Classes e noções de Orientação a Objetos

Sobre paradigmas de programação

Um paradigma, por definição, é um conceito que define um exemplo típico ou modelo de algo. É a representação de um padrão a ser seguido.

Um **paradigma de programação** é uma forma de classificação de linguagens de programação, baseada nas suas funcionalidades. Um paradigma é, portanto, um tipo de estruturação ao qual a linguagem deverá respeitar.

Cada paradigma surgiu de necessidades diferentes. Dado isso, cada um apresenta maiores vantagens sobre os outros dentro do desenvolvimento de determinado sistema. Sendo assim, um paradigma pode oferecer técnicas apropriadas para uma aplicação específica.

Não é o objetivo da nossa aula o detalhamento dos paradigmas de programação. No entanto, vamos falar com um pouco mais sobre dois: o procedural (ou imperativo) e o orientado a objetos.

Paradigma procedural ou imperativo

Nesse paradigma, as instruções devem ser passadas ao computador na sequência em que devem ser executadas. Ou seja, o computador executa, através do código compilado ou através de um interpretador, uma linha por vez, na ordem em que elas foram escritas.

Como linguagens mais conhecidas temos COBOL, FORTRAN e Pascal. O código programado através desse paradigma é uma espécie de passo-a-passo dos procedimentos que a máquina deverá executar.

Nesse paradigma, a solução do problema será muito dependente da experiência e da criatividade de quem trabalha com a programação. O foco das linguagens que usam esse paradigma é mais no "como" as soluções devem ser implementadas.

O paradigma procedural é recomendado em projetos onde não se espera que haja mudanças significativas com o tempo. Ou seja, aplicações de *back-end* em sistemas financeiros, scripts para build de projetos maiores, etc.

Os programas desenvolvidos utilizando este paradigma tendem a ser muito eficientes - o computador não precisa analisar dezenas (ou centenas, ou milhares) de arquivos e pontos de entrada diferentes para decidir por onde seguir. O paradigma também possibilita uma modelagem tal qual o mundo real, porém o código tende a ser de difícil legibilidade, o que dificulta manutenção e evolução da aplicação.

Paradigma orientado a objetos

Este paradigma foi popularizado na década de 1990 com a linguagem Java (1995), porém ele existe há muito mais tempo, em linguagens como Smalltalk (1969) ou C++ (1979).

Apesar de existirem linguagens que adotassem esse paradigma há muito tempo, apenas com a revolução dos computadores pessoais (com a chegada do Windows) e da expectativa dos primeiros dispositivos inteligentes na década de 1990 (televisores, carros, etc.) é que ele se tornou popular. Isto porque o paradigma orientado a objetos (ou simplesmente POO) é ideal para uma programação multiplataforma.

Quando implementamos uma aplicação usando OO podemos aplicar diferentes camadas de abstração, que permitem a transposição rápida da solução de uma plataforma para outra.

O POO apoia-se nas abstrações de classes e objetos ao tentar retratar a programação tal qual se enxerga o mundo real. Todos os objetos têm determinados estados e comportamentos. Esses estados são descritos pelas classes como atributos, e a forma como os objetos se comportam é definida por meio de métodos, que são equivalentes às funções do paradigma funcional.

Qual o paradigma do Python?

Apesar de termos várias linguagens com paradigmas muito bem definidos (FORTRAN é procedural, Java é orientado a objetos, VBA é orientado a eventos, etc.), o Python cai numa categoria que chamamos de **multiparadigma**.

Com esta linguagem, podemos programar utilizando vários paradigmas: procedural, orientado a objetos ou funcional. Dependendo da nossa abordagem, podemos utilizar um ou mais desses paradigmas para atingir o objetivo do nosso projeto.

Apesar de ser possível programar com orientação a objetos, o Python possui algumas restrições de design que não estão completamente de acordo com os princípios da OO (como encapsulamento), de forma que não é recomendado escolher Python para desenvolver um projeto 100% orientado a objetos.

O mais comum é desenvolver uma aplicação Python utilizando recursos de todos os paradigmas, aproveitando os pontos fortes e evitando os pontos fracos.

Introdução a Orientação a Objetos

Dentre os vários paradigmas de programação surgidos desde a década de 1950, o POO veio com a missão de cobrir as insuficiências existentes nos paradigmas anteriores, como o procedural.

O POO tem como principal característica uma melhor e maior expressividade das necessidades do nosso dia-a-dia. Ele possibilita criar unidades de código mais próximas da forma como

pensamos e agimos, facilitando o processo das necessidades diárias para uma linguagem orientada a objetos.

Dá-se o nome de Programação Orientada a Objeto ao processo de usar uma linguagem orientada a objeto. Percebe-se que a sigla POO termina se fundindo entre o paradigma e a programação. Entretanto, é válido ressaltar que, em alguns casos, somente utilizar uma linguagem orientada a objetos não garante que se esteja programando efetivamente orientado a objetos, devido a "vícios procedurais" dos novos programadores desse paradigma, ou mesmo de programadores experientes.

Nesta aula vamos ver alguns conceitos básicos do POO e como podemos aplicá-los em um

Um breve histórico da Orientação a Objetos

O conceito de simulação

O principal insumo para a OO da forma como ela é conhecida hoje foi o conceito de *simulação*, que no mundo da computação significa "simular os eventos do dia-a-dia em sistemas digitais". A simulação pode ser classificada em três tipos:

- *Discrete Events Simulation*: Usa modelos lógicos e matemáticos para retratar mudanças de estado através do tempo, assim como os relacionamentos que levaram a essas mudanças;
- *Continuous Simulation*: Usa equações matemáticas que não se preocupam em representar mudanças de estados e relacionamentos, mas apenas manipular dados brutos que serviram de insumos para outros processamentos;
- *Monte Carlo Simulation*: Usa modelos de incerteza, em que a representação de tempo não é necessária. Uma melhor definição é "um processo onde a solução é atingida através de tentativa e erro, e tal solução se aplicará somente ao problema específico".

Da Noruega para o mundo

Em 1962, no Centro Norueguês de Computação (NCC) da Universidade de Oslo havia dois pesquisadores, Kristen Nygaard e Ole-Johan Dahl, que aceitaram o projeto de criar uma linguagem de simulação de eventos discretos. Eles decidiram batizar sua linguagem de SIMULA I, tendo uma segunda versão chamada SIMULA 67. Essa última é reconhecida como a primeira linguagem de renome no que diz respeito ao universo da Orientação a Objetos.

Um dos conceitos bases na criação de SIMULA I e SIMULA 67 foi de que a nova linguagem deveria ser **orientada a problemas** e não **orientada a computadores**. Isso implicou em um aumento da expressividade e facilidade de uso da linguagem. SIMULA I foi inicialmente baseada em FORTRAN, o que infelizmente se configurou em uma má decisão de projeto.

FORTRAN possuía uma estrutura de bloco que não possibilitava a expressividade essencial para a abordagem desejada por eles. Foi a partir dessa deficiência que SIMULA 67, que se baseou em ALGO 60 por possuir uma estrutura de blocos e dados mais amigável, foi lançada.

A nova roupagem da Orientação a Objetos

Embora SIMULA 67 seja considerada a linguagem que originou a OO e, com isso, tenha modificado de forma radical a maneira como se desenvolvia software até aquele momento, ainda se tinha o seguinte problema: a computação ainda era "fechada". Embora as linguagens citadas anteriorments tenham impactado de forma considerável na evolução das linguagens de programação, elas não podiam ser usadas em qualquer computador. SIMULA 67, por exemplo, rodava apenas em uma UNIVAC 1107.

