PYTHON

CONSTRUINDO PROJETO REAL APENAS ORIENTANDO OBJETOS COM PYTHON

MINICURSO SECIT 2022

OBJETIVOS

AO FINALIZAR ESTE CURSO, OS ESTUDANTES PODERÃO:

- IDENTIFICAR OS **CONCEITOS FUNDAMENTAIS** DO PARADIGMA DE PROGRAMAÇÃO ORIENTANDO A OBJETOS
- UTILIZAR APROPRIADAMENTE AS TÉNICAS DE **MODELAGEM** E **PROGRAMAÇÃO** ORIENTADA A OBJETOS
- DESENVOLVER **APLICAÇÕES** UTILIZANDO O PARADIGMA DO POO

FERRAMENTA

- PYTHON 3.X (VÁRIAS MUDANÇAS EM RELAÇÃO À PYTHON 2.X)
- IDES:
  - SUBLIME
  - ATOM
  - **VISUAL CODE**
  - PYCHARM
  - VIM
  - EMACS
- JUPYTER NOTEBOOKS

CLASSES, OBJETOS E ABSTRAÇÃO

OBJETIVOS

- DEFINIR UMA **CLASSE**
- ADICIONAR **ATRIBUTOS**, **MÉTODOS** E **CONSTRUTORES** A UMA CLASSE
- APRENDER NOVOS COMANDOS BÁSICOS DE PYTHON

CONSIDERE UM CARRO

1. QUAIS SÃO AS CARACTERÍSTICAS DE UM CARRO?
2. QUE COISAS UM CARRO PODE "FAZER"?
3. TODOS OS CARROS ALCANÇAM A MESMA VELOCIDADE?

PROGRAMAÇÃO ORIENTADA A OBJETOS

ENVOLVE IMPLEMENTAR PROGRAMAS QUE POSSUEM ***OBJETOS*** INTERAGINDO ENTRE SI:

- DIFERENTEMENTE DE PROGRAMAÇÃO ESTRUTURADA ONDE FUNÇÕES/COMANDOS INTERAGEM ENTRE SI

QUALQUER COISA PODE SER UM OBJETO:

- COISAS DO MUNDO REAL: CARRO, CASA, AVIÃO...
- GRANDEZAZ MATEMÁTICAS (VETOR, MATRIZ...)
- ABSTRAÇÕES NO CONTEXTO DE UM PROBLEMA (GERENCIADOR DE BANCO DE DADOS, POR EXEMPLO)

CLASSES

- ABSTRAÇÃO PARA AGRUPAR OBJETOS QUE TÊM UM MESMO COMPORTAMENTO COMUM
- DESCREVEM DE MANEIRA ABSTRATA O COMPORTAMENTO DOS OBJETOS
- É UMA ESPECIFICAÇÃO PARA OBJETOS DAQUELA CLASSE, SIMILAR A UMA PLANTA BAIXA

CLASSES

A PALAVRA CLASSE VEM DA TAXONOMIA DA BIOLOGIA. TODOS OS SERES VIVOS DE UMA MESMA CLASSE BIOLÓGICA TÊM UMA SÉRIE DE ATRIBUTOS E COMPORTAMENTOS EM COMUM, MAS ELES NÃO SÃO IGUAIS, PODEM VARIAR NOS VALORES DESSES ATRIBUTOS E COMO REALIZAM ESSES COMPORTAMENTOS.

OBJETOS

- SÃO INSTÂNCIAS DE UMA CLASSE
- ENCAPSULAM UM ESTADO:
  - CONJUNTO DE VALORES QUE OS ATRIBUTOS POSSUEM
  - POR EXEMPLO, UMA INSTÂNCIA DA CLASSE CARRO PODE TER POTÊNCIA IGUAL A 90, MARCA RENAULT, ETC

OBJETOS

- POSSUEM UM COMPORTAMENTO DEFINIDO ATRAVÉS DOS SEUS MÉTODOS
  - TÊM OS SEUS MÉTODOS CHAMADOS PARA EXECUTAR UMA AÇÃO NO PROGRAMA
  - MÉTODOS SÃO FUNÇÕES CHAMADAS/EXECUTADAS POR UM OBJETO

ATRIBUTO E ESTADO

- O ESTADO DE UM OBJETO ESTÁ DEFINIDO PELOS VALORES DOS ATRIBUTOS DE CLASSE

- OS TIPOS DOS ATRIBUTOS PODEM SER:
  - TIPOS PRIMITIVOS: INT, FLOAT, DOUBLE, ETC.
  - TIPOS DEFINIDOS PELO USUÁRIO (OUTRAS CLASSES)

ATRIBUTO E ESTADO

O ESTADO ESTÁ DEFINIDO PELOS VALORES DOS ATRIBUTOS


```
Class Carro:
  // Atributo
    - marca
    - potencia
```

PROGRAMAÇÃO ORIENTADA A OBJETOS

PROGRAMAR ORIENTADO A OBJETOS ENVOLVE:

- IDENTIFICAR CLASSES QUE AGRUPAM OBJETOS COM UM COMPORTAMENTO COMUM
- IDENTIFICAR OS ATRIBUTOS QUE CADA OBJETO DEVE ARMAZENAR
- IDENTIFICAR COMO OS OBJETOS DEVEM SE COMPORTAR
- IDENTIFICAR COMO OS OBJETOS DO SISTEMA DEVEM INTERAGIR ENTRE SI

PROGRAMAÇÃO ORIENTADA A OBJETOS

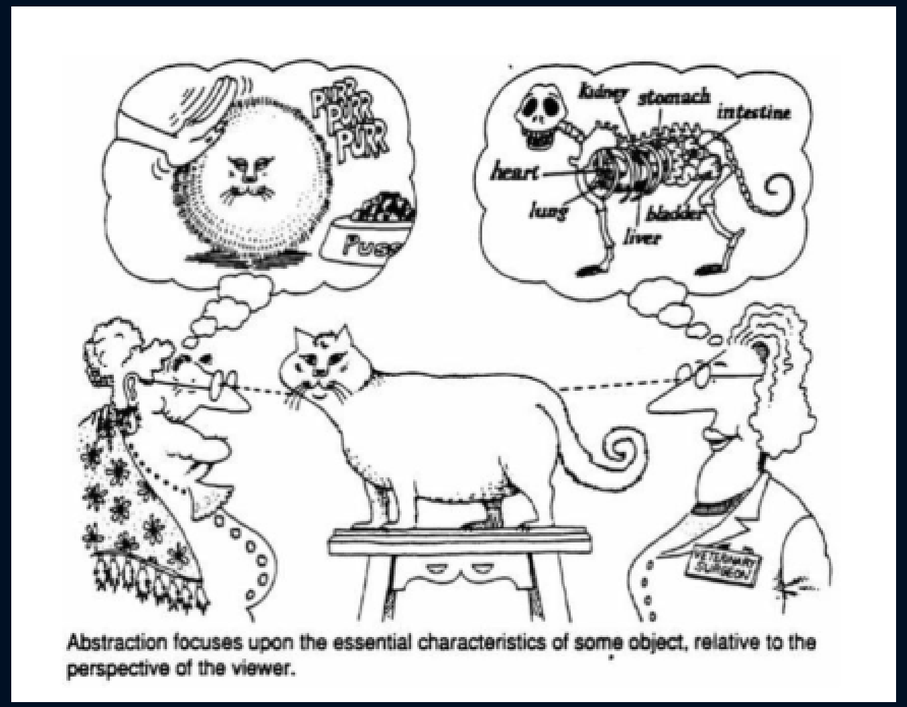
SE EU IDENTIFIQUEI QUE CLASSES (OBJETOS) O MEU PROGRAMA DEVE TER, COMO SABER QUE ATRIBUTOS E MÉTODOS ELAS DEVEM TER?

