Classes e Objetos

Para compreender alguns conceitos técnicos de programação, vamos começar com um exemplo simples. Imagine uma fábrica de sapatos, na qual todos os calçados são produzidos a partir de um molde que define suas características fundamentais, como tamanho e forma. Na programação orientada a objetos, esse molde é a nossa classe, e os sapatos feitos com base nele são os objetos. Com esse exemplo, introduzimos um dos conceitos básicos da orientação a objetos: uma classe é um modelo para criar e definir o tipo de objetos.

Os objetos não precisam ser idênticos. No exemplo dos sapatos, aqueles feitos a partir do mesmo molde compartilham algumas características, como tamanho e forma, mas podem ter outras características exclusivas, como cor ou material de fabricação. É crucial entender que, na programação, dois sapatos idênticos são considerados objetos distintos.

O nome da classe deve seguir a convenção *CapWords*, estabelecida no tópico *Class Names* do *Python Enhancement Proposal 8* (PEP 8). Esta convenção estabelece que o nome de uma classe deve ser escrito em *camel case*, em que cada palavra começa com letra maiúscula, sem separação por espaços ou caracteres especiais.

Uma vez que a classe esteja definida, podemos utilizá-la para criar objetos. Um objeto é uma instância da classe. Para instanciar uma classe, basta escrever seu nome seguido de parênteses, da mesma forma que fazemos ao chamar uma função.

Atributos e métodos

Segundo Phillips (2018), uma instância é um objeto específico com um conjunto próprio de dados e comportamentos. Os dados representam as características individuais de um determinado objeto. A classe define um conjunto de características compartilhadas por todos os objetos criados a partir dessa classe. Os objetos criados podem ter valores diferentes para essas características, também chamadas de atributos. Os comportamentos são ações que podem ser realizadas em um objeto. Já os comportamentos executáveis em uma classe específica de objeto são denominados métodos.

De forma prática, um método é uma função definida dentro de uma classe, mas com algumas características especiais, como a possibilidade de acessar os outros componentes da classe.

Quase todos os métodos exigirão um parâmetro especial, que chamaremos de *self* por convenção (PHILLIPS, 2018). Esse parâmetro está relacionado aos conceitos de método de instância e método de classe. O importante é saber que esse primeiro parâmetro é obrigatório, pois é por meio dele que acessaremos os atributos e métodos da classe.

Classes abstratas e concretas

Até este ponto, aprendemos que as classes são modelos para a criação de objetos. No entanto, na programação orientada a objetos também é possível ter classes abstratas, que servem como modelos para a construção de outras classes, chamadas de classes concretas. Uma classe abstrata define os métodos e atributos que suas classes filhas terão, mas estas podem ter seus próprios atributos e métodos, diferentes dos definidos na classe pai.
Em *Python*, a criação de uma classe abstrata envolve o uso do módulo *abc*, que fornece a infraestrutura necessária para que os mecanismos de herança e abstração funcionem conforme esperado. Na abstração, todos os métodos abstratos da classe abstrata precisam ser implementados nas classes concretas, caso contrário, ocorrerá um erro na instanciação de um objeto.
A *ABC* em *Python* é uma classe que pode conter métodos abstratos, ou seja, que não possuem implementação definida na classe abstrata, mas que devem ser implementados em suas subclasses concretas. Ela é utilizada para definir interfaces ou comportamentos comuns que devem ser seguidos por classes que a herdam, permitindo estabelecer um contrato entre a classe abstrata e suas subclasses. Isso garante que todas as subclasses implementem os métodos necessários.

Parâmetro *self*

De acordo com Phillips (2018), a diferença sintática entre os métodos (funções dentro de uma classe) e as funções em *Python* é que todos os métodos têm *self* como argumento obrigatório. Esse parâmetro nos métodos de uma classe é uma convenção que representa a própria instância da classe.
Quando um método é chamado em uma instância da classe, *Python* automaticamente passa a própria instância como o primeiro argumento para o método. Esse parâmetro permite que os métodos acessem e manipulem os atributos e métodos da instância específica em que foram chamados.
Por convenção, o primeiro parâmetro de qualquer método em uma classe *Python* deve ser *self*, mas esse nome pode ser substituído, embora não seja recomendado, já que *self* é amplamente reconhecido pela comunidade *Python* como o padrão para este papel (PEP 8, 2013).
Ao definir métodos em uma classe, o programador deve incluir *self* como o primeiro parâmetro, mesmo que o método não precise de nenhum argumento. Isso ocorre porque, quando o método é chamado, a instância em que isso acontece é automaticamente passada como o primeiro argumento, e o parâmetro *self* garante que o método possa acessar corretamente os atributos e métodos dessa instância.