Com isso, embora as linguagens trouxessem mais facilidades no processo de desenvolvimento, ainda existia o problema da portabilidade. Essas linguagens eram específicas para os computadores que foram usados para desenvolvê-las.

A linguagem que, efetivamente, tornou a OO conhecida até os dias de hoje foi o *Smalltalk*. Ela foi desenvolvida na década de 1970 por Alan Kay, um pesquisador da Xerox Parc, que recebeu o projeto de criar uma linguagem que pudesse se usada nos emergentes PCs.

O Smalltalk trouxe facilidades como: interface gráfica amigável, um IDE, capacidade de ser executada em máquinas de pequeno porte, entre outras características.

Por que usar a Orientação a Objetos?

Segundo o paradigma procedural, é possível representar todo e qualquer processo do mundo real a partir da utilização de *apenas* três estruturas básicas:

- Sequência: Os passos devem ser executados um após o outro, linearmente. Ou seja, o programa seria uma sequência finita de passos. Em uma unidade de código, todos os passos devem ser feitos para se programar o algoritmo desejado;
- Decisão: Uma determinada sequência de código pode ou não ser executada. Para isto, um teste lógico deve ser realizado para determinar ou não sua execução. A partir disto, verifica-se que duas estruturas de decisão (também conhecida como seleção) podem ser usadas: a `if-else` e a `switch`.
- Iteração: É a execução repetitiva de um segmento (parte do programa). A partir da execução de um teste lógico, a repetição é realizada um número finito de vezes. Estruturas de repetição conhecidas são: `for`, `foreach`, `while`, `do-while`, `repeat-until`, entre outras (dependendo da linguagem de programação).

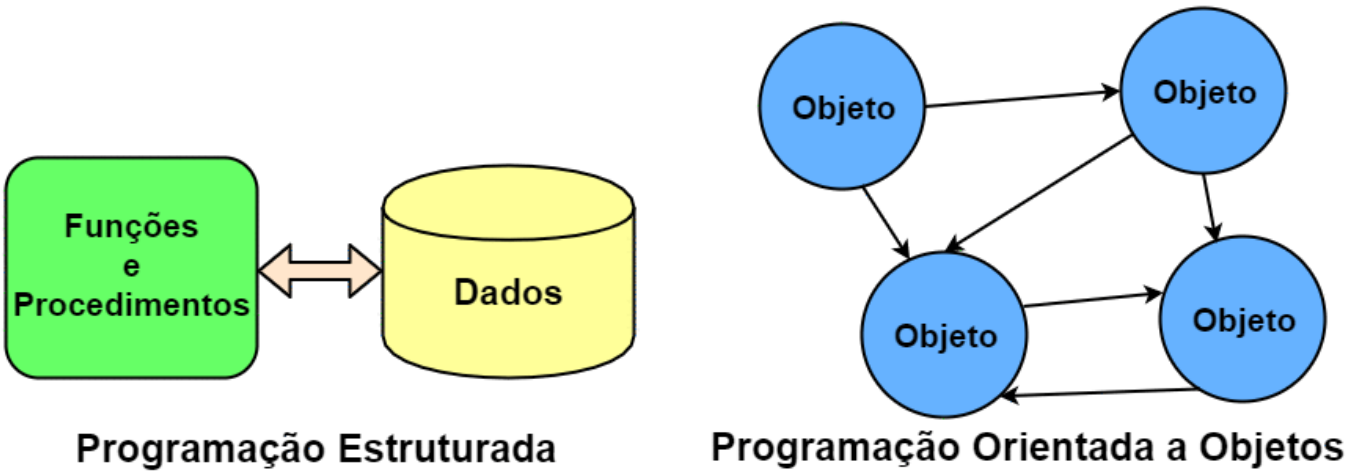
Inicialmente, pode-se pensar que estas três estruturas são o suficiente para trabalhar. Entretanto, ao começarmos a fazer uma avaliação mais minuciosa, é possível notar algumas limitações. Por exemplo, somos acostumados a usar linguagens procedurais para aprender a programar. Ou seja, criamos programas simples como cálculo de média, soma de números, um joguinho da velha, etc. Porém, quanto mais complexo o programa se torna, mais difícil fica a manutenção de uma sequência organizada de código.

E se a necessidade agora for um controle de estoque? Uma aplicação deste tipo manipulará conceitos como produto, venda, estoque, cliente, etc. Este terá operações como vender, comprar, atualizar estoque, cadastrar produto, cadastrar cliente, etc. Logo, nota-se que isso levará a um emaranhado de código, muitas vezes muito extenso e propício à duplicação.

Para tentar amenizar essa situação, podemos recorrer a modularizações que essas linguagens proveem. Entretanto, o código começará a ficar mais complexo.

Em resumo, a simplificação da representação das reais necessidades dos problemas a serem automatizados leva a uma facilidade de entendimento e representação. Porém, isso pode levar a uma complexidade de programação caso o nicho de negócio do sistema-alvo seja complexo.

Devido à sua fraca representatividade do mundo real, a programação procedural foca na representação dos dados e operações desassociadas. Isto é, dados e operações de diversos conceitos são misturados, não ficando claro qual operação realmente está ligada aos específicos dados. A figura abaixo ilustra essa situação e mostra que a OO tem o objetivo de colocar ordem na casa com a interação entre objetos, que tem seu escopo bem delimitado.

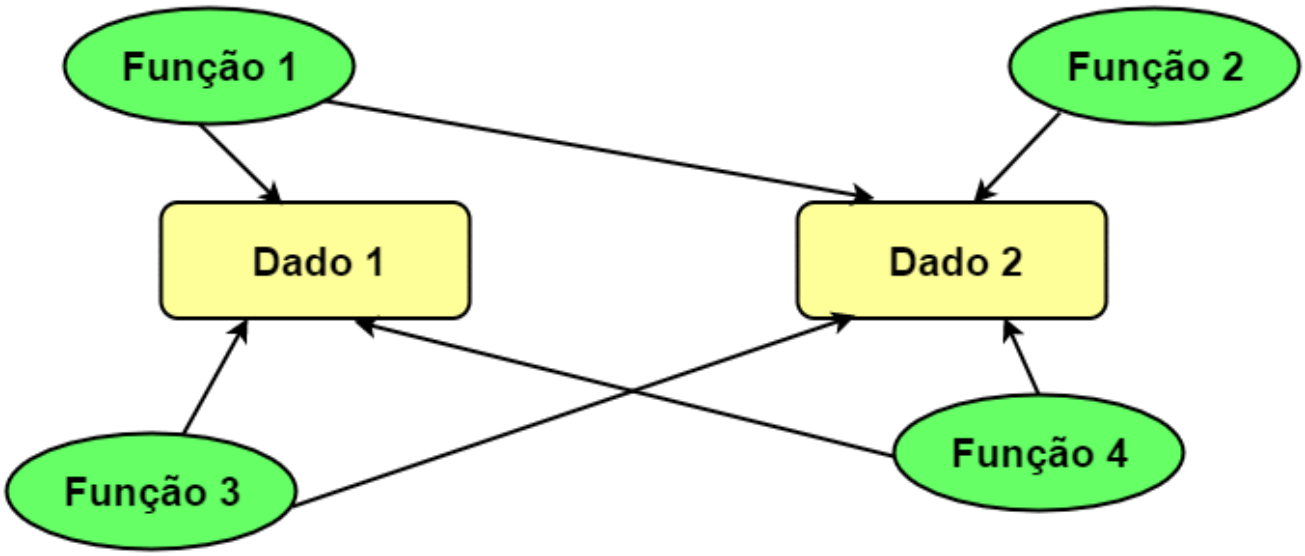


Explicando de forma mais clara a figura anterior, na programação procedural, devido ao fato de os dados não serem intimamente ligados às possíveis operações sobre estes, acabamos encontrando códigos similares ao apresentado a seguir:

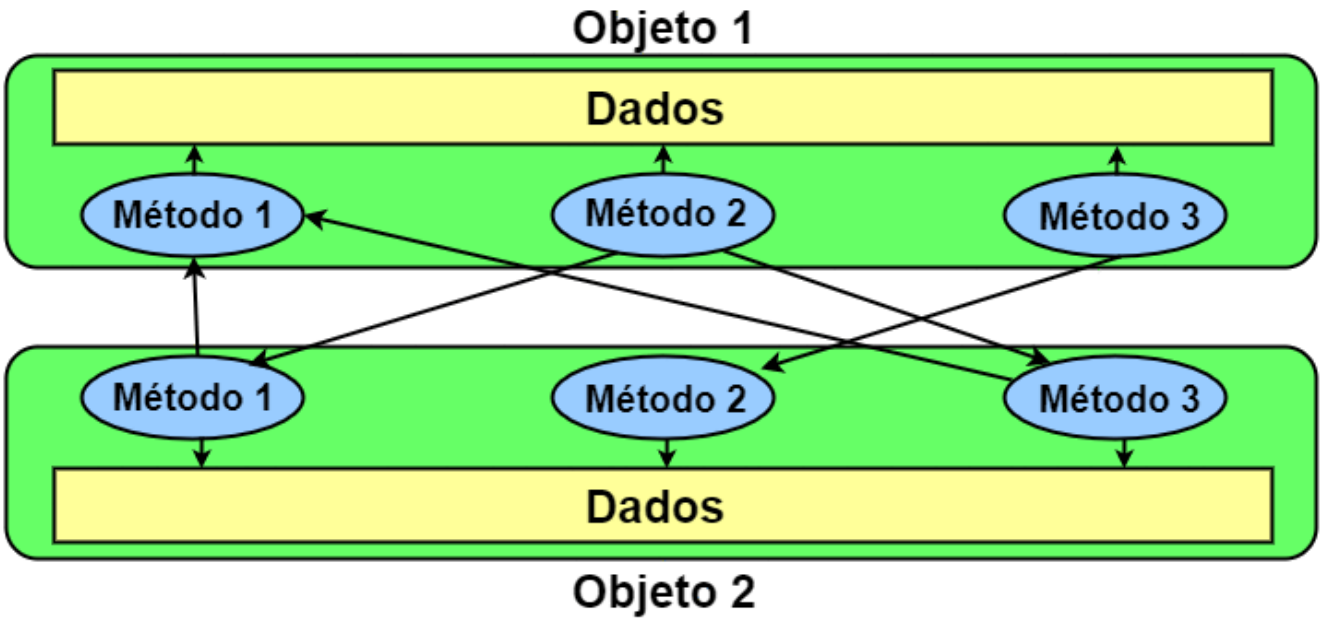
```
struct produto {
    char nome[150];
    double valor;
};
typedef struct produto Produto;

struct venda {
    Produto produtos[];
    double desconto;
};
typedef struct venda Venda;

void finalizarVenda() {
```

```
    ...
}

double calcularTotalVenda(Produto *produtos) {
    ...
}

void adicionarProduto(Venda venda, Produto produto) {
    ...
}
```

Nesse código, há uma mistura de dados diferentes que representam entidades diferentes, mas que estão definidos em uma mesma unidade de código. Isto acaba por levar também a uma mistura das operações que vão manipular tais dados. Assim, nota-se que a programação procedural tem como filosofia que funções afins manipulem diversas variáveis definidas de forma global - no caso, `structs`. Com isso, facilmente seria possível fazer, de forma errônea, uma função usar dados que não lhe dizem respeito.



Ao contrário disto, a OO preconiza que os dados relativos a uma representação de uma entidade do mundo real devem somente estar juntos de suas operações, quais são os responsáveis por manipular - exclusivamente - tais dados. Assim, há uma separação de dados e operações que não dizem respeito a uma mesma entidade. Todavia, se tais entidades necessitarem trocar informações, farão isto através da chamada de seus métodos, e não de acessos diretos a informações da outra. A figura a seguir ilustra tal modo de funcionamento.



Tendo em vista as diferentes formas de funcionamento desses paradigmas, para se fazer uma transição segura do procedural para o OO, é necessário saber que, devido a essa desassociação entre dados/funções na programação procedural, somente os dados são trafegados dentro da aplicação. Já na OO, os dados são transmitidos junto com suas operações, pois, ao contrário do outro paradigma, ambos - dados e operações, estão definidos em uma única e organizada unidade de código. Isso torna a manipulação de tais dados mais segura e simplificada.

A partir do que foi exposto, verifica-se que esta simplicidade culmina em algumas dificuldades, que podem onerar, tornar mais complexo, ser mais propenso à geração de erros no processo de desenvolvimento. A seguir, serão apresentadas quais são essas deficiências e, de forma introdutória, como a OO provê a solução para elas.

Reuso

Em linguagem de programação podemos reutilizar duas coisas:

- Comportamentos - no caso operações, serviços, ações;
- Informações - no caso dados, características.

Sem utilizar a OO, podemos atingir o reuso, porém isso tem um custo. Quanto mais complexo o sistema que queremos desenvolver, teremos cada vez mais redefinições, propícios a erros de esquecimento e pontos de falhas. Linguagens procedurais, como C, possuem mecanismos que permitem o reuso em certo grau, como o uso de `structs` ou de `headers`, porém a quantidade de estruturas necessárias para possibilitar a reutilização do código é extensa e muitas vezes trabalhosa.

Para suprir tais dificuldades, a OO disponibiliza dois mecanismos para reuso de código: a herança e a associação. A partir deles é possível criarmos unidades de código que compartilham códigos de forma procedural, ou seja, não são blocos de código dispersos. Eles criam um relacionamento que, além de possibilitar o reuso de forma mais prática e menos

propícia a erros, ainda gera uma modelagem mais próxima do mundo real. Quando for aprsentado o termo *gap semântico*, a ideia de "modelagem mais próxima do mundo real" ficará mais clara.

Coesão

Este princípio preconiza que cada unidade de código deve ser responsável somente por possuir informações e executar tarefas que dizem respeito somente ao conceito que ela pretende representar. A ideia por detrás da coesão é não misturar responsabilidades, para evitar que a unidade de código fique sobrecarregada com dados e tarefas que não lhe dizem respeito.

Em linguagens procedurais, como C, é necessário recorrermos a `headers` e módulos que, como já vimos, pode trazer complexidade excessiva ao código, gerando maior risco de falhas.

Para agilizar o processo de desenvolvimento, a OO disponibiliza conceitos que facilitam a vida do desenvolvedor: classe e associação. Criar unidades de código mais coesas com esses conceitos é mais simples do que trabalhar com `headers` e módulos. Concomitantemente a esses dois conceitos, o uso de métodos e atributos contribui para a definição de unidades de código que sejam responsáveis somente por tarefas e conceitos às quais elas se propõem, assim evitando uma "salada mista" de responsabilidades.