PRIMEIRO PILAR DE POO: ABSTRAÇÃO

ABSTRAÇÃO: ESCOLHER OS ASPETOS MAIS RELEVANTES PARA AS CLASSES/OBJETOS A SEREM IMPLEMENTADOS

- SIMPLICIDADE: ATRIBUTOS IRRELEVANTES DEVEM SER IGNORADOS
- DIVIDIR E CONQUISTAR: O PROPÓSITO DE UMA CLASSE DEVE ESTAR BEM DEFINIDO
    - IMPLEMENTE VÁRIAS CLASSES COM PROPÓSITOS DIFERENTES, UMA UTILIZANDO A OUTRA, SE FOR O CASO

Primeiro pilar de POO: Abstração



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Programação Orientada a Objetos

Encapsulamento

Revisão

Classe

Abstração para agrupar objetos comuns
Descreve de maneira abstrata o comportamento dos

Objetos

Instâncias de uma classe
Encapsulam um estado

- Atributos: características da entidade sendo modelada
- Métodos: definem o comportamento
- Inicializaodor: inicializa os atributos (construtor em outras linguagens)
- self: referência que um objeto tem dele mesmo

Objetivos

Nesta aula aprenderemos:

- ***Encapsulamento***: ***segundo pilar*** de POO
- Setters/getters e @property: primeiro decorador em Python

Encapsulamento

Capacidade de esconder informações (dados) nos objetos

- Alguns atributos/métodos são usados apenas como parte da lógica da implementação da classe
- Estes atributos/métodos devem ser impedidos de serem utilizados fora da classe
- As classes devem expôr o mínimo necessário para serem utilizadas
  - Similar a funções: os dados mínimos que elas precisam são os parâmetros e as suas variáveis locais não ficam acessíveis por quem as chama

Interface Pública de uma Classe

- Toda classe tem uma interface pública: conjunto de métodos que devem ser chamados para que objetos de uma classe sejam utilizados
- A interface pública da classe informa ao usuário da classe como ele deve utilizá-la
  - Usuário da classe é quem utiliza a classe e portanto é um programador

Usuário da classe não é o mesmo que o usuário do programa (que é uma pessoa que não necessariamente entende de programação)

Encapsulamento

- Os usuários da classe só podem acessar os atributos e métodos públicos da classe
- Ao chamar um método, o usuário da classe pode ignorar como ele foi implementado
  - Basta que ele saiba o que deve ser passado como parâmetro e qual o retorno do método para chamá-lo
- Facilita a manutenção e reaproveitamento do código

Modificadores de Acesso

Os modificadores de acesso valem para atributos e métodos:

- Público: o atributo/método pode ser acessado/chamado de dentro ou fora da classe
- Privado: o atributo/método só pode ser acessado/chamado de dentro da classe; ele não é herdado pelas subclasses
- Protegido: o atributo/método só pode ser acessado/chamado de dentro da classe; ele é herdado pelas subclasses

Modificadores de Acesso

Exemplo em Java

```java
public class Pessoa{
    // Atributos
    private String nome;
    private int idade;
    // Método público
    public String cumprimentar(Pessoa outro)
    {...
    }
}
```

Em Python, não é assim que funciona. Mais sobre isto no notebook da aula.

Exemplo: Estacionamento

- Um estacionamento tem capacidade para um número n > 0 de vagas.
- Devemos controlar quantos carros estão dentro do estacionamento.
- Os carros podem entrar só se há vagas disponíveis.

Exemplo: Estacionamento

Nesse sistema podemos identificar:

- Classes: Carros e o Estacionamento (por enquanto, vamos ignorar os carros)
- Atributos: número de vagas, capacidade máxima
- Métodos: Os carros podem entrar e sair. Além disso, podemos consultar o número de vagas disponíveis.

Exemplo: Estacionamento

Pergunta: o usuário da classe Estacionamento deveria modificar diretamente o atributo vagas?

Para responder, pense na consistência do objeto Estacionamento. Ela pode ser mantida dessa forma?

A resposta é Não!: o valor do atributo vagas não deve ser modificado diretamente.
Isto deve ser feito exclusivamente utilizando chamada aos método que informam que um carro entrou ou saiu do estacionamento. Estes métodos, por sua vez, são quem deve alterar o a quantidade de vagas do estacionamento.

Encapsulamento

Utilizado para:

- Esconder os atributos de uma classe
- Esconder como funcionam os métodos da classe
- Facilitar o reaproveitamento de código
- Garantir a consistência (do estado) dos objetos

04-Encapsulamento

Programação Orientada a Objetos

Biblioteca Padrão

Revisão

Encapsulamento

- Protege o acesso direto aos atributos de um objeto
- Esconde como funcionam as rotinas (métodos) da classe
- As classes devem expôr o mínimo necessário para serem utilizadas
- Ao chamar um método, podemos ignorar como ele foi implementado
- Os usuários da classe só devem acessar os atributos/métodos públicos da classe (interface pública)

Objetivos

Nesta aula, serão mostradas as seguintes classes da biblioteca padrão de Python:

- tuple (pares ordenados, tuplas, etc)
- list (listas de elementos)
- str (strings)
- dict (dicionários)

05-Biblioteca-Padrão

Programação Orientada a Objetos

Relações entre Classes

Revisão

Classe

- Abstração para agrupar objetos comuns
- Descreve de maneira abstrata o comportamento dos objetos

Objetos

- Instâncias de uma classe
- Encapsulam um estado

- Construtor/inicializador: inicializa (instancia) o objeto com valores iniciais para os atributos
- Atributos: características da entidade sendo modelada
- Métodos: definem o comportamento através de métodos que objetos da classe devem executar
- self: referência ao próprio objeto

Interfaces e Encapsulamento

Membros privados

- Métodos e atributos que não devem ser visíveis para os usuários da classe
- Utilizados apenas internamente, pela própria classe

Membros Públicos

- Métodos que podem ser chamados externamente
- Especificam um contrato: operações que o objeto pode realizar

Interfaces e Encapsulamento

- Alterações no estado de um objeto acontecem através da chamada de seus métodos
  - Atributos não devem ser modificados diretamente (a não ser via getters/setters)
- As mudanças na implementação da classe não devem afetar os usuários da classe

Programas do mundo real (mais complexos) são implementados por várias classes que se relacionam entre si

Objetivos desta aula:

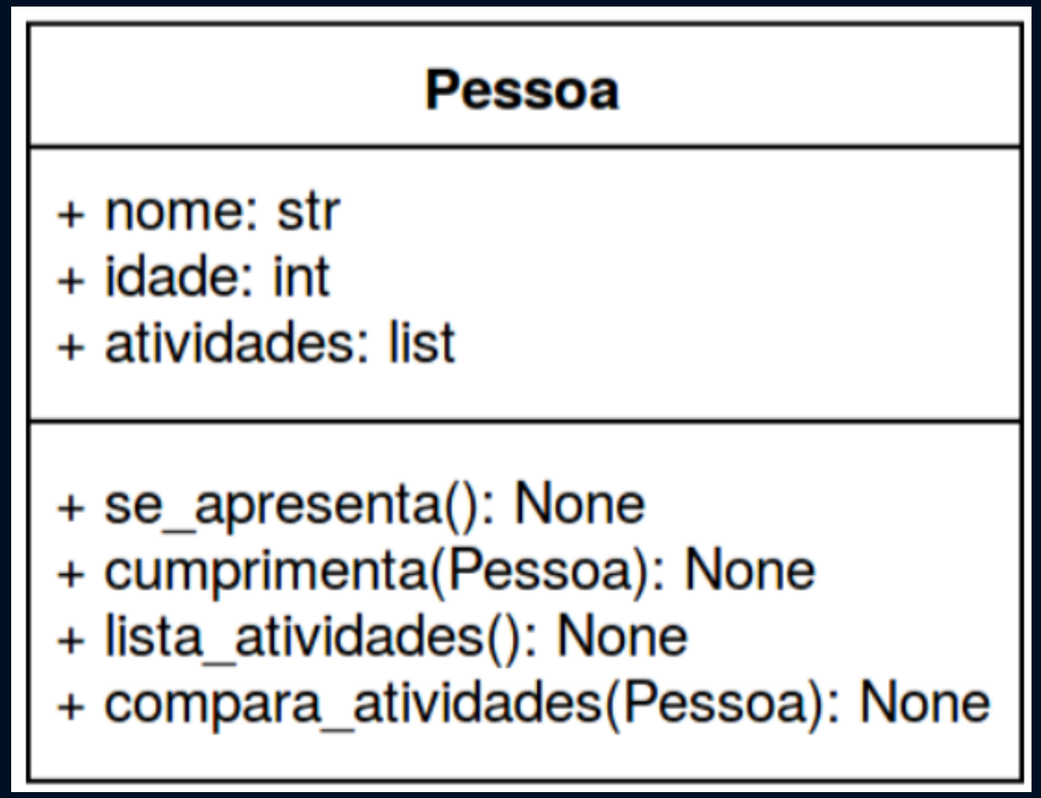
- Introduzir o conceito de diagrama de classes
- Explicar como os objetos se relacionam em um sistema orientado a objetos
- Identificar os diferentes tipos de relacionamento entre classes
- Implementar relações entre classes em Python

Implementação de Classes

- Até então, vocês implementaram uma (ou no máximo duas) classes para os programas solicitados
- A partir desta aula, deverá ficar claro quando mais de uma classe se torna necessário e qual o papel de cada uma delas
    - Como elas se relacionam entre si

Antes disto, para facilitar:
Como podemos mostrar/visualizar rapidamente o que possui uma classe?

