobras_iniciais=0, velocidade_inicial=0):
        self.__manobra = manobras_iniciais
        super().__init__(velocidade_inicial)

    @atualizarstatus
    def executar_manobra(self):
        self.__manobra += 1

    @staticmethod
    def calcular_velocidade_manobra(manobra):
        return manobra/3

bicicleta_propria = BMX(velocidade_inicial=3)
bicicleta_propria.executar_manobra()
BMX.calcular_velocidade_manobra(5)
```

```
Antes de chamar o método: {'_BMX__manobra': 0, '_Bicicleta__velocidade': 3}
Depois de chamar o método: {'_BMX__manobra': 1, '_Bicicleta__velocidade': 3}
1.6666666666666667
```

FINALIZANDO

Esta aula teve como foco apresentar a você o tópico de programação orientada a objetos. Esse paradigma, como você deve ter percebido, possui vantagens e desvantagens. Como vantagens, poderíamos citar a sua facilidade em transcrever requisitos de negócio e conceitos da vida real em algoritmos funcionais (afinal de contas, é fácil entender o que é um “objeto” na realidade). Também é mais fácil de dar manutenção, pois é mais modular – logo, também é possível modificarmos pequenas partes do código com o risco menor de comprometer o todo. Além disso, pode ter vários trechos de código reaproveitados e tem ainda como vantagens a abstração da sua lógica e a segurança.

Por outro lado, também traz uma camada de complexidade bem maior, que se traduz em requisitos computacionais mais pesados, uma curva de aprendizagem bem maior e mais linhas de código. É por esta razão que aqui procuramos tomar um cuidado em traduzir vários conceitos para você de uma forma palatável, uma vez que existe um risco bem grande em tão somente repassar conceitos e não necessariamente **compreendê-los**.