PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON

material produzido pelo prof. Guilherme Derenievicz

adaptado pelo prof. Jônata Tyska Carvalho

Departamento de Informática e Estatística - UFSC

---

4 - Polimorfismo e Composição

Polimorfismo

Polimorfismo deriva dos termos *muitas* + *formas* e está fortemente relacionado com os conceitos de herança, sobreposição e classes abstratas. Informalmente, significa *multiplas formas do mesmo "objeto"*. Aqui, "objeto", entre aspas, refere-se a um objeto do programa que representa, ao mesmo tempo, diferentes tipos de instâncias de classes.

Por exemplo, considere o código para o problema das áreas de formas geométricas do **Notebook 3-Herança e Classes Abstratas**, replicado abaixo:

In [1]:
```
from abc import ABC, abstractmethod
import math

class FormaGeometrica(ABC):
    def __init__(self):
        self.__area = 0

    @property
    def area(self):
        return self.__area

    @area.setter
    def area(self, area: float):
        self.__area = area

    @abstractmethod
    def calcula_area(self):
        pass
```

```
class Quadrado(FormaGeometrica):
    def __init__(self, lado: float):
        super().__init__()
        self.__lado = lado

    def calcula_area(self):
        self.area = self.__lado * self.__lado


class Retangulo(FormaGeometrica):
    def __init__(self, base: float, altura: float):
        super().__init__()
        self.__base = base
        self.__altura = altura

    def calcula_area(self):
        self.area = self.__base * self.__altura


class Circulo(FormaGeometrica):
    def __init__(self, raio: float):
        super().__init__()
        self.__raio = raio

    def calcula_area(self):
        self.area = math.pi * self.__raio**2


figuras = [Quadrado(2), Retangulo(2,3), Circulo(2), Circulo(3), Quadrado(3), Qua

for fig in figuras:
    if isinstance(fig,FormaGeometrica):
        fig.calcula_area()
        print(fig.area)
```

```
4
6
12.566370614359172
28.274333882308138
9
4
3
```

Na função `main`, o polimorfismo é aplicado nas seguintes linhas:

`fig.calcula_area()`
e

`print(fig.area)`
Neste ponto, não nos importamos em saber se `fig` é um objeto da classe `Quadrado`, `Retangulo` ou `Circulo`, pois sabemos que, independentemente, ele possui o método `calcula_area()` e o atributo `area` e podemos acessá-los. Isto é, `fig` representa "muitas formas" de figuras geométricas.

Outra maneira de pensar em polimorfismo é como *muitas formas de um mesmo método*; nesse caso, o método `calcula_area()` possui muitas formas (uma para cada figura geométrica) e acessamos esse método de maneira genérica.

O uso de polimorfismo fica muito mais elegante, seguro e legível se acompanhado de classes abstratas (consequentemente, herança e sobreposição). Vejamos novamente o exemplo dos veículos, em um programa que instancia diversos carros e bicicletas.