Diagrama de Classe UML



Blocos representam classes:

- Parte superior: atributos
- Parte inferior: métodos
  - Apenas o tipo dos parâmetros e tipo de retorno
- - denota um atributo/método privado
- + denota um atributo/método público

Diagrama de classe da linguagem unificada de modelagem (UML)

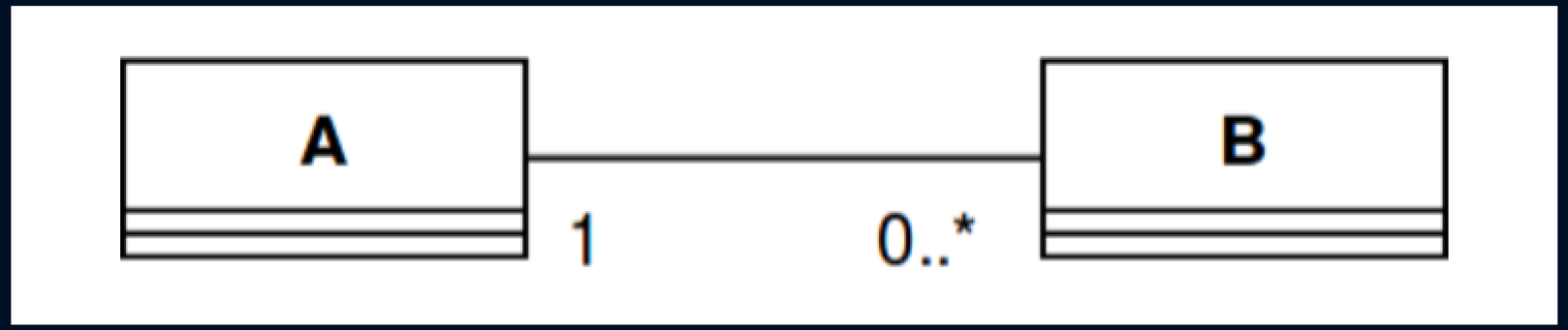
- Facilita a visão geral de sistemas maiores
- Ferramenta sugerida: Draw IO
    - Você pode utilizar qualquer outra

Relacionamentos de Associação

- Indicam algum relacionamento significativo e de interesse entre objetos
- Especificam que objetos de uma classe estão de alguma forma associados a objetos de uma outra classe
    - Pode indicar associação entre objetos de mesma classe

Exemplos:

- Em um Estacionamento estão estacionados os Carros
- Um Professor ministra várias Disciplinas
- Uma Mensagem possui um Remetente e um Destinatário
- Uma Pessoa tem outra Pessoa como a sua mãe



Notação de associação em diagrama de classes: linha conectando as classes que estão associadas

- Um objeto tipo A está associado com zero ou vários objetos tipo B
- Um objeto tipo B está associado com um objeto tipo A

Multiplicidade:

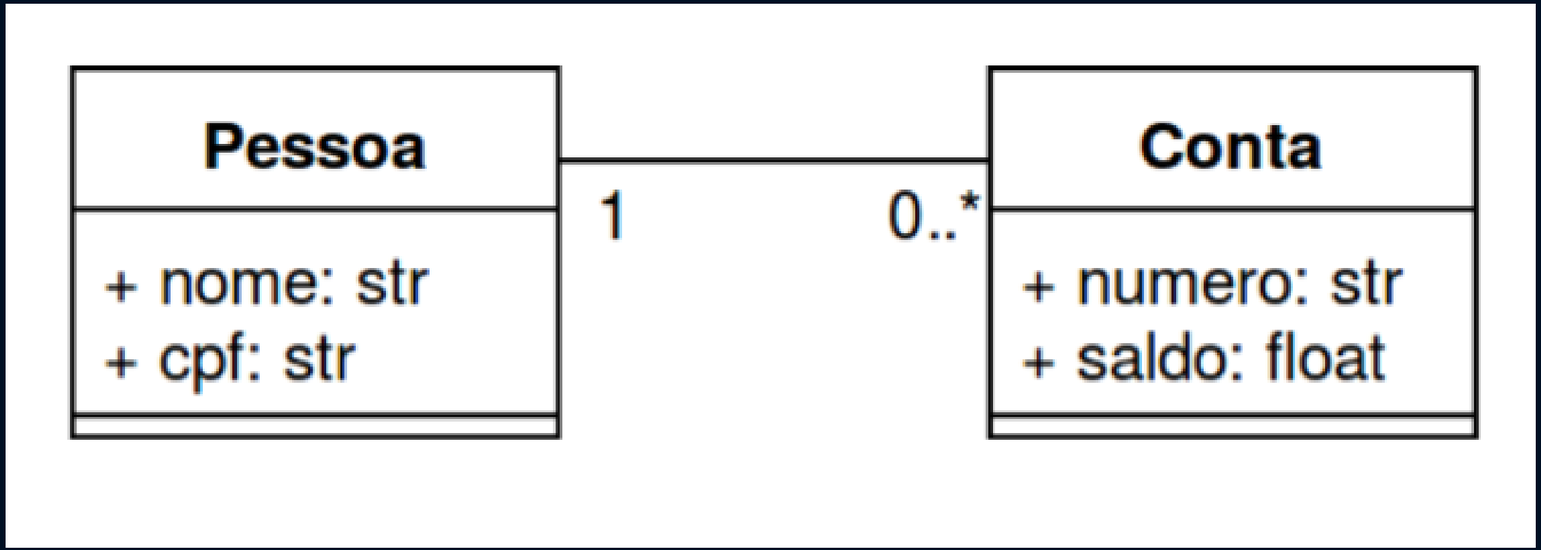
Nas extremidades da linha denotando a associação, está a multiplicidade:

- 0 : zero
- 0..1: zero ou um
- 1 : um
- 1..*: um ou mais
- 0..* ou *: zero ou mais

Relacionamentos de Associação

Exemplo

Um banco, no qual cada conta bancaria está relacionada com uma pessoa (o titular da conta):



Exemplo:

Observe que este diagrama representa uma lógica de negócio específica:

- Uma pessoa pode ter mais de uma conta? →→ Sim
- É possível existirem contas conjuntas? →→ Não

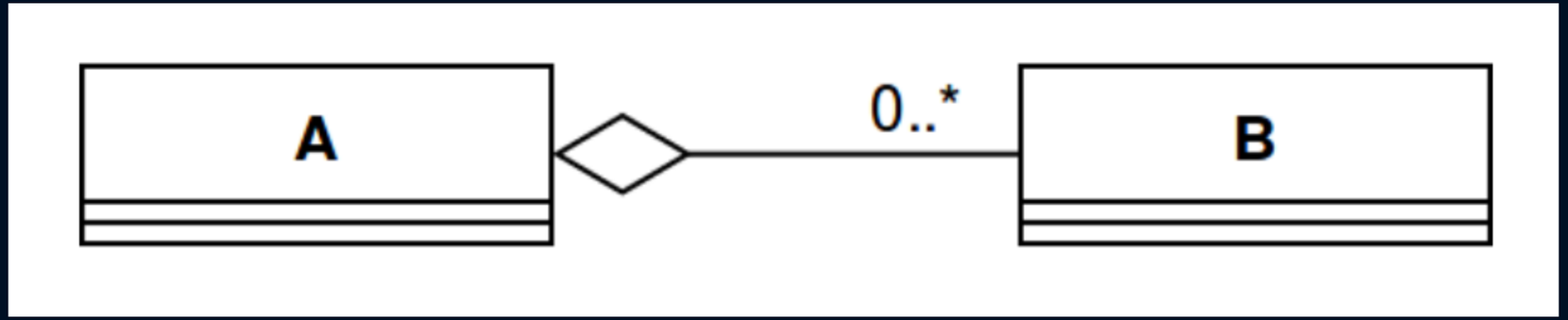
Relacionamentos de Agregação

- Representa uma relação todo-parte fraca
- As partes podem existir sem o todo

Exemplos:

- Relação entre o Carro, as Rodas e o Motor
- Relação entre um Computador, o Teclado, a Tela

