6

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

6

Herança e Polimorfismo

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo veremos como aplicar em Python os conceitos de herança e polimorfismo. Vamos construir uma hierarquia de classes, representando-a com os diagramas da UML e na sequência iremos implementá-las em Python, ilustrando os conceitos estudados. Vamos aprender o que significa a sigla MRO, como usar a função ***super*** *para acessar métodos de classes que estão acima na hierarquia de classes, e veremos como são aplicados em Python o polimorfismo de sobrescrita e de sobrecarga.*

6.1. Introdução

Continuando no estudo dos conceitos básicos de POO, vamos aprender agora como herança e o polimorfismo podem ser implementados em Python. Como vimos nos demais capítulos, a herança é uma das formas que nos permite reutilizar código, mas precisamos tomar bastante cuidado com o seu uso.

Se tentarmos resolver todos os problemas de reutilização de código apenas com o uso da herança, podemos criar uma teia hierárquica de relacionamentos entre as classes, que pode levar a dificuldades de manutenção e comportamentos inesperados no código quando precisarmos editar algo em uma das superclasses nesta rede.

Outra forma de reutilização de código é a composição, que nos permite compor diversas classes para podermos separar as responsabilidades de cada uma e melhorar a manutenabilidade do código.

6.2. Herança

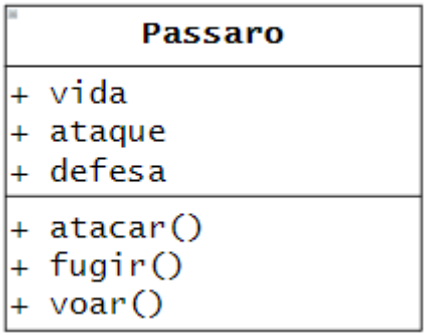
Em POO, um dos objetivos é aproximar a modelagem de um programa do mundo real, para facilitar a escrita e leitura de código. Portanto, o conceito de herança em POO é muito parecido com a taxonomia dos seres vivos, isto é, a forma que os classificamos em grupos de acordo com suas características.

Colocamos seres com características semelhantes em uma espécie, em seguida agrupamos espécies com características semelhantes em um gênero, gêneros semelhantes em uma família, famílias semelhantes em uma ordem, e assim por diante passando por classe e filo, até chegar no reino (animal, vegetal, etc.).

Vamos pensar no reino animal, que inclui todos os animais que conhecemos no mundo, é um conceito muito genérico, portanto ele define apenas os traços comuns a todos os animais, que coloca no mesmo grupo esponjas do mar, jacarés, grilos, pássaros e nós humanos. Em seguida temos diversas subdivisões de acordo com outras características destes seres, até chegar na espécie, cujos indivíduos possuem o maior grau de semelhança entre si.

Com esse exemplo em mente, vamos modelar um exemplo comumente utilizado para ilustrar a herança e um dos problemas que podem surgir se a implementamos sem a devida consideração. Digamos que estejamos programando um jogo de mundo aberto, que irá possuir diversos animais, e nossa tarefa é programar as classes para os pássaros do jogo. Criamos então a classe da Figura 6.1 para agrupar todos os pássaros do jogo.

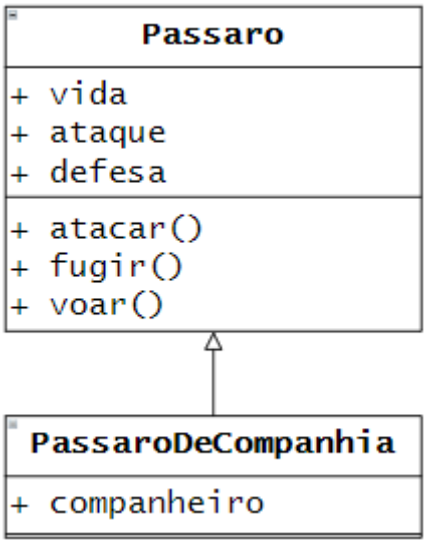
Figura 6.1: Exemplo inicial da classe *Passaro*



Fonte: do autor, 2021

Agora imagine que no design do jogo, alguns dos personagens principais poderão em determinado ponto ganhar um pássaro de companhia, que irá ajudá-los nas missões. Podemos então estender a classe pássaro e criar uma nova classe com um atributo para guardar o personagem associado ao pássaro. Veja a Figura 6.2.