In [5]:
```
class Pessoa:
    def __init__(self, nome: str, cpf: str, dinheiro: float):
        self.__nome = nome
        self.__cpf = cpf
        self.__dinheiro = dinheiro

    @property
    def nome(self):
        return self.__nome

    @property
    def cpf(self):
        return self.__cpf

    @property
    def dinheiro(self):
        return self.__dinheiro

    @dinheiro.setter
    def dinheiro(self, valor: float):
        self.__dinheiro = valor


class Veiculo(ABC):
    def __init__(self, modelo: str, cor: str, dono: Pessoa):
        self.__modelo = modelo
        self.__cor = cor
        self.__dono = dono
        self.__velocidade = 0
        self.__sujo = True

    @property
    def modelo(self):
        return self.__modelo

    @property
    def cor(self):
        return self.__cor

    @cor.setter
    def cor(self, nova_cor: str):
        self.__cor = nova_cor

    @property
    def dono(self):
        return self.__dono

    @dono.setter
    def dono(self, novo_dono: Pessoa):
        self.__dono = novo_dono

    @property
    def velocidade(self):
        return self.__velocidade
```

```
    @velocidade.setter
    def velocidade(self, velocidade: int):
        self.__velocidade = velocidade

    @property
    def sujo(self):
        return self.__sujo

    @sujo.setter
    def sujo(self, sujo: bool):
        self.__sujo = sujo

    @abstractmethod
    def lavar(self):
        pass

    @abstractmethod
    def frear(self):
        pass


class Carro(Veiculo):
    def __init__(self, modelo: str, cor: str, placa: str, dono: Pessoa):
        super().__init__(modelo, cor, dono)
        self.__placa = placa

    @property
    def placa(self):
        return self.__placa

    @placa.setter
    def placa(self, nova_placa: str):
        self.__placa = nova_placa

    def buzinar(self):
        print(self.__modelo, 'buzinou!')

    def acelerar(self, valor: int):
        self.__velocidade += valor

    def lavar(self):
        total = self.dono.dinheiro
        if total >= 30:
            self.dono.dinheiro = total - 30
            self.sujo = False

    def frear(self):
        self.velocidade -= 10
        if self.velocidade < 0:
            self.velocidade = 0

class Bicicleta(Veiculo):
    def __init__(self, modelo: str, cor: str, marchas: int, amortecedor: bool, c
        super().__init__(modelo, cor, dono)
        self.__marchas = marchas
        self.__amortecedor = amortecedor

    @property
    def marchas(self):
        return self.__marchas
```

```
    @marchas.setter
    def marchas(self, novas_marchas: int):
        self.__marchas = novas_marchas

    @property
    def amortecedor(self):
        return self.__amortecedor

    @amortecedor.setter
    def amortecedor(self, novo_amortecedor: bool):
        self.__amortecedor = novo_amortecedor

    def pedalar(self, valor: int):
        self.__velocidade += valor

    def lavar(self):
        total = self.dono.dinheiro
        if total >= 10:
            self.dono.dinheiro = total - 10
            self.sujo = False

    def frear(self):
        self.velocidade -= 1
        if self.velocidade < 0:
            self.velocidade = 0


joao = Pessoa('Joao Silva', '123.456.789-0', 500)
carro1 = Carro('Gol', 'Vermelho', 'ABC-1234', joao)
carro2 = Carro('Fox', 'Prata', 'ABC-9999', joao)
carro3 = Carro('Palio', 'Preto', 'ABC-0000', joao)
bicicleta1 = Bicicleta('Caloi 10', 'Vermelha', 12, False, joao)
bicicleta2 = Bicicleta('Caloi 10', 'Amarela', 12, False, joao)

lista = [carro1, carro2, carro3, bicicleta1, bicicleta2]

print('Lista de veículos que estão sendo lavados e pintados de branco:')
for veiculo in lista:
    print(veiculo.modelo, veiculo.cor, '( Sujo =', veiculo.sujo, '); Dinheiro do
    veiculo.lavar()
    veiculo.cor = 'Branco'

print('Lista de veículos reformados:')
for veiculo in lista:
    print(veiculo.modelo, veiculo.cor, '( Sujo =', veiculo.sujo, '); Dinheiro do
```

```
Lista de veículos que estão sendo lavados e pintados de branco:
Gol Vermelho ( Sujo = True ); Dinheiro do dono Joao Silva = R$ 500
Fox Prata ( Sujo = True ); Dinheiro do dono Joao Silva = R$ 470
Palio Preto ( Sujo = True ); Dinheiro do dono Joao Silva = R$ 440
Caloi 10 Vermelha ( Sujo = True ); Dinheiro do dono Joao Silva = R$ 410
Caloi 10 Amarela ( Sujo = True ); Dinheiro do dono Joao Silva = R$ 400
Lista de veículos reformados:
Gol Branco ( Sujo = False ); Dinheiro do dono Joao Silva = R$ 390
Fox Branco ( Sujo = False ); Dinheiro do dono Joao Silva = R$ 390
Palio Branco ( Sujo = False ); Dinheiro do dono Joao Silva = R$ 390
Caloi 10 Branco ( Sujo = False ); Dinheiro do dono Joao Silva = R$ 390
Caloi 10 Branco ( Sujo = False ); Dinheiro do dono Joao Silva = R$ 390
```

Aqui a classe `Veiculo` foi transformada em classe abstrata, pois ela nunca é instanciada. Do mesmo modo os métodos `lavar()` e `frear()` desta classe são também abstratos. Isto indica que qualquer objeto de suas subclasses **obrigatoriamente** deve definir estes métodos.

Na função `main`, carros e bicicletas são inseridos em uma lista e tratados sem distinção, como sendo apenas veículos. Isso funciona corretamente, pois os atributos e métodos utilizados dentro do `for`, que são `modelo`, `cor`, `sujo`, `dono` e `lavar()`, são da *__superclasse__ `Veiculo`, comuns a ambos os tipos de objetos.