Considere que em ambos os exemplos, o objeto todo pode ser instanciado sem as suas partes (e vice-versa)

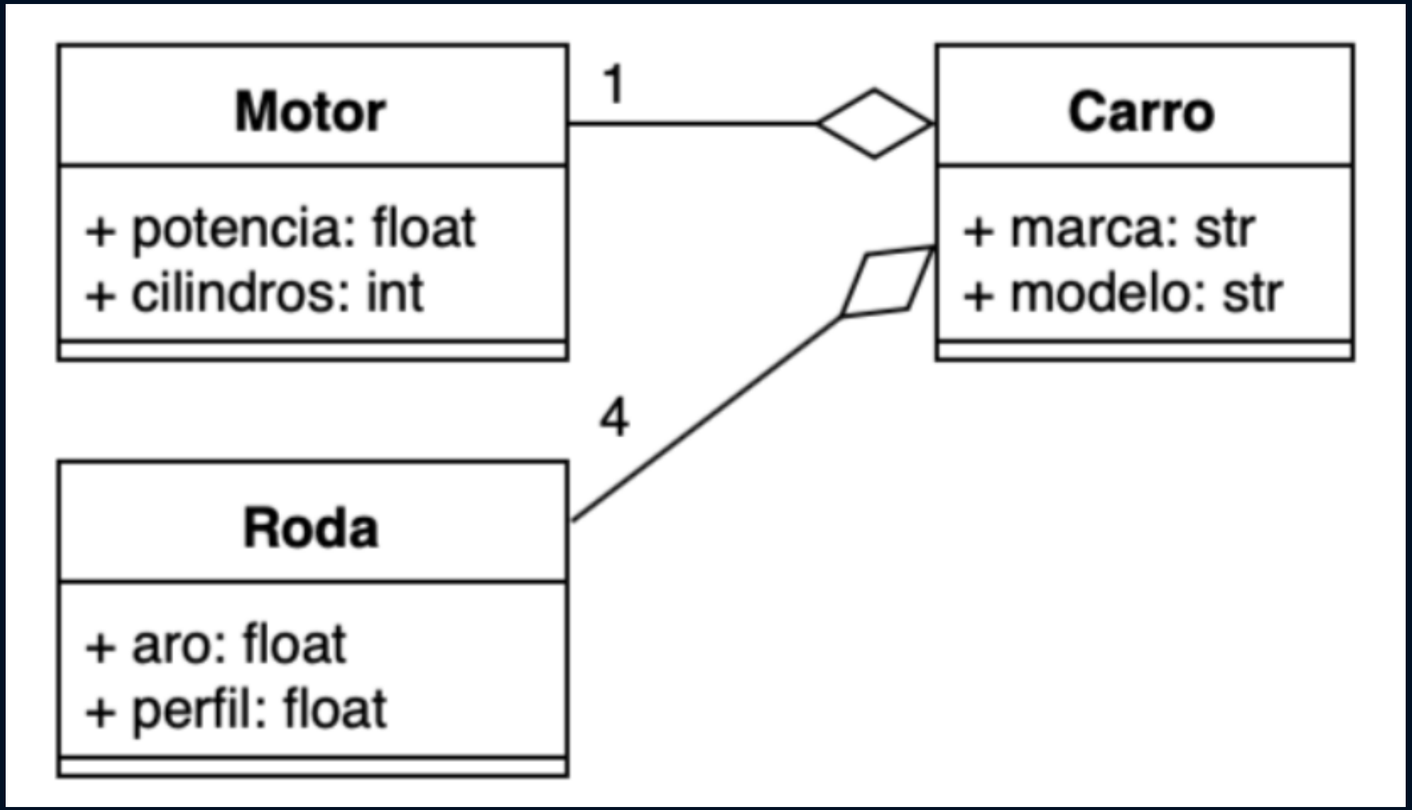


Em um diagrama de classes:

- A é o "todo"
  - Diamante vazado na classe "todo"
- B são as "partes"
- Linha conectando "todo" com "partes"

Exemplo

Um carro que possui 1 motor e 4 rodas



Com agregação:

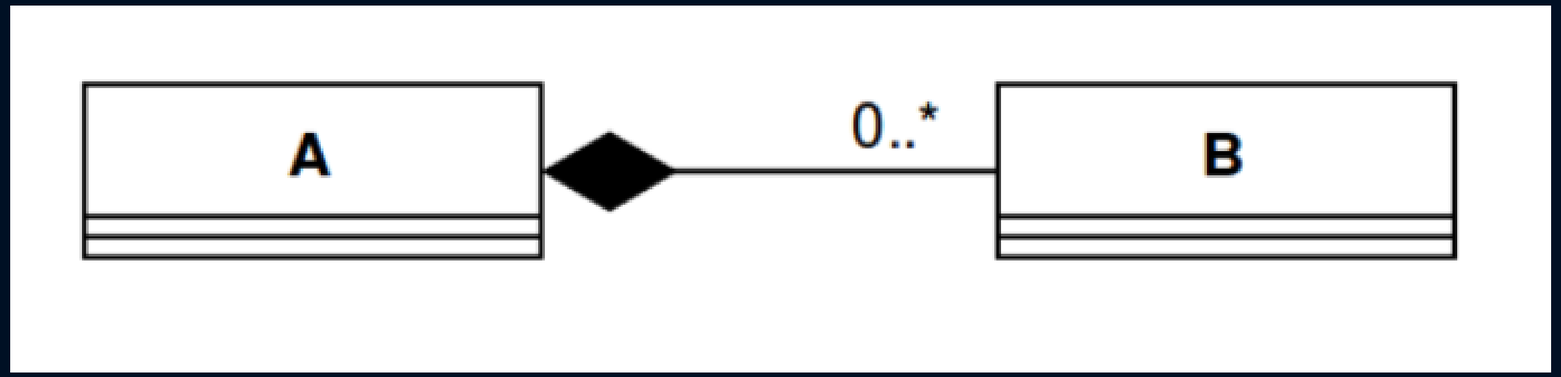
- Objetos Motor e Roda são instanciados
- Estas instâncias são então associadas a um Carro
- Ou seja, Motor, Roda e Carro podem ser instanciados independentemente
- Se um Carro é removido da memória, o Motor e Roda a ele associados continuam a existir

Relacionamentos de Composição

- Representa uma relação todo-parte forte
- Se o objeto todo deixar de existir, os seus objetos parte também devem deixar de existir

Exemplos:

- Um Estacionamento e seus Andares
- Um Prédio e suas Salas

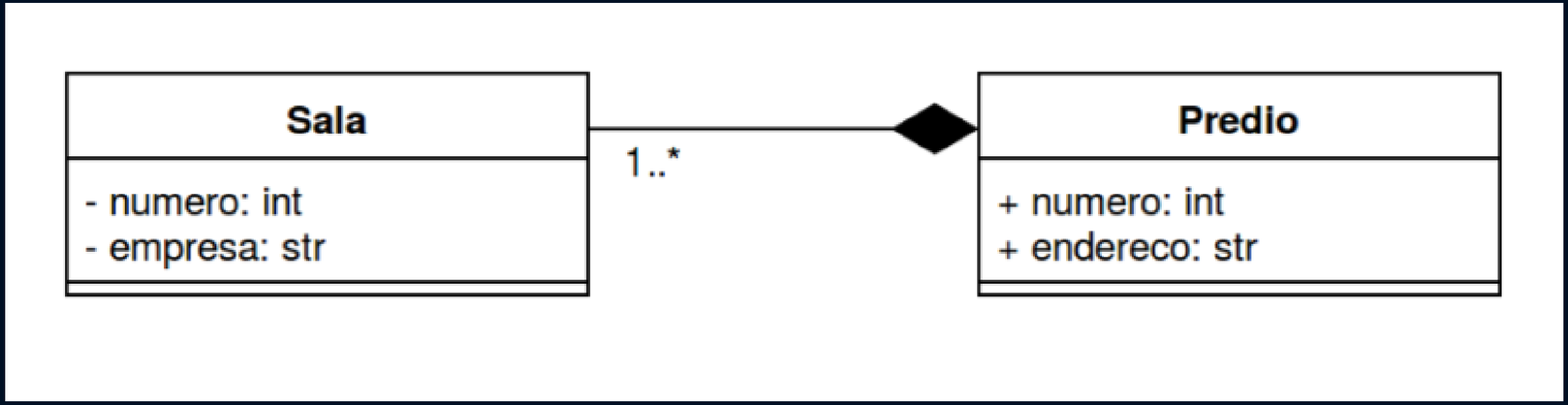


Em um diagrama de classes: juntamente com linha conectando as classes

- A é o "todo"
  - Diamante preenchido na classe "todo"
- Linha conectando "todo" com "partes"