Acoplamento

Acoplamento é um termo usado para medir (quantificar) o relacionamento entre unidades de código que são unidas, acopladas, para que nossa aplicação consiga executar suas atividades da forma desejada. A princípio, podemos pensar que linguagens procedurais não possuem acoplamento, pois elas possuem somente uma unidade de código, o Módulo Principal. Todavia, o conceito de acoplamento é mais amplo.

Em uma linguagem procedural, o acoplamento existe entre o Módulo Principal com suas *funções* - ou mesmo entre funções, com `headers`, módulos e qualquer outra estrutura que possua seu próprio código. Em linguagens procedurais, o acoplamento pode se tornar um problema devido ao processo de compilação ou `linkagem` dessas linguagens. Em resumo, quanto mais baixo for o nível de estruturação do código, mais complexo torna-se o processo de se trabalhar com o acoplamento.

Não obstante, é necessário usar acoplamento para organizar o código e dividir responsabilidades com outras unidades de código. Ao citar "dividir responsabilidades", logo, nota-se que há um relacionamento muito íntimo entre acoplamento e coesão. Ou seja, para atingirmos uma boa coesão, é necessário dividir responsabilidades e acoplar a outras unidades de código. A partir disto, verificamos que este "relacionamento íntimo" é importante, mas deve ser muito bem dosado para não gerar códigos difíceis de serem mantigos.

Na OO, os conceitos de classe e associação podem ser usados para facilitar o uso de acoplamento. Ao usar o conceito de classe, consegue-se criar unidades de códigos mais autocontidas e coesas. A partir disto, o acoplamento entre elas torna-se mais alto nível do que

entre porções de código como funções, `headers` etc. Conseguir criar aplicações com uma boa coesão e acoplamento é um dos desafios da OO.

Gap semântico

Também chamado de *Fosso Semântico*, este termo caracteriza a diferença existente entre duas representações de conceitos por diferentes representações linguísticas. No contexto da computação, refere-se à diferença entre a representação de um contexto do conhecimento em linguagens (paradigmas) de programação.

Representar os conceitos que as aplicações necessitam para se tornarem projetos de sucesso de forma adequada e realista é um desafio. Em linguagens como C - em que é necessário se preocupar mais em definir entradas, processá-las e gerar saídas -, fica difícil trabalhar em alto nível. Trabalhar com variáveis (globais ou locais) e funções que são definidas desassociadamente dessas variáveis - mas que devem operar sobre elas - não é um trabalho amigável, principalmente em aplicações de grande porte, que são mais complexas por natureza.

Por mais que criemos `structs` para tentar aglutinar informações, as funções ainda estariam desassociadas delas. Esse gap da representação procedural em relação ao mundo real é o que torna este paradigma limitado. Essa dificuldade é a grande diferença da Orientação a Objetos. Ela disponibiliza, principalmente, os conceitos de classe, atributo, método e objeto para conseguir representar de forma mais realista os conceitos que a aplicação deseja representar.

Introdução a Orientação a Objetos

Definição

Como já foi apresentado, a OO não se limita em ser uma nova forma de programação. Ela também se preocupa com a modelagem (análise e projeto) dos processos/tarefas que devem ser realizados. Mais do que um tipo de "linguagem de programação", a OO é uma nova forma de se pensar e representar de forma mais realista as necessidades dos softwares.

Os fundamentos

Antes de aplicarmos os conceitos em um caso mais prático, é importante prover um embasamento sobre os pilares (fundamentos) da OO. Todos os conceitos que esta aula apresenta têm como finalidade possibilitar e facilitar a aplicação desses pilares. Mais uma vez, o uso correto destes conceitos eleva e facilita o processo de programação.

Abstração

Dentre as várias definições do termo **abstração**, a seguinte se encaixa bem no nosso contexto: *"Processo pelo qual se isolam características de um objeto, considerando os que tenham em comum certos grupos de objetos"*.

A ideia que essa definição transmite é que não devemos nos preocupar com características menos importantes, ou seja, acidentais. Devemos, neste caso, nos concentrar apenas nos aspectos essenciais. Por natureza, as abstrações devem ser incompletas e imprecisas, mas isto não significa que ela perderá sua utilidade. Na verdade, esta é a sua grande vantagem, pois nos permite, a partir de um contexto inicial, modelar necessidades específicas. Isso possibilita flexibilidade no processo de programação, já que é possível não trabalharmos com o conceito alvo diretamente, mas sim com suas abstrações.

Por exemplo, se uma fábrica de cadeiras fosse representar os produtos que ela já fabrica e vende, ou mesmo que um dia venha a fabricar e vender, ela poderia pensar inicialmente em uma cadeira da forma mais básica (abstrata) possível. Com isto, seu processo de produção seria facilitado, pois ela não saberia inicialmente quais os tipos de cadeiras que ela poderia fabricar, mas saberia que a cadeira teria, pelo menos, pernas, assento e encosto.

A partir disto, ela poderia fabricar diversos tipos: cadeira de praia, cadeira de aula, cadeiras com design contemporâneo, entre vários outros tipos, a medida que novas demandas viessem a surgir. Neste caso, ela adaptaria sua linha de produção a partir de um molde inicial.

Em cada tipo, algo poderia ser acrescentado ou modificado de acordo com sua especificidade. Assim, na cadeira de aula, teria um braço, já a de praia seria reclinável. Por fim, a contemporânea teria o assento acoplado ao encosto.

Os processos de inicialmente se pensar no mais abstrato e, posteriormente, acrescentar ou se adaptar são também conhecidos como generalização e especialização, respectivamente. Mais à frente, serão explicados os conceitos de classe e herança, bases para entendermos melhor o conceito de abstração.

Reuso

Não existe pior prática em programação do que a repetição de código. Isto leva a um código frágil, propício a resultados inesperados. Quanto mais códigos são repetidos pela aplicação, mais difícil vai se tornando sua manutenção. Isso porque facilmente se pode esquecer de atualizar algum ponto que logo levará a uma inconsistência, pois se é o mesmo código que está presente em vários lugares, é de se esperar que ele esteja igual em todos eles.

Para alcançar este fundamento, a OO provê conceitos que visam facilitar sua aplicação. O fato de simplesmente utilizarmos uma linguagem OO não é suficiente para se atingir a reusabilidade, temos de trabalhar de forma eficiente para aplicar os conceitos de herança e associação, por exemplo.

Na herança, é possível criar classes a partir de outras classes. Como consequência disto, ocorre um reaproveitamento de códigos - dados e comportamentos - da chamada classe mãe. Neste caso, a classe filha, além do que já foi reaproveitada, pode acrescentar o que for necessário para si.

Já na associação, o reaproveitamento é diferente. Uma classe pede ajuda a outra para poder fazer o que ela não consegue fazer por si só. Em vez de simplesmente repetir, em si, o código que está em outra classe, a associação permite que uma classe forneça uma porção de código a outra. Assim, esta troca mútua culmina por evitar a repetição de código.