Note como o uso da classe abstrata `Veiculo` e dos métodos abstratos `frear()` e `lavar()` desta classe facilitam o entendimento da função `main()`. Se não houvessem essas definições, precisaríamos verificar uma por uma das subclasses!

Exemplos de relação entre classes: Composição e Agregação

Em um sistema desenvolvido usando o paradigma de Orientação a Objetos, além da definição das classes e seus comportamentos individuais, a forma como elas se relacionam é parte fundamental do funcionamento do sistema. As classes de um sistema normalmente estão associadas a outras classes, por exemplo através de um atributo que referencia um objeto de outra classe. O tipo de relação entre as classes, normalmente relacionada à semântica de cada uma, define a forma de associação. Por exemplo, a classe `Carro` possui o atributo `dono` que é uma referência a um objeto da classe `Pessoa`.

Na prática, essas associações podem ser classificadas em duas formas:

- **Composição:** *a parte não existe sem o objeto*. Por exemplo, todo carro tem um motor, e esse motor só existe dentro de um único carro. Em geral, podemos pensar que um motor não pode existir sem que esteja dentro de um carro. A forma mais usual de tratar composição é instanciar o objeto "parte" diretamente **dentro** da classe que a contém, garantindo que sua existência dependa da existência do objeto "todo".

- **Agregação:** *a parte pode ser compartilhada com outros objetos*. Esse é o caso do nosso exemplo `Pessoa`, pois o atributo `dono` da classe `Carro` faz referência a um objeto do tipo `Pessoa` que pode estar associado a vários objetos (a mesma pessoa pode ser dona de vários carros). Além disso, mesmo que um carro seja destruído, a pessoa (seu dono) continua existindo e continua sendo dona de outros carros. Assim, o objeto da classe `Pessoa` deve ser instanciado **fora** da classe `Carro`.

O exemplo abaixo mostra essas duas formas de associação. As classes `Veiculo` e `Bicicleta` foram retiradas para simplificar o exemplo. Note a adição da classe

`Motor`, do atributo `motor` na classe `Carro` e do parâmetro `potencia` no construtor da classe `Carro`.

In [3]:
```
class Pessoa:
    def __init__(self, nome: str, cpf: str, dinheiro: float):
        self.__nome = nome
        self.__cpf = cpf
        self.__dinheiro = dinheiro

    @property
    def nome(self):
        return self.__nome

    @property
    def cpf(self):
        return self.__cpf

    @property
    def dinheiro(self):
        return self.__dinheiro

    @dinheiro.setter
    def dinheiro(self, valor: float):
        self.__dinheiro = valor


class Motor:
    def __init__(self, potencia: float):
        self.__potencia = potencia

    @property
    def potencia(self):
        return self.__potencia


class Carro:
    def __init__(self, modelo: str, cor: str, placa: str, dono: Pessoa, potencia
        self.__modelo = modelo
        self.__cor = cor
        self.__placa = placa
        self.__dono = dono              # associação por agregação: dono é recebi
        self.__motor = Motor(potencia) # associação por composição: motor é inst
        self.__velocidade = 0
        self.__sujo = True

    @property
    def modelo(self):
        return self.__modelo

    @property
    def cor(self):
        return self.__cor

    @cor.setter
    def cor(self, nova_cor: str):
        self.__cor = nova_cor

    @property
    def placa(self):
```

```
        return self.__placa

    @placa.setter
    def placa(self, nova_placa: str):
        self.__placa = nova_placa

    @property
    def dono(self):
        return self.__dono

    @dono.setter
    def dono(self, novo_dono: Pessoa):
        self.__dono = novo_dono

    @property
    def motor(self):
        return self.__motor

    @property
    def velocidade(self):
        return self.__velocidade

    @property
    def sujo(self):
        return self.__sujo

    def buzinar(self):
        print(self.__modelo, 'buzinou!')

    def acelerar(self, valor: int):
        self.__velocidade += int

    def lavar(self):
        total = self.dono.dinheiro
        if total >= 30:
            self.dono.dinheiro = total - 30
            self.__sujo = False

    def frear(self):
        self.__velocidade -= 10
        if self.__velocidade < 0:
            self.__velocidade = 0


joao = Pessoa('Joao Silva', '123.456.789-0', 500)
carro1 = Carro('Gol', 'Vermelho', 'ABC-1234', joao, 1000)
carro2 = Carro('Fox', 'Prata', 'ABC-9999', joao, 1500)

# Aqui verificamos que os objetos do tipo Motor foram de fato instanciados para
print('Potência do carro 1:', carro1.motor.potencia)
print('Potência do carro 2:', carro2.motor.potencia)
print('Dono do carro 1:', carro1.dono.nome)
print('Dono do carro 2:', carro2.dono.nome)

# Ao deletar o objeto carro1, deletamos também o objeto Motor associado a ele (c
# Mas o objeto Pessoa que é dono do carro1 ainda existe (agregação), e pode ser
del carro1
print('Dono do carro 2:', carro2.dono.nome)
```

```
Potência do carro 1: 1000
Potência do carro 2: 1500
Dono do carro 1: Joao Silva
Dono do carro 2: Joao Silva
Dono do carro 2: Joao Silva
```