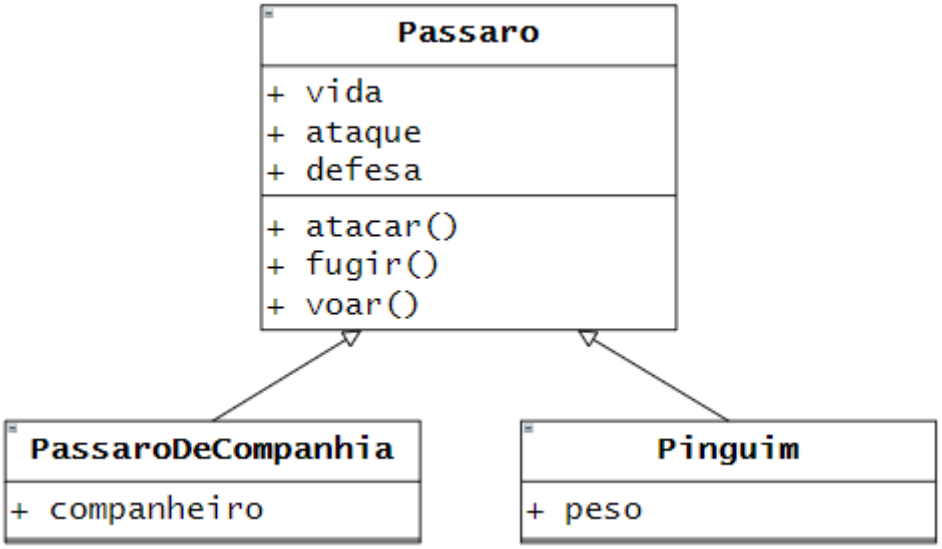
Figura 6.2: Criação da classe *PassaroDeCompanhia*, estendendo a classe *Passaro*



Fonte: do autor, 2021

Inicialmente essa classe funciona perfeitamente, mas imagine agora que em uma vila do jogo, há um mercador de animais exóticos que vende pinguins, precisamos de uma classe para eles também, então podemos criá-la herdando de `Passaro` e teremos a situação da Figura 6.3.

Figura 6.3: Inclusão da classe Penguin, estendendo a classe *Passaro*



Fonte: do autor, 2021

Antes de seguir na leitura, veja se consegue encontrar um problema que introduzimos na modelagem das classes quando fizemos a classe `Pinguim` herdar de `Passaro`.

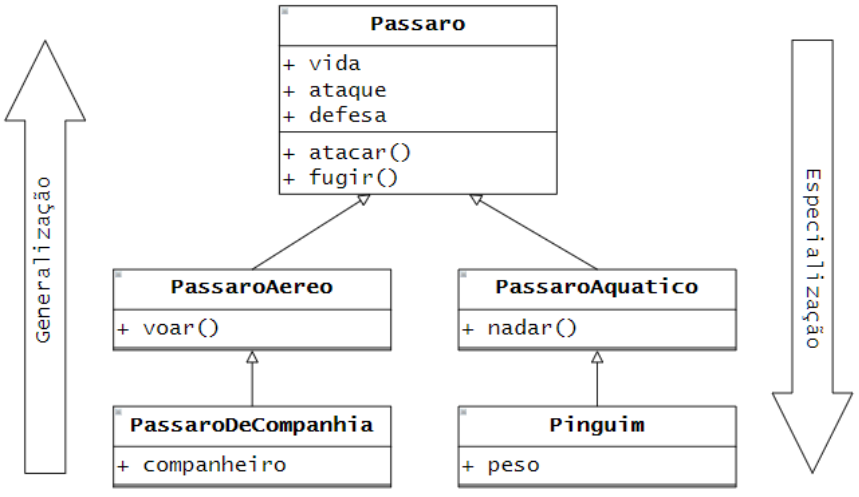
Exatamente, pinguins não voam, mas a classe `Passaro` implementa um método `voar()` que será herdado por `Pinguim`, pois um pinguim é um pássaro. O que deve acontecer quando esse método for chamado em um objeto do tipo `Pinguim`? Podemos sobrescrever o método e fazer com que um erro seja levantado, indicando que aquela ação não pode ser realizada, mas essa não é uma boa abordagem.

É esperado que um objeto de uma classe mãe possa ser substituído por um objeto de qualquer uma das classes filhas, sem que isso altere a expressão do código, ou seja, um método que funciona em uma classe não pode deixar de existir ou levantar um erro em uma classe derivada (esse é um dos princípios do SOLID, que veremos em outro capítulo).

Você pode estar se perguntando agora "mas e o polimorfismo, não é justamente a alteração de um método nas classes filhas, isso não contradiz o parágrafo anterior?", e essa é uma pergunta válida. Como veremos ainda neste capítulo, a forma como o método funciona pode ser diferente no polimorfismo, mas o resultado final não. No caso do método voar, poderíamos ter um pássaro mecânico que implementa tal método com o uso de motores e hélices, ao invés de bater as asas, então a forma (implementação) é diferente, mas o resultado é o mesmo: ao chamar o método `voar()`, o pássaro voa.

Agora, como podemos resolver o problema mencionado acima em nossas classes? Nessa situação, a melhor coisa é repensar a hierarquia que definimos de maneira a acomodar tais mudanças, e por isso este é um processo extremamente importante de ser feito no começo do projeto para evitar mudanças drásticas no futuro, que impactam toda a aplicação e muitas vezes são inviáveis. A Figura 6.4 traz um exemplo da nova hierarquia que evita o problema do método `voar` sendo passado para a classe `Pinguim`.

Figura 6.4: Reestruturação das classes



Fonte: do autor, 2021

Note que conforme subimos na hierarquia de classes, vamos para classes mais genéricas, e conforme descemos, chegamos a classes mais específicas.

Com isso, vemos que antes de programar qualquer linha de código, é importante definir qual será a responsabilidade de cada classe, o que ela está modelando, quais objetos estamos abstraindo e agrupando em uma classe, como estes objetos irão se relacionar, etc. Pois assim evitamos ou reduzimos a necessidade de alterar trechos de código por toda a aplicação para acomodar uma reorganização das classes.