Encapsulamento

Uma analogia com o mundo real será feita para inicialmente entendermos o que vem a ser o encapsulamento. Quando alguém se consulta com um médico, por estar com um resfriado, seria desesperador se ao final da consulta o médico entregasse a seguinte receita:

```
Receituário (Complexo)

- 400mg de ácido acetilsalicílico
- 1mg de maleato de dexclorfeniramina
- 10mg de cloridrato de fenilefrina
- 30mg de cafeína

Misturar bem e ingerir com água. Repetir em momentos de crise.
```

A primeira coisa que viria em mente seria: onde achar essas substâncias? Será que é vendido tão pouco? Como misturá-las? Existe alguma sequência? Seria uma tarefa difícil - até complexa - de ser realizada. Mais simples do que isso é o que os médicos realmente fazer: passam uma cápsula onde todas estas substâncias já estão prontas. Ou seja, elas já vêm encapsuladas.

Com isso, não será preciso se preocupar em saber quanto e como as substâncias devem ser manipuladas para no final termos o comprimido que resolverá o problema. O que interessa é o resultado final, no caso, a cura do resfriado. A complexidade de chegar a essas medidas e como misturá-las não interessa. É um processo que não precisa ser do conhecimento do paciente.

```
Receituário (Encapsulado)

1 comprimido de Resfriol. Ingerir com água. Repetir em momentos de crise.
```

Essa mesma ideia se aplica na OO. No caso, a complexidade que desejamos esconder é a de implementação de alguma necessidade. Com o encapsulamento, podemos esconder a forma como algo foi feito, dando a quem precisa apenas o resultado gerado.

Uma vantagem deste princípio é que as mudanças se tornam transparentes, ou seja, quem usa algum processamento não será afetado quando seu comportamento interno mudar.

POO com Python

Agora que vimos os conceitos básicos do POO, vamos falar um pouco sobre como conseguimos aplicá-los utilizando Python. Veja que o objetivo aqui não é abordar todo o potencial da orientação a objetos com a linguagem - para isso, precisaríamos de um curso inteiro!

Vamos discutir aqui os principais conceitos, de forma que possamos utilizar algumas estruturas básicas nos nossos projetos.

Discutiremos os seguintes conceitos:

- Classes
- Objetos
- Métodos
- Métodos especiais
    - `__init__`
    - `__str__`
    - `__eq__`
- Herança
- Polimorfismo

Para aplicar esses conceitos, vamos considerar o início de uma aplicação bem simplificada de um possível sistema de agendamento de consultas médicas. Construa um projeto com a seguinte estrutura de pacotes e módulos:

```
main.py
consulta/
  __init__.py
  entidades/
    __init__.py
    pessoa.py
    paciente.py
    medico.py
    consulta.py
```

Classes

Embora o nome do paradigma que estamos estudando é o Orientado a Objetos, tudo começa com a definição da classe. Antes mesmo de conseguirmos manipular objetos, precisamos definir uma classe, pois esta é a unidade inicial e mínima de código na OO. É a partir de classes que futuramente será possível criar objetos.

"Classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar."

O objetivo de uma classe é definir, servir de base, para o que futuramente será o objeto. É através dela que criamos o "molde" aos quais os objetos deverão seguir. Este "molde" definirá quais informações serão trabalhadas e como elas serão manipuladas.

A classe é a forma mais básica de se definir apenas uma vez como devem ser todos os objetos criados a partir dela, em vez de dfinir cada objeto separadamente e até repetidamente. A partir disto, logo percebemos que o conceito de classe é fundamental para a aplicação da abstração. Assim, uma classe também pode ser definida como uma abstração de uma entidade, seja ela física (bola, pessoa, carro, etc.) ou conceitual (viagem, venda, estoque, etc.) do mundo real.

É através de criação de classes que se conseguirá codificar todas as necessidades de um sistema. Mas como será possível identificar as necessidades, entidades, de um software? Um bom ponto de partida é pensar em substantivos. Estes são responsáveis por nomear tudo o que conhecemos, então é a partir deles que se possibilitará identificar quais as entidades um software terá de modelar.

Por exemplo, imagine que precisamos desenvolver um site de vendas online. Assim, aparecerão entidades como `Venda`, `Cliente`, `Fornecedor`, `Produto`, entre outras. Vemos que todos estes substantivos fazem parte do contexto de um site de vendas como esse. Logo, é possível especificar (codificar) classes, para assim manipular tais entidades.

Mas como devemos chamar, nomear as classes? Seu nome deve representar bem sua finalidade dentro do contexto ao qual ela foi necessária. Por exemplo, em um sistema de controle hospitalar, podemos ter uma classe chamada `Pessoa` para representar quem está hospitalizado ou apenas sendo consultado. Já em um sistema de pondo de vendas (PDV), temos mais uma vez o conceito de `Pessoa`, que neste caso é quem está comprando os produtos.

A partir disto, é possível termos classes nestes sistemas com estas definições. Entretanto, nota-se que o termo *pessoa* pode gerar uma ambiguidade, embora esteja correto. No hospital, também existem os médicos, enfermeiros; e no supermercado, gerentes e vendedores. Todos são pessoas.

Assim, muito melhor seria definir a classe `Paciente` no hospital e, no PDV, `Cliente`, além de `Médico`, `Vendedor`, respectivamente. Todos estes são pessoas, mas dentro de cada contexto eles representam papéis diferentes, então, para melhorar o entendimento e a representatividade, seria melhor mudar seus nomes.

Embora possa parecer preciosismo, classes com nomes pobremente definidos podem dificultar o entendimento do código e até levar a erros de utilização. Pense bem antes de nomear uma classe.

Em Python, utilizamos a notação **CamelCase** para nomear classes. Ou seja, ao contrário das variáveis, dos módulos e das funções, que utilizamos **snake_case** (todas as letras minúsculas, palavras separadas pelo underscore), em classes utilizamos a primeira letra de cada palavra em maiúscula.

Definindo as classes do projeto

No nosso projeto, as duas classes mais óbvias que precisamos é a de `Medico` e `Paciente`, afinal estamos falando de um sistema de consultas médicas. No entanto, como vamos falar um pouco mais à frente, médicos e pacientes possuem algumas informações em comum, como data de nascimento, nome, cpf, etc. Sendo assim, vamos criar uma classe chamada `Pessoa` para representar essas propriedades comuns às duas classes. Quando falarmos de hierarquia vamos entrar um pouco mais nesse detalhe.

Além dessas três classes, precisamos também abstrair o conceito abstrato de `Consulta`, para ser detalhada e implementada no nosso projeto.

Atributos

Após o processo inicial de identificar as entidades (classes) que devem ser manipuladas, começa a surgir a necessidade de detalhá-las. A primeira coisa que vem à mente é: quais informações devem ser manipuladas através desta classe? A partir disto, começa-se a tarefa de caracterizá-las. Essas características é que vão definir quais informações as classes poderão armazenar e manipular. Na OO, estas características e informações são denominadas de **atributo**.

Atributo é o elemento de uma classe responsável por definir sua estrutura de dados. O conjunto destes será responsável por representar suas características e fará parte dos objetos criados a partir da classe.

Essa definição deixa bem claro que os atributos devem ser definidos dentro da classe. Devido a isso, são responsáveis por definir sua estrutura de dados. É a partir do uso de atributos que será possível caracterizar (detalhar) as classes, sendo possível representar fielmente uma entidade do mundo real.

Assim como nas classes, os atributos podem ser representados a partir de substantivos. Além destes, podemos também usar adjetivos. Pensar em ambos pode facilitar o processo de identificação dos atributos.

Por exemplo, imagine que a entidade `Paciente` foi identificada para o sistema hospitalar. Alguns de seus atributos podem ser nome, CPF, sexo. Todos estes são substantivos, mas alguns de seus valores poderiam ser adjetivos. Quanto mais for realizado o processo de caracterização, mais detalhada será a classe e, com isso, ela terá mais atributos. Porém, é preciso ter parcimônia no processo de identificação dos atributos.