Exemplo:

Prédio com várias salas



Com composição:

- Objetos Predio são instanciados e, dentro do seu inicializador, objetos Sala também o são
- Ou seja, Predio instancia Sala automaticamente
- Não é possível instanciar Predio sem Salas
- Se um Predio é removido da memória, as Sala a ele associadas também são removidas

Resumindo

Associação

- Relação genérica entre duas classes
- Nenhum objeto é considerado dono/proprietário do outro objeto

Agregação

- Relação tem um fraca: um objeto da classe A tem um (ou mais) objeto da classe B
- Objetos da classe A podem existir objetos da classe B
  - Observe que neste caso, o objeto da classe A pode não estar pronto para uso (estado inconsistente →→ ele pode precisar de objetos da classe B para funcionar corretamente)

Composição

- Relação tem um forte: um objeto da classe A tem um (ou mais) objeto da classe B
- Objetos da classe A instanciam também objetos da classe B
- Objetos da classe B deixam de existir quando objetos da classe proprietária A deixam de existir

Relações entre Objetos e Abstração

A escolha de uma forma de relação específica está atrelada à abstração empregada no domínio do problema

As classes Motor, Roda e Carro, por exemplo, podem ser implementadas como:

- Agregação, quando for de interesse que Motor e Roda sejam instanciados sem depender de um Carro
- Composição, quando for de interesse que Motor e Roda só sejam instanciados quando um Carro for instanciado

06-relacoes

Programação Orientada a Objetos

Herança

Objetivo

Apresentar o mecanismo de herança:

- O que é herança em POO
- Como utilizá-la na linguagem Python

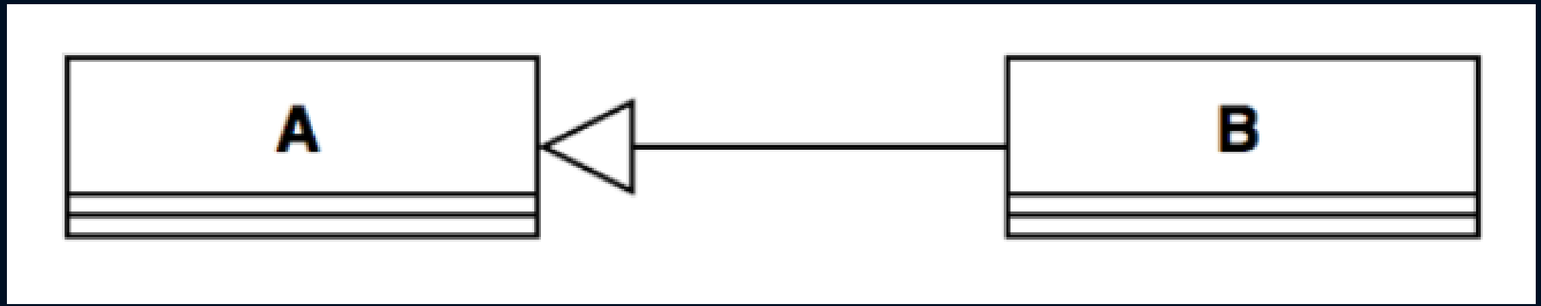
Os Quatro Pilares de POO

- Abstração
- Encapsulamento
- Herança: permite a reutilização de comportamento entre classes
- Polimorfismo

Herança

- Capacidade de uma classe herdar o comportamento definido por outra classe
- Possibilita a reutilização de código entre classes que apresentam alguma similaridade entre si
- Um novo tipo de relação entre classes
- Acontece entre objetos genéricos e objetos específicos

UML: Herança



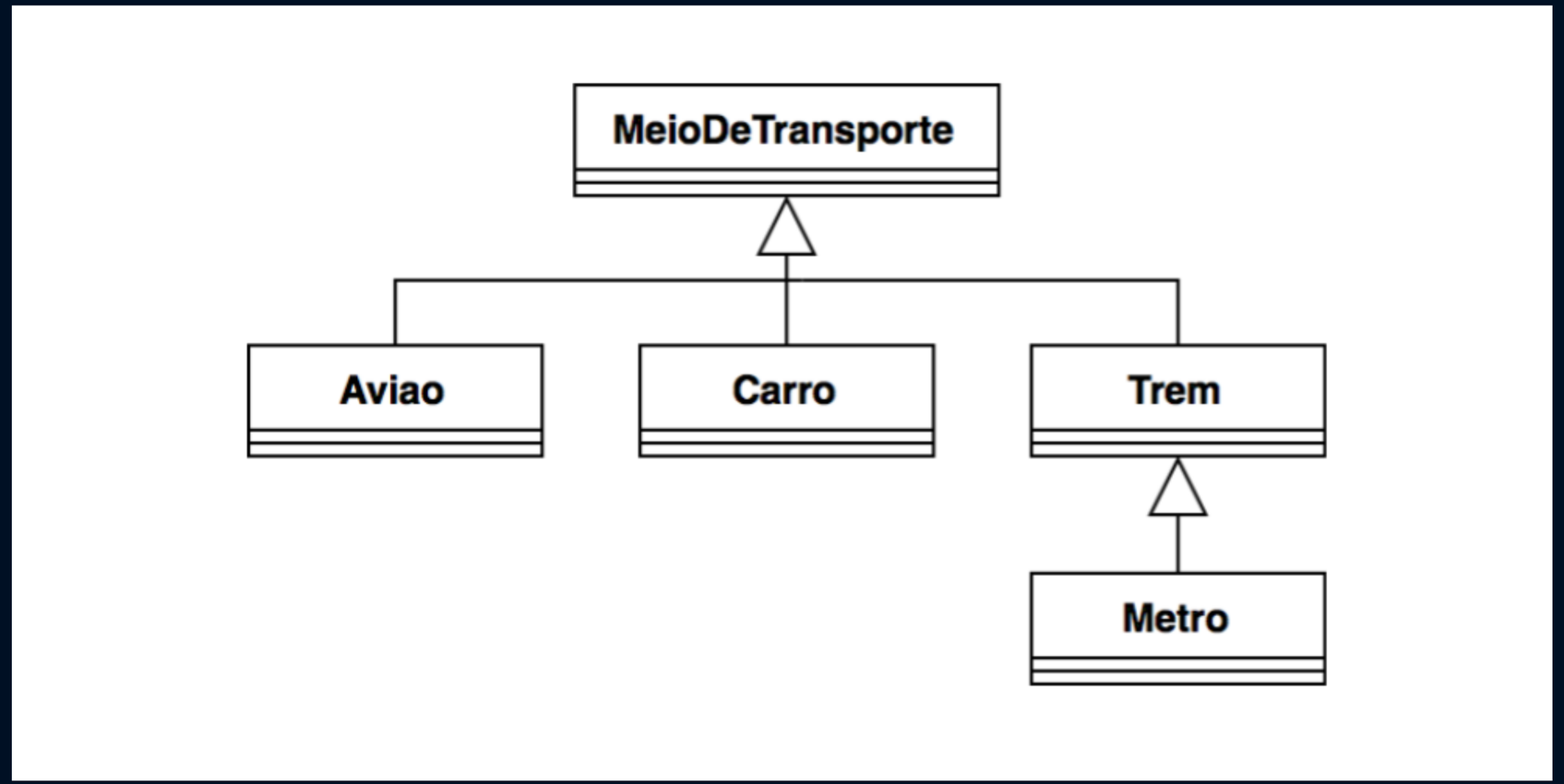
- Relação do tipo "é um": "um objeto B é um objeto A"
- Classe B herda o comportamento (atributos e métodos) da classe A
- A: classe base, classe mãe ou superclasse
- B: classe derivada, classe filha ou subclasse

- As superclasses devem oferecer comportamentos genéricos
- As subclasses devem oferecer comportamentos específicos

Exemplos

Relações de herança entre classes:

- Pessoa e Aluno: todo aluno (objeto específico) é uma pessoa (objeto genérico)
- MeioDeTransporte e Carro: todo carro (objeto específico) é um meio de transporte (objeto genérico)
- Sensor e Sonar: todo sonar (objeto específico) é um sensor (objeto genérico)
- Poligono e Triangulo: todo triângulo (objeto específico) é um polígono (objeto genérico)


MeioDeTransporte
Aviao
Carro
Trem
Metro

Por que precisamos de Herança?