Atributos de classe e de instância

Até aqui, vimos que atributos são dados pertencentes a um objeto. Agora vamos conhecer dois tipos diferentes de atributo: de classe e de instância.
Os atributos de classe são compartilhados por todas as instâncias e definidos fora de qualquer método dessa classe. Eles são acessados por meio do nome da classe, e, se um objeto modificar o valor de um atributo de classe, todas as outras instâncias terão o mesmo valor para esse atributo.

Já os atributos de instância são específicos de cada objeto e definidos dentro dos métodos da classe, geralmente no método construtor. Cada instância possui sua própria cópia desses atributos, e eles são acessados usando o nome do objeto, seguido por um ponto e o nome do atributo.

Construtor

Na nossa classe *Carteira*, temos a variável de classe *saldo* e criamos a variável de instância *self.saldo*, que permite armazenar o valor do saldo para cada objeto. É crucial entender essa diferença: a variável de classe *saldo* compartilhará o mesmo valor para diferentes objetos, enquanto a variável de instância *self.saldo* fará com que cada objeto tenha seu próprio valor.
Por meio de um método especial chamado método construtor, podemos criar e inicializar atributos de instância quando uma classe é construída (instanciada). Esse método deve ser obrigatoriamente chamado de "__init__".

Property e decorator

Ao definir nossa classe da forma apresentada, estamos suscetíveis a um problema que queremos evitar: a falta de controle sobre os valores atribuídos ao atributo, o que poderia resultar em valores sem qualquer verificação. Por exemplo, você poderia atribuir valores negativos a um atributo que não deveria receber valores menores que 0.
Para resolver isso, podemos usar a função integrada *property* para encapsular uma propriedade. Essa abordagem faz com que um método seja tratado como um atributo e requer dois parâmetros obrigatórios: devemos fornecer um método para obter o valor desejado do nosso atributo e outro para alterar esse valor.
Uma prática adotada pelos programadores *Python* é utilizar um *underscore* (_) como prefixo de um atributo ou método para indicar que ele é privado, ou seja, destinado apenas ao uso interno da classe.
Uma maneira ainda mais prática de utilizar o *property* é através da sintaxe de *decorator*, o que resulta em um código mais legível. A função *property* pode ser usada como um *decorator* para transformar um método em uma propriedade e, para que ela funcione adequadamente, é preciso criar um método *setter* que faça a verificação do valor.

Tipos de métodos

1. Método de classe: é definido com o decorador @*classmethod*. Esse tipo de método opera na classe em si, não em instâncias específicas. O primeiro parâmetro de um método de classe é a própria classe, por convenção denominada *cls*. É comumente usado para operações que afetam a classe como um todo, como métodos de fábrica ou de inicialização alternativos.

2. Método de instância: é o tipo mais comum de método em *Python*. Ele opera em instâncias específicas da classe e é chamado com uma instância como primeiro argumento, por convenção denominada *self*. Métodos de instância são capazes de acessar e modificar os atributos específicos de uma instância.

3. Método estático: é definido com o decorador *@staticmethod*. Esse tipo de método não recebe uma referência explícita à instância ou à classe. Métodos estáticos são independentes de instâncias e classes, geralmente utilizados quando a lógica de um método não depende do estado interno da classe ou da instância.

4. Método de classe não estático: é um método regular que não possui nenhum decorador especial. Esse tipo de método opera tanto na classe quanto nas instâncias. Pode acessar a classe usando *self.__class__ ou cls* (se for um método de classe) e os atributos de instância usando *self*.
Cada tipo de método tem seu uso específico, e é importante entender suas diferenças para uma correta aplicação na construção de classes em *Python*.