Extras (*magic methods*)

O método `__str__()`

O que acontece ao imprimir um objeto com o comando `print()`? Uma mensagem mais amigável (e útil) pode ser obtida ao se definir o método especial `__str__()`, que deve retornar uma string com o conteúdo a ser impresso na tela. De modo mais específico, este método diz como um objeto do tipo `str` deve ser instanciado a partir deste objeto, e isso pode ser utilizado não apenas para imprimir o objeto na tela via comando `print()`, mas também para convertê-lo em string utilizando `str()`.

In [4]:
```
class Pessoa:
    def __init__(self, nome: str, cpf: str, dinheiro: float):
        self.__nome = nome
        self.__cpf = cpf
        self.__dinheiro = dinheiro

    @property
    def nome(self):
        return self.__nome

    @property
    def cpf(self):
        return self.__cpf

    @property
    def dinheiro(self):
        return self.__dinheiro

    @dinheiro.setter
    def dinheiro(self, valor: float):
        self.__dinheiro = valor

    def __str__(self):
        return 'Nome: ' + self.__nome + ', CPF: ' + self.__cpf + ' (R$ ' + str(s



joao = Pessoa('Joao Silva', '123.456.789-0', 500)
print(joao)

s = str(joao)
print(s)
```

```
Nome: Joao Silva, CPF: 123.456.789-0 (R$ 500)
Nome: Joao Silva, CPF: 123.456.789-0 (R$ 500)
```

Herança Múltipla e o Problema do Diamante

No código abaixo, a classe `Bilingue` herda de duas classes: `Ingles` e `Portugues` (**herança múltipla**). Como `pessoa` é uma instância de `Bilingue`, ao chamarmos o método `pessoa.cumprimento()`, o que será impresso na tela? A classe `Bilingue` não implementa este método, mas ele está definido nas duas superclasses. Considerando a superclasse `Ingles`, o programa irá mostrar na tela "Hi!". Mas considerando a superclasse `Portugues`, o programa irá mostrar na tela "Oi!". Essa ambiguidade é o chamado **Problema do Diamante**. Veja na execução abaixo como Python lida com este problema.

In [3]:
```
from abc import ABC, abstractmethod

class Idioma(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def cumprimento(self):
        pass


class Ingles(Idioma):
    def __init__(self):
        super().__init__()

    def cumprimento(self):
        print('Hi!')

    def contar_final(self):
        print('Spoiler')


class Portugues(Idioma):
    def __init__(self):
        super().__init__()

    def cumprimento(self):
        print('Oi!')

    def dois_dias_atras(self):
        print('Anteontem')


class Bilingue(Portugues, Ingles):
    def __init__(self):
        super().__init__()

    def cantar(self):
        print("'Money, que é good nóis não have'")

pessoa = Bilingue()
```

```
pessoa.contar_final()
pessoa.dois_dias_atras()
pessoa.cantar()

pessoa.cumprimento()
```

```
Spoiler
Anteontem
'Money, que é good nóis não have'
Oi!
```

Troque a definição `class Biligue(Ingles, Portugues)` para `class Biligue(Portugues, Ingles)` e veja o que acontece!

Convenção de Nomenclatura

Ao escrever um programa, é adequado que se estabeleça certas convenções de nomenclatura e estilos, a fim de tornar o código legível. Cada linguagem de programação possui certas convenções estabelecidades pela comunidade ao longo dos anos.