- As relações de herança definem uma hierarquia de classes onde as subclasses herdam as características das suas superclasses
- É útil para definir um comportamento em comum para objetos de uma mesma hierarquia
  - O código da classe base é reutilizado em todas as subclasses
  - Qualquer alteração no código da classe base é propagado para todas as subclasses

- O comportamento comum a várias classes pode ser definido em uma superclasse
- Além de reutilizar código, as subclasses também podem:
  - Reescrever completamente os métodos que ditam o comportamento da classe (method overriding)

Estender os métodos que ditam o comportamento da classe (utilizando parte da implementação base)

08-heranca

Atributos e Métodos de Classe

Objetivo

Apresentar atributos e métodos de classe:

- Utilidade
- Sintaxe em Python

Atributos e Métodos de Classe

- Frequentemente, é desejável ter atributos/métodos que são "globais", ou seja, que não dizem respeito a uma instância específica de uma classe
- Por exemplo:
  - O endereço IP de um servidor de banco de dados
  - As possíveis cores de um Carro
  - A quantidade de instâncias criadas de uma classe

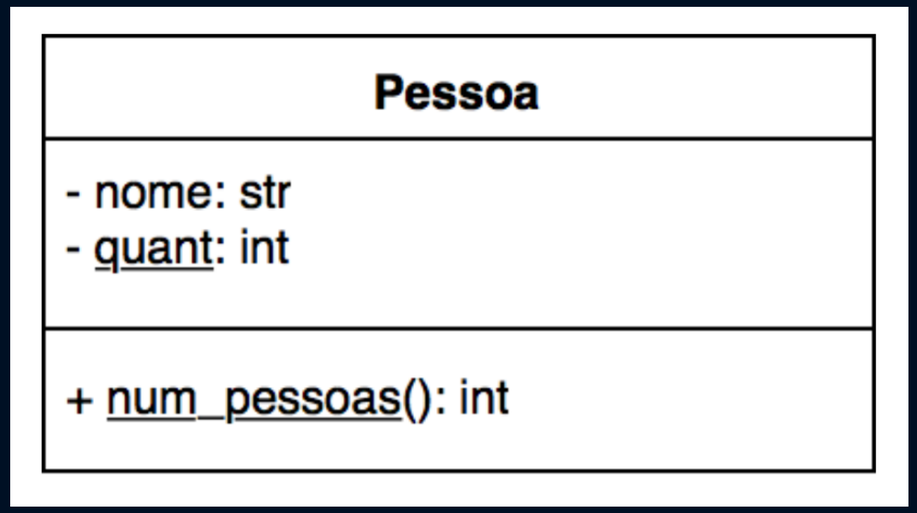
Atributos e Métodos de Instância vs. Atributos e Métodos de Classe

Até então, todos os atributos e métodos vistos são de instância

- Dizem respeito a uma instância/objeto específico daquela classe
- São comumente chamados de atributos/métodos (a parte de instância do nome atributo de instância é implícita)

- Atributos e métodos de classe dizem respeito à classe, e não a um objeto específico daquela classe
  - Também chamados de dados static ou atributos/métodos estáticos
  - Não precisam de uma instância da classe para serem utilizados
- Úteis para compartilhar informações entre todos os objetos de uma mesma classe

UML: Atributos e Métodos de Classe



Notação UML: texto sublinhado indica métodos e atributos de classe (estáticos)

Atributos e Métodos de Classe

Como é em Python (mais detalhes no notebook):

```python
class A:

    # Atributos de classe são declarados fora do __init__
    atributo_de_classe1 = ...
    atributo_de_classe2 = ...

    def __init__(self, ...):
      # Não confundir com os atributos de instância (declarados no __init__)
      self.atributo_de_instancia1 = ...
      self.atributo_de_instancia2 = ...
      self.atributo_de_instancia3 = ...

    # Método de classe não tem parâmetro self
    def metodo_de_classe():
```

09-static

Classes Abstratas

Objetivo da aula

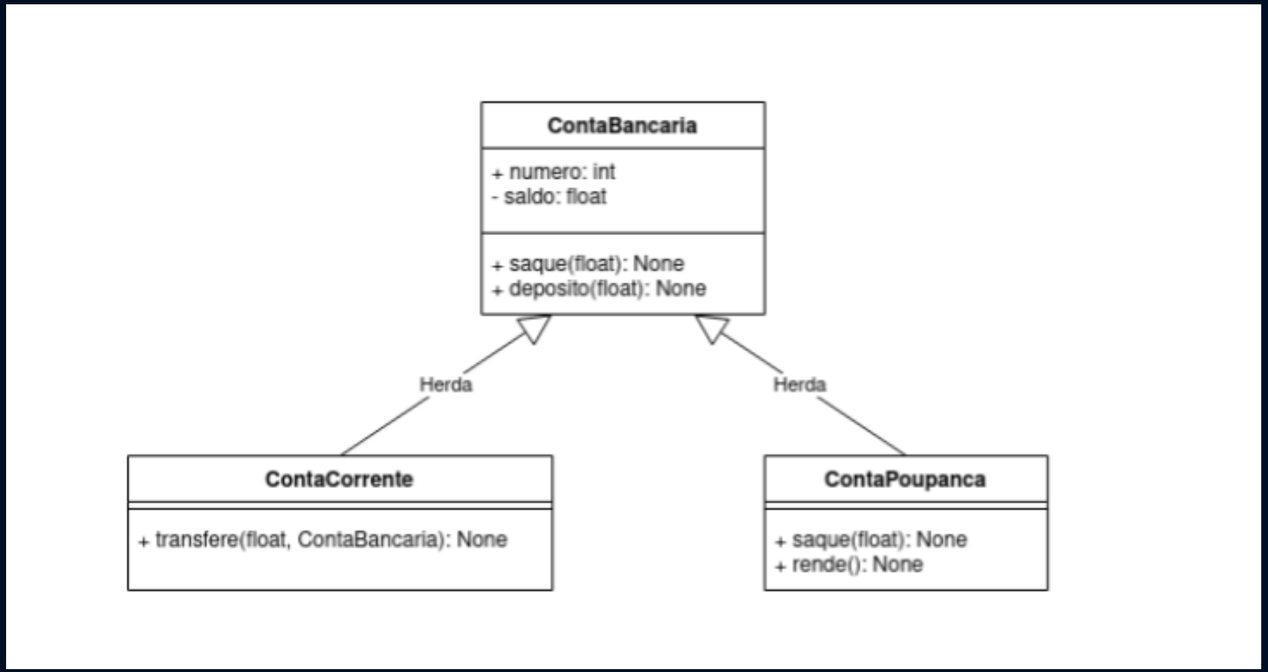
- Apresentar o mecanismo de classes abstratas
  - O que são classes abstratas
  - Identificar quando utilizar classes abstratas em um projeto orientado a objetos
  - Como utilizá-las na linguagem Python

Herança (revisão)

- Permite que classes derivadas herdem o comportamento (atributos e métodos) de uma classe base
- Introduz a relação "é um" (ex.: "trem" é um "meio de transporte")
- O código da superclasse é reutilizado pelas classes derivadas
- As classes derivadas podem sobrescrever métodos da superclasse com funcionalidades específicas

Revisão

Observe um possível diagrama do sistema de contas bancárias:



- Faz sentido criar objetos tipo Conta?
- Senão, qual é a utilidade dessa classe ?