Em uma situação real, estaríamos trabalhando com potencialmente muito mais classes, e é quase certo que novos recursos sejam adicionados com o passar do tempo, então é preciso desenvolver nosso programa ou aplicação de modo que os módulos e classes possam ser estendidos e reutilizados de maneira simples e fácil.

Para nos ajudar nessa tarefa, existe um conjunto de princípios de POO, desenvolvidos por Robert C. Martin entre o final da década de 1990 e começo dos anos 2000, com foco em como projetar um programa ou aplicação para que o código seja reutilizável, robusto e flexível (MARTIN, R. C., 2000). Os primeiros 5 princípios são popularmente conhecidos pelo acrônimo SOLID, e fazem referência específica ao projeto de classes em POO.

PARA ASSISTIR!

Sandi Metz fez uma palestra a respeito dos princípios do SOLID, disponível no Youtube em <https://www.youtube.com/watch?v=v-2yFMzxqwU>, com possibilidade de legendas em português geradas automaticamente. No vídeo ela explica que Robert C. Martin não inventou sozinho todos os princípios, mas foi o responsável por juntar diferentes ideias que estavam circulando à época e nomeá-las em seu artigo, que serviu desde então de base para a discussão do desenvolvimento de software segundo a POO.

Agora vamos implementar a herança das classes acima em Python. Para fazer com que uma classe herde de outra, indicamos a classe mãe entre parênteses no momento de criação da classe filha:

```
class ClasseFilha(ClasseMae):
    pass
```

Começamos então definindo a classe base inicial, para isso crie um arquivo “passaros.py” na pasta “aula06”, com o seguinte código:

```
class Passaro:
    def __init__(self, vida, ataque, defesa):
        self.vida = vida
        self.ataque = ataque
        self.defesa = defesa

    def atacar(self, alvo):
        pass

    def fugir(self, destino):
        pass
```

Não nos importamos com a implementação dos métodos, pois o objetivo aqui é demonstrar as características da herança em POO, então em seguida, vamos implementar as duas subclasses de `Passaro`.

Adicione o seguinte código ao arquivo passaros.py:

```
class PassaroAereo(Passaro):
    def voar(self):
        pass

class PassaroAquatico(Passaro):
    def nadar(self):
        pass
```

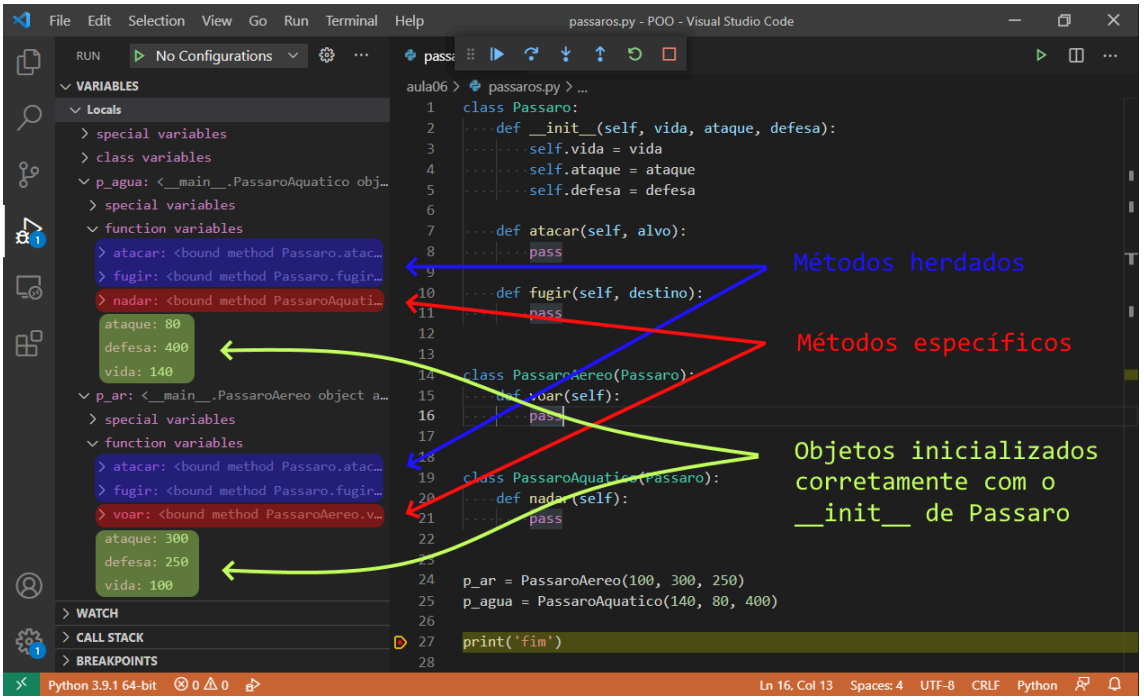
Nestas classes, não precisamos adicionar nenhum atributo, então não há necessidade de implementar o método inicializador, basta implementar os métodos específicos de cada classe, que o Python irá usar o inicializar herdado da classe mãe para inicializar os objetos. Vejamos um exemplo no modo de depuração do VSCode.