Herança

Todo trabalho que realizamos deve ser eficiente e prático, e isso também se aplica à programação. Quando falamos sobre eficiência e reutilização de código na programação, estamos nos referindo à importância de evitar repetições desnecessárias. Um acrônimo muito conhecido na área de tecnologia da informação é o *Don't Repeat Yourself* (DRY), que em português significa "não se repita"
Uma maneira de evitar a repetição do mesmo código em diferentes partes de um programa é usando a herança, um conceito fundamental na programação orientada a objetos. Através da herança, uma classe pode herdar atributos e métodos de outra classe.
Nesse conceito, uma classe pai é chamada de superclasse, e a classe filha é denominada subclasse. Também podemos dizer que uma subclasse é uma classe derivada ou que estende a classe pai

Classe base

Considerando o que foi analisado até agora, podemos identificar uma relação entre classe base e classe abstrata. Conforme Lott (2019, p. 229), as classes base são aquelas que outras classes usarão como superclasse. Por outro lado, uma classe abstrata fornece definições para métodos, e as classes base abstratas geralmente fornecem as assinaturas para os métodos ausentes. Uma subclasse deve fornecer os métodos corretos para criar uma classe concreta que se ajuste à interface definida pela classe abstrata.

Polimorfismo

Polimorfismo é o pilar da orientação a objetos que possibilita a reutilização de códigos. "É a capacidade da linguagem de programação de processar objetos de formas diferentes dependendo de seu tipo de dado ou classe." (DATHAN; RAMNATH, 2015, p. 27). Uma das principais vantagens é que, a partir do momento em que você tem uma classe herdeira, é possível ter métodos iguais (com o mesmo nome), mas que serão implementados de formas diferentes.

Sobrecarga e sobrescrita

Dado o conceito de polimorfismo, podemos avançar analisando o que é sobrecarga e sobrescrita de métodos. Segundo Phillips (2018), sobrecarga significa ter, em uma classe, métodos com o mesmo nome que aceitem diferentes conjuntos de argumentos. Em linguagens de tipagem estática, isso pode ser útil; porém, em *Python* só precisamos de um método que aceite qualquer tipo de objeto. Além disso, não podemos escrever dois métodos com o mesmo nome dentro de uma mesma classe em *Python*. Para criar o efeito de sobrecarga, devemos adotar o uso de parâmetros obrigatórios e opcionais. Se quisermos tornar um argumento opcional dentro de um método, em vez de criar um segundo método com argumentos diferentes, podemos especificar um valor-padrão em um único método, usando o sinal de igual (=).
Sobrescrita em *Python* permite que métodos em classes filhas reescrevam métodos definidos inicialmente na classe pai ou na classe base. Além de manter os parâmetros originais, a classe herdeira pode adicionar novas especificações ao método.

Composição

Além da herança, outro mecanismo da orientação a objetos que nos permite evitar a repetição de código e organizar o projeto de forma mais adequada é a composição. Mais simples que a herança, a composição ocorre quando classes têm uma relação de "ter um" entre elas.
Macoratti (2011) levanta uma discussão importante sobre quando utilizar herança e quando utilizar composição. Segundo ele, a composição estende uma classe, e o seu objeto é quem implementa suas funcionalidades. Por exemplo, considere duas classes, *Livro* e *Autor*, que possuem uma relação de composição. Um livro existe sem um autor? Se a resposta for não, utiliza-se a composição, pois a classe *Autor* faz parte da classe *Livro*.