Classe Abstrata

- É uma classe que não deve ser instanciada
- Define um comportamento comum para outras classes derivadas
- Toda classe que não é abstrata é chamada classe concreta

Classes abstratas são em geral utilizadas para:

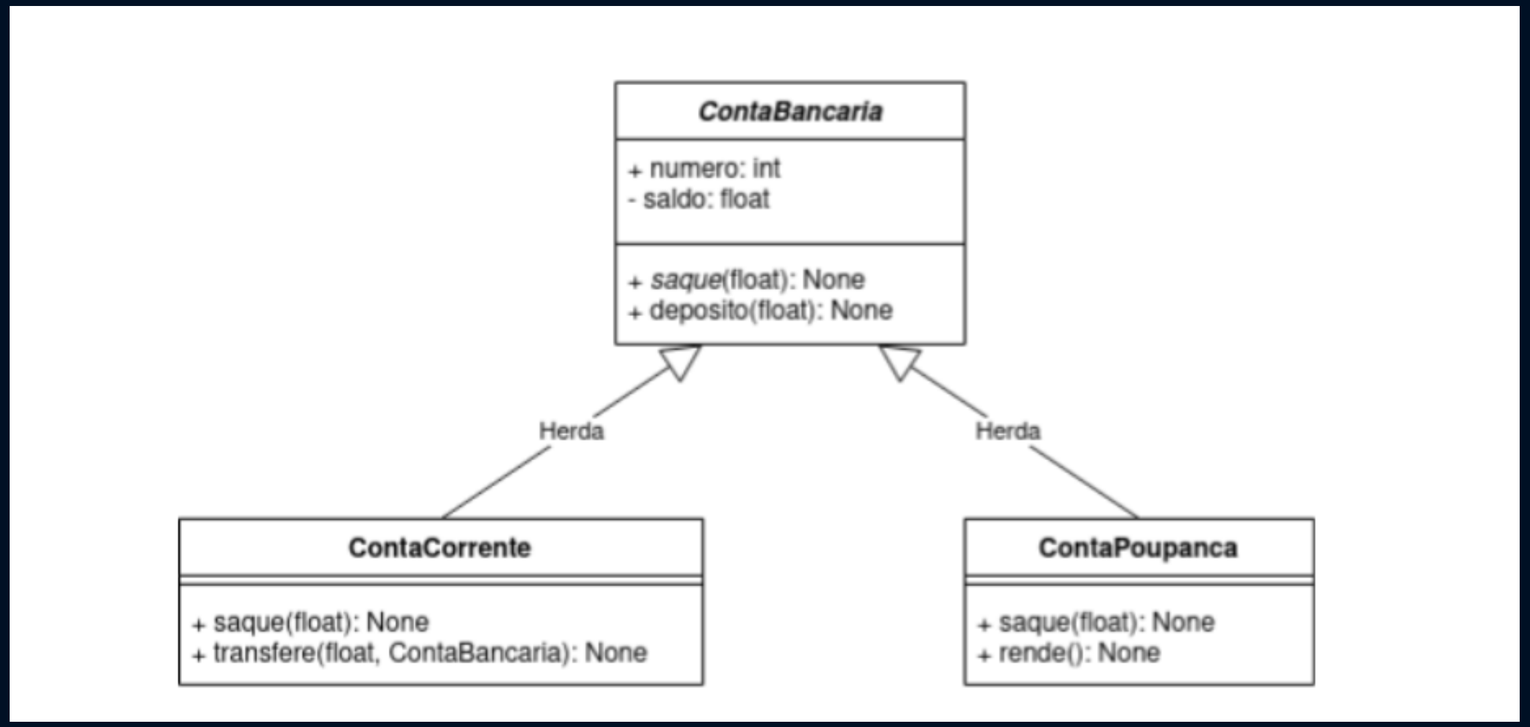
- Oferecer código para ser reutilizado por classes derivadas
- Especificar uma interface pública a ser implementada por classes derivadas
  - Quais métodos a classe derivada deve possuir
  - Que parâmetros eles devem ter

Método Abstrato

- Estão presentes somente em classes abstratas
- É um método que deve ser obrigatoriamente sobrescrito nas classes derivadas
- Se a classe possui pelo menos um método abstrato, então ela é uma classe abstrata (não é possível instanciar objetos desta classe)
- Em geral, um método abstrato não possui implementação
  - Apenas especifica tipo de retorno e parâmetros (igual à assinatura de uma função)
  - Em Python: podem possuir implementação

Classe Abstrata - UML

Na notação UML, uma classe abstrata possui seu nome em itálico:
Os métodos abstratos também são indicados com fonte itálica.



Observações Importantes

- Classes concretas não podem ter métodos abstratos
- Se a subclasse não sobrescreve todos os métodos abstratos da superclasse, ela também é abstrata
- Uma classe abstrata pode ter métodos abstratos e métodos implementados

Classe Abstrata

Uso em sistemas reais:

- Prover funcionalidade comum a uma hierarquia de classes
- Extensão de programas: por exemplo, plugins

Como Funciona na Prática

10-classes-abstratas

10-classes-abstratas

Polimorfismo

Objetivo da aula

Nesta aula veremos:

- O que é polimorfismo
- Duck typing
- Como a linguagem Python utiliza o polimorfismo

Os Pilares de POO

- Abstração
- Encapsulamento
- Herança
- Polimorfismo

Polimorfismo

Mecanismo presente em linguagens OO que permitem a um objeto se comportar de formas diferentes

- Poli: muitos, morfismo: formas
- Mais um recurso utilizado para promover a reutilização de código
- Um mesmo código possui diferentes comportamentos

Na prática, polimorfismo é oferecido por linguagens OO através de vários mecanismos. Estes mecanismos são mostrados a seguir, utilizando códigos de outras linguagens.

Polimorfismo em Linguagens Tipadas

1. Sobrecarga de operadores: algumas linguagens permitem implementar um método que define o comportamento dos operadores +, -, *, /, >, <, ==, etc.

- C++ permite
- C# permite
- Objective-C não permite
- Java não permite

Python Funciona de Outra Forma

1. Objetos da classe base recebendo objetos de classes derivadas:

```
Pessoa p = new Funcionario(...);
```

Não tem sentido, porque em Python, uma variável pode ser atribuída a objetos de classes diferentes (que não necessariamente pertencem à mesma hierarquia)

```
x = 4
x = [1,2,3]
x = "alo"
x = Pessoa(...)
x = Carro(...)
```

2. Chamar funções/métodos com instâncias de alguma subclasse como parâmetro:
Como não temos tipos, podemos chamar uma função/método com objetos de classes diferentes (não necessariamente objetos de uma subclasse):

```
str(3)
str(3.2)
p = Pessoa(...)
str(p)
####
len("alo")
len([1,2,3])
```

3. Sobrecarga de funções:

```
def funcao(x):
 ...

def funcao(x,y):  # a definição anterior de f é substitu
 ...

f(4) # Erro! 2 parâmetros são esperados
```

Em Python, não podemos criar 2 funções com diferentes assinaturas

Entretanto, o seguinte código funciona perfeitamente:

```
class Pessoa:
    def __init__(self, nome, idade): ...

    def compara_idades(self, p2):
        return p1.idade <= p2.idade

class Aluno(Pessoa): ...
class Professor(Pessoa): ...

p = Pessoa('joão', 25)
a = Aluno('maria', 20, 111)
print(p.compara_idades(a)) # método funciona pq um Aluno
```

Polimorfismo

Com polimorfismo, a chamada dos métodos é polimórfica: a mesma chamada pode se comportar de diferentes formas, de acordo com a classe dos objetos envolvidos

O princípio da substituição de Liskov

- Uma classe base deve poder ser substituída pela sua classe derivada
- Considere o método q(x). Se q pode ser utilizado com objetos da superclasse T, então q deve poder também ser chamado com um objeto de uma subclasse S derivada de T

Polimorfismo em Python

Python implementa todas as funcionalidades apresentadas até então seguindo um conceito chamado como duck typing

Duck Typing

Quando eu vejo um pássaro que anda como pato, nada como um pato e grasna como pato, então pra mim este pássaro é um pato

- Princípio utilizado como base da linguagem Python
- Lembre-se que Python possui tipagem dinâmica: o tipo dos objetos só pode ser determinado na execução do programa

Forma de tipagem que está mais interessada no que o objeto possui como atributos/métodos do que se ele é de uma determinada classe
Exemplos de uso de Duck typing:

- Quando usamos print(a): não interessa a classe de a, o objeto vai ser impresso (e o método __str__ é chamado)
- Quando usamos a.liga(): não interessa se o objeto a é da classe Motor ou da classe Lampada. A chamada irá ter sucesso ser o objeto possuir o método liga

4. Sobrecarga de operadores:
Python suporta através da implementação de métodos mágicos.

11-polimorfismo

Herança Múltipla

Objetivo da aula:

- Apresentar o mecanismo de herança múltipla
  - O que é herança múltipla
  - Como utilizá-lo na linguagem Python

Herança:

- Permite que classes derivadas herdem o comportamento (atributos e métodos) de uma classe base
- Introduz a relação "é um"
- Código na classe base pode ser reutilizado nas classes derivadas
- Classe derivada pode reimplementar um método com funcionalidades específicas

Herança Múltipla

Ocorre quando a classe derivada possui mais de uma classe base

Em Python, as classes base são indicadas por uma tupla:

```
class Subclasse(Superclasse1, Superclasse2):
...
```

- Subclasse é a classe derivada
- Todos os atributos e métodos de Superclasse1 e Superclasse2 estão na subclasse

Herança Múltipla

As superclasses também podem ser classes abstratas

Todos os métodos abstratos de todas as superclasses abstratas têm que ser implementados para que a subclasse seja concreta

Caso contrário, a subclasse se torna uma classe abstrata

12-Heranca-Multipla.ipynb

Erros e Exceções

Motivação

Considere os métodos a seguir:

```
@nome.setter
def nome(self, n):
    '''Set para o nome de uma pessoa'''
    if type(n) == str:
        self._nome = n
    else:
        print('n precisa ser do tipo string')
```

```
def registraEntrada(self):
    '''Entra um carro'''
    if self.vagas > 0:
        self.vagas -= 1
        print("Um carro entrou.")
    else:
        print("Estacionamento sem vagas")
```

- O programa apenas imprime uma mensagem de erro
- Entretanto, a execução do programa continua
- Como fazer para o programa encerrar a sua execução?