A convenção de nomenclatura de Python difere um pouco de outras linguagens OO, como Java. Segue abaixo alguns exemplos:

Nomenclatura

- **Classes:** primeira letra de cada palavra maiúscula. Exs: `Carro`, `FormaGeometrica`, `FuncionarioDeUmBanco`;
- **Funções e métodos:** letras minúsculas, palavras separadas por `_`. Exs: `lavar()`, `calcular_area()`, `get_esta_sujo()`;
- **Variáveis, atributos e parâmetros:** mesma convenção das funções. Exs: `velocidade`, `esta_sujo`, `qtd_de_marchas`.

Estilos

- **Identação:** quatro espaços por nível. Não misture espaços com tabulações;
- **Separação:** utilize uma linha em branco para separar as definições de métodos e duas linhas em branco para separar as definições de classes;
- **Linhas de código:** devem ter um máximo de 80 caracteres incluindo espaços, isto é, 80 colunas.

Mais detalhes podem ser consultados em https://wiki.python.org.br/GuiaDeEstilo e https://legacy.python.org/dev/peps/pep-0008/.

---

del, is None, is not None

- `del objeto` : deleta a **referência** para o objeto (se não houver outras referências ele também é deletado);
- `if objeto is not None` : verifica se essa referência é para um objeto instanciado.

Organização de Arquivos

Em Python, usualmente, cria-se um arquivo para cada classe. Por exemplo, um arquivo veiculo.py para a classe `Veiculo` e um arquivo pessoa.py para a classe `Pessoa`. Como a classe `Veiculo` instancia objetos da classe `Pessoa`, é necessário fazer uma importação:

```
from pessoa import Pessoa
```

Verificação de Tipos

Até aqui, utilizamos a notação `<nome_parâmetro>: <tipo_parâmetro>` em métodos e funções para indicar qual é o tipo de cada parâmetro de entrada. Essa indicação, no entanto, é meramente ilustrativa! Note como, no exemplo abaixo, o programa é executado normalmente mesmo com o parâmetro `valor` sendo passado ao método com o tipo errado.

In [2]:
```
class Teste:
    def __init__(self, valor: int):
        self.__valor = valor

    @property
    def valor(self):
        return self.__valor


def main():

    teste1 = Teste(5)
    teste2 = Teste('casa')

    print(teste1.valor)
    print(teste2.valor)

main()
```

```
5
casa
```

Uma verificação de tipos pode ser efetuada, em Python, utilizando a função `isinstance()`. O primeiro argumento é o objeto a ser verificado e o segundo argumento é a classe a qual esse objeto deve pertencer. Se o retorno for `True`, então o objeto no primeiro argumento é uma instância da classe no segundo argumento. Veja abaixo:

In [5]:
```
class Teste:
    def __init__(self, valor: int):
        if isinstance(valor, int):
            self.__valor = valor
        else:
            print('Erro ao inicializar o valor de um objeto. Atribuindo valor 0.
            self.__valor = 0

    @property
    def valor(self):
        return self.__valor


def main():

    teste1 = Teste(5)
    teste2 = Teste('casa')

    print(teste1.valor)
    print(teste2.valor)

main()
```

```
Erro ao inicializar o valor de um objeto. Atribuindo valor 0...
5
0
```

Conclusão

Nestes 4 notebooks vimos uma introdução aos conceitos básicos do paradigma de programação orientada a objetos, contemplando os 4 pilares da orientação a objetos: abstração, encapsulamento, herança e polimorfismo. Foram apresentadas as ferramentas básicas para se aplicar este conceitos utilizando a linguagem de programação Python. Vale ressaltar que este material é introdutório: orientação a objetos não é apenas um conjunto de funcionalidades que auxiliam no desenvolvimento de programas, mas sim um **paradigma de programação**, e como tal exige/possibilita que o programador desenvolva uma nova forma de pensar, de analisar o problema e de obter sua solução. A continuidade deste estudo, em ambas as dimensões, prática e teórica, é essencial para que o programador possa de fato tirar proveito dos benefícios deste paradigma.

Referências

Este material foi desenvolvido com base nas Notas de Aulas da disciplina *Desenvolvimento de Sistemas Orientados a Objetos I* do curso de Sistemas de Informação do INE-UFSC, de autoria do prof. Jean Carlo Rossa Hauck, e no material *Aulas de Introdução à Computação em Python*, do Detartamento de Ciência da Computação do IME-USP, disponível em https://panda.ime.usp.br/aulasPython/static/aulasPython/index.html