Erros e tratamento de exceções

Como programador, é essencial saber como lidar com erros de código. O tratamento de exceções e erros em *Python* é uma parte fundamental da programação, pois permite que os desenvolvedores lidem com situações inesperadas durante a execução de um programa.
Em *Python*, os erros são tratados por meio de blocos *try-except*, onde o código que pode gerar um erro é colocado dentro do bloco *try*, e as ações a serem tomadas em caso de erro são especificadas no bloco *except*.
O bloco *try* contém um comando para imprimir algo na tela. Se um erro ocorrer durante a execução desse comando, o bloco *except* será acionado para tratar o erro e evitar que o programa trave.
No bloco *try* temos um código com um erro na chamada do método *int*, pois estamos passando um ‘a’, o que deve resultado um *except* por *VaueError*.
Além do bloco *try-except*, *Python* também oferece os blocos *else* e *finally*. O comando *else* pode ser usado em conjunto com a estrutura *try*...*except* para executar ações caso nenhum erro seja encontrado
Diferentemente dos seus irmãos, o bloco *finally* é sempre executado, independentemente de ocorrer ou não um erro. Isso é útil para realizar ações de

limpeza ou de fechamento de recursos, como arquivos ou conexões de banco de dados, garantindo que essas operações sejam realizadas mesmo em caso de erro.
Neste exemplo, tentamos abrir o "arquivo.txt" para leitura. Se o arquivo não for encontrado, uma exceção do tipo *FileNotFoundError* será gerada e capturada no bloco *except*, no qual uma mensagem de erro será exibida. Independentemente disso, o bloco *finally* garantirá que o arquivo seja fechado usando o método *close()*, mesmo se ocorrer uma exceção. Isso é útil para garantir que recursos como arquivos sejam sempre liberados adequadamente, mesmo em casos de erro.
Entretanto, quando se trata de objetos de arquivo, é uma boa prática utilizar a palavra-chave *with*. A vantagem é que o arquivo é fechado corretamente após a finalização de sua execução, mesmo se uma exceção for lançada em algum ponto. O uso de *with* também é muito mais conciso do que escrever blocos *try-finally* equivalentes
Se você não estiver usando a palavra-chave *with*, então deve chamar *f.close()* para fechar o arquivo e liberar imediatamente quaisquer recursos do sistema usados por ele. Inclusive, caso esteja usando *with* e fizer uma chamada ao *f.close()*, terá um retorno *true*, o que confirma o fato de o arquivo ser fechado automaticamente.

Conda

O *Conda* é uma ferramenta de código aberto que permite gerenciar ambientes virtuais e pacotes em várias linguagens, incluindo *Python*. Ele simplifica a instalação, atualização e remoção de pacotes, além de lidar com as dependências entre eles. Com o *Conda*, você pode criar ambientes isolados para diferentes projetos, garantindo que cada um tenha suas próprias versões de pacotes sem conflitos.
Por exemplo, suponha que você esteja trabalhando em dois projetos diferentes, ambos exigindo a biblioteca *pandas*, mas um requer a versão 1.0 e o outro a versão 0.25. Com o *Conda*, você pode criar dois ambientes separados, cada um com a versão correta do *pandas* instalada, evitando conflitos entre os projetos.
Além disso, o *Conda* também permite instalar pacotes não relacionados ao *Python*, como bibliotecas em *C/C++* e *R*. Isso o torna uma ferramenta poderosa para gerenciar todas as dependências de um projeto, independentemente da linguagem de programação utilizada.

Conda

O *Conda* é uma ferramenta de código aberto que permite gerenciar ambientes virtuais e pacotes em várias linguagens, incluindo *Python*. Ele simplifica a instalação, atualização e remoção de pacotes, além de lidar com as dependências entre eles. Com o *Conda*, você pode criar ambientes isolados para diferentes projetos, garantindo que cada um tenha suas próprias versões de pacotes sem conflitos.
Por exemplo, suponha que você esteja trabalhando em dois projetos diferentes, ambos exigindo a biblioteca *pandas*, mas um requer a versão 1.0 e o outro a versão 0.25. Com o *Conda*, você pode criar dois ambientes separados, cada um com a versão correta do *pandas* instalada, evitando conflitos entre os projetos.
Além disso, o *Conda* também permite instalar pacotes não relacionados ao *Python*, como bibliotecas em *C/C++* e *R*. Isso o torna uma ferramenta poderosa para

gerenciar todas as dependências de um projeto, independentemente da linguagem de programação utilizada.
Em resumo, o *Jupyter Notebook* é uma ferramenta poderosa para análise de dados, experimentação, documentação e compartilhamento de resultados, tornando-o uma escolha popular entre cientistas de dados e pesquisadores.