Embora um atributo possa pertencer à classe, nem sempre fará sentido ele ser definido. Isso ocorre devido ao *contexto* no qual a classe vai ser usada. Por exemplo, foi visto anteriormente que `Paciente` não deixa de ser uma `Pessoa`, e geralmente elas possuem um hobby. Porém, em um contexto hospitalar, este atributo não agregaria muito valor. Já em `Cliente` - que também é uma pessoa - seria mais interessante, pois a partir de seu hobby poderiam ser apresentados produtos que lhe interessassem mais. Percebe-se, com isso, que o contexto de uso da classe vai impactar diretamente no processo de definição de seus atributos.

Ainda no processo de identificação e criação dos atributos, é válido ressaltar que nem sempre uma informação, mesmo sendo importante (uma característica inerente à entidade), deve ser transformada em um atributo. Um exemplo clássico disso é a idade. Embora essa seja uma característica válida e importante para uma pessoa, seja ela um `Paciente` ou `Cliente`, devido ao trabalho de mantê-la atualizada (todo ano fazemos aniversário), não valeria a pena criá-la.

Neste caso, seria melhor usar o que é conhecido como *atributo calculado* ou *atributo derivado*. A idade não se torna um atributo em si, mas tem seu valor obtido a partir de um método (ainda será explicado o que vem a ser um método mais à frente). Assim, a idade de um paciente poderia ser calculada a partir da data atual menos a data de nascimento, essa sim um atributo da classe `Paciente` ou `Cliente`.

Não diferentemente de linguagens estruturadas, um atributo possui um tipo. Como sua finalidade é armazenar um valor que será usado para caracterizar a classe, ele precisará identificar qual o tipo do valor armazenado em si. Linguagens orientadas a objetos proveem os mesmos tipos de dados básicos - com pequenas variações - que suas antecessoras.

A nomeação de atributos deve seguir a mesma preocupação das classes: deve ser o mais representativo possível. Nomes como `qtd`, `vlr` devem ser evitados. Esses nomes eram válidos na época em que as linguagens de programação e os computadores que as executavam eram limitados, portanto deveríamos sempre abreviar os nomes das variáveis. Entretanto, atualmente não temos essa limitação, e os editores modernos possuem recursos para reaproveitar nomes de atributos e variáveis sem precisar digitar tudo novamente. Portanto, não deixe a preguiça lhe dominar e escreva `quantidade` e `valor`.

Utilize nomes claros. Evite, por exemplo, o atributo `data`. Uma data pode significar várias coisas: nascimento, morte, envio de um produto, cancelamento de venda. Escreva exatamente o que se deseja armazenar nesse atributo: `dataNascimento`, `dataObito`, etc.

Definindo os atributos das nossas classes

Para ambos os tipos de pessoas envolvidos no projeto, `Medico` e `Paciente`, é extremamente necessário termos um `nome` e um `cpf` associados. Informações como `data_nascimento` e `endereco` são importantes, mas não são essenciais.

No caso particular do `Medico`, é imprescindível termos também o seu `crm` cadastrado!

Já no caso da `Consulta`, precisamos saber o dia em que ela ocorrerá, além do médico que vai atender e do paciente.

Métodos

Tendo identificado a classe com seus atributos, as seguintes perguntas podem surgir: mas o que fazer com eles? Como utilizar a classe e manipular os atributos? É nessa hora que o método entra em cena. Este é responsável por identificar e executar as operações que a classe fornecerá. Essas operações, via de regra, têm como finalidade manipular os atributos.

Método é uma porção de código (sub-rotina) que é disponibilizada pela classe. Esta é executada quando é feita uma requisição a ela. Um método serve para identificar quais serviços, ações, que a classe oferece. Eles são responsáveis por definir e realizar um determinado comportamento.

Para facilitar o processo de identificação dos métodos de uma classe, podemos pensar em verbos. Isso ocorre devido à sua própria definição: *ações*. Ou seja, quando se pensa nas ações que uma classe venha a ofereces, estas identificam seus métodos.

No processo de definição de um método, a sua assinatura deve ser identificada. Esta nada mais é do que o nome do método e sua lista de parâmetros. Mas como nomear os métodos? Novamente, uma expressividade ao nome do método deve ser fornecida, assim como foi feito com o atributo.

Por exemplo, no contexto do hospital, imagine termos uma classe `Procedimento`, logo, um péssimo nome de método seria `calcular`. Calcular o quê? O valor total do procedimento, o quanto cada médico deve receber por ele, o lucro do plano? Neste caso, seria mais interessante `calcularTotal`, `calcularGanhosMedico`, `calcularLucro`.

Veja que, ao lermos esses nomes, logo de cara já sabemos o que cada método se propõe a fazer. Já a lista de parâmetros são informações auxiliares que podem ser passadas aos métodos para que estes executem suas ações. Cada método terá sua lista específica, caso haja necessidade. Esta é bem livre e, em determinados momentos, podemos não ter parâmetros, como em outros podemos ter uma classe passada como parâmetro, ou também tipos primitivos e classes ao mesmo tempo.

Há também a possibilidade de passarmos somente tipos primitivos, entretanto, isto remete à programação procedural e deve ser desencorajado. Via de regra, se você passa muitos parâmetros separados, talvez eles pudessem representar algum conceito em conjunto. Neste caso, valeria a pena avaliarmos se não seria melhor criar uma classe para aglutiná-los.

Por fim, embora não faça parte de sua assinatura, os métodos devem possuir um retorno. Se uma ação é disparada, é de se esperar uma reação. O retorno de um método pode ser qualquer um dos tipos primitivos vistos na seção sobre atributos.

Além destes, o método pode também retornar qualquer um dos conceitos (classes) que foram definidos para satisfazer as necessidades do sistema em desenvolvimento, ou também qualquer outra classe - não criada pelo programador - que pertença à linguagem de programação escolhida.

Definindo os métodos das nossas classes

Para ambas as classes, `Medico` e `Paciente`, precisamos ter métodos que cadastrem o endereço e a data de nascimento, caso esses dados sejam fornecidos. Assim, precisamos criar os métodos `adiciona_endereco` e `adiciona_nascimento`. Além disso, pensando no uso da `Consulta`, devemos precisar de um método que retorne o nome da pessoa envolvida na consulta, então vamos precisar criar um método `le_nome`.

Veja que todos os verbos estão na voz ativa. Em alguns casos é comum ver os verbos no infinitivo. Não há diferença entre as duas formas de nomear métodos, no entanto é importante escolher uma forma e seguir com ela o projeto todo, para manter uma uniformidade do código.

Por fim, nossa `Consulta` precisa, nesse momento, apenas comunicar que aquele agendamento foi feito corretamente. Então teremos um método `comunica_agendamento`.

O método construtor

Em uma classe, independente de qual conceito ela queira representar, podemos ter quantos métodos forem necessários. Cada um será responsável por uma determinada operação que a classe deseja oferecer. Muitas vezes, os métodos trabalham em conjunto para realizar seus comportamentos. Além disso, independente da quantidade e da finalidade dos métodos de uma classe o método construtor está presente em todas elas.

O construtor é responsável por criar objetos a partir da classe em questão. Ou seja, sempre que for necessário criar objetos de uma determinada classe, seu construtor deverá ser utilizado. É através do seu uso que será possível instanciar objetos e, a partir disto, manipular de forma efetiva seus atributos e métodos.