Adicione o seguinte código ao arquivo, adicione também um ponto de parada na última linha do código e execute-o pressionando F5.

```
p_ar = PassaroAereo(100, 300, 250)
p_agua = PassaroAquatico(140, 80, 400)
print('fim')
```

A execução do código pode ser vista na Figura 6.5.

Figura 6.5: Visualização dos métodos herdados e específicos nos objetos das classes filhas



Fonte: do autor, 2021

6.2.1. Usando a função integrada *super* em Python

Para implementar as próximas duas classes, precisamos adicionar mais um atributo ao objeto, então a primeira ideia que poderíamos fazer seria:

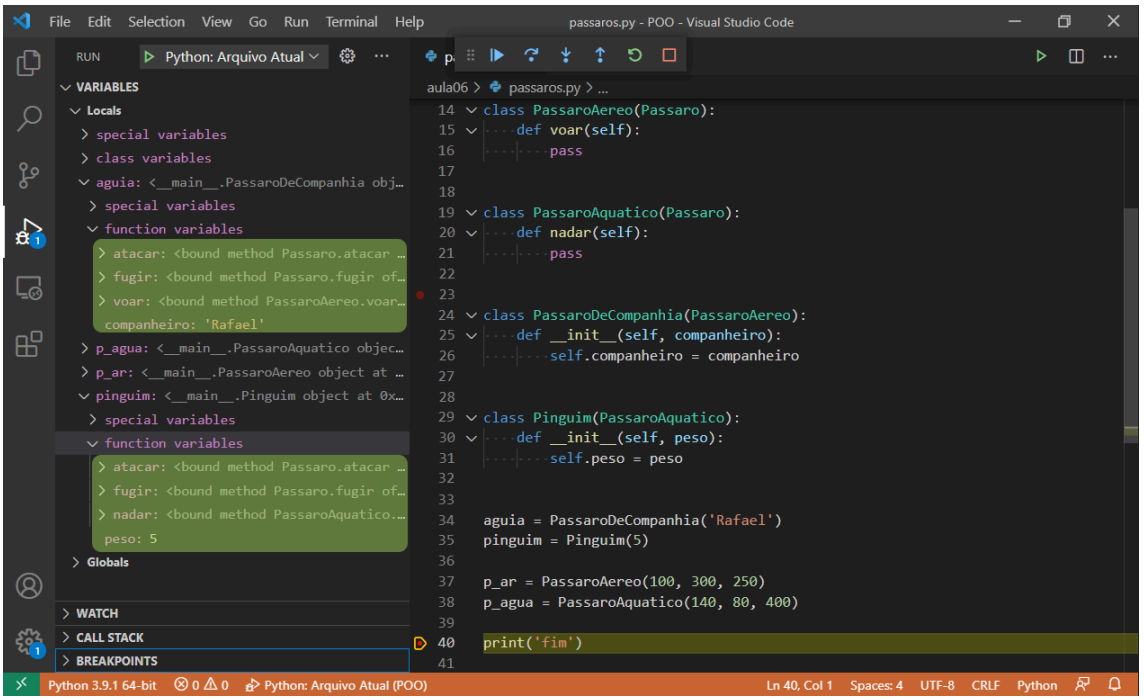
```
class PassaroDeCompanhia(PassaroAereo):
    def __init__(self, companheiro):
        self.companheiro = companheiro

class Pinguim(PassaroAquatico):
    def __init__(self, peso):
        self.peso = peso
```

Em seguida podemos instanciar um objeto de cada classe, da seguinte forma:

```
aguia = PassaroDeCompanhia('Rafael')
pinguim = Pinguim(5)
```

Adicione os trechos de código acima ao arquivo “passaros.py”, após a definição das classes já existentes e antes das linhas em que instanciamos tais classes para testar. O resultado pode ser visto na Figura 6.6.

Figura 6.6: Visualização dos métodos e atributos herdados pelas classes “netas”



Fonte: do autor, 2021

Podemos ver na Figura 6.6 que tanto os métodos gerais quanto os métodos das classes intermediárias foram herdados corretamente, mas o que aconteceu com os demais atributos? Os nossos objetos possuem apenas os atributos específicos, tendo perdido os atributos gerais de `Passaro`. Isso ocorreu porque sobrescrevemos o método inicializador, então foi executado o método `__init__` de `Pinguim` e não mais o de `Passaro`, por exemplo.