Mais ainda:

- Como notificar o restante do programa que a execução do método não foi bem sucedida?
- Como tratar/detectar (por exemplo, por quem chamou o método) que a execução do método não foi bem sucedida?

Estamos interessados em fazer o programa "emitir um erro":

- Emitir erro: notificar e encerrar o programa
- Fazer o programa "Levantar uma exceção"
  - Do inglês "raise an exception"
  - Também conhecido em português como "lançar uma exceção" ou "subir uma exceção"

Você já deve ter se deparado com alguns erros em Python:
Índice não valido (fora dos limites):

```
IndexError: list index out of range
```

Divisão por 0:

```
ZeroDivisionError: division by zero
```

Você já deve ter se deparado com alguns erros em Python:

- Uma variável inexistente é utilizada:

```
NameError: name x is not defined
```

Objetivo

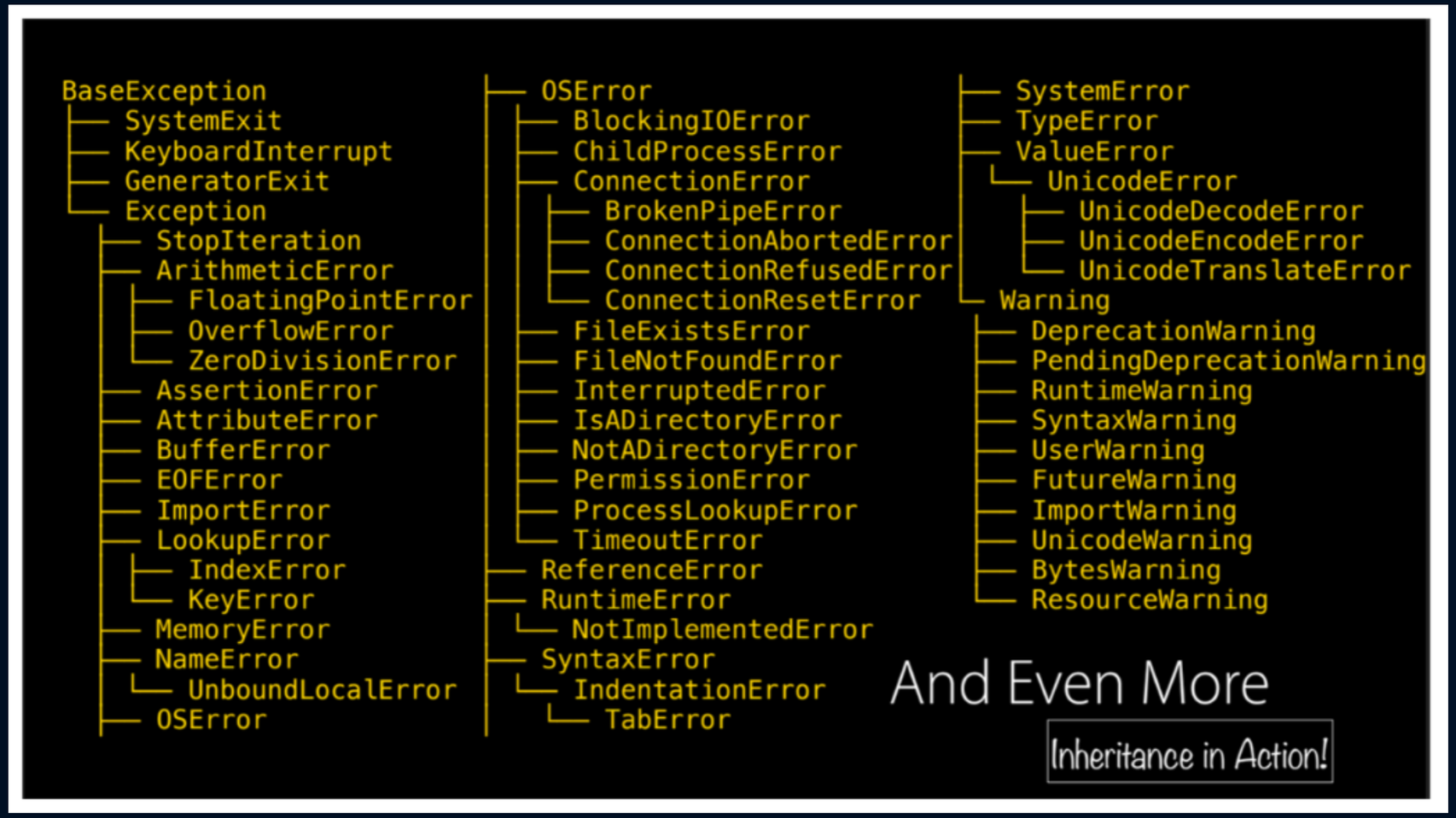
Apresentar o mecanismo de tratamento de exceções

- O que é uma exceção
- Como levantar exceções em Python
- Como tratar exceções em Python
- Como implementar classes que representam exceções em Python

Exceções e Tratamento

- Uma exceção é um erro não necessariamente fatal detectável na execução de um programa
  - Diferente de erro de compilação
- Para levantar uma exceção, utilizamos o comando raise
  - Similar ao comando throw de Java e C++

- Exception é a superclasse das exceções definidas pelo usuário
- A lista de classes de exceções predefinidas em Python está aqui



(Imagem de Stanford Python Course)

Tratamento de Exceções

- Após levantada, uma exceção pode ser tratada
- Tratamento de exceções: trecho de código responsável por fazer o programa se recuperar da exceção levantada
- De acordo com a classe da exceção, o programa pode tratá-la de forma diferente

Ideia Geral

- A cláusula try contém um bloco de código que pode levantar exceções. Ela tenta executar o bloco de comando nela contido
- Se uma exceção for levantada dentro do try, o fluxo do programa é redirecionado para a cláusula except

```
try:
    # Tenta executar o bloco
except:
    #tratamento
```

14-Excecoes.ipynb

Interfaces Gráficas

Motivação

- A grande maioria dos aplicativos utilizados por nós possui uma interface gráfica
  - Janela com vários componentes em que o usuário pode inserir comandos
- Interfaces gráficas melhoram significativamente a usabilidade de um programa

Existem diversas opções de interfaces gráficas em Python:

- GTK (Gnome Toolkit)
- QT
- Wxwidgets
- etc.

Na disciplina de POO, iremos trabalhar com a interface gráfica Tk:

- Tkinter: Tk Interface
- Considerada padrão para a linguagem Python
- Kit de ferramentas com várias opções e de fácil utilização
- Multiplataforma

Interfaces Gráficas

Programar uma aplicação com interface gráfica requer utilizar widgets (window gadgets): componentes de uma interface gráfica

Alguns tipos de widgets:

- Label (rótulo)
- Botão
- Campo para entrada de texto
- Barra de menu
- Barra de rolagem
- etc.

Desenvolver uma interface gráfica envolve:

- Posicionar e configurar os widgets e suas propriedades (cores, fontes, tamanho, etc.)
- Programar como o sistema deve reagir a eventos ao longo da sua execução:
  - Clique de mouse, movimento de mouse, clique de botão, minimização de janela, seleção de texto, seleção de item em uma lista, etc.

- Portanto, o desenvolvimento de interfaces gráficas corresponde a uma camada a mais no desenvolvimento do programa como um todo
- Idealmente: o código correspondente à interface gráfica deve estar em um módulo separado da lógica do programa
- Orientação a objetos facilita esta modularização

Interfaces Gráficas

Jupyter notebook

15-GUI.ipynb

16-MVC.ipynb