Além disto, uma outra função do construtor é prover alguns valores iniciais que o objeto precisa ter inicialmente. O nome do método construtor, em Python, é `__init__`. Diferentemente de outros métodos, no entanto, o construtor é "sem retorno", ou seja, não devolve nenhum dado para a instrução que a chamou.

Implementando nossa primeira classe

Com as informações que temos, vamos programar nossa primeira classe. Vamos pegar todas as propriedades (atributos e métodos) comuns a `Medico` e `Paciente` e vamos escrever a classe `Pessoa`.

Veja que, em Python, todos os métodos possuem como primeiro parâmetro o atributo `self`. Esse é um atributo especial que serve para identificar propriedades daquela instância em particular. Ou seja, se eu quiser usar o `nome` de uma instância específica de uma classe, eu devo usar `self.nome`.

Apesar do parâmetro `self` aparecer em todos os métodos, ele não deve ser identificado na chamada do método. Por exemplo, ao chamar o método assinado como `def adiciona_endereco(self, endereco):`, basta passar `.adiciona_endereco(endereco)`.

Vamos criar então nossa classe! Veja que os campos `endereco` e `data_nasc` são opcionais, então estamos definindo parâmetros padrão no construtor para o caso do usuário não passar esses parâmetros.

```
class Pessoa:
  def __init__(self, nome, cpf, endereco=None, data_nasc=None):
    self.nome = nome
    self.cpf = cpf
```

```
    self.endereco = endereco
    self.data_nasc = data_nasc

  def adiciona_endereco(self, endereco):
    self.endereco = endereco

  def adiciona_nascimento(self, data_nasc):
    self.data_nasc = data_nasc

  def le_nome(self):
    return self.nome
```

Veja que não colocamos parênteses na definição da classe. Isso acontece quando não temos uma herança definida (logo abaixo falamos sobre isso).

Objetos

Embora o nome do paradigma que está sendo estudado seja a Orientação a Objeto, já tínhamos visto que tudo começa com a definição de uma classe. Então, o que é um objeto? É a instanciação de uma classe.

Como já explicado, a classe é a abstração base a partir da qual os objetos serão criados. Quando se usa a OO para criar um software, primeiro pensamos nos objetos que ele vai manipular/representar. Tendo estes sido identificados, devemos então definir as classes que serviram de abstração base para que os objetos venham a ser criados (instanciados).

Um objeto é a representação de um conceito/entidade do mundo real, que pode ser física (bola, carro, árvore, etc.) ou conceitual (viagem, estoque, compra, etc.) e possui um significado bem definido para um determinado software. Para esse conceito/entidade, deve ser definida inicialmente uma classe a partir da qual posteriormente serão instanciados objetos distintos.

Contextualizando melhor, imagine que temos um software de Fluxo de Caixa, logo, um conceito que teria de manipular seria uma *conta*. Então, deveríamos criar uma classe chamada `Conta` e, a partir dela, seriam criados objetos que representariam uma *Conta de Água*, *Conta de Energia*, *Conta de Telefone*, entre outros. Cada um teria seus respectivos atributos e métodos, como: total a ser pago, empresa que fornece tal serviço, verificar pagamento, calcular multa, etc.

A partir disto, percebemos que, no processo de identificação dos conceitos/entidades que são necessários para o software se tornar operante, primeiro devemos identificar os objetos em um alto nível de pensamento. Somente após este processo é que as classes com seus atributos e métodos são definidas para abstrair esses objetos e, finalmente, criar cada objeto distinto a partir da classe definida.

Instanciando uma classe em Python

Para instanciar uma classe, basta chamar a classe que desejamos, como se ela fosse uma função, e atribui-la a uma variável. Essa variável, então, vai possuir todos os métodos e

atributos definidos naquela classe.

Entre no arquivo main.py e digite o seguinte código:

```
from consulta.entidades.pessoa import Pessoa

pessoa = Pessoa("Victor", "123")

print(pessoa)
print(pessoa.nome)
pessoa.cadastra_endereco("rua 1")
print(pessoa.endereco)

outra_pessoa = Pessoa("Ana", "456")

print(outra_pessoa.nome)
```

Quando executar o seu código, você deverá receber um resultado como o seguinte:

```
<consulta.entidades.pessoa.Pessoa object at 0x7f3542718b80>
Victor
rua 1
Ana
```

Pronto! Criamos duas instâncias para a classe `Pessoa`, uma de nome "Victor" e outra de nome "Ana". Ambas possuem as mesmas propriedades, como nome e CPF; e podem fazer as mesmas operações, como cadastrar um endereço ou ler um nome.

Representação de uma instância

Imagino que você tenha reparado na primeira linha do resultado da seção anterior:

```
<consulta.entidades.pessoa.Pessoa object at 0x7f3542718b80>
```

O código que é apresentado aqui, `0x7f3542718b80`, nada mais é do que o endereço na memória no qual aquela instância está armazenada. Certamente a sua execução indicou um outro endereço de memória.

Porém, essa informação não é muito útil para o usuário. Para "consertar" isso, podemos utilizar um método pré-definido em Python, o `__str__`. Esse método faz, basicamente, a conversão da instância da classe para um formato de string.

É uma boa prática definir esse método, para facilitar a visualização dos objetos ao longo do nosso código. Sendo assim, inclua o seguinte método na sua classe `Pessoa`:

```
  def __str__(self):
    ret = f"Nome: {self.nome}\n"
    ret += f"CPF: {self.cpf}"
    return ret
```

Rodando novamente nosso código, passamos a ter o seguinte resultado:

```
Nome: Victor
CPF: 123
Victor
rua 1
Ana
```

Veja agora que o retorno do `print(pessoa)` foi algo bem mais claro! Tivemos uma informação que é, de fato, útil para nós.

Igualdade entre objetos

Agora, altere o seu código na `main.py` para ficar assim:

```
from consulta.entidades.pessoa import Pessoa

pessoa = Pessoa("Victor", "123")
outra_pessoa = Pessoa("Victor", "123")

print(pessoa == outra_pessoa)
```

Era de se esperar que o retorno desse `print` seja `True`, certo? Afinal de contas, ambos os objetos possuem o mesmo nome e o mesmo CPF. No entanto, ao rodar o código, encontramos isso:

```
False
```

Isso acontece porque o interpretador compara simplesmente os endereços na memória. Se os objetos estão apontando para o mesmo endereço, eles são iguais. Do contrário, são diferentes.

Sabemos que isso não deveria acontecer, portanto vamos utilizar mais um método especial do Python, o `__eq__`. Nesse método, sempre passamos como parâmetro o `self` e mais um, representando um outro objeto da mesma classe. O método só precisa retornar uma comparação entre os atributos importantes para que o nosso programa identifique dois objetos como iguais.

Por exemplo, crie o seguinte método na classe `Pessoa`:

```
  def __eq__(self, pessoa):
    return (self.nome == pessoa.nome) and (self.cpf == pessoa.cpf)
```

Rodando novamente, o resultado passou para `True`. Isso porque nosso programa, ao invés de comparar os endereços na memória, agora está realizando a comparação presente no método `__eq__`.

Herança

Veja que, até agora, implementamos apenas a classe `Pessoa`. Entretanto, precisamos também implementar as classes `Medico` e `Paciente`. Ambas as classes possuem as mesmas propriedades de `nome`, `cpf`, `endereco` e `data_nascimento`, além dos mesmos métodos `adiciona_endereco`, `adiciona_nascimento` e `le_nome`.