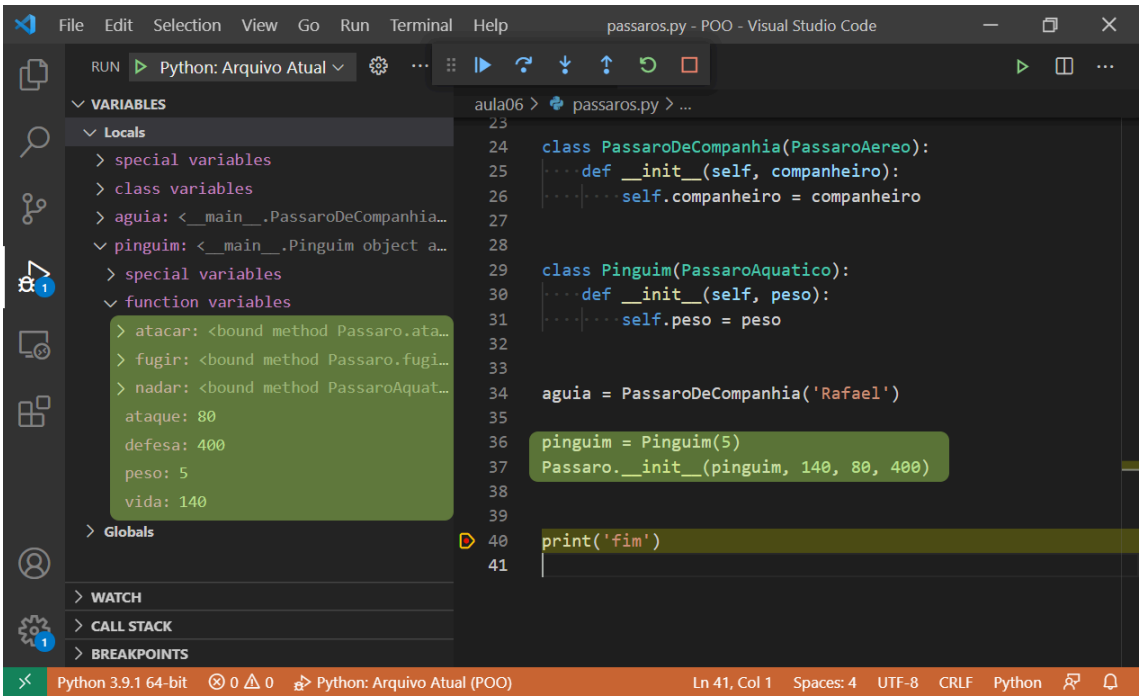
Esse mecanismo é chamado de Ordem de Resolução dos Métodos, e a explicação detalhada de como ele funciona é dada no item 6.3.3 deste capítulo.

Para resolver o problema precisamos chamar manualmente o método inicializador da classe `Passaro`, e isso poderia ser feito usando diretamente o nome da classe e passando para ela os argumentos necessários, incluindo a referência para o objeto que deve ser alterado. Por exemplo, após criar o pinguim, podemos fazer:

```
Passaro.__init__(pinguim, 140, 80, 400)
```

E isso irá executar o método inicializador de `Passaro` com a referência para o objeto `pinguim`. Observe que não estamos invocando o método de um objeto, mas sim diretamente da classe, portanto o Python não fará a injeção automática do `self` e por isso devemos passar o objeto que queremos alterar. Observe o resultado na Figura 6.7.

Figura 6.7: Visualização da criação manual dos atributos “herdados” de *Passaro*



Fonte: do autor, 2021

Com a experiência que você já tem até aqui, o código acima deveria tocar um alarme de “acho que estou fazendo isso errado”, pois imagine a confusão que seria se para cada objeto criado precisássemos adicionar atributos manualmente, em um sistema com dezenas ou até centenas de classes. A chance de introdução de bugs no código dessa forma é altíssima e algo que queremos minimizar.

Portanto, a solução é usar a função integrada `super` para colocar essa chamada que fizemos manualmente no interior da classe, de modo que o Python ficará responsável por buscar o método inicializador das classes mãe e executá-los conforme nossas instruções. De acordo com a documentação da função `super` (PSF, 2021a), podemos usá-la em qualquer parte do nosso código, mas aqui estamos interessados no seu funcionamento quando usada no interior da definição de uma classe.

Quando isso acontece, podemos chamar a função `super` sem passar nenhum argumento e ela irá nos retornar um objeto que automaticamente saberá a ordem em que precisa buscar um determinado método ou atributo, seguindo o conceito de MRO que veremos no item 6.3.3 deste capítulo.

Edite as classes de `Pinguim` e `PassaroDeCompanhia` para corresponder a:

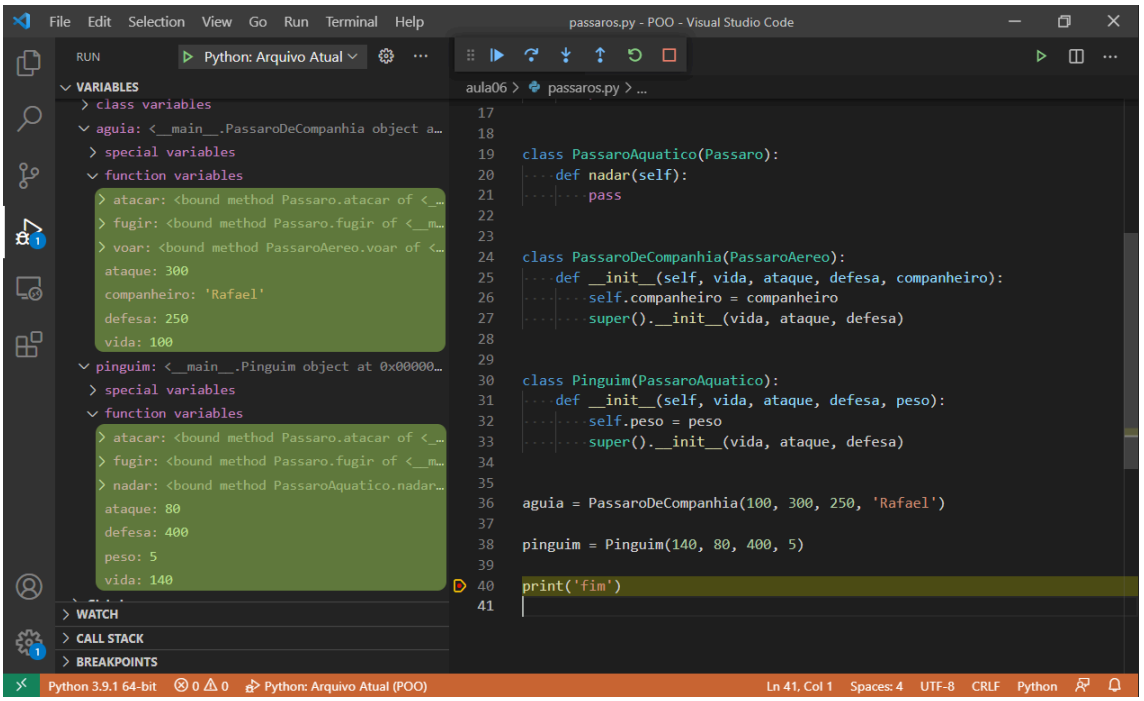
```
class PassaroDeCompanhia(PassaroAereo):
    def __init__(self, vida, ataque, defesa, companheiro):
        self.companheiro = companheiro
        super().__init__(vida, ataque, defesa)

class Pinguim(PassaroAquatico):
    def __init__(self, vida, ataque, defesa, peso):
        self.peso = peso
        super().__init__(vida, ataque, defesa)
```

E em seguida, edite a criação dos objetos para:

```
aguia = PassaroDeCompanhia(100, 300, 250, 'Rafael')
pinguim = Pinguim(140, 80, 400, 5)
```

A Figura 6.8 mostra o resultado após a execução deste código.

Figura 6.8: Visualização da criação dos atributos herdados de *Passaro* com o uso do *super*



Fonte: do autor, 2021

Nesta situação estamos atribuindo na classe mais específica os atributos que são pertinentes à ela, e em seguida, delegamos ao Python o trabalho de procurar nas classes que estão acima na hierarquia de heranças o inicializador que deverá ser executado para atribuição dos demais atributos.

Podemos efetivamente criar uma cadeia, em que cada vez que precisarmos adicionar um atributo específico no inicializador, sobrescrevemos o método `__init__`

com os todos os parâmetros da classe mãe, seguidos pelos parâmetros específicos. Atribuímos os parâmetros específicos localmente e chamamos a função `super` para lidar com os demais parâmetros. Um exemplo desta cadeia é dado no item 6.3.3.

6.2.2. Atributos e métodos "protegidos" em Python

Quando estamos trabalhando com herança em POO, podemos criar atributos que não sejam públicos nem privados, isto é, que estejam "escondidos" do mundo exterior mas acessíveis não só à própria classe, mas também a seus descendentes. São os atributos ditos protegidos.

Mas como já vimos, o Python não força as regras de acessibilidade. Na realidade, a documentação da PEP8 diz explicitamente que em Python não se usam os termos "privado" e "protegido", apenas havendo a diferenciação entre os atributos[1] públicos dos não-públicos (PSF, 2021b). Sendo assim, ela recomenda a adoção dos seguintes critérios para criação dos atributos de uma classe (incluindo os métodos):

- Sempre decida com antecedência quais atributos de uma classe serão públicos e não-públicos, na dúvida, escolha não-públicos. É mais fácil tornar público um atributo interno (não-público), do que internalizar um atributo público. Na primeira situação só precisamos mexer no código da própria classe, já na segunda, precisamos editar também todo código que seja cliente daquela classe e que usavam tal atributo público, e isso pode ser uma tarefa que, além de complicada, introduza comportamentos inesperados na aplicação
- Atributos públicos não devem ser precedidos por nenhum sublinhado;
- Se o nome de um atributo público colide com o nome de uma palavra chave e usar outro nome não é desejável (por exemplo, por piorar a legibilidade do código), deve-se adicionar um sublinhado ao final. Por exemplo: `for_`. A única exceção é que caso esse nome seja usado para referenciar uma classe, a convenção é usar o nome `cls`.
- Para atributos de dados (o que chamamos até agora apenas de atributos) públicos simples, recomenda-se expor diretamente o atributo, e se for necessária a utilização de alguma lógica de validação em seu acesso, deve-se usar o decorador que vimos ao estudar encapsulamento: a *property*.
- Para atributos não-públicos, recomenda-se preceder-los de um único sublinhado.
- Se a classe foi projetada para ser extendida, e há atributos que você não quer que sejam editados pelas classes filhas, nomeie-os precedidos com dois sublinhados, já que isso irá invocar a "desfiguração de nomes" do Python.

[1] O termo atributo aqui faz referência tanto às características quanto aos comportamentos que uma classe define, pois para o Python a única diferença entre eles é que um método é um atributo "chamável", ou seja, refere-se a um objeto que pode ser chamado, como fazemos com funções.

Por fim, o mais importante é lembrar que, de acordo com com a PEP8, um guia de estilo deve prezar pela consistência, em especial a consistência interna de um módulo e do projeto no qual está inserido. Então, ao entender o funcionamento da linguagem, podemos tomar uma decisão consciente por uma ou outra abordagem.

6.3. Polimorfismo

Como vimos na introdução dos pilares de POO, há dois tipos de polimorfismo, o de sobrecarga e o de sobrescrita.

Na sobrecarga, temos a mesma função ou método executando uma ação diferente em função da assinatura da função, isto é, se passamos um número diferente de argumentos e/ou argumentos de tipos diferentes, o comportamento muda.

Já a sobrescrita ocorre quando objetos diferentes possuem implementações diferentes de um mesmo método, como citado no exemplo de um pássaro mecânico vs. um pássaro normal em relação ao método `voar`.

6.3.1. Sobrecarga

Em Python, devido a sua natureza de tipagem dinâmica, não é possível fazer a sobrecarga de um método ou função da maneira tradicional, como é feita em linguagens de tipagem estática como Java, C# ou C++. Isso ocorre porque em Python, o interpretador só saberá o tipo de uma variável ou parâmetro em tempo de execução, então a única forma de diferenciarmos a assinatura de um método é pela quantidade de argumentos que ele recebe.

Com isso, podemos implementar uma variação do polimorfismo de sobrecarga ao utilizarmos valores padrões para alguns dos parâmetros do método ou função, tornando tais parâmetros opcionais e permitindo assim que o método seja chamado com diferentes assinaturas.

Para exemplificar, podemos usar a função que foi provavelmente a primeira a aprendermos em Python, o `print`. Como talvez você já saiba, o `print` aceita alguns parâmetros nomeados[2] como `end` e `sep`, que alteram, respectivamente, o caractere adicionado ao final da *string* e o caractere usado na união dos parâmetros posicionais passados previamente para formar a *string*, antes de exibi-la na tela.

Temos dessa forma, efetivamente comportamentos diferentes para uma mesma função quando chamada com assinaturas diferentes. O comportamento padrão é incluir um espaço entre os valores posicionais passados e finalizar a *string* com uma quebra de linha, denotada pelo caractere `'\n'`.

[2] Para o polimorfismo, não é necessário que os parâmetros sejam nomeados, apenas que sejam opcionais. No caso do *print*, eles precisam ser nomeados pois o *print* pode receber um número variável de argumentos, então essa é a única forma de passar um argumento cujo objetivo não seja ser exibido na tela.

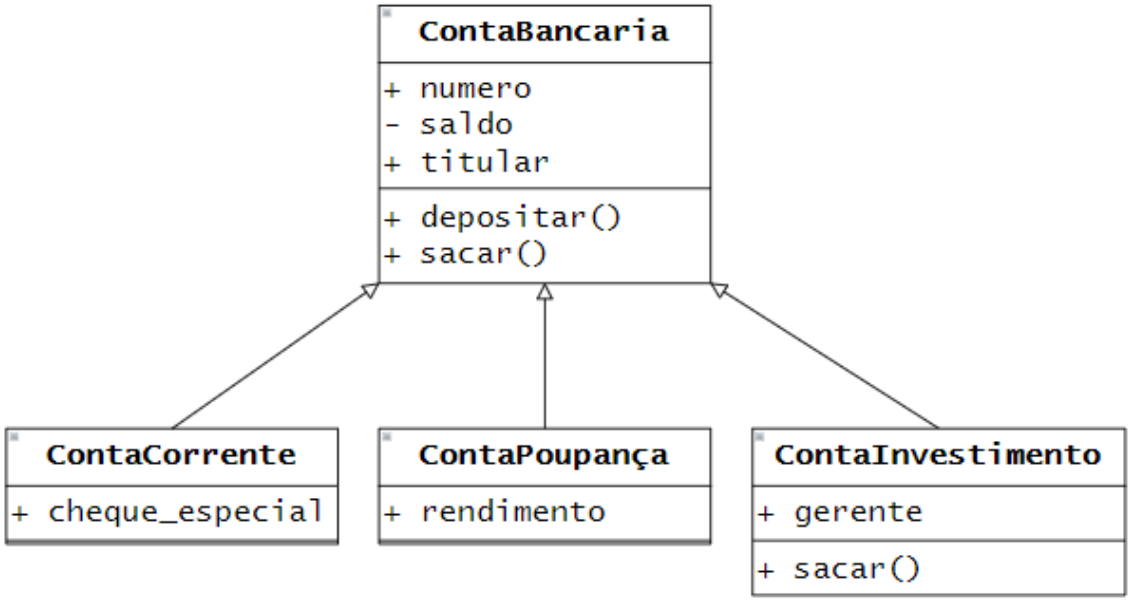
Faça o teste executando o código a seguir e observe o resultado.

```
print('chamada original')
print(1, 2, 3)
print()
print('alterando o sep')
print(1, 2, 3, sep='...')
print()
print('alterando o end')
print('prints separados', end=' ')
print('mas impressos', end=' ')
print('em uma mesma linha')
```

6.3.2. Sobrescrita

O polimorfismo por sobrescrita ocorre quando sobrescrevemos na classe filha um método herdado da classe mãe. Vamos retomar o exemplo da aula de introdução a POO, na qual fizemos classes para representar contas bancárias, mostrado na Figura 6.9.