Veja que, diferente do `Paciente`, o `Medico` precisa de um atributo especial, o `crm`. Além disso, ele também pode emitir atestados, então no caso particular do `Medico` seria necessário incluir um método `emite_atestado`.

Qual seria a solução? Caso estivéssemos trabalhando com uma linguagem procedural, possivelmente isso causaria uma duplicidade de código enorme, já que teríamos que replicar tudo o que fizemos até agora em duas novas classes. No POO, porém, temos o conceito de **herança** que nos ajuda a prevenir que isso aconteça.

Na herança, uma classe (chamada **subclasse** ou **classe filha**) herda, ou seja, incorpora propriedades (atributos e métodos) de uma outra classe (chamada **superclasse** ou **classe mãe**). Sendo assim, não precisamos implementar todo o código novamente.

Podemos colocar quantos níveis de herança quisermos. Por exemplo, poderíamos ter subclasses de `Medico` representando especialidades, como `Oftalmologista`, `Cardiologista` ou `Obstetra`. Por sua vez, essas subclasses poderiam ter outras, e assim sucessivamente. Chamamos esse processo de criar classes com um escopo cada vez mais definido de **especialização**. O processo inverso, de tentar definir similaridades entre classes distintas e definir superclasses em comum, é chamado de **generalização**.

O conceito de herança é um dos mais poderosos dentro do POO, e permite a criação de códigos muito eficientes dentro dos projetos.

Por exemplo, a nossa classe `Paciente`, como não possui nenhuma característica particular, no momento, pode ser implementada simplesmente como:

```
from consulta.entidades.pessoa import Pessoa

class Paciente(Pessoa):
  def __init__(self, nome, cpf, endereco=None, data_nasc=None):
    super().__init__(nome, cpf)
```

Veja que só precisamos definir novamente nosso construtor. Ainda assim, utilizamos a função `super()`, que basicamente indica para o interpretador que queremos chamar um método ou um atributo da nossa superclasse.

Além disso, para indicar que uma classe está herdando propriedades de outra, indicamos a superclasse entre parênteses, na declaração da subclasse (`class Paciente(Pessoa)`).

Agora, se quisermos, podemos instanciar um objeto da classe `Paciente`, passando os mesmos parâmetros da classe `Pessoa`.

Se não há diferença entre `Paciente` e `Pessoa`, por que criamos uma outra classe? Do ponto de vista de implementação, neste momento realmente não é algo necessário. Todavia, precisamos pensar em duas questões críticas:

- Do ponto de vista semântico, uma pessoa é diferente de um paciente. Uma pessoa pode ser um médico, um recepcionista, um almoxarife, etc. Nosso código precisa ser correto não só do ponto de vista lógico, mas também do ponto de vista semântico!
- Caso queiramos evoluir nosso sistema, por exemplo incluindo o histórico médico do paciente, não vamos poder fazer isso na classe `Pessoa`, afinal de contas não nos interessa cadastrar o histórico médico de um recepcionista ou de um almoxarife. Assim, nosso código fica pronto para podermos evoluir o projeto, reduzindo os pontos de alteração no sistema.

Polimorfismo

O **polimorfismo** é outro pilar do POO. Esse conceito indica a capacidade que uma subclasse tem de ter métodos com o mesmo nome de sua superclasse, e ainda assim o programa sabe qual método deve ser chamado, especificamente.

Ou seja, o objeto tem a capacidade de assumir diferentes formas, daí o nome polimorfismo.

No nosso projeto, vamos considerar a classe `Medico`. Precisamos aplicar o conceito de polimorfismo em dois casos: no seu construtor e no método `__str__`. Em ambos os casos, precisamos incluir o atributo `crm` na lógica de implementação.

Para aplicar polimorfismo em Python, basta reimplementar novamente o método desejado, alterando o seu comportamento. Lembre-se que podemos usar a função `super()` para chamar algum método da superclasse.

Portanto, considere a implementação abaixo para a classe `Medico`:

```
from consulta.entidades.pessoa import Pessoa

class Medico(Pessoa):
  def __init__(self, nome, cpf, crm, endereco=None, data_nasc=None):
    super().__init__(nome, cpf, endereco, data_nasc)
    self.crm = crm

  def __str__(self):
```

```
    ret = super().__str__()
    ret += f"\nCRM: {self.crm}"
    return ret

  def emite_atestado(self, paciente):
    print(f"Médico {self.nome} emitiu atestado para paciente {paciente.le_nome()}.")
```

Com esse código implementado, sempre que quisermos exibir os dados de um médico, vamos ver na tela não só o nome e o CPF (como estava previsto na superclasse `Pessoa`), mas também o seu CRM.

Finalizando nosso projeto

Com os conceitos definidos nessa aula, podemos finalizar o nosso projeto, implementando a classe `Consulta` e ajustando nosso módulo principal para realizar um teste básico.

Módulo consulta.py

Observe que, quando queremos chamar um método ou atributo da própria classe, basta usar a expressão `self.` antes do nome do método ou atributo.

Além disso, considerando que os atributos `self.medico` e `self.paciente` são instâncias das classes `Medico` e `Paciente`, para usar algum método dessas classes, basta inseri-lo após a referência do atributo, como `self.medico.le_nome()`.

```
class Consulta:
  def __init__(self, dia, medico, paciente):
    self.dia = dia
    self.medico = medico
    self.paciente = paciente

  def comunica_agendamento(self):
    print(f"Consulta de {self.paciente.le_nome()} agendada para {self.dia} com o medico {self.med

  def __str__(self):
    ret = f"Consulta dia {self.dia}\n"
    ret += f"Médico: {self.medico.le_nome()}\n"
    ret += f"Paciente: {self.paciente.le_nome()}"
    return ret
```

Módulo main.py

```
from consulta.entidades.paciente import Paciente
from consulta.entidades.medico import Medico
from consulta.entidades.consulta import Consulta
```

```
def main():
  medico = Medico("Fulano", "123", "111")
  paciente = Paciente("Ciclano", "456")
  paciente.adiciona_endereco("Rua 1")
  paciente.adiciona_nascimento("10/10/2010")

  print(medico)
  print(paciente)

  consulta = Consulta("25/10/2021", medico, paciente)
  consulta.comunica_agendamento()

if __name__ == "__main__":
  main()
```

Evoluindo o projeto

Você pode evoluir esse projeto com a quantidade de informações que você desejar. Aqui, algumas sugestões:

- Implementar uma classe `Consultorio` para agendar a consulta em um determinado consultório;
- Implementar um pacote `sistema`, incluindo métodos como `cadastro`, para organizar o cadastro de novos objetos e validar informações (não cadastrar dos pacientes iguais, por exemplo);
- Implementar um pacote `persistencia`, que armazene os dados cadastrados localmente, de forma que as informações não sejam perdidas a cada nova execução do programa;
- Construir uma funcionalidade de relatórios, apresentando as consultas agendadas em um dia ou mês;
- Desenvolver classes `Endereco` e `Data`, detalhando mais essas duas propriedades;
- Montar uma funcionalidade para identificar os aniversariantes no dia que o programa estiver sendo executado.

A sua criatividade é o limite! Com essas sugestões (e o que mais você desejar), é possível construir um projeto sofisticado, incluindo até conceitos que não abordamos aqui, como heranças múltiplas, sobrecarga, métodos estáticos, etc.