Figura 4.7: Representação das classes de contas bancárias com polimorfismo do método sacar



Fonte: do autor, 2021

Vamos implementar um exemplo ilustrativo destas classes para visualizar melhor como funciona o polimorfismo de sobrescrita em Python. Crie um arquivo "contas_bancarias.py" na pasta "aula06" e adicione os trechos de código a seguir.

Na classe mãe, implementamos a inicialização do objeto e os métodos para realizar depósitos e saques, que inclui a lógica para verificar o saldo e realizar o saque:

```
class ContaBancaria:
    def __init__(self, numero, titular):
        self.numero = numero
        self.titular = titular
        self.__saldo = 0

    def depositar(self, valor):
        self.__saldo += valor
        print(f'Deposito realizado. Saldo: R$ {self.__saldo}')

    def sacar(self, valor):
        if valor > self.__saldo:
            print(f'Saque falhou. Saldo: R$ {self.__saldo}')
            return 'Saldo insuficiente.'
        self.__saldo -= valor
        print(f'Saque realizado. Saldo: R$ {self.__saldo}')
        return valor
```

Ao criar a classe da conta poupança, precisamos apenas sobrescrever o inicializador do objeto, para incluir o atributo específico referente ao rendimento da poupança, que neste exemplo será fixo, e chamamos o super para continuar com a inicialização do objeto:

```
class ContaPoupanca(ContaBancaria):
    def __init__(self, numero, titular):
        self.rendimento = 0.5
        super().__init__(numero, titular)
```

Já na classe da conta investimento, reescrevemos o inicializador e também o método de sacar, para incluir a lógica específica relacionada a este tipo de conta:

```
class ContaInvestimento(ContaBancaria):
    def __init__(self, numero, titular, gerente):
        self.gerente = gerente
        super().__init__(numero, titular)

    def sacar(self, valor):
        print('verificando prazo do investimento...')
        print('calculando impostos e taxas...')
        print('realizando saque...')
        return super().sacar(valor)
```

Observe que usamos o super também no método de sacar, para delegar à classe mãe a realização de fato do saque. Como criamos o atributo saldo com dois sublinhados, indicando que não devemos alterá-lo fora da classe que o criou, isso é necessário para não violarmos a não-publicidade do atributo. Vale lembrar aqui que em uma situação real, as instruções de print do exemplo seriam substituídas pela lógica que faria as verificações de fato.

Agora crie um outro arquivo "test_contas.py" na mesma pasta, com o seguinte código para testar as classes:

```
from contas_bancarias import ContaPoupanca, ContaInvestimento

# Criação das contas
conta_poupanca = ContaPoupanca('001', 'Rafael')
conta_investimento = ContaInvestimento('001', 'Rafael', 'Ana')

print('\n----Operações na conta poupança----')
conta_poupanca.depositar(1000)
saque_1 = conta_poupanca.sacar(100)
saque_2 = conta_poupanca.sacar(3000)

print(f'Primeiro saque da poupança: R$ {saque_1}')
print(f'Segundo saque da poupança: R$ {saque_2}')

print('\n----Operações na conta investimento----')
conta_investimento.depositar(500)
saque_3 = conta_investimento.sacar(300)
saque_4 = conta_investimento.sacar(300)

print(f'Primeiro saque da conta investimento: R$ {saque_3}')
print(f'Segundo saque da conta investimento: R$ {saque_4}')
```

6.3.3. Ordem de resolução dos métodos

Quando acessamos um método ou atributo de um objeto usando a notação de ponto: `objeto.atributo` ou `objeto.método()`, o Python irá buscá-lo na classe atual e, caso não encontre, ele sobe um nível na hierarquia e busca novamente. Esse processo é repetido até chegar ao fim da linha, que é a classe `object`, da qual todas as outras classes herdam automaticamente em Python. Se então o atributo ou método não for encontrado, o Python levanta um erro de atributo (`AttributeError`).

Para visualizar a ordem das classes em que o Python irá buscar pelos métodos e atributos usando o método `mro`[3] a partir da classe que queremos investigar.

No exemplo dos pássaros, temos:

```
>>> Pinguim.mro()
[<class '__main__.Pinguim'>, <class '__main__.PassaroAquatico'>,
<class '__main__.Passaro'>, <class 'object'>]
```

Podemos ver que ao buscar um método ou atributo, o Python irá percorrer toda a hierarquia que definimos no começo do capítulo, até chegar em `object`, parando a busca na primeira ocorrência encontrada.

Bibliografia

MARTIN, R. C. **Design principles and design patterns**. 2000. Disponível em: <https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>[4]. Acesso em: 15 fev. 2021.

PSF. **Built-in functions:** super. 2021a. Disponível em: <https://docs.python.org/3/library/functions.html#super>. Acesso em: 15 fev. 2021.

PSF. **Style guide for python code:** designing for inheritance. 2021b. Disponível em: <https://www.python.org/dev/peps/pep-0008/#designing-for-inheritance>. Acesso em: 16 fev. 2021.

[3] MRO é o acrônimo para Ordem de Resolução dos Métodos, na sigla em inglês: *Method Resolution Order*.

[4] O site original do autor encontra-se fora do ar e o link direciona a outro site, portanto foi usado aqui o link da página Wayback Machine (archive.org).